Pengantar Sistem Operasi Komputer

Plus Ilustrasi Kernel Linux

Masyarakat Digital Gotong Royong (MDGR)

Pengantar Sistem Operasi Komputer: Plus Ilustrasi Kernel Linux

oleh Masyarakat Digital Gotong Royong (MDGR)

Diterbitkan $Date: 2006-08-28 15:36:08 $
Hak Cipta © 2003-2006 Masyarakat Digital Gotong Royong (MDGR).

Silakan menyalin, mengedarkan, dan/atau, memodifikasi bagian dari dokumen – $Revision: 4.0 $ – – yang dikarang oleh Masyarakat Digital Gotong Royong (MDGR), sesuai dengan ketentuan "*GNU Free Documentation License* versi 1.2" atau versi selanjutnya dari FSF (*Free Software Foundation*); tanpa bagian "*Invariant*", tanpa teks "*Front-Cover*", dan tanpa teks "*Back-Cover*". Lampiran A ini berisi salinan lengkap dari lisensi tersebut. **BUKU INI HASIL KERINGAT DARI RATUSAN JEMAAH MDGR (BUKAN KARYA INDIVIDUAL). JANGAN MENGUBAH/MENGHILANGKAN LISENSI BUKU INI. SIAPA SAJA DIPERSILAKAN UNTUK MENCETAK/MENGEDARKAN BUKU INI!** Seluruh ketentuan di atas **TIDAK** berlaku untuk bagian dan/atau kutipan yang bukan dikarang oleh Masyarakat Digital Gotong Royong (MDGR). Versi digital terakhir dari buku ini dapat diambil dari http://bebas.vlsm.org/ v06/ Kuliah/ SistemOperasi/ BUKU/.

Catatan Revisi

Revisi 4.00	28-Agustus-2006	RMS46
Perbaikan selanjutnya.		
Revisi 4.00	28-Agustus-2006	RMS46
Revisi 4.00		
Revisi 3.64	14-Agustus-2006	RMS46
Mei-Agustus 2006: Pemolesan		
Revisi 3.42	04-Mei-2006	RMS46
April-Mei 2006: Mengosongkan Appendix C: (UPDATE).		
Revisi 3.37	06-April-2006	RMS46
Start Feb2006: Gusur Appendix B: Soal Latihan.		
Revisi 3.27	22-Februari-2006	RMS46
Full XML (was SGML), start update kelompok hingga bab 47.		
Revisi 3.00	26-Agustus-2005	RMS46
Selesai tidak selesai, ini revisi 3.00!		
Revisi 2.34	26-Agustus-2005	RMS46
Memperbaiki sana-sini.		
Revisi 2.24	5-Agustus-2005	RMS46
Mempersiapkan seadanya versi 3.0		
Revisi 2.17	27-Juli-2005	RMS46
Mengubah dari SGML DocBook ke XML DocBook.		
Revisi 2.10	03-Mar-2005	RMS46
Membereskan dan memilah 52 bab.		
Revisi 2.4	02-Dec-2004	RMS46
Update 2.0+. Ubah sub-bab menjadi bab.		
Revisi 2.0	09-09-2004	RMS46
Menganggap selesai revisi 2.0.		
Revisi 1.10	09-09-2004	RMS46
Pesiapan ke revisi 2.0		
Revisi 1.9.2.10	24-08-2004	RMS46
Ambil alih kelompok 51, perbaikan isi buku.		
Revisi 1.9.1.2	15-03-2004	RMS46
Revisi lanjutan: perbaikan sana-sini, ejaan, indeks, dst.		
Revisi 1.9.1.0	11-03-2004	RMS46
Revisi ini diedit ulang serta perbaikan sana-sini.		
Revisi 1.9	24-12-2003	Kelompok 49
Versi rilis final buku OS.		
Revisi 1.8	08-12-2003	Kelompok 49
Versi rilis beta buku OS.		
Revisi 1.7	17-11-2003	Kelompok 49
Versi rilis alfa buku OS.		
Revisi 1.5	17-11-2003	Kelompok 49
Penggabungan pertama (kel 41-49), tanpa indeks dan rujukan utama. ada.		

Revisi 1.4	08-11-2003	Kelompok 49
Pengubahan template versi 1.3 dengan template yang baru yang akan digunakan dalam versi 1.4-2.0		
Revisi 1.3.0.5	12-11-2003	RMS46
Dipilah sesuai dengan sub-pokok bahasan yang ada.		
Revisi 1.3	30-09-2003	RMS46
Melanjutkan perbaikan tata letak dan pengindeksan.		
Revisi 1.2	17-09-2003	RMS46
Melakukan perbaikan struktur SGML, tanpa banyak mengubah isi buku.		
Revisi 1.1	01-09-2003	RMS46
Kompilasi ulang, serta melakukan sedikit perapihan.		
Revisi 1.0	27-05-2003	RMS46
Revisi ini diedit oleh Rahmat M. Samik-Ibrahim (RMS46).		
Revisi 0.21.4	05-05-2003	Kelompok 21
Perapihan berkas dan penambahan entity.		
Revisi 0.21.3	29-04-2003	Kelompok 21
Perubahan dengan menyempurnakan nama file.		
Revisi 0.21.2	24-04-2003	Kelompok 21
Merubah Kata Pengantar.		
Revisi 0.21.1	21-04-2003	Kelompok 21
Menambahkan Daftar Pustaka dan Index.		
Revisi 0.21.0	26-03-2003	Kelompok 21
Memulai membuat tugas kelompok kuliah Sistem Operasi.		

Persembahan



Buku "Kunyuk" ini dipersembahkan *dari* Masyarakat Digital Gotong Royong (MDGR), *oleh* MDGR, *untuk* siapa saja yang ingin mempelajari Sistem Operasi dari sebuah komputer. Buku ini **bukan** merupakan karya individual, melainkan merupakan hasil keringat dari **ratusan** jemaah MDGR! MDGR ini merupakan Gabungan Kelompok Kerja 21–28 Semester Genap 2002/2003, 41–49 Semester Ganjil 2003/2004, 51 Semester Genap 2003/2004, 53–58 Semester Ganjil 2004/2005, 81–89 Semester Genap 2004/2005, 111–120 Semester Ganjil 2005/2006, dan 150 Semester Genal 2005/2006 Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia (http://rms46.vlsm.org/2/133.html -- http://www.cs.ui.ac.id/) yang namanya tercantum berikut ini:

Kelompok 21 (2003). Kelompok ini merupakan penjamin mutu yang bertugas mengkoordinir kelompok 22-28 pada tahap pertama dari pengembangan buku ini. Kelompok ini telah mengakomodir semua ide dan isu yang terkait, serta proaktif dalam menanggapi isu tersebut. Tahap ini cukup sulit dan membingungkan, mengingat sebelumnya belum pernah ada tugas kelompok yang dikerjakan secara bersama dengan jumlah anggota yang besar. Anggota dari kelompok ini ialah: Dhani Yuliarso (Ketua), Fernan, Hanny Faristin, Melanie Tedja, Paramanandana D.M., Widya Yuwanda.

Kelompok 22 (2003). Kelompok ini merancang bagian (bab 1 versi 1.0) yang merupakan penjelasan umum perihal sistem operasi serta perangkat keras/lunak yang terkait. Anggota dari kelompok ini ialah: Budiono Wibowo (Ketua), Agus Setiawan, Baya U.H.S., Budi A. Azis Dede Junaedi, Heriyanto, Muhammad Rusdi.

Kelompok 23 (2003). Kelompok ini merancang bagian (bab 2 versi 1.0) yang menjelaskan managemen proses, *thread*, dan penjadwalan. Anggota dari kelompok ini ialah: Indra Agung (Ketua), Ali Khumaidi, Arifullah, Baihaki Ageng Sela, Christian K.F. Daeli, Eries Nugroho, Eko Seno P., Habrar, Haris Sahlan.

Kelompok 24 (2003). Kelompok ini merancang bagian (bab 3 versi 1.0) yang menjelaskan komunikasi antar proses dan *deadlock*. Anggota dari kelompok ini ialah: Adzan Wahyu Jatmiko (Ketua), Agung Pratomo, Dedy Kurniawan, Samiaji Adisasmito, Zidni Agni.

Kelompok 25 (2003). Kelompok ini merancang bagian (bab 4 versi 1.0) yang menjelaskan segala hal yang berhubungan dengan memori komputer. Anggota dari kelompok ini ialah: Nasrullah (Ketua), Amy S. Indrasari, Ihsan Wahyu, Inge Evita Putri, Muhammad Faizal Ardhi, Muhammad Zaki Rahman, N. Rifka N. Liputo, Nelly, Nur Indah, R. Ayu P., Sita A.R.

Kelompok 26 (2003). Kelompok ini merancang bagian (bab 5 versi 1.0) yang menjelaskan segala hal yang berhubungan dengan managemen sistem berkas. Anggota dari kelompok ini ialah: Rakhmad Azhari (Ketua), Adhe Aries P., Adityo Pratomo, Aldiantoro Nugroho, Framadhan A., Pelangi, Satrio Baskoro Y.

Kelompok 27 (2003). Kelompok ini merancang bagian (bab 6 versi 1.0) yang menjelaskan segala hal yang berhubungan dengan managemen M/K dan Disk. Anggota dari kelompok ini ialah: Teuku Amir F.K. (Ketua), Alex Hendra Nilam, Anggraini Widjanarti, Ardini Ridhatillah, R. Ferdy Ferdian, Ripta Ramelan, Suluh Legowo, Zulkifli.

Kelompok 28 (2003). Kelompok ini merancang bagian (bab 7 versi 1.0) yang menjelaskan segala hal yang berhubungan dengan Studi Kasus GNU/Linux. Anggota dari kelompok ini ialah: Christiono H3ndra (Ketua), Arief Purnama L.K., Arman Rahmanto, Fajar, Muhammad Ichsan, Rama P. Tardan, Unedo Sanro Simon.

Kelompok 41 (2003). Kelompok ini menulis ulang bagian (bab 1 versi 2.0) yang merupakan pecahan bab 1 versi sebelumnya. Anggota dari kelompok ini ialah: Aristo (Ketua), Ahmad Furqan S K., Obeth M S.

Kelompok 42 (2003). Kelompok ini menulis ulang bagian (bab 2 versi 2.0) yang merupakan bagian akhir dari bab 1 versi sebelumnya. Anggota dari kelompok ini ialah: Puspita Kencana Sari (Ketua), Retno Amelia, Susi Rahmawati, Sutia Handayani.

Kelompok 43 (2003). Kelompok ini menulis ulang/memperbaiki bagian (bab 3 versi 2.0, ex bab 2 versi 1.0) yang membahas managemen proses, *thread*, dan penjadwalan. Anggota dari kelompok ini ialah: Agus Setiawan (Ketua), Adhita Amanda, Afaf M, Alisa Dewayanti, Andung J Wicaksono, Dian Wulandari L, Gunawan, Jefri Abdullah, M Gantino, Prita I.

Kelompok 44 (2003). Kelompok ini menulis ulang/memperbaiki bagian (bab 4 versi 2.0, ex bab 3 versi 1.0) yang membahas komunikasi antar proses dan *deadlock*. Anggota dari kelompok ini ialah: Arnold W (Ketua), Antonius H, Irene, Theresia B, Ilham W K, Imelda T, Dessy N, Alex C.

Kelompok 45 (2003). Kelompok ini menulis ulang/memperbaiki bagian (bab 5 versi 2.0, ex bab 4 versi 1.0) yang membahas segala hal yang berhubungan dengan memori komputer. Anggota dari kelompok ini ialah: Bima Satria T (Ketua), Adrian Dwitomo, Alfa Rega M, Boby, Diah Astuti W, Dian Kartika P, Pratiwi W, S Budianti S, Satria Graha, Siti Mawaddah, Vita Amanda.

Kelompok 46 (2003). Kelompok ini menulis ulang/memperbaiki bagian (bab 6 versi 2.0, ex bab 5 versi 1.0) yang membahas segala hal yang berhubungan dengan managemen sistem berkas. Anggota dari kelompok ini ialah: Josef (Ketua), Arief Aziz, Bimo Widhi Nugroho, Chrysta C P, Dian Maya L, Monica Lestari P, Muhammad Alaydrus, Syntia Wijaya Dharma, Wilmar Y Ignesjz, Yenni R.

Kelompok 47 (2003). Kelompok ini menulis ulang/memperbaiki bagian (bab 7 versi 2.0, ex bab 6 versi 1.0) yang membahas segala hal yang berhubungan dengan managemen M/K dan Disk. Anggota dari kelompok ini ialah: Bayu Putera (Ketua), Enrico, Ferry Haris, Franky, Hadyan Andika, Ryan Loanda, Satriadi, Setiawan A, Siti P Wulandari, Tommy Khoerniawan, Wadiyono Valens, William Hutama.

Kelompok 48 (2003). Kelompok ini menulis ulang/memperbaiki bagian (bab 8 versi 2.0, ex bab 7 versi 1.0) yang membahas segala hal yang berhubungan dengan Studi Kasus GNU/Linux. Anggota dari kelompok ini ialah: Amir Murtako (Ketua), Dwi Astuti A, M Abdushshomad E, Mauldy Laya, Novarina Azli, Raja Komkom S.

Kelompok 49 (2003). Kelompok ini merupakan koordinator kelompok 41-48 pada tahap kedua

pengembangan buku ini. Kelompok ini selain kompak, juga sangat kreatif dan inovatif. Anggota dari kelompok ini ialah: Fajran Iman Rusadi (Ketua), Carroline D Puspa.

Kelompok 51 (2004). Kelompok ini bertugas untuk memperbaiki bab 4 (versi 2.0) yang membahas komunikasi antar proses dan *deadlock*. Anggota dari kelompok ini ialah: V.A. Pragantha (Ketua), Irsyad F.N., Jaka N.I., Maharmon, Ricky, Sylvia S.

Kelompok 53 (2004). Kelompok ini bertugas untuk me-*review* bagian 3 versi 3.0 yang berupakan gabungan bab 3 dan bab 8 versi 2.0, yang dipecah ke beberapa bab baru. Bagian 3 ini berisi pokok bahasan Proses/Penjadwalan serta Konsep Perangkat Lunak Bebas. Anggota dari kelompok ini ialah: Endang Retno Nugroho, Indah Agustin, Annisa, Hanson, Jimmy, Ade A. Arifin, Shinta T Effendy, Fredy RTS, Respati, Hafidz Budi, Markus, Prayana Galih PP, Albert Kurniawan, Moch Ridwan J, Sukma Mahendra, Nasikhin, Sapii, Muhammad Rizalul Hak, Salman Azis Alsyafdi, Ade Melani, Amir Muhammad, Lusiana Darmawan, Anthony Steven, Anwar Chandra.

Kelompok 54 (2004). Kelompok ini bertugas untuk me-*review* bagian 4 versi 3.0 yang berupakan gabungan bab 4 dan bab 8 versi 2.0, yang dipecah ke beberapa bab baru. Bagian 4 ini berisi pokok bahasan Sinkronisasi dan Deadlock. Anggota dari kelompok ini ialah: I Christine Angelina, Farania Gama AR, Angga Bariesta H, M.Bayu TS, Muhammad Irfan, Nasrullah, Reza Lesmana, Suryamita H, Fitria Rahma Sari, Api Perdana, Maharmon Arnaldo, Sergio, Tedi Kurniadi, Ferry Sulistiyanto, Ibnu Mubarok, Muhammad Azani HS, Priadhana EK.

Kelompok 55 (2004). Kelompok ini bertugas untuk me-*review* bagian 5 versi 3.0 yang berupakan gabungan bab 5 dan bab 8 versi 2.0, yang dipecah ke beberapa bab baru. Bagian 5 ini berisi pokok bahasan Managemen Memori. Anggota dari kelompok ini ialah: Nilam Fitriah, Nurmaya, Nova Eka Diana, Okky HTF, Tirza Varananda, Yoanna W, Aria WN, Yudi Ariawan, Hendrik Gandawijaya, Johanes, Dania Tigarani S, Desiana NM, Annas Firdausi, Hario Adit W, Kartika Anindya P. Fajar Muharandy, Yudhi M Hamzah K, Binsar Tampahan HS, Risvan Ardiansyah, Budi Irawan, Deny Martan, Prastudy Mungkas F, Abdurrasyid Mujahid, Adri Octavianus, Rahmatri Mardiko.

Kelompok 56 (2004). Kelompok ini bertugas untuk me-*review* bagian 6 versi 3.0 yang berupakan gabungan bab 6 dan bab 8 versi 2.0, yang dipecah ke beberapa bab baru. Bagian 6 ini berisi pokok bahasan Sistem Berkas. Anggota dari kelompok ini ialah: Hipasdo Abrianto, Muhammad Fahrian, Dini Addiati, Titin Farida, Edwin Richardo, Yanuar Widjaja, Biduri Kumala, Deborah YN, Hidayat Febiansyah, M Nizar Kharis, Catur Adi N, M. Faizal Reza,

Kelompok 57 (2004). Kelompok ini bertugas untuk me-*review* bagian 7 versi 3.0 yang berupakan gabungan bab 7 dan bab 8 versi 2.0, yang dipecah ke beberapa bab baru. Bagian 7 ini berisi pokok bahasan M/K. Anggota dari kelompok ini ialah: Dominikus R, Randu Aditara, Dirgantoro Muhammad, Fuady Rosma Hidayat, M Mahdi, Septian Adiwibowo, Muhammad Hasrul M, Riyadi Akbar, A Taufiqurrakhman, Johanes Andria, Irfan Hilmy, Aziiz Surahman.

Kelompok 58 (2004). Kelompok ini bertugas untuk me-*review* yang sebelumnya menjadi bagian dari bab 8 versi 2.0, yang digabungkan ke bagian-bagian lain buku ini. Bagian ini berisi pokok bahasan GNU/Linux dan Perangkat Lunak Bebas. Anggota dari kelompok ini ialah: M Eka Suryana, Rachmad Laksana, Anjar Widianto, Annas, Arie Murdianto, Ranni K, Septina Dian L, Hera Irawati, Renza Azhary.

Kelompok 81 (2005). Kelompok ini bertugas untuk menulis Bab 27 (Masalah Dining Philosophers) serta Bab 7.6, 16.6, 20.2 versi 3.0. Kelompok ini hanya beranggotakan: Andreas Febrian dan Priadhana E. K.

Kelompok 82 (2005). Kelompok ini bertugas untuk menulis Bab 2 (Konsep Perangkat Lunak Bebas) serta Bab 3.5, 10.6, 16.10, 47.6 versi 3.0. Kelompok ini hanya beranggotakan: Agus Anang.

Kelompok 83 (2005). Kelompok ini bertugas untuk menulis Bab 50 (Sistem Terdistribusi) serta Bab 4.2, 14.5, 20.4 versi 3.0. Kelompok ini hanya beranggotakan: Salman Azis Alsyafdi dan Muhamad Rizalul Hak.

Kelompok 84 (2005). Kelompok ini bertugas untuk menulis Bab 49 (Sistem Waktu Nyata dan Multimedia) serta Bab 4.1, 12.3, 17.9, 45.10 versi 3.0. Kelompok ini hanya beranggotakan: Indah Wulansari, Sari W.S, dan Samiaji.

Kelompok 85 (2005). Kelompok ini bertugas untuk menulis Bab 25 (Masalah Bounded Buffer)

serta Bab 10.2, 16.7, 22.2, 47.5 versi 3.0. Kelompok ini hanya beranggotakan: Fahrurrozi Rahman dan Randy S.P.

Kelompok 86 (2005). Kelompok ini bertugas untuk menulis Bab 51 (Keamanan Sistem) serta Bab 10.3, 15.7, 21.11, 46.7 versi 3.0. Kelompok ini hanya beranggotakan: Pamela Indrajati dan Devi Triska Kustiana.

Kelompok 87 (2005). Kelompok ini bertugas untuk menulis Bab 52 (Perancangan dan Pemeliharaan) serta Bab 6.4, 16.8, 29.2 versi 3.0. Kelompok ini hanya beranggotakan: Sri Agustien M. dan Ahlijati N.

Kelompok 88 (2005). Kelompok ini bertugas untuk menulis Bab 26 (Masalah Readers/Writers) serta Bab 4.3, 12.4, 20.3 versi 3.0. Kelompok ini hanya beranggotakan: Muhammad Azani H.S. dan M. Faisal Reza.

Kelompok 89 (2005). Kelompok ini bertugas untuk menulis Bab 8 (Mesin Virtual Java) serta Bab 9.10, 16.9, 17.8, 44.11 versi 3.0. Kelompok ini hanya beranggotakan: Novrizki Primananda dan Zulkifli.

Kelompok 111 (2005). Sub-kelompok 111-10 bertugas menulis ulang Bab 10 (Konsep Proses) versi 4.0. Sub-kelompok ini beranggotakan: Richard Lokasamita, Rado Yanu, Phyllisia Angelia. Sub-kelompok 111-11 bertugas menulis ulang Bab 11 (Konsep Thread) versi 4.0. Sub-kelompok ini beranggotakan: Ario Santoso, Wahyu Mirza, Daniel Cahyadi. Sub-kelompok 111-12 bertugas menulis ulang Bab 12 (Thread Java) versi 4.0. Sub-kelompok ini beranggotakan: Moh. Ibrahim, Hafiz Arraja, Sutanto Sugii Joji. Sub-kelompok 111-13 bertugas menulis ulang Bab 13 (Konsep Penjadwalan) versi 4.0. Sub-kelompok ini beranggotakan: Kresna D.S., Rama Rizki, Wisnu LW.

Kelompok 112 (2005). Sub-kelompok 112-14 bertugas menulis ulang Bab 14 (Penjadwal CPU) versi 4.0. Sub-kelompok ini beranggotakan: Ananda Budi P, Maulana Iman T, Suharjono. Sub-kelompok 112-15 bertugas menulis ulang Bab 15 (Algoritma Penjadwalan I) versi 4.0. Sub-kelompok ini beranggotakan: Daniel Albert Ya, Desmond D. Putra, Rizky A. Sub-kelompok 112-16 bertugas menulis ulang Bab 16 (Algoritma Penjadwalan II) versi 4.0. Sub-kelompok ini beranggotakan: Anthony Steven, Eliza Margaretha, Fandi. Sub-kelompok 112-17 bertugas menulis ulang Bab 17 (Managemen Proses Linux) versi 4.0. Sub-kelompok ini beranggotakan: Abdul Arfan, Akhmad Syaikhul Hadi, Hadaiq Rolis S.

Kelompok 113 (2005). Sub-kelompok 113-18 bertugas menulis ulang Bab 18 (Konsep Interaksi) versi 4.0. Sub-kelompok ini beranggotakan: Adrianus W K, Aziz Yudi Prasetyo, Gregorio Cybill. Sub-kelompok 113-19 bertugas menulis ulang Bab 19 (Sinkronisasi) versi 4.0. Sub-kelompok ini beranggotakan: Candra Adhi, Triastuti C. Sub-kelompok 113-20 bertugas menulis ulang Bab 20 (Pemecahan Masalah Critical Section) versi 4.0. Sub-kelompok ini beranggotakan: Adolf Pandapotan, Ikhsan Putra Kurniawan, Muhammad Edwin Dwi P. Sub-kelompok 113-21 bertugas menulis ulang Bab 21 (Perangkat Sinkronisasi I) versi 4.0. Sub-kelompok ini beranggotakan: Dwi Putro HP, Jeremia Hutabarat, Rangga M Jati. Sub-kelompok 113-22 bertugas menulis ulang Bab 22 (Perangkat Sinkronisasi II) versi 4.0. Sub-kelompok ini beranggotakan: Femphy Pisceldo, Hendra Dwi Hadmanto, Zoni Yuki Haryanda.

Kelompok 114 (2005). Sub-kelompok 114-23 bertugas menulis ulang Bab 23 (Deadlock) versi 4.0. Sub-kelompok ini beranggotakan: Aurora Marsye, Mellawaty, Vidyanita Kumalasari. Sub-kelompok 114-24 bertugas menulis ulang Bab 24 (Diagram Graf) versi 4.0. Sub-kelompok ini beranggotakan: Arief Ristanto, Edwin Kurniawan. Sub-kelompok 114-25 bertugas menulis ulang Bab 25 (Bounded Buffer) versi 4.0. Sub-kelompok ini beranggotakan: Nurilla R I, Vidya Dwi A. Sub-kelompok 114-26 bertugas menulis ulang Bab 26 (Readers/Writers) versi 4.0. Sub-kelompok ini beranggotakan: Astria Kurniawan S, Franova Herdiyanto, Ilham Aji Pratomo. Sub-kelompok 114-27 bertugas menulis ulang Bab 27 (Sinkronisasi Dua Arah) versi 4.0. Sub-kelompok ini beranggotakan: Aprilia, Thoha, Amalia Zahra.

Kelompok 115 (2005). Sub-kelompok 115-28 bertugas menulis ulang Bab 28 (Managemen Memori) versi 4.0. Sub-kelompok ini beranggotakan: Agung Widiyarto, Fahrurrozi, Reynaldo Putra. Sub-kelompok 115-29 bertugas menulis ulang Bab 29 (Alokasi Memori) versi 4.0. Sub-kelompok ini beranggotakan: Rakhmat Adhi Pratama, Akhda Afif Rasyidi, Muhamad Ilyas. Sub-kelompok 115-30 bertugas menulis ulang Bab 30 (Pemberian Halaman) versi 4.0. Sub-kelompok ini beranggotakan: Ardi Darmawan, Iwan Prihartono, Michael B.M. Sub-kelompok 115-31 bertugas menulis ulang Bab 31 (Segmentasi) versi 4.0. Sub-kelompok ini beranggotakan: Andi Nur Mafsah

M, Danang Jaya.

Kelompok 116 (2005). Sub-kelompok 116-32 bertugas menulis ulang Bab 32 (Memori Virtual) versi 4.0. Sub-kelompok ini beranggotakan: Franky, Sadar B S, Yemima Aprilia Sub-kelompok 116-33 bertugas menulis ulang Bab 33 (Permintaan Halaman Pembuatan Proses) versi 4.0. Sub-kelompok ini beranggotakan: Arief Fatchul Huda, Cahyana. Sub-kelompok 116-34 bertugas menulis ulang Bab 34 (Algoritma Pergantian Halaman) versi 4.0. Sub-kelompok ini beranggotakan: Hera Irawati, Renza Azhary, Jaka Ramdani. Sub-kelompok 116-35 bertugas menulis ulang Bab 35 (Strategi Alokasi Frame) versi 4.0. Sub-kelompok ini beranggotakan: Arief Nurrachman, Riska Aprian. Sub-kelompok 116-36 bertugas menulis ulang Bab 36 (Memori Linux) versi 4.0. Sub-kelompok ini beranggotakan: Jani R.R. Siregar, Martin LT, Muhamad Mulki A.

Kelompok 117 (2005). Sub-kelompok 117-37 bertugas menulis ulang Bab 37 (Sistem Berkas) versi 4.0. Sub-kelompok ini beranggotakan: Alida W, Ratih Amalia. Sub-kelompok 117-38 bertugas menulis ulang Bab 38 (Struktur Direktori) versi 4.0. Sub-kelompok ini beranggotakan: Muhamad Rizalul Hak, Mega Puspita. Sub-kelompok 117-39 bertugas menulis ulang Bab 39 (Sistem Berkas Jaringan) versi 4.0. Sub-kelompok ini beranggotakan: Rahmad Mahendra, Rendra Rahmatullah, Rivki Hendriyan.

Kelompok 118 (2005). Sub-kelompok 118-40 bertugas menulis ulang Bab 40 (Implementasi Sistem Berkas) versi 4.0. Sub-kelompok ini beranggotakan: Gita Lystia, Rahmawati. Sub-kelompok 118-41 bertugas menulis ulang Bab 41 (*Filesystem Hierarchy Standard*) versi 4.0. Sub-kelompok ini beranggotakan: Susy Violina, M Rabindra S, Siti Fatihatul Aliyah. Sub-kelompok 118-42 bertugas menulis ulang Bab 42 (Konsep Alokasi Blok Sistem Berkas) versi 4.0. Sub-kelompok ini beranggotakan: Haris Sahlan.

Kelompok 119 (2005). Sub-kelompok 119-43 bertugas menulis ulang Bab 43 (Perangkat Keras Masukan/Keluaran) versi 4.0. Sub-kelompok ini beranggotakan: Intan Sari H H Z, Verra Mukty. Sub-kelompok 119-44 bertugas menulis ulang Bab 44 (Subsistem M/K Kernel) versi 4.0. Sub-kelompok ini beranggotakan: Randy S P, Tunggul Fardiaz. Sub-kelompok 119-45 bertugas menulis ulang Bab 45 (Managemen Disk I) versi 4.0. Sub-kelompok ini beranggotakan: Isnina Eva Hidayati, Sari Dwi Handiny, Rissa Dwi Oktavianty. Sub-kelompok 119-46 bertugas menulis ulang Bab 46 (Managemen Disk II) versi 4.0. Sub-kelompok ini beranggotakan: Ditya Nugraha, Dani Supriyadi, Wahyu Sulistio.

Kelompok 120 (2005). Sub-kelompok 120-47 bertugas menulis ulang Bab 47 (Perangkat Penyimpanan Tersier) versi 4.0. Sub-kelompok ini beranggotakan: Bahtiar, Suharto Anggono. Sub-kelompok 120-48 bertugas menulis ulang Bab 48 (Masukan/Keluaran Linux) versi 4.0. Sub-kelompok ini beranggotakan: M. Danang Pramudya.

Kelompok 150 (2006). Kelompok ini berdiskusi merampungkan versi 4.0. Kelompok ini beranggotakan: Haris Sahlan, Hera Irawati, M. Reza Benaji, Rimphy Darmanegara, V.A. Pragantha.

Daftar Isi

Kata Pengantar xxv
1. Revisi 4.0 (Agustus 2006) xxv
2. Kata Pengantar Revisi 1.0 xxvi
I. Konsep Dasar Perangkat Komputer 29
1. Hari Gini Belajar SO? 31
1.1. Pendahuluan 31
1.2. Mengapa Mempelajari Sistem Operasi? 31
1.3. Definisi Sementara 31
1.4. Sejarah Perkembangan 33
1.5. Bahan Pembahasan 36
1.6. Prasyarat 36
1.7. Sasaran Pembelajaran 36
1.8. Rangkuman 37
2. HaKI Perangkat Lunak 39
2.1. Pendahuluan 39
2.2. Hak Kekayaan Intelektual 39
2.3. HaKI Perangkat Lunak 41
2.4. Komersialisasi Perangkat Lunak 43
2.5. Ancaman dan Tantangan 44
2.6. Rangkuman 44
3. Perangkat Keras Komputer 47
3.1. Pendahuluan 47
3.2. Prosesor 48
3.3. Media Penyimpanan Utama 48
3.4. Penyimpanan Sekunder 49
3.5. Memori Tersier 50
3.6. Struktur Keluaran/Masukan (M/K) 50
3.7. Bus 51
3.8. Interupsi 51
3.9. *Local Area Network* 52
3.10. *Wide Area Network* 52
3.11. Rangkuman 52
4. Proteksi Perangkat Keras 55
4.1. Pendahuluan 55
4.2. Proteksi Fisik 55
4.3. Proteksi Media 55
4.4. Konsep Mode Operasi Ganda 55
4.5. Proteksi Masukan/Keluaran 56
4.6. Proteksi Memori 56
4.7. Proteksi CPU 57
4.8. Rangkuman 57
II. Konsep Dasar Sistem Operasi 59
5. Komponen Sistem Operasi 61
5.1. Pendahuluan 61
5.2. Manajemen Proses 61
5.3. Manajemen Memori Utama 61
5.4. Manajemen Sistem Berkas 61
5.5. Manajemen Sistem Masukan/Keluaran 62
5.6. Manajemen Penyimpanan Sekunder 62
5.7. Sistem Proteksi 62
5.8. Jaringan 62
5.9. *Command-Interpreter System* 62
5.10. Rangkuman 63
6. Sudut Pandang Alternatif 65
6.1. Pendahuluan 65
6.2. *System Program* 65
6.3. *System Calls* 66

6.4. *System Calls* Manajemen Proses 67
6.5. *System Calls* Manajemen Berkas 67
6.6. *System Calls* Manajemen Peranti 67
6.7. *System Calls* Informasi/Pemeliharaan 68
6.8. *System Calls* Komunikasi 68
6.9. Rangkuman 69
7. Struktur Sistem Operasi 71
7.1. Pendahuluan 71
7.2. Struktur Sederhana 71
7.3. Pendekatan Berlapis 71
7.4. Kernel-mikro 74
7.5. *Boot* 74
7.6. Kompilasi Kernel 75
7.7. Komputer Meja 76
7.8. Sistem Prosesor Jamak 76
7.9. Sistem Terdistribusi dan Terkluster 77
7.10. Sistem Waktu Nyata 79
7.11. Aspek Lain 79
7.12. Rangkuman 81
8. Mesin Virtual Java 83
8.1. Pendahuluan 83
8.2. Konsep Mesin Virtual 83
8.3. Konsep Bahasa Java 84
8.4. Mesin Virtual Java 87
8.5. Sistem Operasi Java 88
8.6. Rangkuman 90
9. Sistem GNU/Linux 93
9.1. Pendahuluan 93
9.2. Sistem dan Distribusi GNU/Linux 94
9.3. Lisensi Linux 95
9.4. Linux Saat Ini 95
9.5. Prinsip Rancangan Linux 96
9.6. Modul Kernel Linux 98
9.7. Rangkuman 99
III. Proses dan Penjadwalan 101
10. Konsep Proses 103
10.1. Pendahuluan 103
10.2. Pembentukan Proses 103
10.3. Terminasi Proses 104
10.4. Status Proses 104
10.5. *Process Control Block* 105
10.6. Hirarki Proses 106
10.7. Rangkuman 107
11. Konsep *Thread* 109
11.1. Pendahuluan 109
11.2. Keuntungan *Thread* 110
11.3. *Thread* Pengguna dan Kernel 110
11.4. Model *Multithreading* 110
11.5. *Fork* dan *Exec System Call* 112
11.6. *Cancellation* 112
11.7. Penanganan Sinyal 113
11.8. *Thread Pools* 114
11.9. *Thread Specific Data* 114
11.10. *Pthreads* 114
11.11. Rangkuman 114
12. *Thread* Java 117
12.1. Pendahuluan 117
12.2. Pembuatan *Thread* 117
12.3. Status *Thread* 119
12.4. Penggabungan *Thread* 120
12.5. Terminasi *Thread* 121
12.6. JVM dan *Host Operating System* 121
12.7. Solusi *Multi-Threading* 121

12.8. Rangkuman 122
13. Konsep Penjadwalan 123
13.1. Pendahuluan 123
13.2. Penjadwalan Antrian 123
13.3. *Penjadwal* 125
13.4. *Context Switch* 126
13.5. Rangkuman 127
14. Penjadwal CPU 129
14.1. Pendahuluan 129
14.2. Siklus *Burst* CPU-M/K 129
14.3. *Dispatcher* 130
14.4. Penjadwalan CPU 130
14.5. Kriteria Penjadwalan 132
14.6. Rangkuman 133
15. Algoritma Penjadwalan I 135
15.1. Pendahuluan 135
15.2. FCFS: *First-Come, First-Served* 135
15.3. SJF: *Shortest-Job First* 135
15.4. Prioritas 136
15.5. *Round-Robin* 137
15.6. *Multilevel Queue* 137
15.7. *Multilevel Feedback Queue* 138
15.8. Rangkuman 139
16. Algoritma Penjadwalan II 141
16.1. Pendahuluan 141
16.2. Prosesor Jamak 141
16.3. Sistem Waktu Nyata 141
16.4. Sistem *Hard Real-Time* 142
16.5. Sistem *Soft Real-Time* 142
16.6. Penjadwalan Thread 144
16.7. Penjadwalan Java 145
16.8. Kinerja 148
16.9. Rangkuman 150
17. Manajemen Proses Linux 151
17.1. Pendahuluan 151
17.2. Deskriptor Proses 151
17.3. Proses dan *Thread* 154
17.4. Penjadwalan 158
17.5. *Symmetric Multiprocessing* 159
17.6. Rangkuman 159
IV. Proses dan Sinkronisasi 161
18. Konsep Interaksi 163
18.1. Pendahuluan 163
18.2. Proses yang Kooperatif 163
18.3. Hubungan Antara Proses 163
18.4. Komunikasi Proses Dalam Sistem 165
18.5. Komunikasi Langsung 165
18.6. Komunikasi Tidak Langsung 166
18.7. Sinkronisasi 167
18.8. Buffering 167
18.9. *Mailbox* 167
18.10. *Socket Client*/*Server System* 168
18.11. *Server* dan *Thread* 168
18.12. Rangkuman 171
19. Sinkronisasi 173
19.1. Pendahuluan 173
19.2. *Race Condition* 173
19.3. Problem *Critical Section* 174
19.4. Persyaratan 175
19.5. Rangkuman 175
20. Masalah *Critical Section* 177
20.1. Pendahuluan 177
20.2. Algoritma I 177

20.3. Algoritma 2 178
20.4. Algoritma 3 179
20.5. Algoritma Tukang Roti 180
20.6. Rangkuman 181
21. Perangkat Sinkronisasi I 183
21.1. Pendahuluan 183
21.2. Instruksi Atomik 184
21.3. Semafor 186
21.4. *Wait* dan *Signal* 187
21.5. Jenis Semafor 189
21.6. *Critical Section* Dan Semafor 189
21.7. Solusi Masalah Sinkronisasi Antar Proses Dengan Semafor 190
21.8. Modifikasi *Binary Semaphore* 190
21.9. Pemrograman *Windows* 191
21.10. Rangkuman 191
22. Perangkat Sinkronisasi II 193
22.1. Pendahuluan 193
22.2. Transaksi Atomik 193
22.3. *Critical Region* 194
22.4. Monitor 194
22.5. Pemrograman Java™ 195
22.6. Masalah Umum Sinkronisasi 195
22.7. Sinkronisasi Kernel Linux 195
22.8. Rangkuman 197
23. *Deadlock* 199
23.1. Pendahuluan 199
23.2. Daur Ulang Sumber Daya 200
23.3. Sumber Daya Sekali Pakai 201
23.4. Kondisi untuk Terjadinya *Deadlock* 202
23.5. Mengabaikan Masalah *Deadlock* 202
23.6. Mendeteksi dan Memperbaiki 202
23.7. Menghindari *Deadlock* 203
23.8. Pencegahan *Deadlock* 205
23.9. Rangkuman 206
24. Diagram Graf 207
24.1. Pendahuluan 207
24.2. Komponen Graf Alokasi Sumber Daya 207
24.3. Pendeteksian *Deadlock* 208
24.4. Pencegahan *Deadlock* 210
24.5. Pendeteksian dengan Graf Tunggu 211
24.6. Rangkuman 212
25. *Bounded Buffer* 213
25.1. Pendahuluan 213
25.2. Penggunaan Semafor 216
25.3. Penjelasan Program 216
25.4. Rangkuman 219
26. *Readers/Writers* 221
26.1. Pendahuluan 221
26.2. Program Java 221
26.3. Penjelasan Program 224
26.4. Rangkuman 225
27. Sinkronisasi Dua Arah 227
27.1. Pendahuluan 227
27.2. Program Java 227
27.3. Alur Program 229
27.4. Rangkuman 230
V. Memori 231
28. Manajemen Memori 233
28.1. Pendahuluan 233
28.2. Address Binding 233
28.3. Ruang Alamat Logika dan Fisik 233
28.4. Pemanggilan Dinamis 234
28.5. Link Dinamis dan Pustaka Bersama 234

28.6. *Overlays* 235
28.7. Rangkuman 236
29. Alokasi Memori 239
29.1. Pendahuluan 239
29.2. Proteksi Memori 240
29.3. Alokasi Memori Berkesinambungan 241
29.4. Fragmentasi 243
29.5. Rangkuman 244
30. Pemberian Halaman 247
30.1. Pendahuluan 247
30.2. Metoda Dasar 247
30.3. Dukungan Perangkat Keras 248
30.4. Proteksi Memori 248
30.5. Untung/Rugi Pemberian Halaman 249
30.6. Tabel Halaman 249
30.7. Pemberian Halaman Secara Bertingkat 250
30.8. Tabel Halaman secara *Hashed* 251
30.9. Tabel Halaman secara *Inverted* 252
30.10. Berbagi Halaman (*Share*) 253
30.11. Rangkuman 254
31. Segmentasi 257
31.1. Pendahuluan 257
31.2. Segmentasi Perangkat Keras 257
31.3. Keuntungan Segmentasi 259
31.4. Penggunaan Segmentasi Pentium 260
31.5. Segmentasi Linux 261
31.6. Rangkuman 261
32. Memori Virtual 263
32.1. Pendahuluan 263
32.2. *Demand Paging* 264
32.3. Skema Bit Valid – Tidak Valid 265
32.4. Penanganan Kesalahan Halaman 265
32.5. Kelebihan/Kekurangan Demand Paging 266
32.6. Kinerja *Demand Paging* 267
32.7. Persyaratan Perangkat Keras 268
32.8. Rangkuman 269
33. Permintaan Halaman Proses 271
33.1. Pendahuluan 271
33.2. *Copy-On-Write* 271
33.3. *Memory-Mapped Files* 272
33.4. Rangkuman 274
34. Algoritma Ganti Halaman 277
34.1. Pendahuluan 277
34.2. Algoritma *First In First Out* 278
34.3. Algoritma Optimal 279
34.4. Algoritma *Least Recently Used* (LRU) 279
34.5. Algoritma Perkiraan LRU 280
34.6. Algoritma *Counting* 282
34.7. Algoritma NRU (Not Recently Used) 282
34.8. Algoritma *Page Buffering* 282
34.9. Rangkuman 282
35. Strategi Alokasi Frame 285
35.1. Pendahuluan 285
35.2. *Thrashing* 287
35.3. Membatasi Efek *Thrashing* 288
35.4. *Prepaging* 290
35.5. Ukuran Halaman 290
35.6. Jangkauan *TLB* 291
35.7. Tabel Halaman yang Dibalik 291
35.8. Struktur Program 292
35.9. *Interlock* M/K 292
35.10. Pemrosesan Waktu Nyata 292
35.11. Keluarga Windows NT 292

35.12. Solaris 2 293
35.13. Rangkuman 293
36. Memori Linux 295
36.1. Pendahuluan 295
36.2. Manajemen Memori Fisik 295
36.3. Memori Virtual 297
36.4. *Demand Paging* 299
36.5. *Swaping* 299
36.6. Pengaksesan Memori Virtual Bersama 301
36.7. Efisiensi 301
36.8. Load dan Eksekusi Program 301
36.9. Rangkuman 302
VI. Penyimpanan Sekunder 305
37. Sistem Berkas 307
37.1. Pendahuluan 307
37.2. Konsep Berkas 307
37.3. Atribut berkas 307
37.4. Jenis Berkas 307
37.5. Operasi Berkas 308
37.6. Struktur Berkas 309
37.7. Metode Akses 309
37.8. Rangkuman 310
38. Struktur Direktori 311
38.1. Pendahuluan 311
38.2. Atribut Direktori 311
38.3. Operasi Direktori 312
38.4. Direktori Satu Tingkat 312
38.5. Direktori Dua Tingkat 313
38.6. Direktori dengan Struktur Pohon 313
38.7. Direktori dengan Struktur Graf Asiklik 314
38.8. Direktori dengan Struktur Graf Umum 314
38.9. *Mounting* 316
38.10. Rangkuman 318
39. Aspek Jaringan dan Keamanan 321
39.1. Pendahuluan 321
39.2. *Remote File System* 322
39.3. Model *Client-Server* 322
39.4. Consistency Semantics 323
39.5. Proteksi 323
39.6. Tipe Akses 323
39.7. Kontrol Akses 324
39.8. *File Permission* dalam UNIX 325
39.9. Pendekatan Pengamanan Lainnya 326
39.10. Rangkuman 326
40. Implementasi Sistem Berkas 329
40.1. Pendahuluan 329
40.2. Implementasi Sistem Berkas 331
40.3. Partisi dan *Mounting* 332
40.4. Sistem Berkas Virtual 333
40.5. Implementasi Direktori 334
40.6. Algoritma *Linear List* 334
40.7. Algoritma *Hash Table* 334
40.8. Direktori pada CP/M 335
40.9. Direktori pada MS-DOS 335
40.10. Direktori pada Unix 336
40.11. Rangkuman 337
41. FHS 339
41.1. Pendahuluan 339
41.2. Sistem Berkas 339
41.3. Sistem Berkas *Root* 340
41.4. Hirarki "/usr" 342
41.5. Hirarki "/var" 344
41.6. Tambahan untuk Linux 347

41.7. Rangkuman 347
42. Alokasi Blok Sistem Berkas 349
42.1. Pendahuluan 349
42.2. Metode Alokasi 349
42.3. Manajemen Ruang Kosong 358
42.4. Pengimplementasian Direktori 360
42.5. Efisiensi dan Kinerja 360
42.6. *Recovery* 362
42.7. *Log-Structured File System* 364
42.8. Sistem Berkas Linux Virtual 364
42.9. Operasi-operasi Dalam Inode 364
42.10. Sistem Berkas Linux 365
42.11. Pembagian Sistem Berkas Ortogonal 367
42.12. Rangkuman 368
VII. Masukan/Keluaran (M/K) 371
43. Perangkat Keras M/K 373
43.1. Pendahuluan 373
43.2. Komponen M/K 374
43.3. Penanganan M/K 374
43.4. Polling 375
43.5. Interupsi 376
43.6. *Direct Memory Access* (DMA) 378
43.7. Rangkuman 380
44. Subsistem M/K Kernel 383
44.1. Pendahuluan 383
44.2. Aplikasi Antarmuka M/K 383
44.3. Penjadwalan M/K 385
44.4. *Buffering* 386
44.5. *Caching* 386
44.6. *Spooling* dan Reservasi Perangkat 387
44.7. Penanganan Kesalahan (*Error Handling*) 388
44.8. Struktur Data Kernel 388
44.9. Penanganan Permintaan M/K 389
44.10. Arus M/K 390
44.11. Kinerja M/K 391
44.12. Rangkuman 393
45. Manajemen Disk I 395
45.1. Pendahuluan 395
45.2. Penjadwalan *Disk* 395
45.3. Penjadwalan FCFS 396
45.4. Penjadwalan SSTF 397
45.5. Penjadwalan SCAN 398
45.6. Penjadwalan C-SCAN 399
45.7. Penjadwalan LOOK 401
45.8. Penjadwalan C-LOOK 401
45.9. Pemilihan Algoritma Penjadwalan *Disk* 403
45.10. Rangkuman 403
46. Manajemen Disk II 405
46.1. Pendahuluan 405
46.2. Komponen Disk 405
46.3. Manajemen Ruang *Swap* 407
46.4. Struktur RAID 408
46.5. *Host-Attached Storage* 411
46.6. NAS: *Network-Attached Storage* 412
46.7. SAN: *Storage-Area Network* 413
46.8. Implementasi Penyimpanan Stabil 414
46.9. Rangkuman 415
47. Penyimpanan Tersier 417
47.1. Pendahuluan 417
47.2. Jenis Struktur Penyimpanan Tersier 417
47.3. *Future Technology* 422
47.4. Aplikasi Antarmuka 422
47.5. Masalah Kinerja 424

47.6. Rangkuman 425
48. Keluaran/Masukan Linux 427
48.1. Pendahuluan 427
48.2. *Device Karakter* 427
48.3. *Device* Blok 428
48.4. *Device* Jaringan 429
48.5. Rangkuman 431
VIII. Topik Lanjutan 433
49. Waktu Nyata dan Multimedia 435
49.1. Pendahuluan 435
49.2. Kernel Waktu Nyata 435
49.3. Penjadwalan Berdasarkan Prioritas 435
49.4. Kernel Preemptif 436
49.5. Mengurangi Latency 436
49.6. Penjadual Proses 437
49.7. Penjadual Disk 439
49.8. Manajemen Berkas 440
49.9. Manajemen Jaringan 441
49.10. Unicasting dan Multicasting 441
49.11. Real-Time Streaming Protocol 441
49.12. Kompresi 442
49.13. Rangkuman 443
50. Sistem Terdistribusi 445
50.1. Pendahuluan 445
50.2. Topologi Jaringan 446
50.3. Isu Lainnya 446
50.4. Rangkuman 446
51. Keamanan Sistem 449
51.1. Pendahuluan 449
51.2. Manusia dan Etika 449
51.3. Kebijaksanaan Pengamanan 450
51.4. Keamanan Fisik 450
51.5. Keamanan Perangkat Lunak 450
51.6. Keamanan Jaringan 450
51.7. Kriptografi 451
51.8. Operasional 451
51.9. BCP/DRP 452
51.10. Proses Audit 452
51.11. Rangkuman 453
52. Perancangan dan Pemeliharaan 455
52.1. Pendahuluan 455
52.2. Perancangan Antarmuka 456
52.3. Implementasi 456
52.4. Kinerja 457
52.5. Pemeliharaan Sistem 457
52.6. *Tuning* 458
52.7. Trend 458
52.8. Rangkuman 458
Daftar Rujukan Utama 461
A. *GNU Free Documentation License* 467
A.1. PREAMBLE 467
A.2. APPLICABILITY AND DEFINITIONS 467
A.3. VERBATIM COPYING 468
A.4. COPYING IN QUANTITY 468
A.5. MODIFICATIONS 469
A.6. COMBINING DOCUMENTS 470
A.7. COLLECTIONS OF DOCUMENTS 470
A.8. AGGREGATION WITH INDEPENDENT WORKS 471
A.9. TRANSLATION 471
A.10. TERMINATION 471
A.11. FUTURE REVISIONS OF THIS LICENSE 471
A.12. ADDENDUM: How to use this License for your documents 472
B. Kumpulan Soal Ujian 473

B.1. Pasangan Konsep (2003-2005) 473
B.2. GNU/Linux (2003) 474
B.3. Perangkat Lunak Bebas (2005) 474
B.4. Konsep Sistem Operasi (2005) 475
B.5. Kernel Linux 2.6.X (=KL26) (2004) 475
B.6. Rancangan Sistem (2002) 475
B.7. Tabel Proses I (2003) 476
B.8. Tabel Proses II (2003) 477
B.9. Tabel Proses III (2004) 477
B.10. Status Proses I (2003) 478
B.11. Status Proses II (2005) 478
B.12. *Fork* Proses I (2005) 480
B.13. *Fork* Proses II (2005) 481
B.14. Penjadualan Proses I (2001) 481
B.15. Penjadualan Proses II (2002) 482
B.16. Penjadualan Proses III (2004) 482
B.17. *Deadlock* I (2003) 482
B.18. *Deadlock* II (2003) 484
B.19. *Deadlock* III (2005) 484
B.20. Problem *Reader/Writer* I (2001) 485
B.21. Problem *Reader/Writer* II (2002) 487
B.22. Problem *Reader/Writer* III (2004) 488
B.23. *Bounded Buffer* (2003) 490
B.24. Sinkronisasi I (2005) 493
B.25. Sinkronisasi II(2005) 494
B.26. IPC (2003) 497
B.27. Status Memori I (2004) 499
B.28. Status Memori II (2005) 500
B.29. Managemen Memori dan Utilisasi CPU (2004) 501
B.30. Memori I (2002) 501
B.31. Memori II (2002) 501
B.32. Memori III (2003) 502
B.33. Memori Virtual (2005) 502
B.34. *Multilevel Paging Memory* I (2003) 503
B.35. *Multilevel Paging Memory* II (2004) 504
B.36. *Multilevel Paging Memory* III (2005) 505
B.37. FHS *(File Hierarchy Standards)* (2002) 505
B.38. Sistem Berkas I (2002) 506
B.39. Sistem Berkas II (2003) 506
B.40. Sistem Berkas III (2004) 506
B.41. Sistem Berkas IV (2005) 507
B.42. Sistem Berkas "*ReiserFS*" (2003) 507
B.43. Sistem Berkas "*NTFS*" (2005) 507
B.44. RAID (*Redudant Array of I* Disks*) (2004) 507
B.45. *Mass Storage System* I (2002) 508
B.46. *Mass Storage System* II (2003) 508
B.47. *Mass Storage System* III (2003) 508
B.48. *I/O Interface* (2003) 509
B.49. *I/O* dan *USB* (2004) 509
B.50. Struktur Keluaran/Masukan Kernel (*I/O*) (2004) 510
B.51. Masukan/Keluaran (2005) 510
B.52. *CDROM* (2005) 510
B.53. *HardDisk* I (2001) 510
B.54. *HardDisk* II (2003) 511
B.55. *HardDisk* III (2004) 511
B.56. *HardDisk* IV (2005) 512
B.57. Waktu Nyata/Multimedia (2005) 513
B.58. Tugas Kelompok/Buku Sistem Operasi (2004) 513
C. *UPDATE/WishList* 515
C.1. *WishList* 515
Indeks 517

Daftar Gambar

1.1. Abstraksi Komponen Sistem Komputer 32
1.2. Arsitektur Komputer von-Neumann 34
1.3. Bagan Sebuah Komputer Personal 34
1.4. Bagan Memori Untuk Sistem *Monitor Batch* Sederhana 35
1.5. Bagan Memori untuk Model *Multiprogram System* 35
3.1. Arsitektur Umum Komputer 47
3.2. Arsitektur PC Modern 48
3.3. Penyimpanan Hirarkis 48
3.4. Struktur *Harddisk* 49
3.5. Struktur *Optical Drive* 50
3.6. Struktur M/K 50
3.7. *Local Area Network* 52
3.8. *Wide Area Network* 52
4.1. *Dual Mode Operation* 56
4.2. Proteksi M/K 56
4.3. *Memory Protection* 56
6.1. Memberikan parameter melalui tabel 66
6.2. Eksekusi MS-DOS 67
6.3. Multi program pada Unix 68
6.4. Mekanisme komunikasi 69
7.1. Lapisan pada Sistem Operasi 72
7.2. Tabel Level pada Sistem Operasi 72
7.3. Lapisan Sistem Operasi secara umum 74
7.4. Sistem Terdistribusi I 77
7.5. Sistem Terdistribusi II 78
8.1. Struktur Mesin Virtual 83
8.2. JVM: Java Virtual Machine 87
8.3. Struktur sistem operasi JavaOS 89
8.4. PL3 89
9.1. Logo Linux. 96
10.1. Status Utama Proses 105
10.2. *Process Control Block* 105
10.3. Status Proses 106
11.1. *Thread* 109
11.2. *Many-To-One* 111
11.3. *One-To-One* 111
11.4. *Many-To-Many* 112
12.1. Bagan Thread 119
13.1. *Device Queue* 123
13.2. Diagram Antrian 124
13.3. *Medium-term Scheduler* 125
13.4. *Context Switch* 126
14.1. Siklus *Burst* 129
14.2. Diagram *Burst* 130
14.3. *Burst* 130
14.4. *Preemptive vs Non Preemptive* 131
14.5. *Dispatch Latency* 132
15.1. FCFS 135
15.2. SJF *Preemptive* 136
15.3. SJF Non *Preemptive* 136
15.4. Round Robin 137
15.5. Multilevel Queue 138
15.6. Multilevel Feedback Queue 139
16.1. Grafik *Hard Real-Time* 143
16.2. Grafik *Soft Real-Time* 143
18.1. Client Server 165
20.1. Algoritma I 178
20.2. Algoritma II 178

20.3. Algoritma III 179
23.1. Contoh Deadlock di Jembatan 199
23.2. Contoh Deadlock di Persimpangan Jalan 199
23.3. Tabel 1 204
23.4. Tabel 2 205
24.1. Proses Pi meminta sumber daya Rj 207
24.2. Sumber daya Rj yang mengalokasikan salah satu 208
24.3. Graf Alokasi Sumber Daya 208
24.4. Graf dengan *deadlock* 209
24.5. Tanpa *deadlock* 209
24.6. Graf alokasi sumber daya dalam status aman 210
24.7. Graf alokasi sumber daya dalam status tidak aman 211
24.8. Graf alokasi sumber daya 211
24.9. Graf tunggu 212
28.1. MMU: *Memory Management Unit* 234
28.2. *Two-Pass Assembler* 235
29.1. Proteksi Memori 241
29.2. Proses Dalam Memori 242
29.3. Permasalahan alokasi penyimpanan dinamis 243
30.1. Penerjemahan Halaman 247
30.2. Skema Tabel Halaman Dua tingkat 250
30.3. Tabel Halaman secara Bertingkat 250
30.4. *Hierarchical Paging* 251
30.5. Hashed Page Tabels 252
30.6. Tabel Halaman secara *Inverted* 252
30.7. *Inverted Page Tables* 253
30.8. Berbagi Halaman 253
30.9. *Share Page* 254
31.1. Alamat Lojik 257
31.2. Arsitektur Segmentasi 258
31.3. Segmentasi 258
31.4. Segmentasi dengan Pemberian Halaman 259
31.5. Segmentasi dengan Pemberian Halaman (INTEL 30386) 260
31.6. Selektor 260
31.7. Segmentasi-Intel-Pentium 261
32.1. Memori Virtual 263
32.2. Tabel halaman untuk skema bit valid-tidak valid 265
32.3. Gambaran pada saat penanganan kesalahan halaman 266
33.1. Sebelum 271
33.2. Sesudah 272
33.3. MMF 273
33.4. MMF pada Win32 API 274
34.1. Ilustrasi Kondisi yang Memerlukan Pemindahan Halaman 277
34.2. Pemindahan Halaman 278
34.3. Contoh Penerapan Algoritma FIFO 279
34.4. Contoh Algoritma Optimal 279
34.5. Algoritma LRU 280
35.1. Derajat dari *Multiprogramming* 287
35.2. Frekuensi-kesalahan-halaman 289
35.3. Solar Halaman Scanner 293
36.1. Contoh pembagian zona memori pada arsitektur Intel x86. 295
36.2. Contoh skema alokasi memori dengan algoritma buddy 296
36.3. Contoh skema alokasi slab di GNU/Linux 297
36.4. Pemetaan Memori Virtual ke Alamat Fisik. 298
36.5. Skema Algoritma Clock 300
36.6. Clock 300
36.7. ELF 302
38.1. File Organization 311
38.2. Single Level Directory 312
38.3. Two Level Directory 313
38.4. Directori Struktur Pohon 314
38.5. Acyclic-Structured Directory 314
38.6. Graf Umum 315

38.7. Path 316
38.8. Mounting 317
39.1. Client Server 322
40.1. Organisasi Disk 329
40.2. File Control Block 330
40.3. *Layered File System* 330
40.4. *Schematic View of Virtual File System* 333
40.5. Direktori CPM 335
40.6. Direktori MS-DOS 336
40.7. Direktori Unix 336
42.1. Alokasi Berkesinambungan 349
42.2. Kondisi Disk 350
42.3. Alokasi dengan penghubung blok 351
42.4. Alokasi Selain Berkesinambungan 352
42.5. Ruang Blok Dialokasikan Penghubung 353
42.6. Cluster 353
42.7. Penggunaan FAT 354
42.8. Pengalokasikan dengan Indeks 355
42.9. Skema Terhubung 355
42.10. Pengindeksan bertingkat 356
42.11. Skema Gabungan 357
42.12. Ruang kosong *linked list* 359
42.13. Menggunakan *unified buffer cache* 361
42.14. Tanpa *unified buffer cache* 362
42.15. Macam-macam lokasi *disk-caching* 363
42.16. Struktur Sistem Berkas EXT2. 365
42.17. Inode Sistem Berkas EXT2. 366
43.1. Abstraksi 373
43.2. Model Bus Tunggal 374
43.3. Proses *Polling* 375
43.4. DMA 379
44.1. Struktur Kernel 383
44.2. Spooling 387
44.3. Struktur Stream 390
44.4. Antar Komputer 391
45.1. FCFS 396
45.2. SSTF 397
45.3. SCAN 399
45.4. CSCAN 400
45.5. LOOK 401
45.6. CLOOK 402
46.1. Format Sektor 405
46.2. Manajemen Ruang Swap: Pemetaan Swap Segmen Teks 4.3 BSD 407
46.3. Manajemen Ruang Swap: Pemetaan Swap Segmen Data 4.3 BSD 408
46.4. RAID 0 + 1 dan 1 + 0 410
46.5. RAID level 0, 1, 2, 3 411
46.6. RAID level 4, 5, 6 411
46.7. NAS 412
46.8. SAN 413
47.1. Komponen internal dasar floppy disk 3.5 inch 417
47.2. Magneto-Optical Disc 418
47.3. DVD-RW disc pada sebuah gelendong 419
47.4. CD-R 419
47.5. DVD-R 420
47.6. CDROM Drive 420
47.7. DDS Tape Drives 421
47.8. USB Drive 421
48.1. CharDev. 427
48.2. Buffer 429

Daftar Tabel

1.1. Perbandingan Sistem Dahulu dan Sekarang 31
17.1. Tabel Flag dan Fungsinya 155
41.1. Direktori/link yang wajib dalam "/." 340
41.2. Direktori/link yang optional dalam "/." 341
41.3. Direktori/link yang dibutuhkan dalam "/usr". 342
41.4. Direktori/link yang merupakan pilihan dalam "/usr". 343
41.5. Direktori/link yang dibutuhkan dalam "/var" 344
41.6. Direktori/link yang dibutuhkan di dalam "/var" 345

Daftar Contoh

8.1. Contoh penggunaan class objek dalam Java 85
8.2. Contoh penggunaan Java API 86
12.1. Thread 118
12.2. *Thread* Sederhana 120
16.1. DialogThread 146
16.2. DialogThread 147
17.1. Isi Deskriptor Proses 151
17.2. Antrian Tunggu 153
17.3. Thread Kernel 155
17.4. Contoh pengambilan nilai PID proses 156
17.5. Fork Tanpa Menggunakan Wait 156
17.6. Penggunaan Fork Dengan Wait 157
17.7. TOP 157
17.8. Tread vs. Fork 158
18.1. *Bounded Buffer* 164
18.2. *Mailbox* 167
18.3. *WebServer* 169
18.4. *Client* 170
19.1. Produser/Konsumer 173
19.2. *Counter* (1) 174
19.3. *Counter* (2) 174
19.4. *Critical Section (1)* 175
20.1. *Algoritma Tukang Roti* 180
21.1. *Critical Section* 183
21.2. *testANDset* 184
21.3. Asembler 185
21.4. Asembler 185
21.5. Asembler 186
21.6. Asembler 186
21.7. Asembler 187
21.8. *waitSpinLock* 188
21.9. *signalSpinLock* 188
22.1. Sintaks Monitor 194
23.1. Lalulintas 200
23.2. P-Q 201
23.3. Deadlock 201
25.1. Class Produsen Konsumen 213
25.2. Keluaran Program 216
25.3. Class Semaphore 217
25.4. Semaphore P 217
25.5. Semaphore V 217
25.6. Enter 218
25.7. Remove 218
26.1. Class ReaderWriterServer 221
26.2. Keluaran 224
27.1. Class SuperProses 227
27.2. Keluaran Program 229

Kata Pengantar

1. Revisi 4.0 (Agustus 2006)

"*I said you robbed me before, so I'm robbing you back!* "
—Paul McCartney: Deliver Your Children (1978)

Penyusunan buku text "bebas" ini bertujuan untuk mengurangi – mudah-mudahan nantinya menghapus – budaya penyalinan (*fotocopy*) bahan pengajaran secara tidak sah. Ironisnya, walaupun sudah memasuki revisi 4.0, aroma Silberschatz dan kawan-kawan masih terasa "sengit" pada karya MDGR ini. Mudah-mudahan, aroma tersebut kian berkurang serta akhirnya hilang dalam revisi-revisi berikutnya.

Buku ini berisi hal-hal dasar yang berhubungan dengan Sistem Operasi yang perlu diketahui oleh para mahasiswa jurusan Ilmu Komputer, baik tingkatan S_1 mau pun tingkatan S_2. Semenjak 2004, buku ini dipergunakan sebagai rujukan utama Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia (http://rms46.vlsm.org/2/133.html – http://www.cs.ui.ac.id/). Agar lebih memahami kandungan buku ini, diharapkan bahwa para pembaca memiliki pengetahuan dasar beberapa komponen dasar Ilmu Komputer seperti sistem dijital, organisasi komputer, struktur data, serta menguasai bahasa Java.

Buku ini masih jauh dari sempurna, sehingga masih diperbaiki secara berkesinambungan. Diskusi yang terkait dengan bidang Sistem Operasi secara umum, maupun yang khusus seputar buku ini, diselenggarakan melalui milis Sistem Operasi. Kritik/tanggapan/usulan juga dapat disampaikan ke `<vlsm.org <at> gmail.com>`. Versi digital terakhir dari buku ini dapat diambil dari http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/BUKU/.

Pokok Bahasan

Tulisan ini terbagi menjadi delapan pokok bahasan. Setiap pokok bahasan akan terdiri dari beberapa sub-pokok bahasan dalam bentuk bab. Setiap bab tersebut dirancang untuk **mengisi satu jam kuliah (tatap muka)**. Jadi ke-52 bab buku ini, cocok untuk sebuah mata ajar dengan bobot empat Satuan Kredit Semester (4 SKS).

Bagian I, "Konsep Dasar Perangkat Komputer". Pembahasan buku ini diawali dengan pengenalan konsep dasar sistem komputer Bagian ini, akan menguraikan secara umum komponen-komponen komputer seperti Sistem Operasi, perangkat keras, proteksi, keamanan, jaringan, hak kekayaan intelektual, lisensi, konsep perangkat lunak bebas (PLB), beserta tantangannya. Aspek-aspek tersebut diperlukan untuk memahami konsep-konsep Sistem Operasi yang akan dijabarkan dalam buku ini. Tentunya tidak dapat diharapkan pembahasan yang dalam. Rincian lanjut, sebaiknya dilihat pada rujukan yang berhubungan dengan "Pengantar Organisasi Komputer", "Pengantar Struktur Data", serta "Pengantar Jaringan Komputer". Bagian ini terdiri dari 4 bab.

Bagian II, "Konsep Dasar Sistem Operasi". Para pembaca sepertinya pernah mendengar istilah "Sistem Operasi". Mungkin pula pernah berhubungan secara langsung atau pun tidak langsung dengan istilah tersebut. Namun, belum tentu dapat menjabarkan perihal apa yang sebetulnya dimaksud dengan kata "Sistem Operasi". Bagian ini merupakan penjabaran secara garis besar dari sebuah Sistem Operasi secara umum, serta GNU/Linux secara khusus. Ini akan membantu pemahaman konsep secara terpadu, tanpa harus terganggu dengan rinciannya. Bagian III - VII akan menjabarkan secara lebih rinci, konsep-konsep dalam bab ini. Pastikan bahwa bagian I dan II buku ini dipahami sepenuhnya. Menguasai kedua bagian tersebut, akan mempermudah pemahaman sisa dari buku ini. Bagian ini terdiri dari 5 bab.

Bagian III, "Proses dan Penjadwalan". Proses merupakan inti pembahasan dari sebuah Sistem Operasi. Bagian ini akan memperkenalkan konsep proses serta penjadwalannya. Dalam delapan sub-pokok bahasan (bab), akan dibahas definisi, konsep *thread*, implementasi *thread* dalam Java, konsep dan algoritma penjadwalan. Bagian ini akan ditutup dengan memperkenalkan implementasi proses Linux. Bagian ini terdiri dari 8 bab.

Bagian IV, "Proses dan Sinkronisasi". Pada bagian ini, akan diperkenalkan bagaimana proses saling berinteraksi. Dalam tujuh sub-pokok bahasan (bab), akan dibahas konsep-konsep seperti interaksi proses, sinkronisasi, *race condition*, *critical section*, serta *deadlock*. Untuk mempermudah pemahaman konsep-konsep abstrak ini, tiga bab penutup akan merupakan ilustrasi aspek sinkronisasi dalam bahasa Java. Membahas keempat bagian tersebut di atas memerlukan waktu sekitar tujuh minggu semester. Dengan demikian, merupakan saat yang tepat untuk melakukan ujian tengah semester. Bagian ini terdiri dari 10 bab.

Bagian V, "Memori". Pengelolaan memori merupakan komponen penting lainnya dari sebuah Sistem Operasi. Pada bagian ini akan diperkenalkan semua aspek yang berhubungan dengan pengelolaan memori seperti, pengalamatan logika dan fisik, *swap*, halaman (*page*), bingkai (*frame*), memori virtual. segmentasi, serta alokasi memori. Bagian ini akan ditutup dengan penguraian pengelolaan memori Linux. Bagian ini terdiri dari 9 bab.

Bagian VI, "Penyimpanan Sekunder". Pada umumnya, penyimpanan sekunder berbentuk disk magnetik. Kecepatan pengaksesan memori sekunder ini jauh lebih lambat dibandingkan memori utama. Pada bagian ini akan diperkenalkan konsep-konsep yang berhubungan dengan memori sekunder seperti sistem berkas, atribut dan operasi sistem berkas, struktur direktori, atribut dan operasi struktur direktori, sistem berkas jaringan, sistem berkas virtual, sistem berkas GNU/Linux, keamanan sistem berkas, FHS (*File Hierarchy System*), serta alokas blok sistem berkas. Bagian ini terdiri dari 6 bab.

Bagian VII, "Masukan/Keluaran (M/K)". Sistem Masukan/Keluaran (M/K) merupakan bagian penting yang menentukan kinerja sistem secara keseluruhan. Pada bagian ini akan diperkenalkan perangkat-perangkat M/K, konsep *polling*, interupsi, *Direct Memory Access* (DMA), subsistem kernel, aplikasi antarmuka, *streams*, penjadwalan, RAID, kinerja, M/K sistem Linux, serta sistem penyimpanan tersier. Bagian ini terdiri dari 6 bab.

Bagian VIII, "Topik Lanjutan". Sebagai penutup, akan dibahas empat topik lanjutan, yaitu: Sistem Waktu Nyata dan Multimedia, Sistem Terdistribusi, Keamanan Sistem, serta Perancangan dan Pemeliharaan. Bagian ini terdiri dari 4 bab.

Secara khusus, semua soal latihan dihimpun pada Lampiran B ini. Soal-soal tersebut pernah diberikan pada Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia antara tahun 2003 hingga dewasa ini (2006).

2. Kata Pengantar Revisi 1.0

Buku ini merupakan hasil karya Masyarakat Digital Gotong Royong (MDGR) Fakultas Ilmu Komputer Universitas Indonesia (Fasilkom UI). Kelompok Kerja 21-28 mengawali penulisan buku ini, lalu Kelompok Kerja 41-49, 51 melakukan revisi dan perbaikan. Tujuan utama penulisan buku ini ialah untuk dimanfaatkan sendiri sebagai rujukan utama pada mata ajar IKI-20230 Sistem Operasi di Fakultas Ilmu Komputer Universitas Indonesia. Versi digital terakhir dari buku ini dapat diambil dari http://bebas.vlsm.org/ v06/ Kuliah/ SistemOperasi/ BUKU/.

Buku ini mencakup delapan pokok bahasan bidang Sistem Operasi. Setiap pokok bahasan dipisahkan ke dalam bab yang tersendiri, yang kemudian dibagi menjadi beberapa sub-pokok bahasan. Setiap sub-pokok bahasan dirancang untuk mengisi satu jam tatap muka kuliah. Buku yang terdiri dari 52 sub-pokok bahasan ini, sehingga cocok untuk sebuah mata ajar dengan bobot empat Satuan Kredit Semester (SKS).

Pembahasan buku ini diawali dengan pengenalan Konsep Dasar Sistem Komputer (Bab 1). Bab ini akan membahas tiga sub-pokok bahasan, yaitu Pengenalan Sistem Operasi Komputer, Pengenalan Perangkat Keras Komputer, serta Aspek Lainnya seperti: Proteksi, Keamanan, Jaringan. Bab ini bersifat pengulangan hal-hal yang menjadi prasyarat untuk buku ini. Jika mengalami kesulitan memahami bab ini, sebaiknya mendalami kembali subyek yang berhubungan dengan Pengantar Organisasi Komputer serta Pengantar Struktur Data.

Bab 2 akan membahas Konsep Dasar Sistem Operasi. Ini merupakan bab yang paling penting dari buku ini.

Bab 3 akan membahas Proses dan Penjadwalan.

Bab 4 akan membahas Sinkronisasi dan *Deadlock*.

Bab 5 akan membahas Manajemen Memori.

Bab 6 akan membahas Sistem Berkas.

Bab 7 akan membahas M/K.

Akhirnya, bab 8 akan membahas sebuah studi kasus yaitu sistem GNU/Linux.

Setiap bab berisi soal-soal latihan agar para pembaca dapat mengulas kembali pembahasan pada bab tersebut dan mengevaluasi sejauh mana pengetahuan mengenai bab tersebut. Gambar dipilih sedemikian rupa sehingga dapat memberikan ilustrasi yang membantu pembaca untuk lebih memahami pembahasan.

Kami menyadari bahwa pada buku ini sangat *berbau* buku karya Schilberschatz dan kawan-kawan. Kebanyakan sub-pokok bahasan buku ini memang berbasis kerangka kerja (*framework*) buku tersebut. Diharapkan secara perlahan, bau tersebut akan pudar pada revisi-revisi yang mendatang. Silakan menyampaikan kritik/tanggapan/usulan anda.

Bagian I. Konsep Dasar Perangkat Komputer

Komputer modern merupakan sistem yang kompleks. Secara fisik, komputer tersebut terdiri dari beberapa bagian seperti prosesor, memori, disk, pencetak (*printer*), serta perangkat lainnya. Perangkat keras tersebut digunakan untuk menjalankan berbagai perangkat lunak aplikasi (*software aplication*). Sebuah *Sistem Operasi* merupakan perangkat lunak penghubung antara perangkat keras (*hardware*) dengan perangkat lunak aplikasi tersebut di atas.

Bagian ini (Bagian I, “Konsep Dasar Perangkat Komputer”), menguraikan secara umum komponen-komponen komputer seperti Sistem Operasi, perangkat keras, proteksi, keamanan, serta jaringan komputer. Aspek-aspek tersebut diperlukan untuk memahami konsep-konsep Sistem Operasi yang akan dijabarkan dalam buku ini. Tentunya tidak dapat diharapkan pembahasan yang dalam. Rincian lanjut, sebaiknya dilihat pada rujukan yang berhubungan dengan "Pengantar Organisasi Komputer", "Pengantar Struktur Data", serta "Pengantar Jaringan Komputer". Bagian II, “Konsep Dasar Sistem Operasi” akan memperkenalkan secara umum seputar Sistem Operasi. Bagian selanjutnya, akan menguraikan yang lebih rinci dari seluruh aspek Sistem Operasi.

Bab 1. Hari Gini Belajar SO?

1.1. Pendahuluan

Mengapa Sistem Operasi masih menjadi bagian dari inti kurikulum bidang Ilmu Komputer? Bab pendahuluan ini akan memberikan sedikit gambaran perihal posisi Sistem Operasi di abad 21 ini.

1.2. Mengapa Mempelajari Sistem Operasi?

Setelah lebih dari 60 tahun sejarah perkomputeran, telah terjadi pergeseran yang signifikan dari peranan sebuah Sistem Operasi. Perhatikan Tabel 1.1, “Perbandingan Sistem Dahulu dan Sekarang” berikut ini. Secara sepintas, terlihat bahwa telah terjadi perubahan sangat drastis dalam dunia Teknologi Informasi dan Ilmu Komputer.

Tabel 1.1. Perbandingan Sistem Dahulu dan Sekarang

	Dahulu	**Sekarang**
Komputer Utama	*Mainframe*	Jaringan Komputer Personal
Memori	Beberapa Kbytes	Beberapa Gbytes
Disk	Beberapa Mbytes	Beberapa ratus Gbytes
Peraga	Terminal Teks	Grafik beresolusi Tinggi
Arsitektur	Beragam arsitektur	Dominasi keluarga i386
Sistem Operasi	Beda Sistem Operasi untuk Setiap Arsitektur	Dominasi *Microsoft* dengan beberapa pengecualian

Hal yang paling terlihat secara kasat mata ialah perubahan (pengecilan) fisik yang luar biasa. Penggunaan memori dan disk pun meningkat dengan tajam, terutama setelah multimedia mulai dimanfaatkan sebagai antarmuka interaksi. Saat dahulu, setiap arsitektur komputer memiliki Sistem Operasi yang tersendiri. Jika dewasa ini telah terjadi penciutan arsitektur yang luar biasa, dengan sendirinya menciutkan jumlah variasi Sistem Operasi. Hal ini ditambah dengan trend Sistem Operasi yang dapat berjalan diberbagai jenis arsitektur. Sebagian dari pembaca yang budiman mungkin mulai bernalar: mengapa "hari gini" (terpaksa) mempelajari Sistem Operasi?! Secara pasti-pasti, dimana relevansi dan "*job* (duit)"-nya?

Terlepas dari perubahan tersebut di atas; banyak aspek yang tetap sama seperti dahulu. Komputer abad lalu menggunakan model arsitektur von-Neumann, dan demikian pula model komputer abad ini. Aspek pengelolaan sumber-daya Sistem Operasi seperti proses, memori, masukan/keluaran (m/k), berkas, dan seterusnya masih menggunakan prinsip-prinsip yang sama. Dengan sendirinya, mempelajari Sistem Operasi masih tetap serelevan abad lalu; walaupun telah terjadi berbagai perubahan fisik.

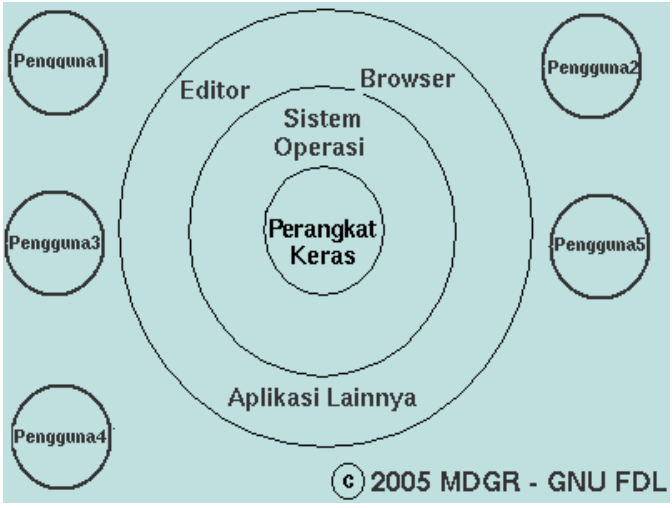
1.3. Definisi Sementara

Buku ini merupakan sebuah rujukan mata-ajar Sistem Operasi (SO). Hampir seluruh isi buku akan menggunjingkan secara panjang-lebar, semua aspek yang berhubungan dengan Sistem Operasi tersebut. Namun sebelum pergunjingan dimulai, perlu ditetapkan sebuah pegangan sementara, perihal apa yang dimaksud dengan "Sistem Operasi" itu sendiri.

Mendefinisikan istilah "Sistem Operasi" mungkin merupakan hal yang mudah, namun mungkin juga merupakan hal yang sangat ribet! Para pembaca sepertinya pernah mendengar istilah "Sistem Operasi". Mungkin pula pernah berhubungan secara langsung ataupun tidak langsung dengan istilah tersebut. Namun, belum tentu dapat menjabarkan perihal apa yang sebetulnya dimaksud dengan kata

"Sistem Operasi". Sebaliknya, banyak pula yang pernah mendengar merek dagang "*Windows*™ [1])" ataupun istilah "GNU/Linux [2])", lalu mengidentikkan nama *Windows* ™ atau GNU/Linux dengan istilah "Sistem Operasi" tersebut.

Gambar 1.1. Abstraksi Komponen Sistem Komputer



Sebuah sistem komputer dapat dibagi ke dalam beberapa komponen utama, seperti "para pengguna", "perangkat keras", serta "perangkat lunak" (Gambar 1.1, “Abstraksi Komponen Sistem Komputer”). "Para pengguna" (*users*) ini merupakan pihak yang memanfaatkan sistem komputer tersebut. Para pengguna di sini bukan saja manusia, namun mungkin berbentuk program aplikasi lain, ataupun perangkat komputer lain. "Perangkat keras" (*hardware*) ini berbentuk benda konkret yang dapat dilihat dan disentuh. Perangkat keras ini merupakan inti dari sebuah sistem, serta penyedia sumber-daya (*resources*) untuk keperluan komputasi. Diantara "para pengguna" dan "perangkat keras" terdapat sebuah lapisan abstrak yang disebut dengan "perangkat lunak" (*software*). Secara keseluruhan, perangkat lunak membantu para pengguna untuk memanfaatkan sumber-daya komputasi yang disediakan perangkat keras.

Perangkat lunak secara garis besar dibagi lagi menjadi dua yaitu "program aplikasi" dan "Sistem Operasi". "Program aplikasi" merupakan perangkat lunak yang dijalankan oleh para pengguna untuk mencapat tujuan tertentu. Umpama, kita menjelajah internet dengan menggunakan aplikasi "*Browser*". Atau mengubah (edit) sebuah berkas dengan aplikasi "*Editor*". Sedangkan, "Sistem Operasi" dapat dikatakan merupakan sebuah perangkat lunak yang "membungkus" perangkat keras agar lebih mudah dimanfaatkan oleh para pengguna melalui program-program aplikasi tersebut.

Sistem Operasi berada di antara perangkat keras komputer dan perangkat aplikasinya. Namun, bagaimana caranya menentukan secara pasti, letak perbatasan antara "perangkat keras komputer" dan "Sistem Operasi", dan terutama antara "perangkat lunak aplikasi" dan "Sistem Operasi"? Umpamanya, apakah "*Internet Explorer*™ [3])" merupakan aplikasi atau bagian dari Sistem Operasi? Siapakah yang berhak menentukan perbatasan tersebut? Apakah para pengguna? Apakah perlu didiskusikan habis-habisan melalui milis? Apakah perlu diputuskan oleh sebuah pengadilan? Apakah para politisi (busuk?) sebaiknya mengajukan sebuah Rencana Undang Undang Sistem Operasi terlebih dahulu? Ha!

Secara lebih rinci, Sistem Operasi didefinisikan sebagai sebuah program yang mengatur perangkat keras komputer, dengan menyediakan landasan untuk aplikasi yang berada di atasnya, serta bertindak sebagai penghubung antara para pengguna dengan perangkat keras. Sistem Operasi bertugas untuk mengendalikan (kontrol) serta mengkoordinasikan pengunaan perangkat keras untuk berbagai program aplikasi untuk bermacam-macam pengguna. Dengan demikian, sebuah Sistem Operasi **bukan** merupakan bagian dari perangkat keras komputer, dan juga **bukan** merupakan

[1] Windows merupakan merek dagang terdaftar dari *Microsoft*.
[2] GNU merupakan singkatan dari GNU is Not Unix, sedangkan Linux merupakan merek dagang dari Linus Torvalds.
[3] Internet Explorer merupakan merek dagang terdaftar dari Microsoft.

bagian dari perangkat lunak aplikasi komputer, apalagi tentunya **bukan** merupakan bagian dari para pengguna komputer.

Pengertian dari Sistem Operasi dapat dilihat dari berbagai sudut pandang. Dari sudut pandang pengguna, Sistem Operasi merupakan sebagai alat untuk mempermudah penggunaan komputer. Dalam hal ini Sistem Operasi seharusnya dirancang dengan mengutamakan kemudahan penggunaan, dibandingkan mengutamakan kinerja ataupun utilisasi sumber-daya. Sebaliknya dalam lingkungan berpengguna-banyak (*multi-user*), Sistem Operasi dapat dipandang sebagai alat untuk memaksimalkan penggunaan sumber-daya komputer. Akan tetapi pada sejumlah komputer, sudut pandang pengguna dapat dikatakan hanya sedikit atau tidak ada sama sekali. Misalnya *embedded computer* pada peralatan rumah tangga seperti mesin cuci dan sebagainya mungkin saja memiliki lampu indikator untuk menunjukkan keadaan sekarang, tetapi Sistem Operasi ini dirancang untuk bekerja tanpa campur tangan pengguna.

Dari sudut pandang sistem, Sistem Operasi dapat dianggap sebagai alat yang menempatkan sumber-daya secara efisien (*Resource Allocator*). Sistem Operasi ialah manager bagi sumber-daya, yang menangani konflik permintaan sumber-daya secara efisien. Sistem Operasi juga mengatur eksekusi aplikasi dan operasi dari alat M/K (Masukan/Keluaran). Fungsi ini dikenal juga sebagai program pengendali (*Control Program*). Lebih lagi, Sistem Operasi merupakan suatu bagian program yang berjalan setiap saat yang dikenal dengan istilah kernel.

Dari sudut pandang tujuan Sistem Operasi, Sistem Operasi dapat dipandang sebagai alat yang membuat komputer lebih nyaman digunakan (*convenient*) untuk menjalankan aplikasi dan menyelesaikan masalah pengguna. Tujuan lain Sistem Operasi ialah membuat penggunaan sumber-daya komputer menjadi efisien.

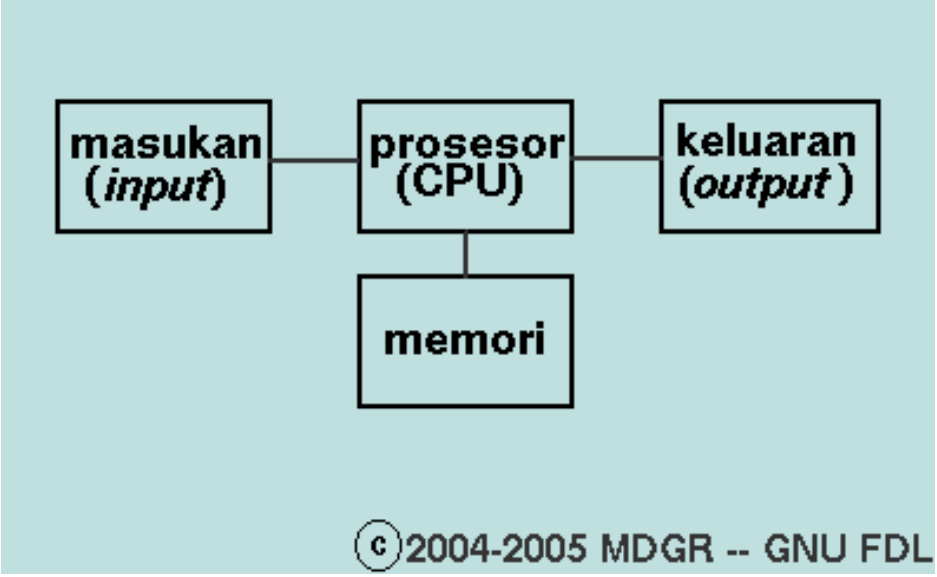
Dapat disimpulkan, bahwa Sistem Operasi merupakan komponen penting dari setiap sistem komputer. Akibatnya, pelajaran "Sistem Operasi" selayaknya merupakan komponen penting dari sistem pendidikan berbasis "ilmu komputer". Konsep Sistem Operasi dapat lebih mudah dipahami, jika juga memahami jenis perangkat keras yang digunakan. Demikian pula sebaliknya. Dari sejarah diketahui bahwa Sistem Operasi dan perangkat keras saling mempengaruhi dan saling melengkapi. Struktur dari sebuah Sistem Operasi sangat tergantung pada perangkat keras yang pertama kali digunakan untuk mengembangkannya. Sedangkan perkembangan perangkat keras sangat dipengaruhi dari hal-hal yang diperlukan oleh sebuah Sistem Operasi. Dalam sub bagian-bagian berikut ini, akan diberikan berbagai ilustrasi perkembangan dan jenis Sistem Operasi beserta perangkat kerasnya.

1.4. Sejarah Perkembangan

Arsitektur perangkat keras komputer tradisional terdiri dari empat komponen utama yaitu "Prosesor", "Memori Penyimpanan", "Masukan" (*Input*), dan "Keluaran" (*Output*). Model tradisional tersebut sering dikenal dengan nama arsitektur von-Neumann (Gambar 1.2, “Arsitektur Komputer von-Neumann”). Pada saat awal, komputer berukuran sangat besar sehingga komponen-komponennya dapat memenuhi sebuah ruangan yang sangat besar. Sang pengguna -- menjadi programer yang sekali gus merangkap menjadi menjadi operator komputer -- juga bekerja di dalam ruang komputer tersebut.

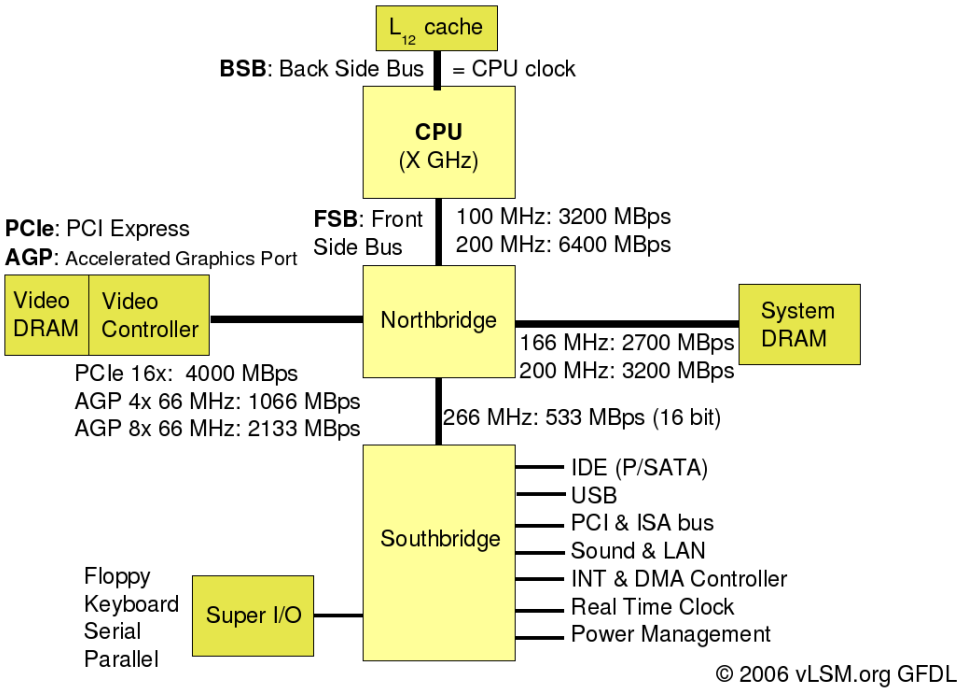
Walaupun berukuran besar, sistem tersebut dikategorikan sebagai "komputer pribadi" (PC). Siapa saja yang ingin melakukan komputasi; harus memesan/antri untuk mendapatkan alokasi waktu (rata-rata 30-120 menit). Jika ingin melakukan kompilasi Fortran, maka pengguna pertama kali akan me-*load* kompilator Fortran, yang diikuti dengan "*load*" program dan data. Hasil yang diperoleh, biasanya berbentuk cetakan (*print-out*). Timbul beberapa masalah pada sistem PC tersebut. Umpama, alokasi pesanan harus dilakukan dimuka. Jika pekerjaan rampung sebelum rencana semula, maka sistem komputer menjadi "*idle*"/tidak tergunakan. Sebaliknya, jika perkerjaan rampung lebih lama dari rencana semula, para calon pengguna berikutnya harus menunggu hingga pekerjaan selesai. Selain itu, seorang pengguna kompilator Fortran akan beruntung, jika pengguna sebelumnya juga menggunakan Fortran. Namun, jika pengguna sebelumnya menggunakan Cobol, maka pengguna Fortran harus me-"*load*". Masalah ini ditanggulangi dengan menggabungkan para pengguna kompilator sejenis ke dalam satu kelompok *batch* yang sama. Medium semula yaitu *punch card* diganti dengan *tape*.

Gambar 1.2. Arsitektur Komputer von-Neumann



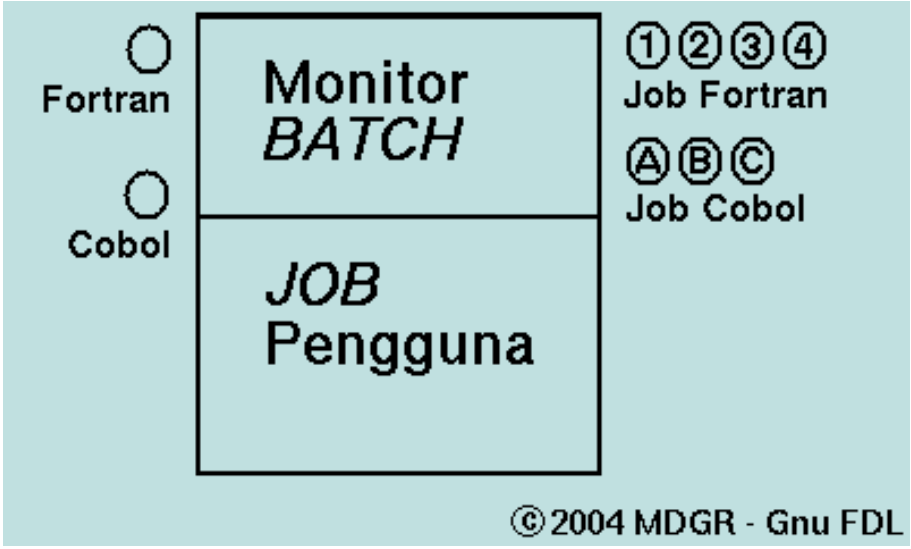
Selanjutnya, terjadi pemisahan tugas antara programer dan operator. Para operator biasanya secara eksklusif menjadi penghuni "ruang kaca" seberang ruang komputer. Para programer yang merupakan pengguna (*users*), mengakses komputer secara tidak langsung melalui bantuan para operator. Para pengguna mempersiapkan sebuah *job* yang terdiri dari program aplikasi, data masukan, serta beberapa perintah pengendali program. Medium yang lazim digunakan ialah kartu berlubang (*punch card*). Setiap kartu dapat menampung informasi satu baris hingga 80 karakter Set kartu *job* lengkap tersebut kemudian diserahkan kepada para operator.

Gambar 1.3. Bagan Sebuah Komputer Personal



Perkembangan Sistem Operasi dimulai dari sini, dengan memanfaatkan sistem *batch* (Gambar 1.4, “Bagan Memori Untuk Sistem *Monitor Batch* Sederhana”). Para operator mengumpulkan *job-job* yang mirip yang kemudian dijalankan secara berkelompok. Umpama, *job* yang memerlukan kompilator Fortran akan dikumpulkan ke dalam sebuah *batch* bersama dengan *job-job* lainnya yang juga memerlukan kompilator Fortran. Setelah sebuah kelompok *job* rampung, maka kelompok *job* berikutnya akan dijalankan secara otomatis.

Gambar 1.4. Bagan Memori Untuk Sistem *Monitor Batch* Sederhana



Pada perkembangan berikutnya, diperkenalkan konsep *Multiprogrammed System*. Dengan sistem ini *job-job* disimpan di memori utama di waktu yang sama dan *CPU* dipergunakan bergantian. Hal ini membutuhkan beberapa kemampuan tambahan yaitu: penyediaan *I/O routine* oleh sistem, pengaturan memori untuk mengalokasikan memori pada beberapa *Job*, penjadwalan *CPU* untuk memilih *job* mana yang akan dijalankan, serta pengalokasian perangkat keras lain (Gambar 1.5, “Bagan Memori untuk Model *Multiprogram System*”).

Gambar 1.5. Bagan Memori untuk Model *Multiprogram System*



Peningkatan lanjut dikenal sistem "bagi waktu"/"tugas ganda"/"komputasi interaktif" (*Time-Sharing System*/*Multitasking*/*Interactive Computing*). Sistem ini, secara simultan dapat diakses lebih dari satu pengguna. *CPU* digunakan bergantian oleh *job-job* di memori dan di disk. *CPU* dialokasikan hanya pada *job* di memori dan *job* dipindahkan dari dan ke disk. Interaksi langsung antara pengguna dan komputer ini melahirkan konsep baru, yaitu *response time* yang diupayakan wajar agar tidak terlalu lama menunggu.

Hingga akhir tahun 1980-an, sistem komputer dengan kemampuan yang "normal", lazim dikenal dengan istilah *main-frame*. Sistem komputer dengan kemampuan jauh lebih rendah (dan lebih murah) disebut "komputer mini". Sebaliknya, komputer dengan kemampuan jauh lebih canggih disebut komputer super (*super-computer*). CDC 6600 merupakan yang pertama dikenal dengan sebutan komputer super menjelang akhir tahun 1960-an. Namun prinsip kerja dari Sistem Operasi dari semua komputer tersebut lebih kurang sama saja.

Komputer klasik seperti diungkapkan di atas, hanya memiliki satu prosesor. Keuntungan dari sistem ini ialah lebih mudah diimplementasikan karena tidak perlu memperhatikan sinkronisasi antar prosesor, kemudahan kontrol terhadap prosesor karena sistem proteksi tidak, teralu rumit, dan cenderung murah (bukan ekonomis). Perlu dicatat yang dimaksud satu buah prosesor ini ialah satu buah prosesor sebagai *Central Processing Unit* (CPU). Hal ini ditekankan sebab ada beberapa perangkat yang memang memiliki prosesor tersendiri di dalam perangkatnya seperti *VGA Card AGP*, *Optical Mouse*, dan lain-lain.

1.5. Bahan Pembahasan

Mudah-mudahan para pembaca telah yakin bahwa hari gini pun masih relevan mempelajari Sistem Operasi! Buku ini terdiri dari delapan bagian yang masing-masing akan membahas satu pokok pembahasan. Setiap bagian akan terdiri dari beberapa bab yang masing-masing akan membahas sebuah sub-pokok pembahasan untuk sebuah jam pengajaran (sekitar 40 menit). Setiap sub-pokok pengajaran ini, terdiri dari sekitar 5 hingga 10 seksi yang masing-masing membahas sebuah ide. Terakhir, setiap ide merupakan unit terkecil yang biasanya dapat dijabarkan kedalam satu atau dua halaman peraga seperti lembaran transparan. Dengan demikian, setiap jam pengajaran dapat diuraikan ke dalam 5 hingga 20 lembaran transparan peraga.

Lalu, pokok bahasan apa saja yang akan dibahas di dalam buku ini? Bagian I, “Konsep Dasar Perangkat Komputer” akan berisi pengulangan – terutama konsep organisasi komputer dan perangkat keras – yang diasumsikan telah dipelajari di mata ajar lain. Bagian II, “Konsep Dasar Sistem Operasi” akan membahas secara ringkas dan pada aspek-aspek pengelolaan sumber-daya Sistem Operasi yang akan dijabarkan pada bagian-bagian berikutnya. Bagian-bagian tersebut akan membahas aspek pengelolaan proses dan penjadwalannya, proses dan sinkronisasinya, memori, memori sekunder, serta masukan/keluaran (m/k). Bagian terakhir (Bagian VIII, “Topik Lanjutan”) akan membahas beberapa topik lanjutan yang terkait dengan Sistem Operasi.

Buku ini bukan merupakan tuntunan praktis menjalankan sebuah Sistem Operasi. Pembahasan akan dibatasi pada tingkat konseptual. Penjelasan lanjut akan diungkapan berikut.

1.6. Prasyarat

Memiliki pengetahuan dasar struktur data, algoritma pemrograman, dan organisasi sistem komputer. Bagian pertama ini akan mengulang secara sekilas sebagian dari prasyarat ini. Jika mengalami kesulitan memahami bagian ini, sebaiknya mencari informasi tambahan sebulum melanjutkan buku ini. Selain itu, diharapkan menguasai bahasa Java.

1.7. Sasaran Pembelajaran

Sasaran utama yang diharapkan setelah mendalami buku ini ialah:
- Mengenal komponen-komponen yang membentuk Sistem Operasi.
- Dapat menjelaskan peranan dari masing-masing komponen tersebut.
- Seiring dengan pengetahuan yang didapatkan dari Organisasi Komputer, dapat menjelaskan atau meramalkan kinerja dari aplikasi yang berjalan di atas Sistem Operasi dan perangkat keras

tersebut.
- Landasan/fondasi bagi mata ajar lainnya, sehingga dapat menjelaskan konsep-konsep bidang tersebut.

1.8. Rangkuman

Sistem Operasi telah berkembang selama lebih dari 40 tahun dengan dua tujuan utama. Pertama, Sistem Operasi mencoba mengatur aktivitas-aktivitas komputasi untuk memastikan pendaya-gunaan yang baik dari sistem komputasi tersebut. Kedua, menyediakan lingkungan yang nyaman untuk pengembangan dan jalankan dari program.

Pada awalnya, sistem komputer digunakan dari depan konsol. Perangkat lunak seperti *assembler*, *loader*, *linker* dan kompilator meningkatkan kenyamanan dari sistem pemrograman, tapi juga memerlukan waktu set-up yang banyak. Untuk mengurangi waktu set-up tersebut, digunakan jasa operator dan menggabungkan tugas-tugas yang sama (sistem *batch*).

Sistem *batch* mengizinkan pengurutan tugas secara otomatis dengan menggunakan Sistem Operasi yang resident dan memberikan peningkatan yang cukup besar dalam utilisasi komputer. Komputer tidak perlu lagi menunggu operasi oleh pengguna. Tapi utilisasi CPU tetap saja rendah. Hal ini dikarenakan lambatnya kecepatan alat-alat untuk M/K relatif terhadap kecepatan CPU. Operasi *off-line* dari alat-alat yang lambat bertujuan untuk menggunakan beberapa sistem reader-to-tape dan tape-to-printer untuk satu CPU. Untuk meningkatkan keseluruhan kemampuan dari sistem komputer, para developer memperkenalkan konsep *multiprogramming*.

Rujukan

[Morgan1992] K Morgan. “The RTOS Difference”. *Byte*. August 1992. 1992.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

Bab 2. HaKI Perangkat Lunak

2.1. Pendahuluan

Ketergantungan terhadap Sistem Informasi telah merambah ke berbagai bidang, mulai dari Sistem Otomasi Perkantoran sebuah Usaha Kecil, hingga Sistem Kendali sebuah Instalasi Nuklir berpresisi tinggi. Peranan dari komponen-komponen sebuah Sistem Informasi pun menjadi vital. Salah satu komponen tersebut ialah Perangkat Lunak (PL), baik dalam bentuk kernel Sistem Operasi beserta utilisasinya, maupun Aplikasi yang berjalan di atas Sistem tersebut.

Walau pun PL memegang peranan yang penting, pengertian publik terhadap Hak atas Kekayaan Intelektual Perangkat Lunak (HaKI PL) masih relatif minim. Kebinggungan ini bertambah dengan peningkatan pemanfaatan dari Perangkat Lunak Bebas (PLB) – Free Software – dan Perangka Lunak Sistem Terbuka (PLST) – Open Source Software (OSS). PLB ini sering disalahkaprahkan sebagai ST, walau pun sebetulnya terdapat beberapa berbedaan yang mendasar diantara kedua pendekatan tersebut. Bab ini mencoba memantapkan pengertian atas HaKI PL. Pembahasan dimulai dengan menerangkan konsep HaKI secara umum, serta HaKI PL secara lebih dalam. Secara khusus akan dibahas konsep Perangkat Lunak Bebas/Sumber Terbuka (Free/Open Source Software – F/OSS).

Pembahasan ini bukan bertujuan sebagai indoktrinasi faham tersebut! Justru yang diharapkan:
- Pelurusan atas persepsi keliru PLB dan ST, serta penjelasan perbedaan dan persamaan dari kedua konsep tersebut.
- Apa yang boleh dan apa yang tidak boleh dilakukan dengan PLB/ST.
- Pelurusan atas persepsi bahwa para penulis program komputer tidak berhak digaji layak.
- Pelurusan atas persepsi bahwa PLB tidak boleh dijual/dikomersialkan.
- Pelurusan atas persepsi bahwa PLB wajib disebarluaskan.
- Pelurusan atas persepsi bahwa saat distribusi tidak wajib menyertakan kode sumber.

Setelah menyimak tulisan ini, diharapkan akan lebih memahami dan lebih menghargai makna PLB/ST secara khusus, serta HaKI/PL secara umum.

2.2. Hak Kekayaan Intelektual

Latar belakang

"Hak atas Kekayaan Intelektual" (HaKI) merupakan terjemahan atas istilah "*Intellectual Property Right*" (IPR). Istilah tersebut terdiri dari tiga kata kunci yaitu: "Hak", "Kekayaan" dan "Intelektual". Kekayaan merupakan abstraksi yang dapat: dimiliki, dialihkan, dibeli, maupun dijual. Sedangkan "Kekayaan Intelektual" merupakan kekayaan atas segala hasil produksi kecerdasan daya pikir seperti teknologi, pengetahuan, seni, sastra, gubahan lagu, karya tulis, karikatur, dan seterusnya. Terakhir, HaKI merupakan hak-hak (wewenang/kekuasaan) untuk berbuat sesuatu atas Kekayaan Intelektual tersebut, yang diatur oleh norma-norma atau hukum-hukum yang berlaku.

``Hak" itu sendiri dapat dibagi menjadi dua. Pertama, ``Hak Dasar (Azasi)", yang merupakan hak mutlak yang tidak dapat diganggu-gugat. Umpama: hak untuk hidup, hak untuk mendapatkan keadilan, dan sebagainya. Kedua, ``Hak Amanat/ Peraturan" yaitu hak karena diberikan oleh masyarakat melalui peraturan/perundangan. Di berbagai negara, termasuk Amrik dan Indonesia, HaKI merupakan "Hak Amanat/Pengaturan", sehingga masyarakatlah yang menentukan, seberapa besar HaKI yang diberikan kepada individu dan kelompok. Sesuai dengan hakekatnya pula, HaKI dikelompokkan sebagai hak milik perorangan yang sifatnya tidak berwujud (intangible).

Terlihat bahwa HaKI merupakan Hak Pemberian dari Umum (Publik) yang dijamin oleh Undang-undang. HaKI bukan merupakan Hak Azazi, sehingga kriteria pemberian HaKI merupakan hal yang dapat diperdebatkan oleh publik. Apa kriteria untuk memberikan HaKI? Berapa lama pemegang HaKI memperoleh hak eksklusif? Apakah HaKI dapat dicabut demi kepentingan umum? Bagaimana dengan HaKI atas formula obat untuk para penderita HIV/AIDs?

Tumbuhnya konsepsi kekayaan atas karya-karya intelektual pada akhirnya juga menimbulkan untuk

melindungi atau mempertahankan kekayaan tersebut. Pada gilirannya, kebutuhan ini melahirkan konsepsi perlindungan hukum atas kekayaan tadi, termasuk pengakuan hak terhadapnya. Sesuai dengan hakekatnya pula, HaKI dikelompokkan sebagai hak milik perorangan yang sifatnya tidak berwujud (*intangible*).

Undang-undang mengenai HaKI pertama kali ada di Venice, Italia yang menyangkut masalah paten pada tahun 1470. Caxton, Galileo, dan Guttenberg tercatat sebagai penemu-penemu yang muncul dalam kurun waktu tersebut dan mempunyai hak monopoli atas penemuan mereka. Hukum-hukum tentang paten tersebut kemudian diadopsi oleh kerajaan Inggris di jaman TUDOR tahun 1500-an dan kemudian lahir hukum mengenai paten pertama di Inggris yaitu *Statute of Monopolies* (1623). Amerika Serikat baru mempunyai undang-undang paten tahun 1791. Upaya harmonisasi dalam bidang HaKI pertama kali terjadi tahun 1883 dengan lahirnya konvensi Paris untuk masalah paten, merek dagang dan desain. Kemudian konvensi Berne 1886 untuk masalah Hak Cipta (*Copyright*).

Aneka Ragam HaKI

- **Hak Cipta (*Copyright*).** Berdasarkan pasal 1 ayat 1 Undang-Undang Nomor 19 Tahun 2002 Tentang Hak Cipta:

  > Hak Cipta adalah hak eksklusif bagi Pencipta atau penerima hak untuk mengumumkan atau memperbanyak ciptaannya atau memberikan izin untuk itu dengan tidak mengurangi pembatasan-pembatasan menurut peraturan perundang-undangan yang berlaku.

- **Paten (*Patent*).** Berdasarkan Pasal 1 ayat 1 Undang-Undang Nomor 14 Tahun 2001 Tentang Paten:

  > Paten adalah hak eksklusif yang diberikan oleh Negara kepada Inventor atas hasil Invensinya di bidang teknologi, yang untuk selama waktu tertentu melaksanakan sendiri Invensinya tersebut atau memberikan persetujuannya kepada pihak lain untuk melaksanakannya.

  Berbeda dengan hak cipta yang melindungi sebuah karya, paten melindungi sebuah ide, bukan ekspresi dari ide tersebut. Pada hak cipta, seseorang lain berhak membuat karya lain yang fungsinya sama asalkan tidak dibuat berdasarkan karya orang lain yang memiliki hak cipta. Sedangkan pada paten, seseorang tidak berhak untuk membuat sebuah karya yang cara bekerjanya sama dengan sebuah ide yang dipatenkan.

- **Merk Dagang (*Trademark*).** Berdasarkan pasal 1 ayat 1 Undang-Undang Nomor 15 Tahun 2001 Tentang Merek:

  > Merek adalah tanda yang berupa gambar, nama, kata, huruf-huruf, angka-angka, susunan warna, atau kombinasi dari unsur-unsur tersebut yang memiliki daya pembeda dan digunakan dalam kegiatan perdagangan barang atau jasa.

  Contoh: Kacang Atom cap Ayam Jantan.

- **Rahasia Dagang (*Trade Secret*).** Menurut pasal 1 ayat 1 Undang-Undang Nomor 30 Tahun 2000 Tentang Rahasia Dagang:

  > Rahasia Dagang adalah informasi yang tidak diketahui oleh umum di bidang teknologi dan/atau bisnis, mempunyai nilai ekonomi karena berguna dalam kegiatan usaha, dan dijaga kerahasiaannya oleh pemilik Rahasia Dagang.

  Contoh: rahasia dari formula Parfum.

- ***Service Mark*.** Adalah kata, prase, logo, simbol, warna, suara, bau yang digunakan oleh sebuah bisnis untuk mengindentifikasi sebuah layanan dan membedakannya dari kompetitornya. Pada prakteknya perlindungan hukum untuk merek dagang sedang *service mark* untuk identitasnya. Contoh: "Pegadaian: menyelesaikan masalah tanpa masalah".

- **Desain Industri.** Berdasarkan pasal 1 ayat 1 Undang-Undang Nomor 31 Tahun 2000 Tentang Desain Industri:

  > Desain Industri adalah suatu kreasi tentang bentuk, konfigurasi, atau komposisi garis atau warna, atau garis dan warna, atau gabungan daripadanya yang berbentuk tiga dimensi atau dua dimensi yang memberikan kesan estetis dan dapat diwujudkan dalam pola tiga dimensi atau dua dimensi serta dapat dipakai untuk menghasilkan suatu produk, barang, komoditas industri, atau kerajinan tangan.

- **Desain Tata Letak Sirkuit Terpadu.** Berdasarkan pasal 1 Undang-Undang Nomor 32 Tahun 2000 Tentang Desain Tata Letak Sirkuit Terpadu;

  > Ayat 1: Sirkuit Terpadu adalah suatu produk dalam bentuk jadi atau setengah jadi, yang di dalamnya terdapat berbagai elemen dan sekurang-kurangnya satu dari elemen tersebut adalah elemen aktif, yang sebagian atau seluruhnya saling berkaitan serta dibentuk secara terpadu di dalam sebuah bahan semikonduktor yang dimaksudkan untuk menghasilkan fungsi elektronik.
  >
  > Ayat 2: Desain Tata Letak adalah kreasi berupa rancangan peletakan tiga dimensi dari berbagai elemen, sekurang-kurangnya satu dari elemen tersebut adalah elemen aktif, serta sebagian atau semua interkoneksi dalam suatu Sirkuit Terpadu dan peletakan tiga dimensi tersebut dimaksudkan untuk persiapan pembuatan Sirkuit Terpadu.

- **Indikasi Geografis.** Berdasarkan pasal 56 ayat 1 Undang-Undang No. 15 Tahun 2001 Tentang Merek:

  > Indikasi-geografis dilindungi sebagai suatu tanda yang menunjukkan daerah asal suatu barang yang karena faktor lingkungan geografis termasuk faktor alam, faktor manusia, atau kombinasi dari kedua faktor tersebut, memberikan ciri dan kualitas tertentu pada barang yang dihasilkan.

2.3. HaKI Perangkat Lunak

Di Indonesia, HaKI PL termasuk ke dalam kategori Hak Cipta (*Copyright*). Beberapa negara, mengizinkan pematenan perangkat lunak. Pada industri perangkat lunak, sangat umum perusahaan besar memiliki portfolio paten yang berjumlah ratusan, bahkan ribuan. Sebagian besar perusahaan-perusahaan ini memiliki perjanjian *cross-licensing*, artinya "Saya izinkan anda menggunakan paten saya asalkan saya boleh menggunakan paten anda". Akibatnya hukum paten pada industri perangkat lunak sangat merugikan perusahaan-perusahaan kecil yang cenderung tidak memiliki paten. Tetapi ada juga perusahaan kecil yang menyalahgunakan hal ini.

Banyak pihak tidak setuju terhadap paten perangkat lunak karena sangat merugikan industri perangkat lunak. Sebuah paten berlaku di sebuah negara. Jika sebuah perusahaan ingin patennya berlaku di negara lain, maka perusahaan tersebut harus mendaftarkan patennya di negara lain tersebut. Tidak seperti hak cipta, paten harus didaftarkan terlebih dahulu sebelum berlaku.

Perangkat Lunak Berpemilik

Perangkat lunak berpemilik (*propriety*) ialah perangkat lunak yang tidak bebas atau pun semi-bebas. Seseorang dapat dilarang, atau harus meminta izin, atau akan dikenakan pembatasan lainnya sehingga menyulitkan – jika menggunakan, mengedarkan, atau memodifikasinya.

Perangkat Komersial

Perangkat lunak komersial adalah perangkat lunak yang dikembangkan oleh kalangan bisnis untuk memperoleh keuntungan dari penggunaannya. ``Komersial'' dan ``kepemilikan'' adalah dua hal yang

berbeda! Kebanyakan perangkat lunak komersial adalah berpemilik, tapi ada perangkat lunak bebas komersial, dan ada perangkat lunak tidak bebas dan tidak komersial. Sebaiknya, istilah ini tidak digunakan.

Perangkat Lunak Semi-Bebas

Perangkat lunak semibebas adalah perangkat lunak yang tidak bebas, tapi mengizinkan setiap orang untuk menggunakan, menyalin, mendistribusikan, dan memodifikasinya (termasuk distribusi dari versi yang telah dimodifikasi) untuk tujuan tertentu (Umpama nirlaba). PGP adalah salah satu contoh dari program semibebas. Perangkat lunak semibebas jauh lebih baik dari perangkat lunak berpemilik, namun masih ada masalah, dan seseorang tidak dapat menggunakannya pada sistem operasi yang bebas.

Public Domain

Perangkat lunak *public domain* ialah perangkat lunak yang tanpa hak cipta. Ini merupakan kasus khusus dari perangkat lunak bebas non-*copyleft*, yang berarti bahwa beberapa salinan atau versi yang telah dimodifikasi bisa jadi tidak bebas sama sekali. Terkadang ada yang menggunakan istilah ``*public domain*" secara bebas yang berarti ``cuma-cuma" atau ``tersedia gratis". Namun ``public domain" merupakan istilah hukum yang artinya ``tidak memiliki hak cipta". Untuk jelasnya, kami menganjurkan untuk menggunakan istilah ``public domain" dalam arti tersebut, serta menggunakan istilah lain untuk mengartikan pengertian yang lain.

Sebuah karya adalah public domain jika pemilik hak ciptanya menghendaki demikian. Selain itu, hak cipta memiliki waktu kadaluwarsa. Sebagai contoh, lagulagu klasik sebagian besar adalah public domain karena sudah melewati jangka waktu kadaluwarsa hak cipta.

Freeware

Istilah ``*freeware*" tidak terdefinisi dengan jelas, tapi biasanya digunakan untuk paket-paket yang mengizinkan redistribusi tetapi bukan pemodifikasian (dan kode programnya tidak tersedia). Paket-paket ini bukan perangkat lunak bebas.

Shareware

Shareware ialah perangkat lunak yang mengizinkan orang-orang untuk meredistribusikan salinannya, tetapi mereka yang terus menggunakannya diminta untuk membayar biaya lisensi. Dalam prakteknya, orang-orang sering tidak mempedulikan perjanjian distribusi dan tetap melakukan hal tersebut, tapi sebenarnya perjanjian tidak mengizinkannya.

Perangkat Lunak Bebas

Perangkat lunak bebas ialah perangkat lunak yang mengizinkan siapa pun untuk menggunakan, menyalin, dan mendistribusikan, baik dimodifikasi atau pun tidak, secara gratis atau pun dengan biaya. Perlu ditekankan, bahwa kode sumber dari program harus tersedia. Jika tidak ada kode program, berarti bukan perangkat lunak. Perangkat Lunak Bebas mengacu pada kebebasan para penggunanya untuk menjalankan, menggandakan, menyebarluaskan, mempelajari, mengubah dan meningkatkan kinerja perangkat lunak. Tepatnya, mengacu pada empat jenis kebebasan bagi para pengguna perangk at lunak:

- **Kebebasan 0.** Kebebasan untuk menjalankan programnya untuk tujuan apa saja.
- **Kebebasan 1.** Kebebasan untuk mempelajari bagaimana program itu bekerja serta dapat disesuaikan dengan kebutuhan anda. Akses pada kode program merupakan suatu prasyarat.
- **Kebebasan 2.** Kebebasan untuk menyebarluaskan kembali hasil salinan perangkat lunak tersebut sehingga dapat membantu sesama anda.
- **Kebebasan 3.** Kebebasan untuk meningkatkan kinerja program, dan dapat menyebarkannya ke khalayak umum sehingga semua menikmati keuntungannya. Akses pada kode program merupakan suatu prasyarat juga.

Suatu program merupakan perangkat lunak bebas, jika setiap pengguna memiliki semua dari

kebebasan tersebut. Dengan demikian, anda seharusnya bebas untuk menyebarluaskan salinan program itu, dengan atau tanpa modifikasi (perubahan), secara gratis atau pun dengan memungut biaya penyebarluasan, kepada siapa pun dimana pun. Kebebasan untuk melakukan semua hal di atas berarti anda tidak harus meminta atau pun membayar untuk izin tersebut.

Perangkat lunak bebas bukan berarti ``tidak komersial''. Program bebas harus boleh digunakan untuk keperluan komersial. Pengembangan perangkat lunak bebas secara komersial pun tidak merupakan hal yang aneh; dan produknya ialah perangkat lunak bebas yang komersial.

Copylefted/Non-Copylefted

Perangkat lunak *copylefted* merupakan perangkat lunak bebas yang ketentuan pendistribusinya tidak memperbolehkan untuk menambah batasan-batasan tambahan – jika mendistribusikan atau memodifikasi perangkat lunak tersebut. Artinya, setiap salinan dari perangkat lunak, walaupun telah dimodifikasi, haruslah merupakan perangkat lunak bebas.

Perangkat lunak bebas non-*copyleft* dibuat oleh pembuatnya yang mengizinkan seseorang untuk mendistribusikan dan memodifikasi, dan untuk menambahkan batasan-batasan tambahan dalamnya. Jika suatu program bebas tapi tidak *copyleft*, maka beberapa salinan atau versi yang dimodifikasi bisa jadi tidak bebas sama sekali. Perusahaan perangkat lunak dapat mengkompilasi programnya, dengan atau tanpa modifikasi, dan mendistribusikan file tereksekusi sebagai produk perangkat lunak yang berpemilik. Sistem X Window menggambarkan hal ini.

Perangkat Lunak Kode Terbuka

Konsep Perangkat Lunak Kode Terbuka (*Open Source Software*) pada intinya adalah membuka kode sumber (*source code*) dari sebuah perangkat lunak. Konsep ini terasa aneh pada awalnya dikarenakan kode sumber merupakan kunci dari sebuah perangkat lunak. Dengan diketahui logika yang ada di kode sumber, maka orang lain semestinya dapat membuat perangkat lunak yang sama fungsinya. *Open source* hanya sebatas itu. Artinya, tidak harus gratis. Kita bisa saja membuat perangkat lunak yang kita buka kode-sumber-nya, mempatenkan algoritmanya, medaftarkan hak cipta, dan tetap menjual perangkat lunak tersebut secara komersial (alias tidak gratis). definisi open source yangasli seperti tertuang dalam OSD (Open Source Definition) yaitu:

- Free Redistribution
- Source Code
- Derived Works
- Integrity of the Authors Source Code
- No Discrimination Against Persons or Groups
- No Discrimination Against Fields of Endeavor
- Distribution of License
- License Must Not Be Specific to a Product
- License Must Not Contaminate Other Software

GNU General Public License (GNU/GPL)

GNU/GPL merupakan sebuah kumpulan ketentuan pendistribusian tertentu untuk meng-copyleft-kan sebuah program. Proyek GNU menggunakannya sebagai perjanjian distribusi untuk sebagian besar perangkat lunak GNU. Sebagai contoh adalah lisensi GPL yang umum digunakan pada perangkat lunak Open Source. GPL memberikan hak kepada orang lain untuk menggunakan sebuah ciptaan asalkan modifikasi atau produk derivasi dari ciptaan tersebut memiliki lisensi yang sama. Kebalikan dari hak cipta adalah public domain. Ciptaan dalam public domain dapat digunakan sekehendaknya oleh pihak lain.

2.4. Komersialisasi Perangkat Lunak

Bebas pada kata perangkat lunak bebas tepatnya adalah bahwa para pengguna bebas untuk menjalankan suatu program, mengubah suatu program, dan mendistribusi ulang suatu program dengan atau tanpa mengubahnya. Berhubung perangkat lunak bebas bukan perihal harga, harga yang murah tidak menjadikannya menjadi lebih bebas, atau mendekati bebas. Jadi jika anda mendistribusi

ulang salinan dari perangkat lunak bebas, anda dapat saja menarik biaya dan mendapatkan uang. Mendistribusi ulang perangkat lunak bebas merupakan kegiatan yang baik dan sah; jika anda melakukannya, silakan juga menarik keuntungan.

Beberapa bentuk model bisnis yang dapat dilakukan dengan Open Source:
- Support/seller, pendapatan diperoleh dari penjualan media distribusi, branding, pelatihan, jasa konsultasi, pengembangan custom, dan dukungan setelah penjualan.
- Loss leader, suatu produk Open Source gratis digunakan untuk menggantikan perangkat lunak komersial.
- Widget Frosting, perusahaan pada dasarnya menjual perangkat keras yang menggunakan program open source untuk menjalankan perangkat keras seperti sebagai driver atau lainnya.
- Accecorizing, perusahaan mendistribusikan buku, perangkat keras, atau barang fisik lainnya yang berkaitan dengan produk Open Source, misal penerbitan buku O Reilly.
- Service Enabler, perangkat lunak Open Source dibuat dan didistribusikan untuk mendukung ke arah penjualan service lainnya yang menghasilkan uang.
- Brand Licensing, Suatu perusahaan mendapatkan penghasilan dengan penggunaan nama dagangnya.
- Sell it, Free it, suatu perusahaan memulai siklus produksinya sebagai suatu produk komersial dan lalu mengubahnya menjadi produk open Source.
- Software Franchising, ini merupakan model kombinasi antara brand licensing dan support/seller.

2.5. Ancaman dan Tantangan

Perangkat Keras Rahasia

Para pembuat perangkat keras cenderung untuk menjaga kerahasiaan spesifikasi perangkat mereka. Ini menyulitkan penulisan driver bebas agar Linux dan XFree86 dapat mendukung perangkat keras baru tersebut. Walau pun kita telah memiliki sistem bebas yang lengkap dewasa ini, namun mungkin saja tidak di masa mendatang, jika kita tidak dapat mendukung komputer yang akan datang.

Pustaka tidak bebas

Pustaka tidak bebas yang berjalan pada perangkat lunak bebas dapt menjadi perangkap bagi pengembang perangkat lunak bebas. Fitur menarik dari pustaka tersebut merupakan umpan; jika anda menggunakannya; anda akan terperangkap, karena program anda tidak akan menjadi bagian yang bermanfaat bagi sistem operasi bebas. Jadi, kita dapat memasukkan program anda, namun tidak akan berjalan jika pustaka-nya tidak ada. Lebih parah lagi, jika program tersebut menjadi terkenal, tentunya akan menjebak lebih banyak lagi para pemrogram.

Paten perangkat Lunak

Ancaman terburuk yang perlu dihadapi berasal dari paten perangkat lunak, yang dapat berakibat pembatasan fitur perangkat lunak bebas lebih dari dua puluh tahun. Paten algoritma kompresi LZW diterapkan 1983, serta hingga baru-baru ini, kita tidak dapat membuat perangkat lunak bebas untuk kompresi GIF. Tahun 1998 yang lalu, sebuah program bebas yang menghasilkan suara MP3 terkompresi terpaksa dihapus dari distro akibat ancaman penuntutan paten.

Dokumentasi Bebas

Perangkat lunak bebas seharusnya dilengkapi dengan dokumentasi bebas pula. Sayang sekali, dewasa ini, dokumentasi bebas merupakan masalah yang paling serius yang dihadapi oleh masyarakat perangkat lunak bebas.

2.6. Rangkuman

Arti bebas yang salah, telah menimbulkan persepsi masyarakat bahwa perangkat lunak bebas merupakan perangkat lunak yang gratis. Perangkat lunak bebas ialah perihal kebebasan, bukan

harga. Konsep kebebasan yang dapat diambil dari kata bebas pada perangkat lunak bebas adalah seperti kebebasan berbicara bukan seperti bir gratis. Maksud dari bebas seperti kebebasan berbicara adalah kebebasan untuk menggunakan, menyalin, menyebarluaskan, mempelajari, mengubah, dan meningkatkan kinerja perangkat lunak.

Suatu perangkat lunak dapat dimasukkan dalam kategori perangkat lunak bebas bila setiap orang memiliki kebebasan tersebut. Hal ini berarti, setiap pengguna perangkat lunak bebas dapat meminjamkan perangkat lunak yang dimilikinya kepada orang lain untuk dipergunakan tanpa perlu melanggar hukum dan disebut pembajak. Kebebasan yang diberikan perangkat lunak bebas dijamin oleh *copyleft*, suatu cara yang dijamin oleh hukum untuk melindungi kebebasan para pengguna perangkat lunak bebas. Dengan adanya *copyleft* maka suatu perangkat lunak bebas beserta hasil perubahan dari kode sumbernya akan selalu menjadi perangkat lunak bebas. Kebebasan yang diberikan melalui perlindungan *copyleft* inilah yang membuat suatu program dapat menjadi perangkat lunak bebas.

Keuntungan yang diperoleh dari penggunaan perangkat lunak bebas adalah karena serbaguna dan efektif dalam keanekaragaman jenis aplikasi. Dengan pemberian kode-sumber-nya, perangkat lunak bebas dapat disesuaikan secara khusus untuk kebutuhan pemakai. Sesuatu yang tidak mudah untuk terselesaikan dengan perangkat lunak berpemilik. Selain itu, perangkat lunak bebas didukung oleh milis-milis pengguna yang dapat menjawab pertanyaan yang timbul karena permasalahan pada penggunaan perangkat lunak bebas.

Rujukan

[UU2000030] RI . 2000. *Undang-Undang Nomor 30 Tahun 2000 Tentang Rahasia Dagang.* .

[UU2000031] RI. 2000. *Undang-Undang Nomor 31 Tahun 2000 Tentang Desain Industri*.

[UU2000032] RI. 2000. *Undang-Undang Nomor 32 Tahun 2000 Tentang Desain Tata Letak Sirkuit Terpadu*.

[UU2001014] RI. 2001. *Undang-Undang Nomor 14 Tahun 2001 Tentang Paten*.

[UU2001015] RI. 2001. *Undang-Undang Nomor 15 Tahun 2001 Tentang Merek*.

[UU2002019] RI. 2002. *Undang-Undang Nomor 19 Tahun 2002 Tentang Hak Cipta*.

[WEBFSF1991a] Free Software Foundation. 1991. *GNU General Public License – http://gnui.vLSM.org/ licenses/ gpl.txt* . Diakses 29 Mei 2006.

[WEBFSF2001a] Free Software Foundation. 2001. *Definisi Perangkat Lunak Bebas – http://gnui.vlsm.org/ philosophy/ free-sw.id.html* . Diakses 29 Mei 2006.

[WEBFSF2001b] Free Software Foundation. 2001. *Frequently Asked Questions about the GNU GPL – http://gnui.vlsm.org/licenses/gpl-faq.html* . Diakses 29 Mei 2006.

[WEBHuham2005] Departemen Hukum dan Hak Asasi Manusia Republik Indonesia. 2005. *Kekayaan Intelektual – http://www.dgip.go.id/ article/ archive/ 2* . Diakses 29 Mei 2006.

[WEBRamelan1996] Rahardi Ramelan. 1996. *Hak Atas Kekayaan Intelektual Dalam Era Globalisasi http://leapidea.com/ presentation?id=6* . Diakses 29 Mei 2006.

[WEBSamik2003a] Rahmat M Samik-Ibrahim. 2003. *Pengenalan Lisensi Perangkat Lunak Bebas – http://rms46.vlsm.org/ 1/ 70.pdf* . vLSM.org. Pamulang. Diakses 29 Mei 2006.

[WEBStallman1994a] Richard M Stallman. 1994. *Mengapa Perangkat Lunak Seharusnya Tanpa Pemilik – http://gnui.vlsm.org/ philosophy/ why-free.id.html* . Diakses 29 Mei 2006.

[WEBWiki2005a] From Wikipedia, the free encyclopedia. 2005. *Intellectual property – http://en.wikipedia.org/ wiki/ Intellectual_property* . Diakses 29 Mei 2006.

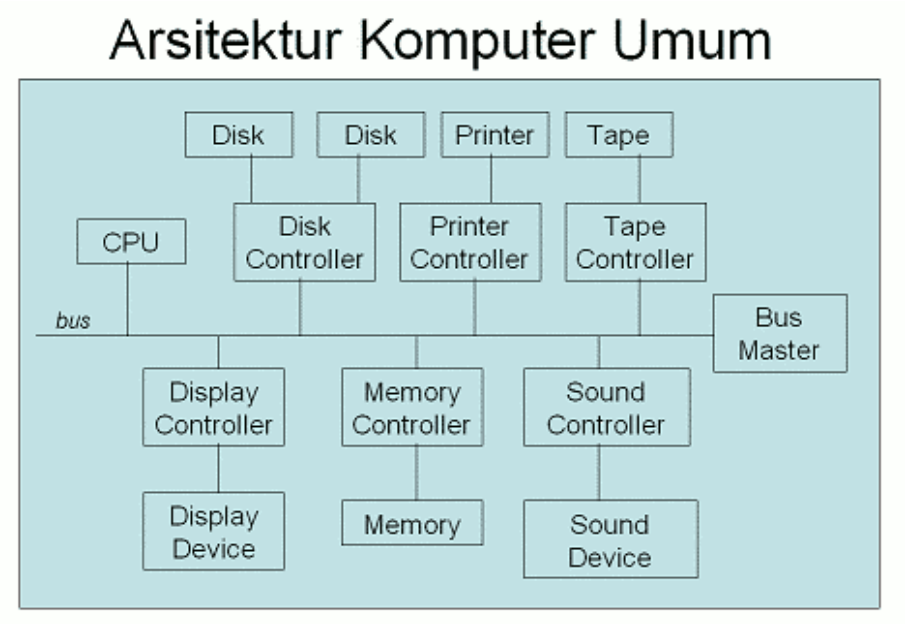
[WEBWIPO2005] World Intellectual Property Organization. 2005. *About Intellectual Property – http://www.wipo.int/ about-ip/ en/* . Diakses 29 Mei 2006.

Bab 3. Perangkat Keras Komputer

3.1. Pendahuluan

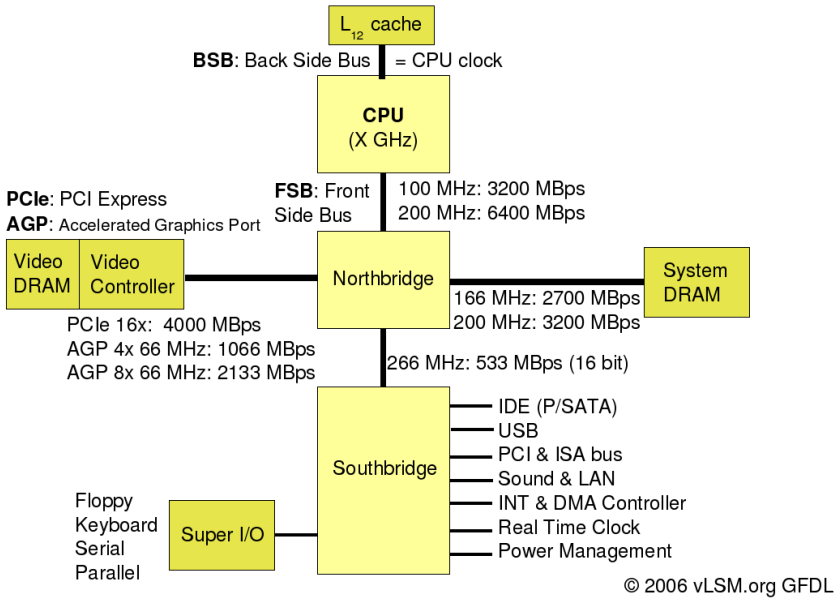
Tidak ada suatu ketentuan khusus tentang bagaimana seharusnya struktur sistem sebuah komputer. Para ahli serta perancang arsitektur komputer memiliki pandangannya masing-masing. Akan tetapi, untuk mempermudah pemahaman rincian dari sistem operasi di bab-bab berikutnya, kita perlu memiliki pengetahuan umum tentang struktur sistem komputer.

Gambar 3.1. Arsitektur Umum Komputer



GPU = Graphics Processing Unit;
AGP = Accelerated Graphics Port;
HDD = Hard Disk Drive;
FDD = Floppy Disk Drive;
FSB = Front Side Bus;
USB = Universal Serial Bus;
PCI = Peripheral Component Interconnect;
RTC = Real Time Clock;
PATA = Pararel Advanced Technology Attachment;
SATA = Serial Advanced Technology Attachment;
ISA = Industry Standard Architecture;
IDE = Intelligent Drive Electronics/Integrated Drive Electronics;
MCA = Micro Channel Architecture;
PS/2 = Sebuah *port* yang dibangun IBM untuk menghubungkan mouse ke *PC*;

Gambar 3.2. Arsitektur PC Modern



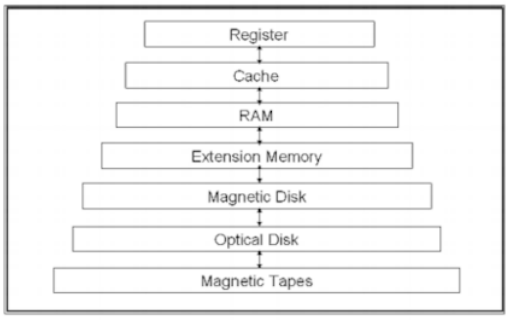
3.2. Prosesor

Secara umum, sistem komputer terdiri atas CPU dan sejumlah perangkat pengendali yang terhubung melalui sebuah *bus* yang menyediakan akses ke memori. Umumnya, setiap *device controller* bertanggung-jawab atas sebuah hardware spesifik. Setiap *device* dan CPU dapat beroperasi secara konkuren untuk mendapatkan akses ke memori. Adanya beberapa *hardware* ini dapat menyebabkan masalah sinkronisasi. Karena itu untuk mencegahnya sebuah *memory controller* ditambahkan untuk sinkronisasi akses memori.

3.3. Media Penyimpanan Utama

Dasar susunan media penyimpanan ialah kecepatan, biaya, sifat volatilitas. *Caching* menyalin informasi ke media penyimpanan yang lebih cepat; Memori utama dapat dilihat sebagai cache terakhir untuk media penyimpanan sekunder. Menggunakan memori berkecepatan tinggi untuk memegang data yang diakses terakhir. Dibutuhkan *cache management policy*. *Cache* juga memperkenalkan tingkat lain di hirarki penyimpanan. Hal ini memerlukan data untuk disimpan bersama-sama di lebih dari satu level agar tetap konsisten.

Gambar 3.3. Penyimpanan Hirarkis



Register

Tempat penyimpanan beberapa buah data *volatile* yang akan diolah langsung di prosesor yang berkecepatan sangat tinggi. Register ini berada di dalam prosesor dengan jumlah yang sangat terbatas karena fungsinya sebagai tempat perhitungan/komputasi data.

Cache Memory

Tempat penyimpanan sementara (*volatile*) sejumlah kecil data untuk meningkatkan kecepatan pengambilan atau penyimpanan data di memori oleh prosesor yang berkecepatan tinggi. Dahulu *cache* disimpan di luar prosesor dan dapat ditambahkan. Misalnya *pipeline burst* cache yang biasa ada di komputer awal tahun 90-an. Akan tetapi seiring menurunnya biaya produksi *die* atau *wafer* dan untuk meningkatkan kinerja, *cache* ditanamkan di prosesor. Memori ini biasanya dibuat berdasarkan desain memori statik.

Random Access Memory

Tempat penyimpanan sementara sejumlah data *volatile* yang dapat diakses langsung oleh prosesor. Pengertian langsung di sini berarti prosesor dapat mengetahui alamat data yang ada di memori secara langsung. Sekarang, *RAM* dapat diperoleh dengan harga yang cukup murah dangan kinerja yang bahkan dapat melewati *cache* pada komputer yang lebih lama.

Memori Ekstensi

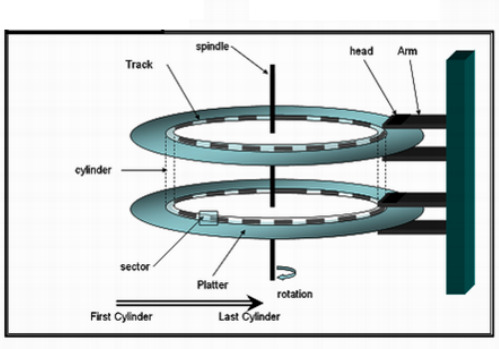
Tambahan memori yang digunakan untuk membantu proses-proses dalam komputer, biasanya berupa buffer. Peranan tambahan memori ini sering dilupakan akan tetapi sangat penting artinya untuk efisiensi. Biasanya tambahan memori ini memberi gambaran kasar kemampuan dari perangkat tersebut, sebagai contoh misalnya jumlah memori VGA, memori *soundcard*.

Direct Memory Access

Perangkat DMA digunakan agar perangkat M/K (*I/O device*) yang dapat memindahkan data dengan kecepatan tinggi (mendekati frekuensi bus memori). Perangkat pengendali memindahkan data dalam blok-blok dari buffer langsung ke memory utama atau sebaliknya tanpa campur tangan prosesor. Interupsi hanya terjadi tiap blok bukan tiap word atau byte data. Seluruh proses DMA dikendalikan oleh sebuah controller bernama *DMA Controller (DMAC). DMA Controller* mengirimkan atau menerima signal dari memori dan *I/O device.* Prosesor hanya mengirimkan alamat awal data, tujuan data, panjang data ke pengendali DMA. Interupsi pada prosesor hanya terjadi saat proses transfer selesai. Hak terhadap penggunaan *bus memory* yang diperlukan pengendali DMA didapatkan dengan bantuan *bus arbiter* yang dalam PC sekarang berupa *chipset Northbridge.*

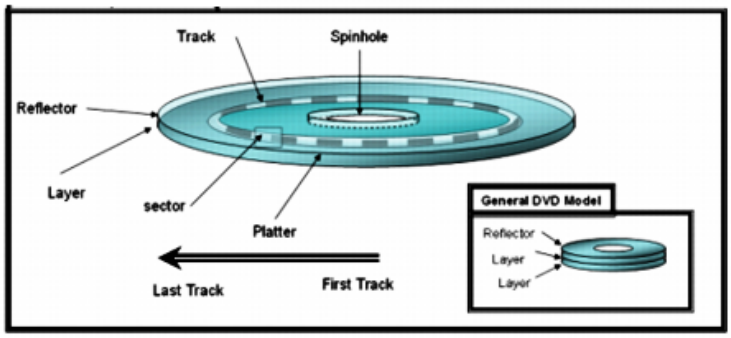
3.4. Penyimpanan Sekunder

Gambar 3.4. Struktur *Harddisk*



Media penyimpanan data yang non-volatile yang dapat berupa *Flash Drive*, *Optical Disc*, *Magnetic Disk*, *Magnetic Tape*. Media ini biasanya daya tampungnya cukup besar dengan harga yang relatif murah. *Portability*-nya juga relatif lebih tinggi.

Gambar 3.5. Struktur *Optical Drive*



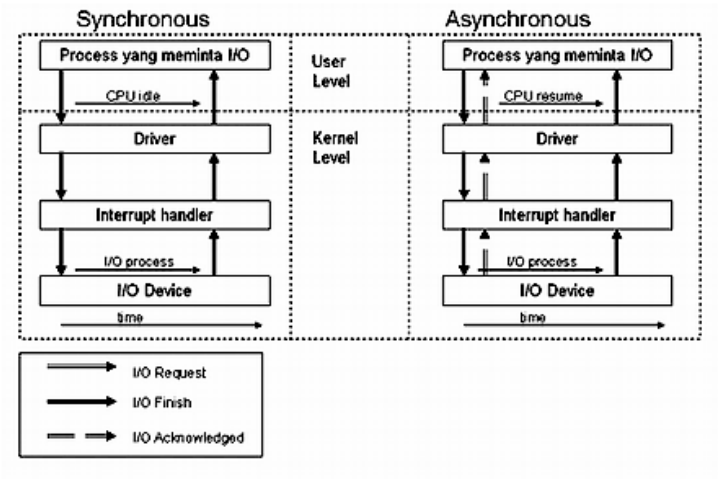
3.5. Memori Tersier

Pada standar arsitektur sequential komputer ada tiga tingkatan utama penyimpanan: primer, sekunder, and tersier. Memori tersier menyimpan data dalam jumlah yang besar (terabytes, atau 10^{12} bytes), tapi waktu yang dibutuhkan untuk mengakses data biasanya dalam hitungan menit sampai jam. Saat ini, memori tersiser membutuhkan instalasi yang besar berdasarkan/bergantung pada disk atau tapes. Memori tersier tidak butuh banyak operasi menulis tapi memori tersier tipikal-nya write ones atau read many. Meskipun per-megabites-nya pada harga terendah, memory tersier umumnya yang paling mahal, elemen tunggal pada modern supercomputer installations.

Ciri-ciri lain: non-volatile, penyimpanan off-line , umumnya dibangun pada removable media contoh optical disk, flash memory.

3.6. Struktur Keluaran/Masukan (M/K)

Gambar 3.6. Struktur M/K



Ada dua macam tindakan jika ada operasi *M/K*. Kedua macam tindakan itu adalah:

i. Setelah proses M/K dimulai, kendali akan kembali ke user program saat proses M/K selesai (*Synchronous*). Instruksi wait menyebabkan CPU idle sampai interupsi berikutnya. Akan terjadi *Wait loop* (untuk menunggu akses berikutnya). Paling banyak satu proses M/K yang berjalan dalam satu waktu.
ii. Setelah proses M/K dimulai, kendali akan kembali ke user program tanpa menunggu proses M/K selesai (*Asynchronous*). *System call* permintaan pada sistem operasi untuk mengizinkan user menunggu sampai M/K selesai. *Device-status table* mengandung data masukkan untuk tiap M/K device yang menjelaskan tipe, alamat, dan keadaannya. Sistem operasi memeriksa M/K device untuk mengetahui keadaan device dan mengubah tabel untuk memasukkan interupsi. Jika M/K device mengirim/mengambil data ke/dari memori hal ini dikenal dengan nama *Direct Memory Access* (DMA).

3.7. Bus

Pada sistem komputer yang lebih maju, arsitekturnya lebih kompleks. Untuk meningkatkan kinerja, digunakan beberapa buah *bus*. Tiap *bus* merupakan jalur data antara beberapa *device* yang berbeda. Dengan cara ini *RAM, Prosesor, GPU (VGA AGP)* dihubungkan oleh *bus* utama berkecepatan tinggi yang lebih dikenal dengan nama *FSB (Front Side Bus)*. Sementara perangkat lain yang lebih lambat dihubungkan oleh *bus* yang berkecepatan lebih rendah yang terhubung dengan *bus* lain yang lebih cepat sampai ke bus utama. Untuk komunikasi antar bus ini digunakan sebuah *bridge*.

Tanggung-jawab sinkronisasi *bus* yang secara tak langsung juga mempengaruhi sinkronisasi memori dilakukan oleh sebuah *bus controller* atau dikenal sebagai *bus master*. *Bus master* akan mengendalikan aliran data hingga pada satu waktu, bus hanya berisi data dari satu buah *device*. Pada prakteknya *bridge* dan *bus master* ini disatukan dalam sebuah *chipset*.

Suatu jalur transfer data yang menghubungkan setiap *device* pada komputer. Hanya ada satu buah *device* yang boleh mengirimkan data melewati sebuah bus, akan tetapi boleh lebih dari satu *device* yang membaca data bus tersebut. Terdiri dari dua buah model: *Synchronous bus* di mana digunakan dengan bantuan clock tetapi berkecepatan tinggi, tapi hanya untuk device berkecepatan tinggi juga; *Asynchronous bus* digunakan dengan sistem *handshake* tetapi berkecepatan rendah, dapat digunakan untuk berbagai macam *device*.

3.8. Interupsi

Kejadian ini pada komputer modern biasanya ditandai dengan munculnya interupsi dari software atau hardware, sehingga Sistem Operasi ini disebut *Interrupt-driven. Interrupt* dari *hardware* biasanya dikirimkan melalui suatu signal tertentu, sedangkan *software* mengirim interupsi dengan cara menjalankan *system call* atau juga dikenal dengan istilah *monitor call. System/Monitor call* ini akan menyebabkan *trap* yaitu interupsi khusus yang dihasilkan oleh software karena adanya masalah atau permintaan terhadap layanan sistem operasi.

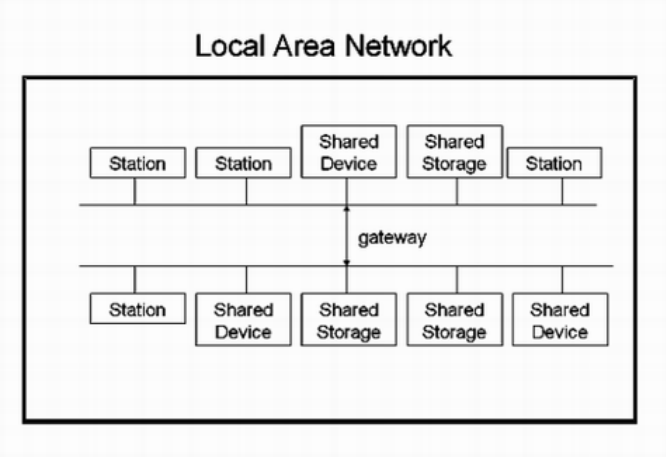
Trap ini juga sering disebut sebagai *exception*.

Setiap interupsi terjadi, sekumpulan kode yang dikenal sebagai *ISR (Interrupt Service Routine)* akan menentukan tindakan yang akan diambil. Untuk menentukan tindakan yang harus dilakukan, dapat dilakukan dengan dua cara yaitu *polling* yang membuat komputer memeriksa satu demi satu perangkat yang ada untuk menyelidiki sumber interupsi dan dengan cara menggunakan alamat-alamat *ISR* yang disimpan dalam array yang dikenal sebagai *interrupt vector* di mana sistem akan memeriksa *Interrupt Vector* setiap kali interupsi terjadi.

Arsitektur interupsi harus mampu untuk menyimpan alamat instruksi yang di-interupsi Pada komputer lama, alamat ini disimpan di tempat tertentu yang tetap, sedangkan pada komputer baru, alamat itu disimpan di *stack* bersama-sama dengan informasi state saat itu.

3.9. *Local Area Network*

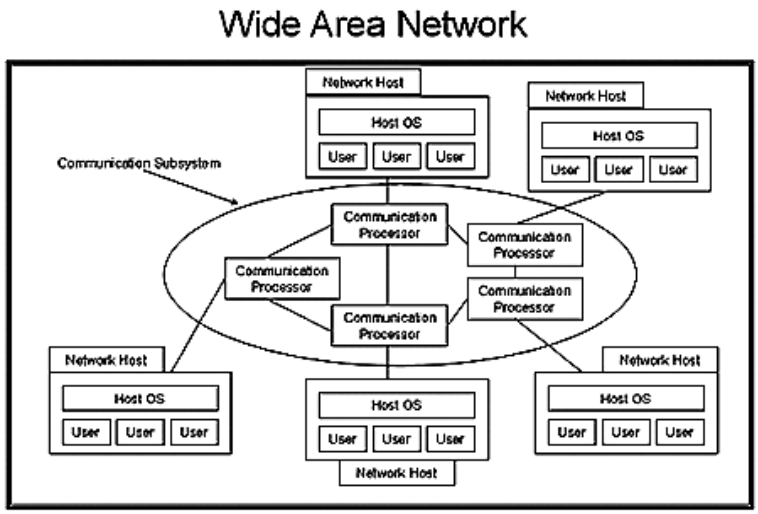
Gambar 3.7. ***Local Area Network***



Muncul untuk menggantikan komputer besar. Dirancang untuk melingkupi suatu daerah yang kecil. Menggunakan peralatan berkecepatan lebih tinggi daripada WAN. Hanya terdiri atas sejumlah kecil komputer.

3.10. *Wide Area Network*

Gambar 3.8. ***Wide Area Network***



Menghubungkan daerah yang lebih luas. Lebih lambat, dihubungkan oleh *router* melalui jaringan data telekomunikasi.

3.11. Rangkuman

Memori utama adalah satu-satunya tempat penyimpanan yang besar yang dapat diakses secara langsung oleh prosessor, merupakan suatu *array* dari *word* atau *byte*, yang mempunyai ukuran

ratusan sampai jutaan ribu. Setiap word memiliki alamatnya sendiri. Memori utama adalah tempat penyimpanan yang volatile, dimana isinya hilang bila sumber energinya (energi listrik) dimatikan. Kebanyakan sistem komputer menyediakan secondary penyimpanan sebagai perluasan dari memori utama. Syarat utama dari penyimpanan sekunder ialah dapat menyimpan data dalam jumlah besar secara permanen.

Media penyimpanan sekunder yang paling umum adalah disk magnetik, yang meyediakan penyimpanan untuk program maupun data. Disk magnetik adalah alat penyimpanan data yang *non-volatile* yang juga menyediakan akses secara random. Tape magnetik digunakan terutama untuk backup, penyimpanan informasi yang jarang digunakan, dan sebagai media pemindahan informasi dari satu sistem ke sistem yang lain.

Beragam sistem penyimpanan dalam sistem komputer dapat disusun dalam hirarki berdasarkan kecepatan dan biayanya. Tingkat yang paling atas adalah yang paling mahal, tapi cepat. Semakin kebawah, biaya perbit menurun, sedangkan waktu aksesnya semakin bertambah (semakin lambat).

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

Bab 4. Proteksi Perangkat Keras

4.1. Pendahuluan

Pada awalnya semua operasi pada sebuah sistem komputer ditangani oleh hanya seorang pengguna. Sehingga semua pengaturan terhadap perangkat keras maupun perangkat lunak dilakukan oleh pengguna tersebut. Namun seiring dengan berkembangnya Sistem Operasi pada sebuah sistem komputer, pengaturan ini pun diserahkan kepada Sistem Operasi tersebut. Segala macam manajemen sumber daya diatur oleh Sistem Operasi.

Pengaturan perangkat keras dan perangkat lunak ini berkaitan erat dengan proteksi dari perangkat keras maupun perangkat lunak itu sendiri. Sehingga, apabila dahulu segala macam proteksi terhadap perangkat keras dan perangkat lunak agar sistem dapat berjalan stabil dilakukan langsung oleh pengguna maka sekarang Sistem Operasi-lah yang banyak bertanggung jawab terhadap hal tersebut. Sistem Operasi harus dapat mengatur penggunaan segala macam sumber daya perangkat keras yang dibutuhkan oleh sistem agar tidak terjadi hal-hal yang tidak diinginkan. Seiring dengan maraknya berbagi sumberdaya yang terjadi pada sebuah sistem, maka Sistem Operasi harus dapat secara pintar mengatur mana yang harus didahulukan. Hal ini dikarenakan, apabila pengaturan ini tidak dapat berjalan lancar maka dapat dipastikan akan terjadi kegagalan proteksi perangkat keras.

Dengan hadirnya multiprogramming yang memungkinkan adanya utilisasi beberapa program di memori pada saat bersamaan, maka utilisasi dapat ditingkatkan dengan penggunaan sumberdaya secara bersamaan tersebut, akan tetapi di sisi lain akan menimbulkan masalah karena sebenarnya hanya ada satu program yang dapat berjalan pada satuan waktu yang sama. Akan banyak proses yang terpengaruh hanya akibat adanya gangguan pada satu program.

Sebagai contoh saja apabila sebuah harddisk menjadi sebuah sumberdaya yang dibutuhkan oleh berbagai macam program yang dijalankan, maka bisa-bisa terjadi kerusakan harddisk akibat suhu yang terlalu panas akibat terjadinya sebuah situasi kemacetan penggunaan sumber daya secara bersamaan akibat begitu banyak program yang mengirimkan request akan penggunaan harddisk tersebut.

Di sinilah proteksi perangkat keras berperan. Sistem Operasi yang baik harus menyediakan proteksi yang maksimal, sehingga apabila ada satu program yang tidak bekerja maka tidak akan menggangu kinerja Sistem Operasi tersebut maupun program-program yang sedang berjalan lainnya.

4.2. Proteksi Fisik

Proteksi fisik merupakan fungsi Sistem Operasi dalam menjaga, memproteksi fisik daripada sumberdaya (perangkat keras). Misal proteksi CUP dan proteksi hardisk. Contohnya adalah dalam kasus dual-mode operation (dibahas di sub-bab berikutnya).

4.3. Proteksi Media

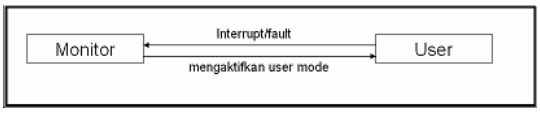
Dalam keseharian kita ada beberapa jenis media yang digunakan untuk penyimpanan data, antara lain tape, disket, CD, USB flash disk, dan lainnya. Untuk menjamin keamanan data yang tersimpan dalam media-media tersebut, maka perlu sebuah mekanisme untuk menanganinya. Mekanisme proteksi antara satu media dengan media yang lain tidak sama. Umpamanya, media disket menggunakan katup yang dapat di geser. Jika katupnya dibuka maka disket bisa ditulis dan di baca, jika ditutup maka disket hanya bisa dibaca tetapi tidak bisa ditulis.

4.4. Konsep Mode Operasi Ganda

Membagi sumber daya sistem yang memerlukan Sistem Operasi untuk menjamin bahwa program yang salah tidak menyebabkan program lain berjalan salah juga. Menyediakan dukungan perangkat keras untuk membedakan minimal dua mode operasi yaitu: *User Mode* - Eksekusi dikendalikan oleh pengguna; *Monitor/Kernel/System Mode* - Eksekusi dikendalikan oleh Sistem Operasi. Instruksi

tertentu hanya berjalan di mode ini (*Privileged Instruction*). Ditambahkan sebuah bit penanda operasi. Jika terjadi *interrupt*, maka perangkat keras berpindah ke *monitor mode* (*Dual Mode Operation*).

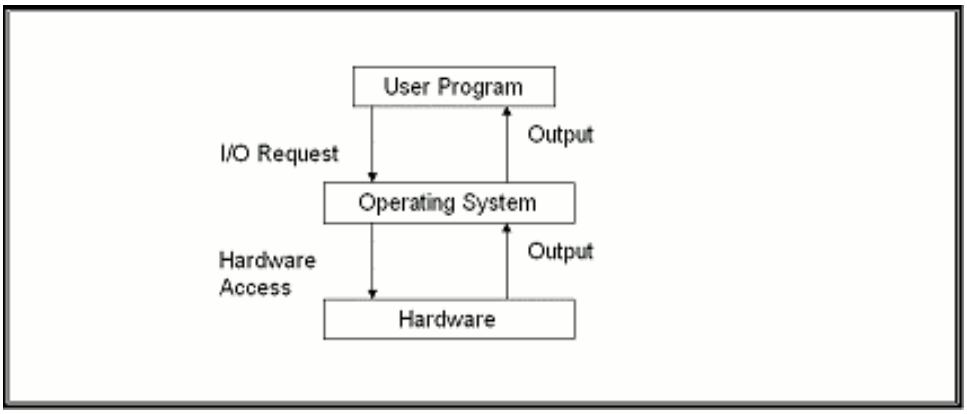
Gambar 4.1. *Dual Mode Operation*



4.5. Proteksi Masukan/Keluaran

Semua instruksi masukan/keluaran umumnya *Privileged Instruction* (kecuali pada DOS, dan program tertentu). Harus menjamin pengguna program tidak dapat mengambil alih kontrol komputer di *monitor mode*.

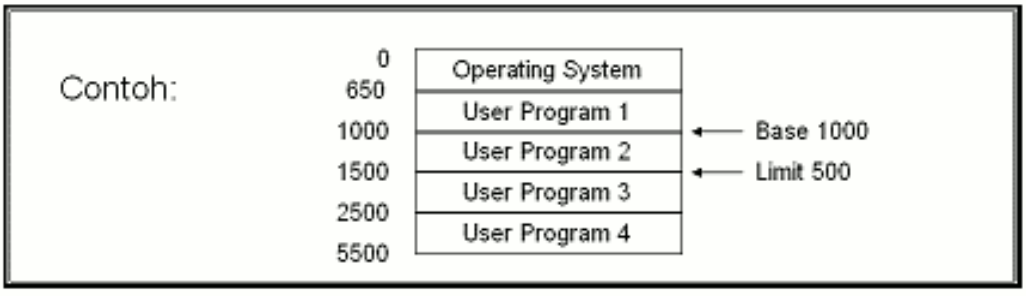
Gambar 4.2. Proteksi M/K



4.6. Proteksi Memori

Harus menyediakan perlindungan terhadap memori minimal untuk *interrupt vector* dan *interrupt service routine*. Ditambahkan dua register yang menentukan di mana alamat legal sebuah program boleh mengakses, yaitu *base register* untuk menyimpan alamat awal yang legal dan *limit register* untuk menyimpan ukuran memori yang boleh diakses Memori di luar jangkauan dilindungi.

Gambar 4.3. *Memory Protection*



4.7. Proteksi CPU

Timer melakukan *interrupt* setelah perioda waktu tertentu untuk menjamin kendali Sistem Operasi. *Timer* diturunkan setiap clock. Ketika timer mencapai nol, sebuah Interrupt terjadi. Timer biasanya digunakan untuk mengimplementasikan pembagian waktu. Timer dapat juga digunakan untuk menghitung waktu sekarang walaupun fungsinya sekarang ini sudah digantikan *Real Time Clock (RTC). System Clock Timer* terpisah dari Pencacah Waktu. *Timer* sekarang secara perangkat keras lebih dikenal sebagai *System Timer*/*CPU Timer*. *Load Timer* juga *Privileged Instruction*.

4.8. Rangkuman

Sistem Operasi harus memastikan operasi yang benar dari sistem komputer. Untuk mencegah pengguna program mengganggu operasi yang berjalan dalam sistem, perangkat keras mempunyai dua mode: mode pengguna dan mode monitor. Beberapa perintah (seperti perintah M/K dan perintah halt) adalah perintah khusus, dan hanya dapat dijalankan dalam mode monitor. Memori juga harus dilindungi dari modifikasi oleh pengguna. Timer mencegah terjadinya pengulangan secara terus menerus (infinite loop). Hal-hal tersebut (dual mode, perintah khusus, pengaman memori, timer interrupt) adalah blok bangunan dasar yang digunakan oleh Sistem Operasi untuk mencapai operasi yang sesuai.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

Bagian II. Konsep Dasar Sistem Operasi

Para pembaca sepertinya pernah mendengar istilah "Sistem Operasi". Mungkin pula pernah berhubungan secara langsung atau pun tidak langsung dengan istilah tersebut. Namun, belum tentu dapat menjabarkan perihal apa yang sebetulnya dimaksud dengan kata "Sistem Operasi". Bagian ini akan mengungkapan secara singkat dan padat, apa yang dimaksud dengan "Sistem Operasi".

Bab 5. Komponen Sistem Operasi

5.1. Pendahuluan

Sebuah sistem operasi dapat dibagi menjadi beberapa komponen. Secara umum, para pakar sepakat bahwa terdapat sekurangnya empat komponen manajeman utama yaitu:
- Manajemen Proses,
- Manajemen Memori, dan
- Manajamen Sistem Berkas.
- Manajemen Masukan/Keluaran

Selain keempat komponen di atas, Avi Silberschatz, dan kawan-kawan menambahkan beberapa komponen seperti:
- Manajemen Penyimpanan Sekunder.
- Manajemen Sistem Proteksi.
- Manajemen Jaringan.
- *Command-Interpreter System*.

5.2. Manajemen Proses

Proses adalah sebuah program yang sedang dieksekusi. Sebuah proses membutuhkan beberapa sumber daya untuk menyelesaikan tugasnya. Alokasi sumber daya tersebut dikelola oleh Sistem Operasi. Umpamanya, walku *CPU*, memori, berkas-berkas, dan perangkat-perangkat M/K. Ketika proses tersebut berhenti dijalankan, sistem operasi akan mendapatkan kembali semua sumber daya yang bisa digunakan kembali.

Sistem operasi bertanggung-jawab atas aktivitas-aktivitas yang berkaitan dengan manajemen proses seperti:
- Membuat dan menghapus proses pengguna dan sistem proses.
- Menunda atau melanjutkan proses.
- Menyediakan mekanisme untuk sinkronisasi proses.
- Menyediakan mekanisme untuk komunikasi proses.
- Menyediakan mekanisme untuk penanganan *deadlock*.

5.3. Manajemen Memori Utama

Memori utama atau lebih dikenal sebagai memori adalah sebuah *array* yang besar dari *word* atau *byte*, yang ukurannya mencapai ratusan, ribuan, atau bahkan jutaan. Setiap *word* atau byte mempunyai alamat tersendiri. Memori utama berfungsi sebagai tempat penyimpanan instruksi/data yang akses datanya digunakan oleh CPU dan perangkat Masukan/Keluaran. Memori utama termasuk tempat penyimpanan data yang yang bersifat volatile -- tidak permanen -- yaitu data akan hilang kalau komputer dimatikan.

Sistem operasi bertanggung-jawab atas aktivitas-aktivitas yang berkaitan dengan manajemen memori seperti:
- Menjaga *track* dari memori yang sedang digunakan dan siapa yang menggunakannya.
- Memilih program yang akan di-*load* ke memori.

5.4. Manajemen Sistem Berkas

Berkas adalah kumpulan informasi yang berhubungan, sesuai dengan tujuan pembuat berkas tersebut. Umumnya berkas merepresentasikan program dan data. Berkas dapat mempunyai struktur yang bersifat hirarkis (direktori, volume, dll.). Sistem operasi mengimplementasikan konsep abstrak dari berkas dengan mengatur media penyimpanan massa, misalnya *tapes* dan *disk*.

Sistem operasi bertanggung-jawab dalam aktivitas yang berhubungan dengan manajemen berkas:
- Pembuatan dan penghapusan berkas.
- Pembuatan dan penghapusan direktori.

- Mendukung manipulasi berkas dan direktori.
- Memetakan berkas ke *secondary-storage*.
- Mem-*back-up* berkas ke media penyimpanan yang permanen (*non-volatile*).

5.5. Manajemen Sistem Masukan/Keluaran

Sistem ini sering disebut dengan *device manager*. Menyediakan *device driver* yang umum sehingga operasi Masukan/Keluaran dapat seragam (membuka, membaca, menulis, menutup). Contoh: pengguna menggunakan operasi yang sama untuk membaca berkas pada perangkat keras, *CD-ROM* dan *floppy disk.*

Komponen Sistem Operasi untuk sistem Masukan/Keluaran:
- Penyangga: menampung sementara data dari/ke perangkat Masukan/Keluaran.
- *Spooling*: melakukan penjadwalan pemakaian Masukan/Keluaran sistem supaya lebih efisien (antrian dsb.).
- Menyediakan *driver*: untuk dapat melakukan operasi rinci untuk perangkat keras Masukan/Keluaran tertentu.

5.6. Manajemen Penyimpanan Sekunder

Data yang disimpan dalam memori utama bersifat sementara dan jumlahnya sangat kecil. Oleh karena itu, untuk menyimpan keseluruhan data dan program komputer dibutuhkan penyimpanan sekunder yang bersifat permanen dan mampu menampung banyak data, sebagai *back-up* dari memori utama. Contoh dari penyimpanan sekunder adalah *hard-disk*, disket, dll.

Sistem operasi bertanggung-jawab atas aktivitas-aktivitas yang berkaitan dengan manajemen disk seperti:
- *free space management*.
- alokasi penyimpanan.
- penjadwalan disk.

5.7. Sistem Proteksi

Proteksi mengacu pada mekanisme untuk mengontrol akses yang dilakukan oleh program, prosesor, atau pengguna ke sistem sumber daya. Mekanisme proteksi harus:

- Membedakan antara penggunaan yang sudah diberi izin dan yang belum.
- Menspesifikasi kontrol untuk dibebankan/diberi tugas.
- Menyediakan alat untuk pemberlakuan sistem.

5.8. Jaringan

Sistem terdistribusi adalah sekumpulan prosesor yang tidak berbagi memori, atau *clock*. Setiap prosesor mempunyai memori dan clock tersendiri. Prosesor-prosesor tersebut terhubung melalui jaringan komunikasi Sistem terdistribusi menyediakan akses pengguna ke bermacam sumber-daya sistem. Akses tersebut menyebabkan peningkatan kecepatan komputasi dan meningkatkan kemampuan penyediaan data.

5.9. *Command-Interpreter System*

Sistem Operasi menunggu instruksi dari pengguna (*command driven*). Program yang membaca instruksi dan mengartikan control statements umumnya disebut: *control-card interpreter*, *command-line interpreter* dan terkadang dikenal sebagai *shell*. *Command-Interpreter System* sangat bervariasi dari satu sistem operasi ke sistem operasi yang lain dan disesuaikan dengan tujuan dan teknologi perangkat Masukan/Keluaran yang ada. Contohnya: *CLI*, *Windows*, *Pen-based* (*touch*),

dan lain-lain.

5.10. Rangkuman

Pada umumnya, komponen sistem operasi terdiri dari manajemen proses, manajemen memori utama, manajemen berkas, manajemen sistem M/K, manajemen penyimpanan sekunder, sistem proteksi, jaringan dan *Command-Interpreter System*.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

Bab 6. Sudut Pandang Alternatif

6.1. Pendahuluan

Layanan sistem operasi dirancang untuk membuat pemrograman menjadi lebih mudah.

1. **Pembuatan Program.** Sistim operasi menyediakan berbagai fasilitas yang membantu programer dalam membuat program seperti editor. Walaupun bukan bagian dari sistim operasi, tapi layanan ini diakses melalui sistim operasi.
2. **Eksekusi Program.** Sistem harus bisa me-*load* program ke memori, dan menjalankan program tersebut. Program harus bisa menghentikan pengeksekusiannya baik secara normal maupun tidak (ada *error*).
3. **Operasi Masukan/Keluaran.** Program yang sedang dijalankan kadang kala membutuhkan Masukan/Keluaran. Untuk efisiensi dan keamanan, pengguna biasanya tidak bisa mengatur peranti Masukan/Keluaran secara langsung, untuk itulah sistem operasi harus menyediakan mekanisme dalam melakukan operasi Masukan/Keluaran.
4. **Manipulasi Sistem Berkas.** Program harus membaca dan menulis berkas, dan kadang kala juga harus membuat dan menghapus berkas.
5. **Komunikasi.** Kadang kala sebuah proses memerlukan informasi dari proses yang lain. Ada dua cara umum dimana komunikasi dapat dilakukan. Komunikasi dapat terjadi antara proses dalam satu komputer, atau antara proses yang berada dalam komputer yang berbeda, tetapi dihubungkan oleh jaringan komputer. Komunikasi dapat dilakukan dengan *share-memory* atau *message-passing*, dimana sejumlah informasi dipindahkan antara proses oleh sistem operasi.
6. **Deteksi *Error*.** Sistem operasi harus selalu waspada terhadap kemungkinan *error*. *Error* dapat terjadi di CPU dan memori perangkat keras, Masukan/Keluaran, dan di dalam program yang dijalankan pengguna. Untuk setiap jenis *error* sistem operasi harus bisa mengambil langkah yang tepat untuk mempertahankan jalannya proses komputasi. Misalnya dengan menghentikan jalannya program, mencoba kembali melakukan operasi yang dijalankan, atau melaporkan kesalahan yang terjadi agar pengguna dapat mengambil langkah selanjutnya.

Disamping pelayanan di atas, sistem operasi juga menyediakan layanan lain. Layanan ini bukan untuk membantu pengguna tapi lebih pada mempertahankan efisiensi sistem itu sendiri. Layanan tambahan itu yaitu:

1. **Alokasi Sumber Daya.** Ketika beberapa pengguna menggunakan sistem atau beberapa program dijalankan secara bersamaan, sumber daya harus dialokasikan bagi masing-masing pengguna dan program tersebut.

2. ***Accounting*.** Kita menginginkan agar jumlah pengguna yang menggunakan sumber daya, dan jenis sumber daya yang digunakan selalu terjaga. Untuk itu maka diperlukan suatu perhitungan dan statistik. Perhitungan ini diperlukan bagi seseorang yang ingin merubah konfigurasi sistem untuk meningkatkan pelayanan.

3. **Proteksi.** Layanan proteksi memastikan bahwa segala akses ke sumber daya terkontrol. Dan tentu saja keamanan terhadap gangguan dari luar sistem tersebut. Keamanan bisa saja dilakukan dengan terlebih dahulu mengidentifikasi pengguna. Ini bisa dilakukan dengan meminta *password* bila ingin menggunakan sumber daya.

6.2. *System Program*

System program menyediakan lingkungan yang memungkinkan pengembangan program dan eksekusi berjalan dengan baik. Dapat dikategorikan:

- **Manajemen/manipulasi berkas.** Membuat, menghapus, *copy*, *rename*, *print*, memanipulasi berkas dan direktori.
- **Informasi status.** Beberapa program meminta informasi tentang tanggal, jam, jumlah memori dan disk yang tersedia, jumlah pengguna dan informasi lain yang sejenis.
- **Modifikasi berkas.** Membuat berkas dan memodifikasi isi berkas yang disimpan pada disk atau tape.

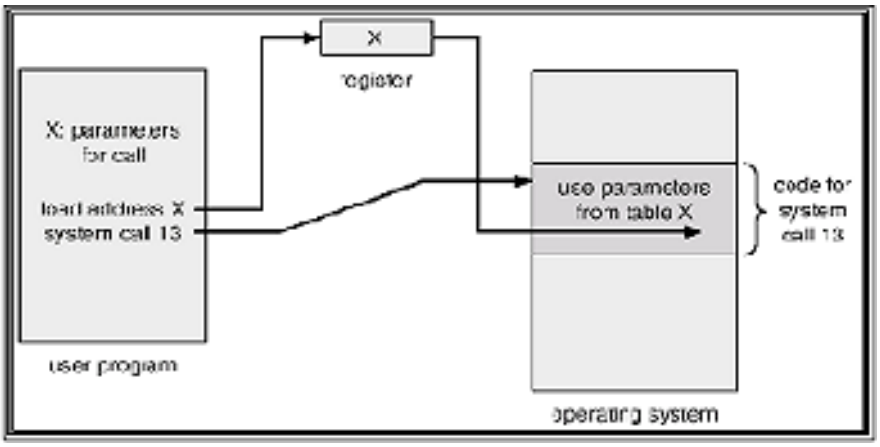
- **Pendukung bahasa pemrograman.** Kadang kala kompilator, *assembler*, *interpreter* dari bahasa pemrograman diberikan kepada pengguna dengan bantuan sistem operasi.
- ***Loading* dan eksekusi program.** Ketika program di-*assembly* atau dikompilasi, program tersebut harus di-*load* ke dalam memori untuk dieksekusi. Untuk itu sistem harus menyediakan *absolute loaders*, *relocatable loaders*, *linkage editors*, dan *overlay loaders*.
- Komunikasi Menyediakan mekanisme komunikasi antara proses, pengguna, dan sistem komputer yang berbeda. Sehingga pengguna bisa mengirim pesan, *browse* web pages, mengirim e-mail, atau mentransfer berkas.

Umumnya sistem operasi dilengkapi oleh *system-utilities* atau program aplikasi yang di dalamnya termasuk *web browser*, *word prossesor* dan format teks, sistem database, *games*. *System program* yang paling penting adalah *command interpreter* yang mengambil dan menerjemahkan *user-specified command* selanjutnya.

6.3. *System Calls*

Biasanya tersedia sebagai instruksi bahasa *assembly*. Beberapa sistem mengizinkan *system calls* dibuat langsung dari program bahasa tingkat tinggi. Beberapa bahasa pemrograman (contoh: C, C++) telah didefenisikan untuk menggantikan bahasa *assembly* untuk sistem pemrograman.

Gambar 6.1. Memberikan parameter melalui tabel



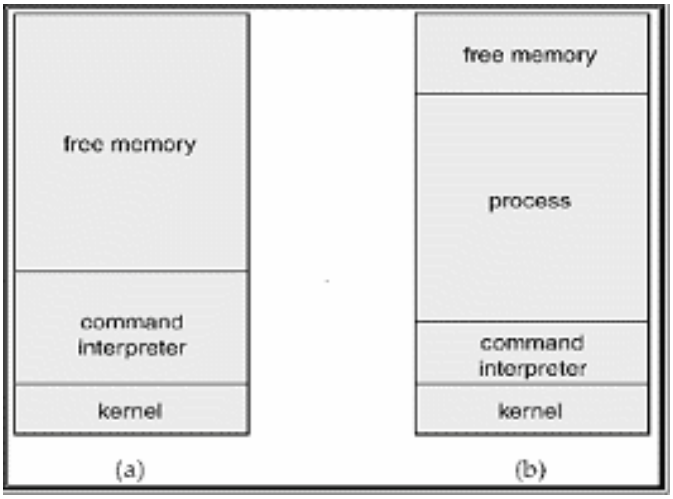
Tiga metoda umum yang digunakan dalam memberikan parameter kepada sistem operasi:
- Melalui *register*.
- Menyimpan parameter dalam *block* atau tabel pada memori dan alamat *block* tersebut diberikan sebagai parameter dalam *register*.
- Menyimpan parameter (*push*) ke dalam *stack* oleh program, dan melakukan *pop off* pada *stack* oleh sistem operasi.

Jenis *System Calls*

System calls yang berhubungan dengan kontrol proses antara lain ketika penghentian pengeksekusian program. Baik secara normal (*end*) maupun tidak normal (*abort*). Selama proses dieksekusi kadang kala diperlukan untuk me-*load* atau mengeksekusi program lain, disini diperlukan lagi suatu *system calls*. Juga ketika membuat suatu proses baru dan menghentikan sebuah proses. Ada juga *system calls* yang dipanggil ketika kita ingin meminta dan merubah atribut dari suatu proses.

MS-DOS adalah contoh dari sistem *single-tasking*. MS-DOS menggunakan metoda yang sederhana dalam menjalankan program aan tidak menciptakan proses baru. Program di-*load* ke dalam memori, kemudian program dijalankan. Berkeley Unix adalah contoh dari sistem *multi-tasking*. *Command Interpereter* masih tetap bisa dijalankan ketika program lain dieksekusi.

Gambar 6.2. Eksekusi MS-DOS



6.4. *System Calls* Manajemen Proses

System Call untuk manajemen proses diperlukan untuk mengatur proses-proses yang sedang berjalan. Kita dapat melihat penggunaan system calls untuk manajemen proses pada Sistem Operasi Unix. Contoh yang paling baik untuk melihat bagaimana system call bekerja untuk manajemen proses adalah Fork. Fork adalah satu satunya cara untuk membuat sebuah proses baru pada sistem Unix. Fork membuat duplikasi yang mirip dengan proses aslinya, termasuk file descriptor, register, dan lainnya.

Setelah perintah Fork, child akan mengeksekusi kode yang berbeda dengan parentnya. Bayangkan yang terjadi pada shell. Shell akan membaca command dari terminal, melakukan fork pada child, menunggu child untuk mengeksekusi command tersebut, dan membaca command lainnya ketika child terminate.

Untuk menunggu child selesai, parent akan mengeksekusi system call waitpid, yang hanya akan menunggu sampai child selesai. Proses child harus mengeksekusi command yang dimasukkan oleh user(pada kasus shell). Proses child melakukannya dengan menggunakan system call exec.

Dari ilustrasi tersebut kita dapat mengetahui bagaimana system call dipakai untuk manajemen proses. Kasus lainnya bukan hanya pada Fork, tetapi hampir setiap proses memerlukan system call untuk melakukan manajement proses.

6.5. *System Calls* Manajemen Berkas

System calls yang berhubungan dengan berkas sangat diperlukan. Seperti ketika kita ingin membuat atau menghapus suatu berkas. Atau ketika ingin membuka atau menutup suatu berkas yang telah ada, membaca berkas tersebut, dan menulis berkas itu. *System calls* juga diperlukan ketika kita ingin mengetahui atribut dari suatu berkas atau ketika kita juga ingin merubah atribut tersebut. Yang termasuk atribut berkas adalah nama berkas, jenis berkas, dan lain-lain.

Ada juga *system calls* yang menyediakan mekanisme lain yang berhubungan dengan direktori atau sistim berkas secara keseluruhan. Jadi bukan hanya berhubungan dengan satu spesifik berkas. Contohnya membuat atau menghapus suatu direktori, dan lain-lain.

6.6. *System Calls* Manajemen Peranti

Program yang sedang dijalankan kadang kala memerlukan tambahan sumber daya. Jika banyak pengguna yang menggunakan sistem, maka jika memerlukan tambahan sumber daya maka harus

meminta peranti terlebih dahulu. Dan setelah selesai penggunakannnya harus dilepaskan kembali. Ketika sebuah peranti telah diminta dan dialokasikan maka peranti tersebut bisa dibaca, ditulis, atau direposisi.

6.7. *System Calls* Informasi/Pemeliharaan

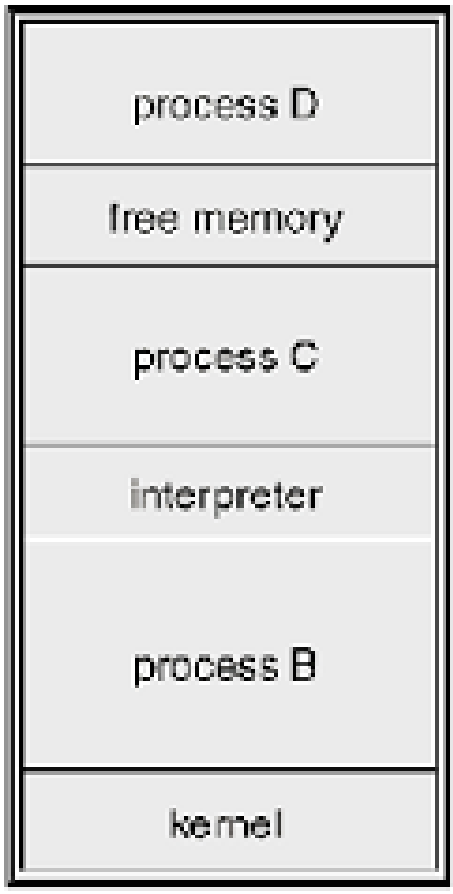
Beberapa *system calls* disediakan untuk membantu pertukaran informasi antara pengguna dan sistem operasi. Contohnya *system calls* untuk meminta dan mengatur waktu dan tanggal. Atau meminta informasi tentang sistem itu sendiri, seperti jumlah pengguna, jumlah memori dan disk yang masih bisa digunakan, dan lain-lain. Ada juga *system calls* untuk meminta informasi tentang proses yang disimpan oleh sistem dan *system calls* untuk merubah (*reset*) informasi tersebut.

6.8. *System Calls* Komunikasi

Dua model komunikasi:

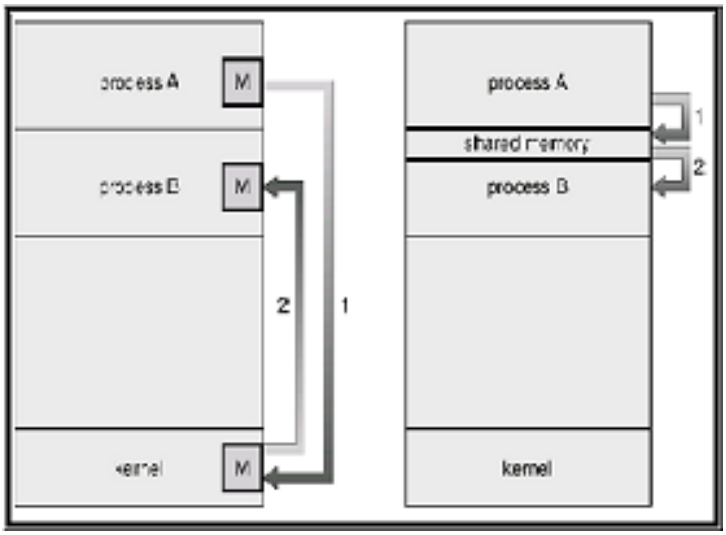
- ***Message-passing*.** Pertukaran informasi dilakukan melalui fasilitas komunikasi antar proses yang disediakan oleh sistem operasi.
- ***Shared-memory*.** Proses menggunakan memori yang bisa digunakan oleh berbagai proses untuk pertukaran informasi dengan membaca dan menulis data pada memori tersebut.

Gambar 6.3. Multi program pada Unix



Dalam *message-passing*, sebelum komunikasi dapat dilakukan harus dibangun dulu sebuah koneksi. Untuk itu diperlukan suatu *system calls* dalam pengaturan koneksi tersebut, baik dalam menghubungkan koneksi tersebut maupun dalam memutuskan koneksi tersebut ketika komunikasi sudah selesai dilakukan. Juga diperlukan suatu *system calls* untuk membaca dan menulis pesan (*message*) agar pertukaran informasi dapat dilakukan.

Gambar 6.4. Mekanisme komunikasi



6.9. Rangkuman

Layanan sistem operasi dirancang untuk membuat programming menjadi lebih mudah. Sistem operasi mempunyai lima layanan utama dan tiga layanan tambahan. *System calls* ada lima jenis. *System program* menyediakan *environment* yang memungkinkan pengembangan program dan eksekusi berjalan dengan baik.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

Bab 7. Struktur Sistem Operasi

7.1. Pendahuluan

Sebuah sistem yang besar dan kompleks seperti sistem operasi modern harus diatur dengan cara membagi *task* kedalam komponen-komponen kecil agar dapat berfungsi dengan baik dan mudah dimodifikasi. Pada bab ini, kita akan membahas cara komponen-komponen ini dihubungkan satu sama lain. Menurut Silberschatz dan kawan-kawan, ada tiga cara yaitu:

- Struktur Sederhana.
- Pendekatan Berlapis.
- Kernel Mikro.

Sedangkan menurut Stallings, kita bisa memandang sistem sebagai seperangkat lapisan. Tiap lapisan menampilkan bagian fungsi yang dibutuhkan oleh sistem operasi. Bagian yang terletak pada lapisan yang lebih rendah akan menmpilkan fungsi yang lebih primitif dan menyimpan detail fungsi tersebut.

7.2. Struktur Sederhana

Banyak sistem yang tidak terstruktur dengan baik, sehingga sistem operasi seperti ini dimulai dengan sistem yang lebih kecil, sederhana, dan terbatas. Kemudian berkembang dengan cakupan yang original. Contoh sistem seperti ini adalah MS-DOS, yang disusun untuk mendukung fungsi yang banyak pada ruang yang sedikit karena keterbatasan perangkat keras untuk menjalankannya.

Contoh sistem lainnya adalah UNIX, yang terdiri dari dua bagian yang terpisah, yaitu kernel dan program sistem. Kernel selanjutnya dibagi dua bagian, yaitu antarmuka dan *device drivers*. Kernel mendukung sistem berkas, penjadwalan CPU, manajemen memori, dan fungsi sistem operasi lainnya melalui *system calls*.

7.3. Pendekatan Berlapis

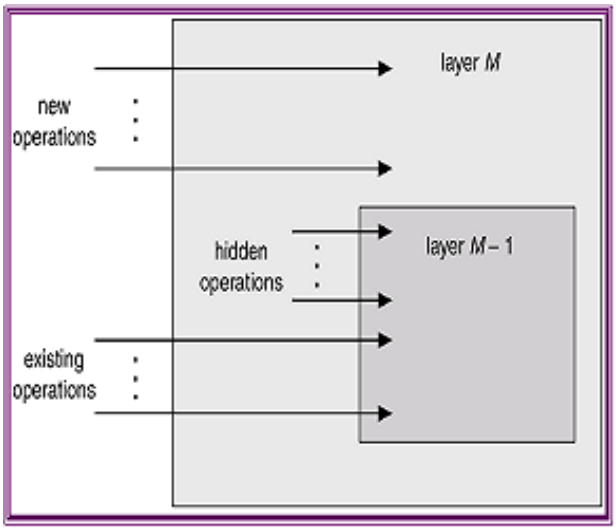
Sistem operasi dibagi menjadi sejumlah lapisan yang masing-masing dibangun di atas lapisan yang lebih rendah. Lapisan yang lebih rendah menyediakan layanan untuk lapisan yang lebih tinggi. Lapisan yang paling bawah adalah perangkat keras, dan yang paling tinggi adalah *user-interface*.

Sebuah lapisan adalah implementasi dari obyek abstrak yang merupakan enkapsulasi dari data dan operasi yang bisa memanipulasi data tersebut. Keuntungan utama dengan sistem ini adalah modularitas. Pendekatan ini mempermudah *debug* dan verifikasi sistem. Lapisan pertama bisa di *debug* tanpa mengganggu sistem yang lain karena hanya menggunakan perangkat keras dasar untuk implementasi fungsinya. Bila terjadi error saat *debugging* sejumlah lapisan, error pasti pada lapisan yang baru saja di *debug*, karena lapisan dibawahnya sudah di *debug.*

Sedangkan menurut Tanenbaum dan Woodhull, sistem terlapis terdiri dari enam lapisan, yaitu:

- **Lapisan 0.** Mengatur alokasi prosesor, pertukaran antar proses ketika interupsi terjadi atau waktu habis. Lapisan ini mendukung dasar *multi-programming* pada CPU.
- **Lapisan 1.** Mengalokasikan ruang untuk proses di memori utama dan pada 512 kilo word *drum* yang digunakan untuk menahan bagian proses ketika tidak ada ruang di memori utama.
- **Lapisan 2.** Menangani komunikasi antara masing-masing proses dan operator *console*. Pada lapis ini masing-masing proses secara efektif memiliki opertor *console* sendiri.
- **Lapisan 3.** Mengatur peranti M/K dan menampung informasi yang mengalir dari dan ke proses tersebut.
- **Lapisan 4.** Tempat program pengguna. Pengguna tidak perlu memikirkan tentang proses, memori, *console*, atau manajemen M/K.
- **Lapisan 5.** Merupakan operator sistem.

Gambar 7.1. Lapisan pada Sistem Operasi



Menurut Stallings, model tingkatan sistem operasi yang mengaplikasikan prinsip ini dapat dilihat pada tabel berikut, yang terdiri dari level-level dibawah ini:

- **Level 1.** Terdiri dari sirkuit elektronik dimana obyek yang ditangani adalah *register memory cell*, dan gerbang logika. Operasi pada obyek ini seperti membersihkan register atau membaca lokasi memori.
- **Level 2.** Pada level ini adalah set instruksi pada prosesor. Operasinya adalah instruksi bahasa-mesin, seperti menambah, mengurangi, *load* dan *store*.
- **Level 3.** Tambahan konsep prosedur atau subrutin ditambah operasi *call* atau *return*.
- **Level 4.** Mengenalkan interupsi yang menyebabkan prosesor harus menyimpan perintah yang baru dijalankan dan memanggil rutin penanganan interupsi.

Gambar 7.2. Tabel Level pada Sistem Operasi

Level	nama	objek
13	shell	user programming environment
12	proses pengguna	proses pengguna
11	direktori	direktori
10	peranti	peranti eksternal
9	sistem berkas	berkas
8	komunikasi	pipa
7	memori virtual	segmen, halaman
6	penyimpanan sekunder lokal	blok data, saluran peranti
5	proses primitif	proses primitif, semafor, ready list
4	interupsi	program penanganan interupsi
3	prosedur	prosedur, call-stack, tampilan
2	set instruksi	stack,microprogram interpreter, scalar, dan array data
1	sirkuit elektronik	register, gerbang, bus, dll

Empat level pertama bukan bagian sistem operasi tetapi bagian perangkat keras. Meski pun demikian beberapa elemen sistem operasi mulai tampil pada level-level ini, seperti rutin penanganan interupsi. Pada level 5, kita mulai masuk kebagian sistem operasi dan konsepnya berhubungan dengan *multi-programming*.

- **Level 5.** Level ini mengenalkan ide proses dalam mengeksekusi program. Kebutuhan-kebutuhan dasar pada sistem operasi untuk mendukung proses ganda termasuk kemampuan men-*suspend* dan me-*resume* proses. Hal ini membutuhkan register perangkat keras untuk menyimpan agar eksekusi bisa ditukar antara satu proses ke proses lainnya.
- **Level 6.** Mengatasi penyimpanan sekunder dari komputer. Level ini untuk menjadwalkan operasi dan menanggapi permintaan proses dalam melengkapi suatu proses.
- **Level 7.** Membuat alamat logik untuk proses. Level ini mengatur alamat virtual ke dalam blok yang bisa dipindahkan antara memori utama dan memori tambahan. Cara-cara yang sering dipakai adalah menggunakan ukuran halaman yang tetap, menggunakan segmen sepanjang variabelnya, dan menggunakan cara keduanya. Ketika blok yang dibutuhkan tidak ada dimemori utama, alamat logis pada level ini meminta transfer dari level 6.

Sampai point ini, sistem operasi mengatasi sumber daya dari prosesor tunggal. Mulai level 8, sistem operasi mengatasi obyek eksternal seperti peranti bagian luar, jaringan, dan sisipan komputer kepada jaringan.

- **Level 8.** Mengatasi komunikasi informasi dan pesan-pesan antar proses. Dimana pada level 5 disediakan mekanisme penanda yang kuno yang memungkinkan untuk sinkronisasi proses, pada level ini mengatasi pembagian informasi yang lebih banyak. Salah satu peranti yang paling sesuai adalah *pipe* (pipa) yang menerima output suatu proses dan memberi input ke proses lain.
- **Level 9.** Mendukung penyimpanan jangka panjang yang disebut dengan berkas. Pada level ini, data dari penyimpanan sekunder ditampilkan pada tingkat abstrak, panjang variabel yang terpisah. Hal ini bertentangan tampilan yang berorientasikan perangkat keras dari penyimpanan sekunder.
- **Level 10.** Menyediakan akses ke peranti eksternal menggunakan antarmuka standar.
- **Level 11.** Bertanggung-jawab mempertahankan hubungan antara internal dan eksternal *identifier* dari sumber daya dan obyek sistem. Eksternal *identifier* adalah nama yang bisa dimanfaatkan oleh aplikasi atau pengguna. Internal *identifier* adalah alamat atau indikasi lain yang bisa digunakan oleh level yang lebih rendah untuk meletakkan dan mengontrol obyek.
- **Level 12.** Menyediakan suatu fasilitator yang penuh tampilan untuk mendukung proses. Hal ini merupakan lanjutan dari yang telah disediakan pada level 5. Pada level 12, semua info yang dibutuhkan untuk manajemen proses dengan berurutan disediakan, termasuk alamat virtual di proses, daftar obyek dan proses yang berinteraksi dengan proses tersebut serta batasan interaksi tersebut, parameter yang harus dipenuhi proses saat pembentukan, dan karakteristik lain yang mungkin digunakan sistem operasi untuk mengontrol proses.
- **Level 13.** Menyediakan antarmuka dari sistem operasi dengan pengguna yang dianggap sebagai *shell* atau dinding karena memisahkan pengguna dengan sistem operasi dan menampilkan sistem operasi dengan sederhana sebagai kumpulan servis atau pelayanan.

Dapat disimpulkan bahwa lapisan sistem operasi secara umum terdiri atas empat bagian, yaitu:

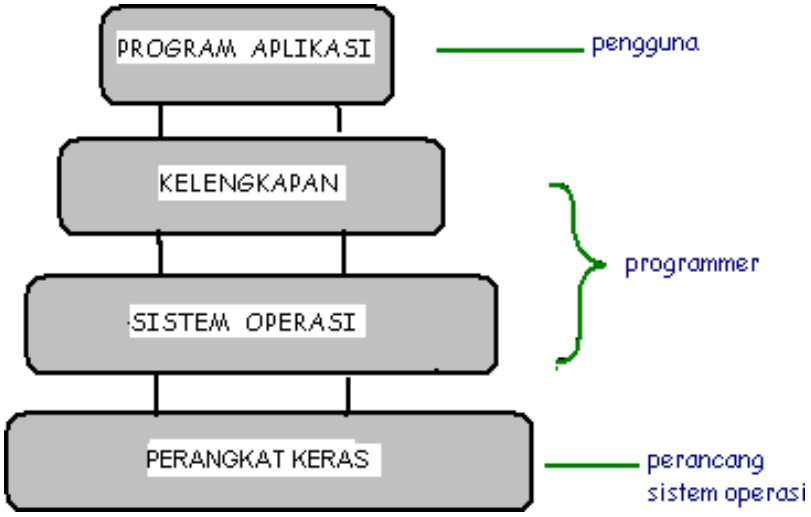
1. **Perangkat keras.** Lebih berhubungan kepada perancang sistem. Lapisan ini mencakup lapisan 0 dan 1 menurut Tanenbaum, dan level 1 sampai dengan level 4 menurut Stallings.

2. **Sistem operasi.** Lebih berhubungan kepada programer. Lapisan ini mencakup lapisan 2 menurut Tanenbaum, dan level 5 sampai dengan level 7 menurut Stallings.

3. **Kelengkapan.** Lebih berhubungan kepada programer. Lapisan ini mencakup lapisan 3 menurut Tanenbaum, dan level 8 sampai dengan level 11 menurut Stallings.

4. **Program aplikasi.** Lebih berhubungan kepada pengguna aplikasi komputer. Lapisan ini mencakup lapisan 4 dan lapisan 5 menurut Tanebaum, dan level 12 dan level 13 menurut Stallings.

Gambar 7.3. Lapisan Sistem Operasi secara umum



Salah satu kesulitan besar dalam sistem terlapis disebabkan karena sebuah lapisan hanya bisa menggunakan lapisan-lapisan dibawahnya, misalnya: *backing-store driver*, normalnya di atas penjadwal CPU sedangkan pada sistem yang besar, penjadwal CPU punya informasi tentang proses yang aktif yang ada di memori. Oleh karena itu, info ini harus dimasukkan dan dikeluarkan dari memori, sehingga membutuhkan *backing-store driver* dibawah penjadwal CPU. Kesulitan lainnya adalah paling tidak efisien dibandingkan tipe lain. Ketika pengguna mengeksekusi M/K, akan mengeksekusi lapisan M/K, lapisan manajemen memori, yang memanggil lapisan penjadwal CPU.

7.4. Kernel-mikro

Metode ini menyusun sistem operasi dengan mengeluarkan semua komponen yang tidak esensial dari *kernel*, dan mengimplementasikannya sebagai program sistem dan level pengguna. Hasilnya *kernel* yang lebih kecil. Pada umumnya mikrokernel mendukung proses dan menajemen memori yang minimal, sebagai tambahan utnuk fasilitas komunikasi.

Fungsi utama mikrokernel adalah mendukung fasilitas komunikasi antara program klien dan bermacam-macam layanan yang juga berjalan di *user space*. Komunikasi yang dilakukan secara tidak langsung, didukung oleh sistem *message passing*, dengan bertukar pesan melalui mikrokernel.

Salah satu keuntungan mikrokernel adalah ketika layanan baru akan ditambahkan ke *user space*, *kernel* tidak perlu dimodifikasi. Kalau pun harus, perubahan akan lebih sedikit. Hasil sistem operasinya lebih mudah untuk ditempatkan pada suatu desain perangkat keras ke desain lainnya. kernel-mikro juga mendukung keamanan reliabilitas lebih, karena kebanyakan layanan berjalan sebagai pengguna proses. Jika layanan gagal, sistem operasi lainnya tetap terjaga. Beberapa sistem operasi yang menggunakan metode ini adalah TRU64 UNIX, MacOSX, dan QNX.

7.5. *Boot*

Saat awal komputer dihidupkan, disebut dengan *booting*. Komputer akan menjalankan *bootstrap program* yaitu sebuah program sederhana yang disimpan dalam ROM yang berbentuk chip *CMOS*. (*Complementary Metal Oxide Semiconductor*). Chip CMOS modern biasanya bertipe *Electrically Erasable Programmable Read Only Memory* (EEPROM), yaitu memori *non-volatile* (tak terhapus jika power dimatikan) yang dapat ditulis dan dihapus dengan pulsa elektronik. Lalu *bootsrap program* ini lebih dikenal sebagai *BIOS (Basic Input Output System)*.

Bootstrap program utama, yang biasanya terletak pada *motherboard* akan memeriksa perangkat keras utama dan melakukan inisialisasi terhadap program dalam *hardware* yang dikenal dengan nama *firmware*.

Bootstrap program utama kemudian akan mencari dan meload *kernel* sistem operasi ke memori lalu dilanjutkan dengan inisialisasi sistem operasi. Dari sini program sistem operasi akan menunggu kejadian tertentu. Kejadian ini akan menentukan apa yang akan dilakukan sistem operasi berikutnya (*event-driven*).

7.6. Kompilasi Kernel

Seperti yang telah diketahui, kernel adalah program yang dimuat pada saat *boot* yang berfungsi sebagai interface antara user-level program dengan hardware. Secara teknis linux hanyalah sebuah kernel. Pogram lain seperti editor, kompilator dan manager yang disertakan dalam paket (SuSE, RedHat, Mandrake, dll.) hanyalah distribusi yang melengkapi kernel menjadi sebuah sistem operasi yang lengkap. Kernel membutuhkan konfigurasi agar dapat bekerja secara optimal.

Konfigurasi ulang dilakukan jika ada device baru yang belum dimuat. Setelah melakukan konfigurasi, lakukan kompilasi untuk mendapatkan kernel yang baru. Tahap ini memerlukan beberapa tool, seperti kompilator dsb. Kompilasi kernel ini dilakukan jika ingin mengupdate kernel dengan keluaran terbaru. Kernel ini mungkin lebih baik dari pada yang lama.

Tahap kompilasi ini sangat potensial untuk menimbulkan kesalahan atau kegagalan, oleh karena itu sangat disarankan untuk mempersiapkan *emergency boot* disk, sebab kesalahan umumnya mengakibatkan sistem mogok.

Ada beberapa langkah yang umumnya dilakukan dalam mengkompilasi kernel, yaitu:

- **Download Kernel.** Tempat untuk mendownload kernel ada di beberapa situs internet. Silakan dicari sendiri. Tetapi biasanya di "kambing.vlsm.org" ada versi-versi kernel terbaru. Source kernel tersebut biasanya dalam format linux-X.Y.ZZ.tar.gz, di mana X.Y.ZZ menunjukkan nomor versi kernel. Misalnya 2.6.11. Nomor tersebut dibagi menjadi tiga bagian, yaitu Major number, Minor number, Revision number. Pada contoh versi kernel di atas (2.6.40), angka 2 menunjukkan major number. Angka ini jarang berubah. Perubahan angka ini menandakan adanya perubahan besar (upgrade) pada kernel. Kemudian angka 6 menunjukkan minor number. Angka ini menunjukkan stabilitas.
  i. Angka genap (0, 2, 4, 6, dst.) menunjukkan kernel tersebut telah stabil.
  ii. Angka ganjil menandakan bahwa kernel tersebut dalam tahap pengembangan Kernel ganjil mengandung experimental-code atau fitur terbaru yang ditambahkan oleh developernya.

  Kernel genap pada saat dirilis tidak ada penambahan lagi dan dianggap sudah stabil. Percobaan terhadap fitur terbaru biasanya dilakukan pada kernel dengan nomor minor yang ganjil. Dua angka terakhir (11) menunjukkan nomor revisi. Ini menandakan current path versi tersebut. Selama tahap pengambangan, nomor ini cepat berubah. Kadang sampai dua kali dalam seminggu.

- **Kompilasi Kernel.** Kompilasi akan memakan waktu lama, dan seperti telah disebutkan di atas, sangat mungkin untuk menimbulkan kegagalan. Di direktori /usr/src/linux, jalankan: make dep; make clean; make zImage. Perintah make dep: membaca file konfigurasi dan membentuk dependency tree. proses ini mengecek apa yang dikompilasi dan apa yang tidak. make clean: menghapus seluruh jejak kompilasi yang telah dilakukan sebelumnya. Ini memastikan agar tidak ada fitur lama yang tersisa. make zImage: Kompilasi yang sesungguhnya. Jika tidak ada kesalahan akan terbentuk kernel terkompresi dan siap dikompilasi. Sebelum dikompilasi, modul-modul yang berhubungan perlu dikompilasi juga dengan make modules. Cek lokasi kernel, umumnya nama kernel dimulai dengan vmlinuz, biasanya ada di direktori /boot, atau buka /etc/lilo.conf untuk memastikannya. Sebelum kernel modul diinstalasi, sebaiknya back-up dulu modul lama. Keuntungan memback-up modul lama adalah bila nanti modul baru tidak berjalan baik, maka modul lama bisa digunakan lagi dengan menghapus modul baru. Setelah tahap ini selesai, jalankan lilo, reboot sistem dan lihat hasilnya.

- **Konfigurasi Kernel.** Konfigurasi kernel adalah tahap terpenting yang menentukan kualitas sebuah kernel. Mana yang harus diikutkan, dan mana yang harus ditinggal sesuai tuntutan hardware dan keperluan. Konfigurasi dimulai dari directori /usr/src/linux. Ada tiga cara: make config, berupa text base interface, cocok untuk user yang memiliki terminal mode lama dan tidak memiliki setting termcap. make menuconfig, berupa text base juga tapi memiliki puldown menu berwarna, digunakan untuk user yang memiliki standar console. make xconfig, interface menggunakan layar grafik penuh, untuk user yang sudah memiliki X Window. Ada sekitar 14

menu pilihan dimulai dari Code maturity level options sampai kernel hacking. Masing-masing memiliki submenu bila dipilih dan pilihan yes(y), module(m), atau no(n). Setiap pilihan dimuat/kompilasi ke dalam kernel akan memperbesar ukuran kernel. Karena itu pilih fitur-fitur yang sering digunakan atau jadikan module untuk fitur yang tidak sering digunakan. Dan jangan memasukkan fitur-fitur yang tidak dibutuhkan. Setelah selesai melakukan pilihan konfigurasi, simpanlah sebelum keluar dari layar menu konfigurasi.

- **Patch Kernel.** Setiap dikeluarkan kernel versi terbaru juga dikeluarkan sebuah file patch. File patch ini jauh lebih kecil dari file source kernel sehingga jauh lebih cepat bila digunakan untuk upgrade kernel. File ini hanya bekerja untuk mengupgrade satu versi kernel dibawahnya. Misalnya, versi kernel 2.4.19 hanya bisa diupgrade dengan file patch 2.4.20 menjadi kernel 2.4.20. Umumnya file patch ini tersedia pada direktori yang sama di FTP dan website yang menyediakan source kernel. File-file patch tersedia dalam format .gz

7.7. Komputer Meja

Dalam pembahasan ini, semua yang layak diletakan di atas meja kerja dikategorikan ke dalam keluarga "komputer meja" (desktop). Salah satu jenis desktop yang paling mudah dikenal ialah komputer personal (PC). Pada awalnya, perangkat keras dari jenis komputer ini relatif sederhana. Sedangkan sistem operasinya hanya mampu secara nyaman, melayani satu pengguna dengan satu job per saat.

Baik komputer meja maupun sistem operasinya, sudah sangat popular sehingga mungkin tidak perlu perkenalan lebih lanjut. Bahkan, mungkin banyak yang belum menyadari bahwa masih terdapat banyak jenis komputer dan sistem operasi lainnya. Dewasa ini (2006), komputer meja lebih canggih ribuan kali dibandingkan IBM PC yang pertama (1981, prosesor 8088, 4.77 MHz). Sedangkan PC pertama tersebut, beberapa kali lebih canggih dibandingkan *main-frame* tahun 1960-an.

Titik fokus perancangan sistem operasi jenis komputer meja agak berbeda dibadingkan dengan sistem operasi "*main-frame*". *Pertama*, kinerja serta derajat kerumitan komponen perangkat keras komputer meja jauh lebih sederhana (dan murah). Karena itu, "*utilisasi*" perangkat keras tidak lagi menjadi masalah utama. *Kedua*, para pengguna komputer meja tidak selalu merupakan "pakar", sehingga kemudahan penggunaan menjadi prioritas utama dalam perancangan sistem operasinya. Ketiga, akibat dari butir kedua di atas, "keamanan" dan "perlindungan" kurang mendapatkan perhatian. Dewasa ini (2006), "virus" dan "cacing" (*worm*) telah menjadi masalah utama yang dihadapi para pengguna sistem operasi komputer meja tertentu.

Yang juga termasuk keluarga komputer meja ini ialah komputer jinjing. Pada dasarnya, tidak terdapat banyak perbedaan, kecuali:
a. sistem *portable* ini pada dasarnya mirip dengan sistem komputer meja, namun harganya relatif lebih mahal.
b. penggunaan catu daya internal (baterei) agar catu daya dapat bertahan selama mungkin (rata-rata 3-6 jam).
c. bobot komputer yang lebih ringan, serta ukuran komputer yang nyaman untuk dijinjing. Sistem ini nyaman digunakan untuk bekerja di perjalanan atau pekerjaan yang menuntut fleksibilitas tempat.

7.8. Sistem Prosesor Jamak

Pada umumnya, setiap komputer dilengkapi dengan satu buah prosesor (CPU). Namun, dewasa ini (2006) mulai umum, jika sebuat sistem komputer memiliki lebih dari satu prosesor (*multi-procesor*). Terdapat dua jenis sistem prosesor jamak, yaitu *Symmetric MultiProcessing (SMP)* dan *Asymmetric MultiProcessing (ASMP)*. Dalam *SMP* setiap prosesor menjalankan salinan identik dari sistem operasi dan banyak job yang dapat berjalan di suatu waktu tanpa pengurangan kinerja. Sementara itu dalam *ASMP* setiap prosesor diberikan suatu tugas yang spesifik. Sebuah prosesor bertindak sebagai *Master processor* yang bertugas menjadwalkan dan mengalokasikan pekerjaan pada prosesor lain yang disebut *slave processors*. Umumnya ASMP digunakan pada sistem yang besar.

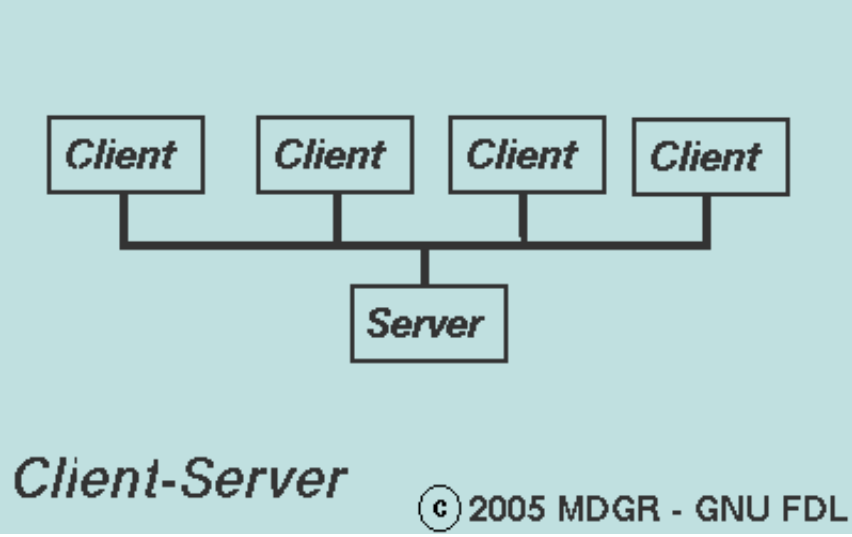
Sistem Operasi Jamak memiliki beberapa keunggulan [Silbeschatz 2004]:
a. Peningkatan *throughput* karena lebih banyak proses/*thread* yang dapat dijalankan sekali gus.

Perlu diingat hal ini tidak berarti daya komputasinya menjadi meningkat sejumlah prosesornya. Yang meningkat ialah jumlah pekerjaan yang bisa dilakukannya dalam waktu tertentu.

b. Economy of Scale: Ekonomis dalam peralatan yang dibagi bersama. Prosesor-prosesor terdapat dalam satu komputer dan dapat membagi *peripheral* (ekonomis) seperti disk dan catu daya listrik.
c. Peningkatan Kehandalan: Jika satu prossor mengalami suatu gangguan, maka proses yang terjadi masih dapat berjalan dengan baik karena tugas prosesor yang terganggu diambil alih oleh prosesor lain. Hal ini dikenal dengan istilah *Graceful Degradation*. Sistemnya sendiri dikenal bersifat *fault tolerant* atau *fail-soft system*.

7.9. Sistem Terdistribusi dan Terkluster

Gambar 7.4. Sistem Terdistribusi I



Melaksanakan komputasi secara terdistribusi diantara beberapa prosesor. Hanya saja komputasinya bersifat *loosely coupled system* yaitu setiap prosesor mempunyai memori lokal sendiri. Komunikasi terjadi melalui bus atau jalur telepon. Keuntungannya hampir sama dengan prosesor jamak (*multi-processor*), yaitu adanya pembagian sumber daya dan komputasi lebih cepat. Namun, pada distributed system juga terdapat keuntungan lain, yaitu memungkinkan komunikasi antar komputer.

Sistem terdistribusi merupakan kebalikan dari Sistem Operasi Prosesor Jamak. Pada sistem tersebut, setiap prosesor memiliki memori lokal tersendiri. Kumpulan prosesornya saling berinteraksi melalui saluran komunikasi seperti LAN dan WAN menggunakan protokol standar seperti TCP/IP. Karena saling berkomunikasi, kumpulan prosesor tersebut mampu saling berbagi beban kerja, data, serta sumber daya lainnya. Namun, keduanya berbagi keunggulan yang serupa seperti dibahas sebelum ini.

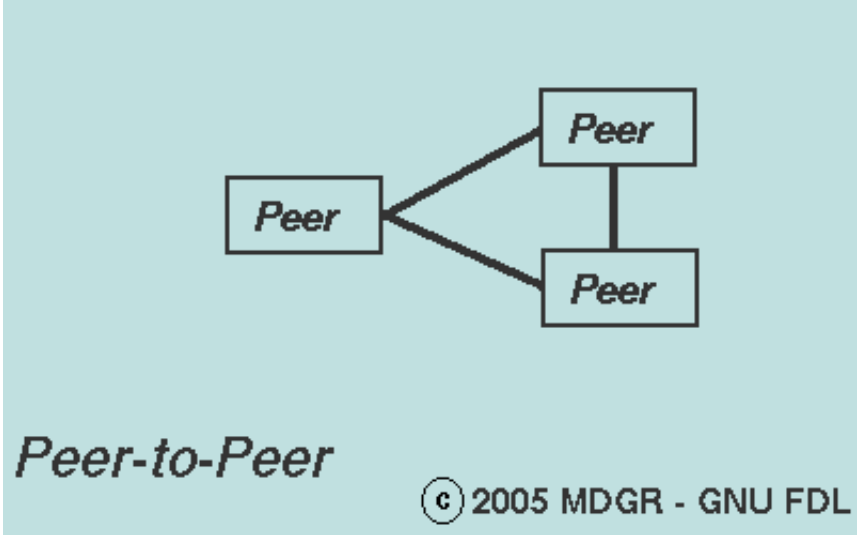
Terdapat sekurangnya tiga model dalam sistem terdistribusi ini. Pertama, sistem *client*/*server* yang membagi jaringan berdasarkan pemberi dan penerima jasa layanan. Pada sebuah jaringan akan didapatkan: *file server*, *time server*, *directory server*, *printer server*, dan seterusnya. Kedua, sistem *point to point* dimana sistem dapat sekali gus berfungsi sebagai *client* maupun *server*. Terakhir sistem terkluster, yaitu beberapa sistem komputer yang digabungkan untuk mendapatkan derajat kehandalan yang lebih baik.

Sistem operasi tersebut di atas, ialah NetOS/Distributed OS. Contoh penerapan *Distributed System*: *Small Area Network (SAN)*, *Local Area Network (LAN)*, *Metropolitan Area Network* (MAN), *Online Service (OL)/Outernet*, *Wide Area Network* (WAN)/*Internet*.

Sistem kluster ialah gabungan dari beberapa sistem individual (komputer) yang dikumpulkan pada suatu lokasi, saling berbagi tempat penyimpanan data (*storage*), dan saling terhubung dalam jaringan lokal (*Local Area Network*). Sistem kluster memiliki persamaan dengan sistem paralel

dalam hal menggabungkan beberapa CPU untuk meningkatkan kinerja komputasi. Jika salah satu mesin mengalami masalah dalam menjalankan tugas maka mesin lain dapat mengambil alih pelaksanaan tugas itu. Dengan demikian, sistem akan lebih andal dan *fault tolerant* dalam melakukan komputasi.

Gambar 7.5. Sistem Terdistribusi II



Dalam hal jaringan, sistem kluster mirip dengan sistem terdistribusi (*distributed system*). Bedanya, jika jaringan pada sistem terdistribusi melingkupi komputer-komputer yang lokasinya tersebar maka jaringan pada sistem kluster menghubungkan banyak komputer yang dikumpulkan dalam satu tempat.

Dalam ruang lingkup jaringan lokal, sistem kluster memiliki beberapa model dalam pelaksanaannya: asimetris dan simetris. Kedua model ini berbeda dalam hal pengawasan mesin yang sedang bekerja. Pengawasan dalam model asimetris menempatkan suatu mesin yang tidak melakukan kegiatan apa pun selain bersiap-siaga mengawasi mesin yang bekerja. Jika mesin itu masalah maka pengawas akan segera mengambil alih tugasnya. Mesin yang khusus bertindak pengawas ini tidak diterapkan dalam model simetris. Sebagai gantinya, mesin-mesin yang melakukan komputasi saling mengawasi keadaan mereka. Mesin lain akan mengambil alih tugas mesin yang sedang mengalami masalah.

Jika dilihat dari segi efisiensi penggunaan mesin, model simetris lebih unggul daripada model asimetris. Hal ini disebabkan terdapat mesin yang tidak melakukan kegiatan apa pun selain mengawasi mesin lain pada model asimetris. Mesin yang 'menganggur' ini dimanfaatkan untuk melakukan komputasi pada model simetris. Inilah yang membuat model simetris lebih efisien.

Isu yang menarik tentang sistem kluster ialah bagaimana mengatur mesin-mesin penyusun sistem dalam berbagi tempat penyimpanan data (*storage*). Untuk saat ini, biasanya sistem kluster hanya terdiri dari dua hingga empat mesin berhubung kerumitan dalam mengatur akses mesin-mesin ini ke tempat penyimpanan data.

Isu di atas juga berkembang menjadi bagaimana menerapkan sistem kluster secara paralel atau dalam jaringan yang lebih luas (*Wide Area Network*). Hal penting yang berkaitan dengan penerapan sistem kluster secara paralel ialah kemampuan mesin-mesin penyusun sistem untuk mengakses data di *storage* secara serentak. Berbagai *software* khusus dikembangkan untuk mendukung kemampuan itu karena kebanyakan sistem operasi tidak menyediakan fasilitas yang memadai. Salah satu contoh perangkat-lunak-nya-nya ialah *Oracle Parallel Server* yang khusus didesain untuk sistem kluster paralel.

Seiring dengan perkembangan pesat teknologi kluster, sistim kluster diharapkan tidak lagi terbatas pada sekumpulan mesin pada satu lokasi yang terhubung dalam jaringan lokal. Riset dan penelitian sedang dilakukan agar pada suatu saat sistem kluster dapat melingkupi berbagai mesin yang tersebar

di seluruh belahan dunia.

Komputasi model terbaru ini juga berbasis jaringan dengan *clustered system*. Digunakan *super computer* untuk melakukan komputasinya. Pada model ini komputasi dikembangkan melalui *pc-farm*. Perbedaan yang nyata dengan komputasi berbasis jaringan ialah bahwa komputasi berbasis *grid* dilakukan bersama-sama seperti sebuah *multiprocessor* dan tidak hanya melakukan pertukaran data seperti pada komputasi berbasis jaringan.

7.10. Sistem Waktu Nyata

Sistem waktu nyata (*Real Time Systems*) ialah suatu sistem yang mengharuskan suatu komputasi selesai dalam jangka waktu tertentu. Jika komputasi ternyata belum selesai maka sistem dianggap gagal dalam melakukan tugasnya. Sistem waktu nyata memiliki dua model dalam pelaksanaannya: *hard real time system* dan *soft real time system*.

Hard real time system menjamin suatu proses yang paling penting dalam sistem akan selesai dalam jangka waktu yang valid. Jaminan waktu yang ketat ini berdampak pada operasi dan perangkat keras (*hardware*) yang mendukung sistem. Operasi M/K dalam sistem, seperti akses data ke *storage*, harus selesai dalam jangka waktu tertentu. Dari segi (*hardware*), memori jangka pendek (*short-term memory*) atau *read-only memory* (ROM) menggantikan *hard-disk* sebagai tempat penyimpanan data. Kedua jenis memori ini dapat mempertahankan data mereka tanpa suplai energi. Ketatnya aturan waktu dan keterbatasan *hardware* dalam sistem ini membuat ia sulit untuk dikombinasikan dengan sistem lain, seperti sistim multiprosesor dengan sistem *time-sharing*.

Soft real time system tidak memberlakukan aturan waktu seketat *hard real time system.* Namun, sistem ini menjamin bahwa suatu proses terpenting selalu mendapat prioritas tertinggi untuk diselesaikan diantara proses-proses lainnya. Sama halnya dengan *hard real time system*, berbagai operasi dalam sistem tetap harus ada batas waktu maksimum.

Aplikasi sistem waktu nyata banyak digunakan dalam bidang penelitian ilmiah, sistem pencitraan medis, sistem kontrol industri, dan industri peralatan rumah tangga. Dalam bidang pencitraan medis, sistem kontrol industri, dan industri peralatan rumah tangga, model waktu nyata yang banyak digunakan ialah model *hard real time system*. Sedangkan dalam bidang penelitian ilmiah dan bidang lain yang sejenis digunakan model *soft real time system*.

Menurut [Morgan1992], terdapat sekurangnya lima karakteristik dari sebuah sistem waktu nyata
- **Deterministik.** Dapat ditebak berapa waktu yang dipergunakan untuk mengeksekusi operasi.
- **Responsif.** Kapan secara pasti eksekusi dimulai serta diakhiri.
- **Kendali Pengguna.** Dengan menyediakan pilihan lebih banyak daripada sistem operasi biasa.
- **Kehandalan.** Sehingga dapat menanggulangi masalah-masalah pengecualian dengan derajat tertentu.
- **Penanganan Kegagalan.** Agar sistem tidak langsung *crash*.

7.11. Aspek Lain

Masih terdapat banyak aspek sistem operasi yang lain; yang kurang cocok diuraikan dalam bab pendahuluan ini. Sebagai penutup dari sub-pokok bahasan ini; akan disinggung secara singkat perihal:
- Sistem Multimedia
- *Embeded System*
- Komputasi Berbasis Jaringan
- PDA dan Telepon Seluler
- *Smart Card*

Sistem MultiMedia

Sistem multimedia merupakan sistem yang mendukung sekali gus berbagai medium seperti gambar tidak bergerak, video (gambar bergerak), data teks, suara, dan seterusnya. Sistem operasi yang mendukung multimedia seharusnya memiliki karakteristik sebagai berikut:
- Handal: para pengguna tidak akan gembira jika sistem terlalu sering *crash*/tidak handal.

- Sistem Berkas: ukuran berkas multimedia cenderung sangat besar. Sebagai gambaran, berkas video dalam format MPEG dengan durasi 60 menit akan berukuran sekitar 650 *MBytes*. Untuk itu, diperlukan sistem operasi yang mampu menangani berkas-berkas dengan ukuran tersebut secara efektif dan efisien.
- *Bandwidth*: diperlukan *bandwidth* (ukuran saluran data) yang besar untuk multimedia.
- Waktu Nyata (*Real Time*): selain *bandwidth* yang besar, berkas multimedia harus disampaikan secara lancar berkesinambungan, serta tidak terputus-putus. Walaupun demikian, terdapat toleransi tertentu terhadap kualitas gambar/suara (*soft real time*).

Embeded System

Komputasi *embedded* melibatkan komputer *embedded* yang menjalankan tugasnya secara *real-time*. Lingkungan komputasi ini banyak ditemui pada bidang industri, penelitian ilmiah, dan lain sebagainya.

Mengacu pada sistem komputer yang bertugas mengendalikan tugas spesifik dari suatu alat seperti mesin cuci digital, tv digital, radio digital. Terbatas dan hampir tak memiliki *user-interface*. Biasanya melakukan tugasnya secara *real-time* merupakan sistem paling banyak dipakai dalam kehidupan.

Komputasi Berbasis Jaringan

Pada awalnya komputasi tradisional hanya meliputi penggunaan komputer meja (*desktop*) untuk pemakaian pribadi di kantor atau di rumah. Namun, seiring dengan perkembangan teknologi maka komputasi tradisional sekarang sudah meliputi penggunaan teknologi jaringan yang diterapkan mulai dari *desktop* hingga sistem genggam. Perubahan yang begitu drastis ini membuat batas antara komputasi tradisional dan komputasi berbasis jaringan sudah tidak jelas lagi.

Komputasi berbasis jaringan menyediakan fasilitas pengaksesan data yang luas oleh berbagai perangkat elektronik. Akses tersedia asalkan perangkat elektronik itu terhubung dalam jaringan, baik dengan kabel maupun nirkabel.

PDA dan Telepon Seluler

Secara umum, keterbatasan yang dimiliki oleh sistem genggam sesuai dengan kegunaan/layanan yang disediakan. Sistem genggam biasanya dimanfaatkan untuk hal-hal yang membutuhkan portabilitas suatu mesin seperti kamera, alat komunikasi, MP3 Player dan lain lain.

Sistem genggam ialah sebutan untuk komputer-komputer dengan kemampuan tertentu, serta berukuran kecil sehingga dapat digenggam. Beberapa contoh dari sistem ini ialah *Palm Pilots, PDA,* dan telepon seluler.

Isu yang berkembang tentang sistem genggam ialah bagaimana merancang perangkat lunak dan perangkat keras yang sesuai dengan ukurannya yang kecil.

Dari sisi perangkat lunak, hambatan yang muncul ialah ukuran memori yang terbatas dan ukuran monitor yang kecil. Kebanyakan sistem genggam pada saat ini memiliki memori berukuran 512 KB hingga 8 MB. Dengan ukuran memori yang begitu kecil jika dibandingkan dengan PC, sistem operasi dan aplikasi yang diperuntukkan untuk sistem genggam harus dapat memanfaatkan memori secara efisien. Selain itu mereka juga harus dirancang agar dapat ditampilkan secara optimal pada layar yang berukuran sekitar 5 x 3 inci.

Dari sisi perangkat keras, hambatan yang muncul ialah penggunaan sumber tenaga untuk pemberdayaan sistem. Tantangan yang muncul ialah menciptakan sumber tenaga (misalnya baterai) dengan ukuran kecil tapi berkapasitas besar atau merancang *hardware* dengan konsumsi sumber tenaga yang sedikit.

Smart Card

Smart Card (Kartu Pintar) merupakan sistem komputer dengan ukuran kartu nama. Kemampuan komputasi dan kapasitas memori sistem ini sangat terbatas sehingga optimasi merupakan hal yang

paling memerlukan perhatian. Umumnya, sistem ini digunakan untuk menyimpan informasi rahasia untuk mengakses sistem lain. Umpamanya, telepon seluler, kartu pengenal, kartu bank, kartu kredit, sistem wireless, uang elektronis, dst.

Dewasa ini (2006), *smart card* dilengkapi dengan prosesor 8 bit (5 MHz), 24 kB ROM, 16 kB EEPROM, dan 1 kB RAM. Namun kemampuan ini meningkat drastis dari waktu ke waktu.

7.12. Rangkuman

Komponen-komponen sistem operasi dapat dihubungkan satu sama lain dengan tiga cara. Pertama, dengan struktur sederhana, kemudian berkembang dengan cakupan yang original. Kedua, dengan pendekatan terlapis atau level. Lapisan yang lebih rendah menyediakan layanan untuk lapisan yang lebih tinggi. Model sistem operasi seperti ini terdiri dari tiga belas level. Ketiga, dengan metode kernelmikro, dimana sistem operasi disusun dalam bentuk *kernel* yang lebih kecil.

Paralel System mempunyai lebih dari satu CPU yang mempunyai hubungan yang erat; CPU-CPU tersebut berbagi bus komputer, dan kadang-kadang berbagi memori dan perangkat yang lainnya. Sistem seperti itu dapat meningkatkan throughput dan reliabilititas.

Sistem hard real-time sering kali digunakan sebagai alat pengontrol untuk applikasi yang dedicated. Sistem operasi yang hard real-time mempunyai batasan waktu yang tetap yang sudah didefinisikan dengan baik.Pemrosesan harus selesai dalam batasan-batasan yang sudah didefinisikan, atau sistem akan gagal.

Sistem *soft real-time* mempunyai lebih sedikit batasan waktu yang keras, dan tidak mendukung penjadwalan dengan menggunakan batas akhir. Pengaruh dari internet dan World Wide Webbaru-baru ini telah mendorong pengembangan sistem operasi modern yang menyertakan web browser serta perangkat lunak jaringan dan komunikasi sebagai satu kesatuan.

Multiprogramming dan sistem time-sharing meningkatkan kemampuan komputer dengan melampaui batas operasi (overlap) CPU dan M/K dalam satu mesin. Hal seperti itu memerlukan perpindahan data antara CPU dan alat M/K, ditangani baik dengan polling atau interrupt-driven akses ke M/K port, atau dengan perpindahan DMA. Agar komputer dapat menjalankan suatu program, maka program tersebut harus berada di memori utama.

Rujukan

[Morgan1992] K Morgan. “The RTOS Difference”. *Byte*. August 1992. 1992.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

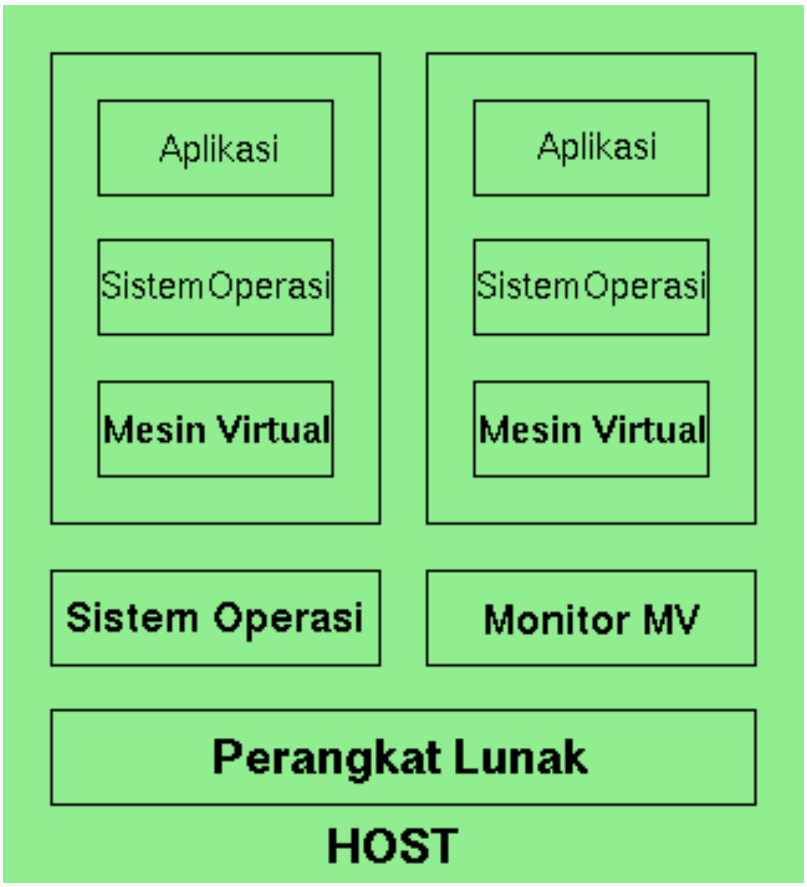
[WEBJones2003] Dave Jones. 2003. *The post-halloween Document v0.48 (aka, 2.6 - what to expect) – http://zenii.linux.org.uk/ ~davej/ docs/ post-halloween-2.6.txt* . Diakses 29 Mei 2006.

Bab 8. Mesin Virtual Java

8.1. Pendahuluan

Mesin virtual sebenarnya bukan merupakan hal yang baru dalam dunia komputer. Mesin virtual biasa digunakan dalam dunia komputer untuk memecahkan beberapa masalah serius, namun sesungguhnya mesin virtual adalah nyata penggunaanya untuk pengguna komputer karena mesin virtual secara khas telah digunakan dalam program aplikasi yang biasa digunakan sehari-hari. Beberapa masalah tersebut misalnya pembagian hardware yang sama yang diakses banyak program atau untuk memungkinkan perangkat lunak agar lebih portabel di antara berbagai jenis sistem operasi. Dalam bab ini kita akan membahas tentang mesin virtual beserta penerapannya dalam sistem operasi, khususnya mesin virtual Java, yang dewasa ini sangat populer dalam ilmu komputer.

Gambar 8.1. Struktur Mesin Virtual



8.2. Konsep Mesin Virtual

Dasar logika dari konsep mesin virtual atau virtual machine adalah dengan menggunakan pendekatan lapisan-lapisan (*layers*) dari sistem komputer. Sistem komputer dibangun atas lapisan-lapisan. Urutan lapisannya mulai dari lapisan terendah sampai lapisan teratas adalah sebagai berikut:

- Perangkat keras
- Kernel
- Sistem program

Kernel, yang berada pada lapisan kedua, menggunakan instruksi perangkat keras untuk menciptakan

seperangkat system call yang dapat digunakan oleh komponen-komponen pada level sistem program. Sistem program kemudian dapat menggunakan system call dan perangkat keras seolah-olah pada level yang sama. Meski sistem program berada di level tertinggi, namun program aplikasi bisa melihat segala sesuatu di bawahnya (pada tingkatan) seakan-akan mereka adalah bagian dari mesin. Pendekatan dengan lapisan-lapisan inilah yang kemudian menjadi kesimpulan logis pada konsep mesin virtual atau virtual machine (VM).

Kelemahan Mesin Virtual

Kesulitan utama dari konsep VM adalah dalam hal sistem penyimpanan dan pengimplementasian. Sebagai contoh, kesulitan dalam sistem penyimpanan adalah sebagai berikut. Andaikan kita mempunyai suatu mesin yang memiliki 3 disk drive namun ingin mendukung 7 VM. Keadaan ini jelas tidak memungkinkan bagi kita untuk dapat mengalokasikan setiap disk drive untuk tiap VM, karena perangkat lunak untuk mesin virtual sendiri akan membutuhkan ruang disk secara substansi untuk menyediakan memori virtual dan spooling.

Solusinya adalah dengan menyediakan disk virtual, atau yang dikenal pula dengan minidisk, di mana ukuran daya penyimpanannya identik dengan ukuran sebenarnya. Sistem disk virtual mengimplementasikan tiap minidisk dengan mengalokasikan sebanyak mungkin track dari disk fisik sebanyak kebutuhan minidisk itu. Secara nyata, total kapasitas dari semua minidisk harus lebih kecil dari kapasitas disk fisik yang tersedia. Dengan demikian, pendekatan VM juga menyediakan sebuah antarmuka yang identik dengan underlying bare hardware. VM dibuat dengan pembagian sumber daya oleh physical computer. Pembagian minidisk sendiri diimplementasikan dalam perangkat lunak.

Kesulitan yang lainnya adalah pengimplementasian. Meski konsep VM cukup baik, namun VM sulit diimplementasikan. Ada banyak hal yang dibutuhkan untuk menyediakan duplikat yang tepat dari underlying machine. VM dapat dieksekusi hanya pada user mode, sehingga kita harus mempunyai user mode virtual sekaligus monitor mode virtual yang keduanya berjalan di physical user mode. Ketika instruksi yang hanya membutuhkan virtual user mode dijalankan, ia akan mengubah isi register yang berefek pada virtual monitor mode, sehingga dapat merestart VM tersebut. Sebuah instruksi M/K yang membutuhkan waktu 100 ms, dengan menggunakan VM bisa dieksekusi lebih cepat karena spooling atau dapat pula lebih lambat karena interpreter. Terlebih lagi, CPU menjadi multiprogrammed di antara banyak VM. Jika setiap user diberi satu VM, dia akan bebas menjalankan sistem operasi (kernel) yang diinginkan pada VM tersebut.

Keunggulan Mesin Virtual

Terlepas dari segala kelemahan-kelemahannya, VM memiliki beberapa keunggulan, antara lain:

Pertama, dalam hal hal keamanan, VM memiliki perlindungan yang lengkap pada berbagai sistem sumber daya, yaitu dengan meniadakan pembagian resources secara langsung, sehingga tidak ada masalah proteksi dalam VM. Sistem VM adalah kendaraan yang sempurna untuk penelitian dan pengembangan sistem operasi. Dengan VM, jika terdapat suatu perubahan pada satu bagian dari mesin, maka dijamin tidak akan mengubah komponen lainnya.

Kedua, dimungkinkan untuk mendefinisikan suatu jaringan dari mesin virtual, di mana tiap-tiap bagian mengirim informasi melalui jaringan komunikasi virtual. Sekali lagi, jaringan dimodelkan setelah komunikasi fisik jaringan diimplementasikan pada perangkat lunak.

Contoh Mesin Virtual

Contoh penerapan VM saat ini terdapat pada sistem operasi Linux. Mesin virtual saat ini memungkinkan aplikasi Windows untuk berjalan pada komputer yang berbasis Linux. VM juga berjalan pada aplikasi Windows dan sistem operasi Windows.

8.3. Konsep Bahasa Java

Sun Microsystems mendesain bahasa Java, yang pada mulanya dikenal dengan nama Oak. James Gosling, sang pencipta Oak, menciptakannya sebagai bagian dari bahasa C++. Bahasa ini harus

cukup kecil agar dapat bertukar informasi dengan cepat di antara jaringan kabel perusahaan dan pertelevisian dan cukup beragam agar dapat digunakan lebih dari satu jaringan kabel. Sun Microsystems lalu merubah nama Oak menjadi Java, kemudian membuatnya tersedia di dalam Internet. Perkenalan dengan Java di Internet ini dimulai pada tahun 1995.

Java didesain dengan tujuan utama portabilitas, sesuai dengan konsep write once run anywhere. Jadi, hasil kompilasi bahasa Java bukanlah native code, melainkan bytecode. Bytecode dieksekusi oleh interpreter Java yang juga merupakan Java Virtual Machine. Penjelasan mengenai Java Virtual Machine (JVM) akan dijelaskan pada Bagian 8.4, “Mesin Virtual Java”.

Ada beberapa hal yang membedakan Java dengan bahasa pemrograman lain yang populer pada saat ini, yakni:
- Bersifat portable, artinya program Java dapat dijalankan pada platform yang berbeda tanpa perlu adanya kompilasi ulang.
- Memiliki garbage collection yang berfungsi untuk mendelokasi memori secara otomatis.
- Menghilangkan pewarisan ganda, yang merupakan perbaikan dari bahasa C++.
- Tidak ada penggunaan pointer, artinya bahasa Java tidak membolehkan pengaksesan memori secara langsung.

Teknologi Java terdiri dari komponen, yakni:
- Application Programming Interface (API)
- Spesifikasi mesin virtual

Penjelasan lebih lanjut mengenai komponen Java adalah sebagai berikut.

Bahasa Pemrograman

Bahasa Java merupakan bahasa pemrograman yang berorientasi pada objek (object-oriented), memiliki arsitektur yang netral (architecture-neutral), dapat didistribusikan, dan mendukung multithread. Objek-objek dalam Java dispesifikasikan ke dalam class; program Java terdiri dari satu atau beberapa class.

Contoh 8.1. Contoh penggunaan class objek dalam Java

```
01 class Objek1
02 {
03    private int attribut1;
04    private String attribut2;
05
06    public void changeAttribut1()
07    {
08       // melakukan sesuatu terhadap attribut1 harus dengan method
09       // ini. Jadi variabel attribut1 aman di dalam objeknya,
10       // tidak mudah diakses begitu saja...
11      }
12 }
13
14 class Objek2
15 {
16      private int attribut1;
17      private String attribut2;
19      public Objek1 objekSatu;
20
21      public void interfensi()
22      {
23              objekSatu.changeAttribut1();
24              // valid karena akses modifiernya public
26              System.out.print( objekSatu.attribut1 );
27              // invalid karena akses modifiernya private
28      }
29 }
```

Dari setiap class dalam Java, kompilator menghasilkan keluaran berupa berkas bytecode yang bersifat architecture-neutral. Artinya, berkas tersebut akan dapat berjalan pada mesin virtual Java (JVM) manapun. Pada awalnya, Java digunakan untuk pemrograman Internet, karena Java menyediakan sebuah layanan yang disebut dengan applet, yaitu program yang berjalan dalam sebuah web browser dengan akses sumber daya yang terbatas. Java juga menyediakan layanan untuk jaringan dan distributed objects. Java adalah sebuah bahasa yang mendukung multithread, yang berarti sebuah program Java dapat memiliki beberapa thread.

Java termasuk sebuah bahasa yang aman. Hal ini sangat penting mengingat program Java dapat berjalan dalam jaringan terdistribusi. Java juga memiliki suatu pengendalian memori dengan menjalankan garbage collection, yaitu suatu fasilitas untuk membebaskan memori dari objek-objek yang sudah tidak dipergunakan lagi dan mengembalikannya kepada sistem.

API

API merupakan suatu metode yang menggunakan sebuah aplikasi program untuk mengakses sistem operasi dari komputer. API memungkinkan kita untuk memprogram antarmuka pre-constructed sebagai pengganti memprogram device atau bagian dari perangkat lunak secara langsung. API menyediakan sarana bagi para programmer untuk mendesain antarmuka dengan komponen yang disediakan. Hal ini membuat pengembangan dan pendesainan antarmuka menjadi cepat, tanpa harus memiliki pengetahuan secara mendetail tentang device atau perangkat lunak yang digunakan. Sebagai contoh, API dari OpenGL memungkinkan kita untuk membuat efek 3D tanpa perlu mengetahui bagian dalam dari kartu grafis.

API dalam Java

Terdiri dari tiga bagian, yaitu:
- API standar yang dipergunakan untuk aplikasi dan applet dengan layanan bahasa dasar untuk grafik, M/K, utilitas, dan jaringan.
- API enterprise untuk mendesain aplikasi server dengan layanan database dan aplikasi server-side (dikenal dengan servlet).
- API untuk device kecil seperti komputer genggam, pager, dan ponsel.

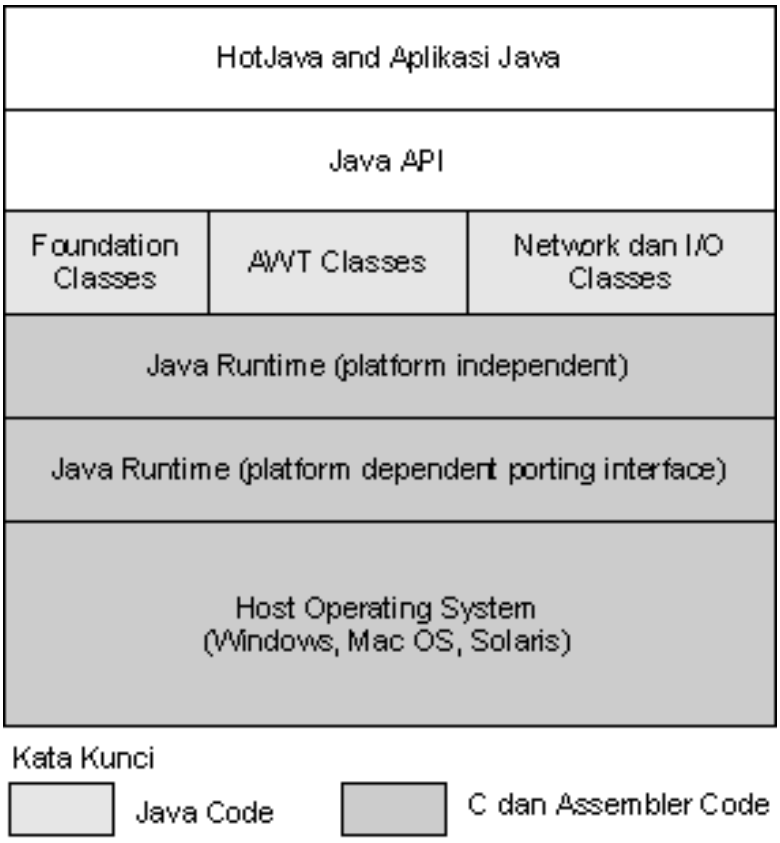
Contoh 8.2. Contoh penggunaan Java API

```
01 import java.util.Date;
02
03 class Tanggal
04 {
05      public void cetak()
06      {
07              Date tanggal = new Date();
08              // membuat objek baru untuk tanggal
09
10              String cetak1 = tanggal.toString();
11
12              System.out.println( cetak1 );
13              // mencetak tanggal hari ini
14      }
15 }
```

8.4. Mesin Virtual Java

Mesin Virtual Java atau Java Virtual Machine (JVM) terdiri dari sebuah class loader dan Java interpreter yang mengeksekusi architecture-neutral bytecode. Java interpreter merupakan suatu fasilitas penerjemah dalam JVM. Fungsi utamanya adalah untuk membaca isi berkas bytecode (.class) yang dibuat kompilator Java saat berkas berada dalam memori, kemudian menerjemahkannya menjadi bahasa mesin lokal. Java interpreter dapat berupa perangkat lunak yang menginterpretasikan bytecode setiap waktu, atau hanya *Just-In-Time* (*JIT*), yang mengubah *architecture-neutral bytecode* menjadi bahasa mesin lokal. Interpreter bisa juga diimplementasikan pada sebuah chip perangkat keras. Instance dari JVM dibentuk ketika aplikasi Java atau applet dijalankan. JVM mulai berjalan saat method main() dipanggil.

Gambar 8.2. JVM: Java Virtual Machine



Meski sistem program berada di level tertinggi, program aplikasi bisa melihat segala sesuatu di bawahnya (pada tingkatan) seakan mereka adalah bagian dari mesin. Pendekatan dengan lapisan-lapisan inilah yang diambil sebagai kesimpulan logis pada konsep mesin virtual atau ***virtual machine*** (VM). Pendekatan VM menyediakan sebuah antarmuka yang identik dengan *underlying bare hardware*. VM dibuat dengan pembagian sumber daya oleh *physical computer*. VM perangkat lunak membutuhkan ruang pada disk untuk menyediakan memori virtual dan *spooling* sehingga perlu ada disk virtual.

Pada applet, JVM menciptakan method main() sebelum membuat applet itu sendiri. Java Development Environment terdiri dari sebuah Compile-Time Environment dan Runtime Environment. Compile berfungsi mengubah sourcecode Java menjadi bytecode, sedangkan Runtime merupakan Java Platform untuk sistem Host.

Meski sangat berguna, VM sulit untuk diimplementasikan. Banyak hal yang dibutuhkan untuk menyediakan duplikat yang tepat dari *underlying machine*. VM dapat dieksekusi pada *only user mode* sehingga kita harus mempunyai *virtual user mode* sekaligus *virtual memory mode* yang keduanya berjalan di *physical user mode*. Ketika instruksi yang hanya membutuhkan *virtual user*

mode dijalankan, ia akan mengubah isi register yang berefek pada *virtual monitor mode* sehingga dapat memulai ulang VM tersebut. Sebuah instruksi M/K yang membutuh waktu 100 ms, dengan menggunakan VM bisa dieksekusi lebih cepat karena *spooling* atau lebih lambat karena interpreter. Terlebih lagi, CPU menjadi *multiprogrammed* di antara banyak VM. Jika setiap user diberi 1 VM, dia akan bebas menjalankan sistem operasi (kernel) yang diinginkan pada VM tersebut.

Selain kekurangan yang telah disebutkan diatas, jelas VM memiliki kelebihan-kelebihan, yaitu: Keamanan yang terjamin karena VM mempunyai perlindungan lengkap pada berbagai sistem sumber daya, tidak ada pembagian resources secara langsung. Pembagian disk mini dan jaringan diimplementasikan dalam perangkat lunak. Sistem VM adalah kendaraan yang sempurna untuk penelitian dan pengembangan Sistem Operasi. Dengan VM, perubahan satu bagian dari mesin dijamin tidak akan mengubah komponen lainnya.

Mesin Virtual Java atau *Java Virtual Machine* (JVM) terdiri dari sebuah kelas loader dan java interpreter yang mengeksekusi *the architecture-neutral bytecodes*. Java interpreter bisa berupa perangkat lunak yang menginterpretasikan kode byte setiap waktu atau hanya ***Just-In-Time*** (JIT) yang mengubah *architecture-neutral bytecodes* menjadi bahasa mesin lokal. Interpreter bisa juga diimplementasikan pada sebuah *chip* perangkat keras. *Instance* dari JVM dibentuk ketika aplikasi java atau applet dijalankan. JVM mulai berjalan saat *method main* dipanggil. Pada applet, JVM menciptakan method *main* sebelum membuat applet itu sendiri.

Java Development Environment terdiri dari sebuah *Compile Time Environment* dan *RunTime Environment*. *Compile* berfungsi mengubah *java sourcecode* menjadi kode byte. Sedangkan *RunTime* merupakan *Java Platform* untuk sistem *Host*.

8.5. Sistem Operasi Java

Kebanyakan dari sistem operasi yang ada dewasa ini dibuat dari kombinasi bahasa C dan bahasa assembly. Hal ini disebabkan karena keuntungan performa serta kemudahan dalam berinteraksi dengan perangkat keras. Kami menyebut ini sebagai sistem operasi tradisional.

Namun, akhir-akhir ini banyak usaha yang dilakukan dalam membuat sistem operasi berbasis bahasa pemrogramanan, terutama sistem operasi berbasis bahasa pemrograman Java, di antaranya adalah sistem operasi JavaOS yang telah merilis versi 1.0 dan juga JX. Perbedaan antara keduanya adalah pada fungsionalitas bahasa pemrograman yang digunakan. JavaOS sepenuhnya menggunakan fungsionalitas bahasa Java, sementara JX menggunakan gabungan fungsionalitas dari bahasa Java, C, dan assembly.

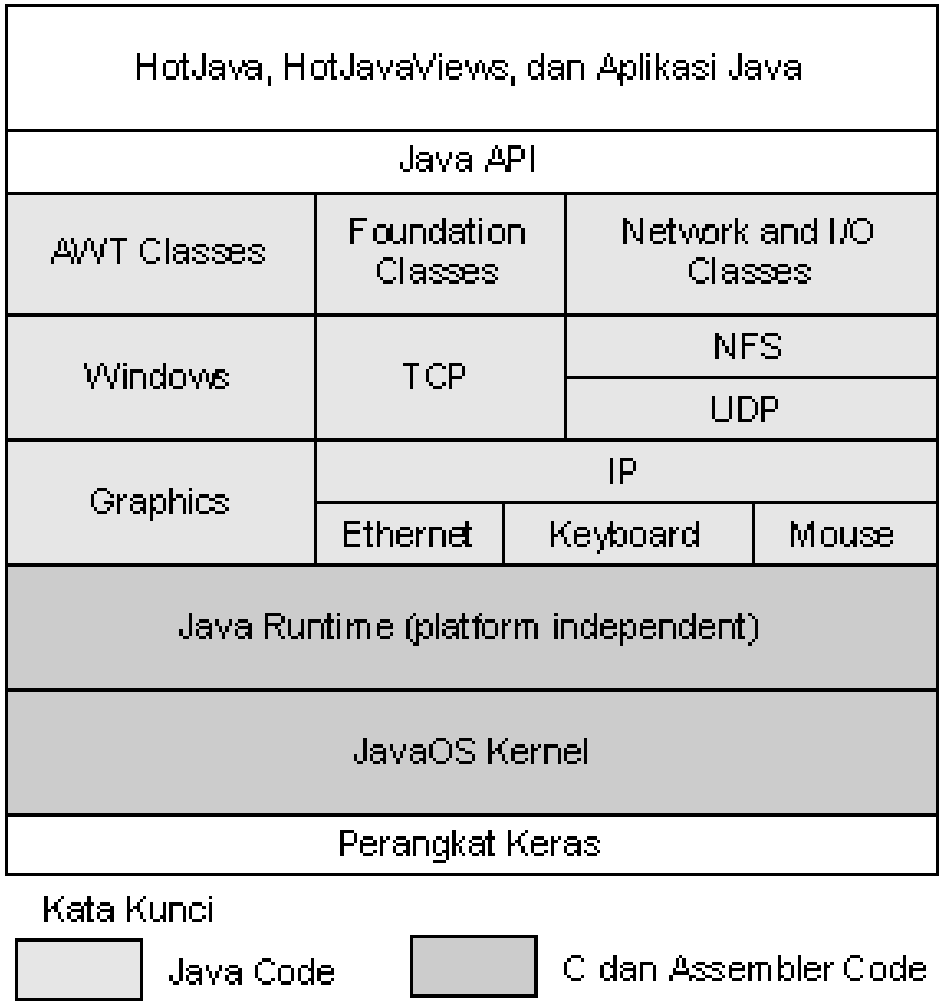
Sistem Operasi JavaOS

JavaOS adalah satu-satunya sistem yang mencoba untuk mengimplementasi fungsi sistem operasi dalam bahasa Java secara lengkap. JavaOS mengimplementasi platform Java agar dapat menjalankan aplikasi atau applet yang mengakses fasilitas dari beberapa objek. Selain itu, JavaOS juga mengimplementasikan JVM dan lapisan fungsionalitas untuk windowing, jaringan, dan sistem berkas tanpa membutuhkan dukungan dari sistem operasi lokal. JavaOS mendefinisikan platform seperti halnya CPU, memori, bus, dan perangkat keras lainnya. Platform independen dari sistem operasinya disebut JavaOS runtime, sedangkan bagian platform yang non-independen dari sistem operasinya disebut JavaOS kernel.

JavaOS menyediakan lingkungan Java yang standalone. Dengan kata lain, aplikasi yang dikembangkan untuk platform Java yang menggunakan JavaOS dapat berjalan pada perangkat keras tanpa dukungan sistem operasi lokal. Selain itu, aplikasi yang ditulis untuk berjalan pada satu mesin tanpa adanya sistem operasi lokal dapat pula berjalan pada mesin yang memiliki sistem operasi lokal.

JavaOS terbagi menjadi dua, yaitu kode platform independen dan platform non-independen. Kode platform non-independen merujuk kepada kernel dan terdiri atas mikrokernel dan JVM. Mikrokernel menyediakan layanan menajemen memori, interupsi dan penganan trap, multithread, DMA, dan fungsi level rendah lainnya. JVM menerjemahkan dan mengeksekusi bytecode Java. Tujuan dari kernel adalah untuk meringkaskan spesifikasi perangkat keras dan menyediakan platform antarmuka yang netral dari sistem operasi.

Gambar 8.3. Struktur sistem operasi JavaOS



Kernel JavaOS

Kernel JavaOS membutuhkan antarmuka untuk underlying machine dengan JVM. Hal ini memungkinkan kernel untuk menjadi lebih kecil, cepat, dan portabel. Beberapa fungsi yang disediakan oleh kernel di antaranya adalah:

Gambar 8.4. PL3

```
1. Sistem Booting       5. Monitor              9. Debug
2. Exceptions           6. Sistem berkas
3. Thread               7. Timing
4. Manajemen Memori     8. DMA
```

Sedangkan kode platform independen dari JavaOS merujuk pada JavaOS runtime. Runtime sepenuhnya ditulis dalam bahasa Java, yang memungkinkan untuk dijalankan pada platform yang berbeda. Java runtime terdiri dari device driver, dukungan jaringan, sistem grafik, sistem windowing, dan elemen lain dari Java API. Device driver mendukung komunikasi dengan monitor,

mouse, keyboard, dan kartu jaringan.

Komponen JavaOS Runtime

JavaOS runtime terdiri dari fungsi spesifik sistem operasi yang ditulis dalam bahasa Java. Komponen dari JavaOS runtime di antaranya Device Driver, Jaringan TCP/IP, Sistem Grafik, dan Sistem Window.

Sistem Operasi JX

Mayoritas sistem operasi JX ditulis dalam bahasa Java, sementara kernel mikronya ditulis dalam bahasa C dan assembly yang mengandung fungsi yang tidak terdapat di Java. Struktur utama dari JX adalah tiap kode Java diorganisasikan sebagai komponen, di mana kode di-load langsung ke domain dan diterjemahkan ke native code. Domain mengencapsulate objek dan thread. Komunikasi antara domain ditangani menggunakan portal.

Berikut penjelasan lebih lanjut mengenai arsitektur sistem operasi JX:
- Domain; yaitu unit proteksi dan manajemen sumber daya di mana semua domain kecuali domain zero mengandung 100% kode bahasa Java.
- Portal; merupakan dasar mekanisme komunikasi inter-domain. Cara kerjanya mirip dengan Java RMI yang membuat programmer mudah menggunakannya.
- Objek memori, merupakan abstraksi dari representasi area memori yang diakses dengan method invocations.
- Komponen; tiap kode java yang di-load ke suatu domain diatur pada komponen. Suatu komponen mengandung class, antarmuka, dan tambahan informasi.
- Manajemen memori; proteksi memori berbasiskan bahasa type-safe.
- Verifier dan Translator; merupakan bagian penting dari sistem JX. Mekanismenya, semua kode diverifikasi sebelum diterjemahkan ke dalam bentuk native code dan dieksekusi.
- Device driver; semua device driver sistem JX ditulis dalam bahasa Java.
- Penjadwalan; sistem JX menggunakan pendekatan di mana penjadwalan diimplementasikan di luar kernel mikro.
- Locking; terdapat kernel-level locking, domain-level locking, dan inter-domain locking.

Kelemahan sistem operasi berbasis bahasa pemrograman

Salah satu kesulitan dalam mendesain sistem berbasis bahasa pemrograman adalah menyangkut masalah proteksi, khususnya proteksi memori. Sistem operasi tradisional menyandarkan pada fitur perangkat keras untuk menyediakan proteksi memori. Sistem berbasis bahasa pemrograman sendiri memiliki ketergantungan pada fitur type-safety dari bahasa pemrograman tersebut untuk mengimplementasikan proteksi memori. Hasilnya, sistem berbasis bahasa pemrograman memerlukan perangkat keras yang mampu menutupi kekurangan dalam hal fitur proteksi memori.

8.6. Rangkuman

Konsep mesin virtual sangat baik, namun cukup sulit untuk diimplementasikan, karena mesin virtual harus mampu berjalan pada dua keadaan sekaligus, yaitu virtual user mode dan virtual monitor mode. Mesin virtual juga memiliki keunggulan, yaitu proteksi sistem yang sangat cocok untuk riset dan pengembangan sistem operasi.

Java didesain dengan tujuan utama adalah portabilitas. Dengan konsep write once run anywhere, maka hasil kompilasi bahasa Java yang berupa bytecode dapat dijalankan pada platform yang berbeda. Teknologi Java terdiri dari tiga komponen penting, yakni spesifikasi bahasa pemrograman, Application Programming Interface (API) dan spesifikasi mesin virtual. Bahasa Java mendukung paradigma berorientasi objek serta dilengkapi juga dengan library API yang sangat lengkap.

Mesin virtual Java atau Java Virtual Machine (JVM) terdiri dari sebuah class loader dan Java interpreter yang mengeksekusi architecture-neutral bytecode.

JavaOS dibangun dari kombinasi native code dan Java code, di mana platformnya independen. Sedangkan JX merupakan sistem operasi di mana setiap kode Java diorganisasikan sebagai komponen.

Penggunaan mesin virtual amat berguna, tapi sulit untuk diimplementasikan. Sebagaimana perangkat-perangkat lainnya, penggunaan mesin virtual ini memiliki kelebihan dan kekurangan. Masalah utama dari desain sistem adalah menentukan kebijakan dan mekanisme yang sesuai dengan keinginan pengguna dan pendisainnya. *System generation* adalah proses mengkonfigurasikan sistem untuk masing-masing komputer.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Venners1998] Bill Venners. 1998. *Inside the Java Virtual Machine* . McGraw-Hill.

[WEBGolmFWK2002] Michael Golm, Meik Felser, Christian Wawersich, dan Juerge Kleinoede. 2002. *The JX Operating System – http://www.jxos.org/ publications/ jx-usenix.pdf* . Diakses 31 Mei 2006.

[WEBRyan1998] Tim Ryan. 1998. *Java 1.2 Unleashed – http://utenti.lycos.it/ yanorel6/ 2/ ch52.htm* . Diakses 31 Mei 2006.

Bab 9. Sistem GNU/Linux

9.1. Pendahuluan

Linux adalah sebuah sistem operasi yang sangat mirip dengan sistem-sistem UNIX, karena memang tujuan utama rancangan dari proyek Linux adalah UNIX compatible. Sejarah Linux dimulai pada tahun 1991, ketika mahasiswa Universitas Helsinki, Finlandia bernama Linus Benedict Torvalds menulis Linux, sebuah kernel untuk prosesor 80386, prosesor 32-bit pertama dalam kumpulan CPU intel yang cocok untuk PC.

Pada awal perkembangannya, kode sumber (*source code*) Linux disediakan secara bebas melalui internet. Hasilnya, pengembangan Linux merupakan kolaborasi para pengguna dari seluruh dunia, semuanya dilakukan secara eksklusif melalui internet. Bermula dari kernel awal yang hanya mengimplementasikan subset kecil dari sistem UNIX, kini sistem Linux telah tumbuh sehingga mampu memasukkan banyak fungsi UNIX.

Kernel Linux berbeda dengan sistem Linux. Kernel Linux merupakan sebuah perangkat lunak orisinil yang dibuat oleh komunitas Linux, sedangkan sistem Linux, yang dikenal saat ini, mengandung banyak komponen yang dibuat sendiri atau dipinjam dari proyek pengembangan lain.

Kernel Linux pertama yang dipublikasikan adalah versi 0.01, pada tanggal 14 Maret 1991. Sistem berkas yang didukung hanya sistem berkas Minix. Kernel pertama dibuat berdasarkan kerangka Minix (sistem UNIX kecil yang dikembangkan oleh Andy Tanenbaum). Tetapi, kernel tersebut sudah mengimplementasi proses UNIX secara tepat.

Pada tanggal 14 Maret 1994 dirilis versi 1.0, yang merupakan tonggak sejarah Linux. Versi ini adalah kulminasi dari tiga tahun perkembangan yang cepat dari kernel Linux. Fitur baru terbesar yang disediakan adalah jaringan. Versi 1.0 mampu mendukung protokol standar jaringan TCP/IP. Kernel 1.0 juga memiliki sistem berkas yang lebih baik tanpa batasan-batasan sistem berkas Minix. Sejumlah dukungan perangkat keras ekstra juga dimasukkan ke dalam rilis ini. Dukungan perangkat keras telah berkembang termasuk diantaranya floppy-disk, CD-ROM, sound card, berbagai mouse, dan keyboard internasional. Dukungan juga diberikan terhadap modul kernel yang *loadable* dan *unloadable* secara dinamis.

Satu tahun kemudian dirilis kernel versi 1.2. Kernel ini mendukung variasi perangkat keras yang lebih luas. Pengembang telah memperbaharui networking stack untuk menyediakan support bagi protokol IPX, dan membuat implementasi IP lebih lengkap dengan memberikan fungsi accounting dan firewalling. Kernel 1.2 ini merupakan kernel Linux terakhir yang PC-only. Konsentrasi lebih diberikan pada dukungan perangkat keras dan memperbanyak implementasi lengkap pada fungsi-fungsi yang ada.

Pada bulan Juni 1996, kernel Linux 2.0 dirilis. Versi ini memiliki dua kemampuan baru yang penting, yaitu dukungan terhadap multiple architecture dan multiprocessor architectures. Kode untuk manajemen memori telah diperbaiki sehingga kinerja sistem berkas dan memori virtual meningkat. Untuk pertama kalinya, file system caching dikembangkan ke networked file systems, juga sudah didukung writable memory mapped regions. Kernel 2.0 sudah memberikan kinerja TCP/IP yang lebih baik, ditambah dengan sejumlah protokol jaringan baru. Kemampuan untuk memakai remote netware dan SMB (Microsoft LanManager) network volumes juga telah ditambahkan pada versi terbaru ini. Tambahan lain adalah dukungan internal kernel threads, penanganan dependencies antara modul-modul loadable, dan loading otomatis modul berdasarkan permintaan (on demand). Konfigurasi dinamis dari kernel pada run time telah diperbaiki melalui konfigurasi interface yang baru dan standar.

Semenjak Desember 2003, telah diluncurkan kernel versi 2.6, yang dewasa ini (2006) telah mencapai *patch* versi 2.6.17.11. Hal-hal yang berubah dari versi 2.6 ini ialah:

- Subsitem M/K yang dipercanggih.
- Kernel yang pre-emptif.

- Penjadwalan Proses yang dipercanggih.
- Threading yang dipercanggih.
- Implementasi ALSA (Advanced Linux Sound Architecture) dalam kernel.
- Dukungan sistem berkas seperti: ext2, ext3, reiserfs, adfs, amiga ffs, apple macintosh hfs, cramfs, jfs, iso9660, minix, msdos, bfs, free vxfs, os/2 hpfs, qnx4fs, romfs, sysvfs, udf, ufs, vfat, xfs, BeOS befs (ro), ntfs (ro), efs (ro).

9.2. Sistem dan Distribusi GNU/Linux

Dalam banyak hal, kernel Linux merupakan inti dari proyek Linux, tetapi komponen lainlah yang membentuk secara lengkap sistem operasi Linux. Dimana kernel Linux terdiri dari kode-kode yang dibuat khusus untuk proyek Linux, kebanyakan perangkat lunak pendukungnya tidak eksklusif terhadap Linux, melainkan biasa dipakai dalam beberapa sistem operasi yang mirip UNIX. Contohnya, sistem operasi BSD dari Berkeley, X Window System dari MIT, dan proyek GNU dari Free Software Foundation.

Pembagian (*sharing*) alat-alat telah bekerja dalam dua arah. Sistem pustaka utama Linux awalnya dimulai oleh proyek GNU, tetapi perkembangan pustakanya diperbaiki melalui kerjasama dari komunitas Linux terutama pada pengalamatan, ketidak- efisienan, dan bugs. Komponen lain seperti GNU C Compiler, gcc, kualitasnya sudah cukup tinggi untuk dipakai langsung dalam Linux. Alat-alat administrasi network di bawah Linux berasal dari kode yang dikembangkan untuk 4.3 BSD, tetapi BSD yang lebih baru, salah satunya FreeBSD, sebaliknya meminjam kode dari Linux, contohnya adalah pustaka matematika Intel floating-point-emulation.

Sistem Linux secara keseluruhan diawasi oleh network tidak ketat yang terdiri dari para pengembang melalui internet, dengan grup kecil atau individu yang memiliki tanggung-jawab untuk menjaga integritas dari komponen-komponen khusus. Dokumen 'File System Hierarchy Standard' juga dijaga oleh komunitas Linux untuk memelihara kompatibilitas ke seluruh komponen sistem yang berbeda-beda. Aturan ini menentukan rancangan keseluruhan dari sistem berkas Linux yang standar.

Siapa pun dapat menginstall sistem Linux, ia hanya perlu mengambil revisi terakhir dari komponen sistem yang diperlukan melalui situs ftp lalu dikompilasi. Pada awal keberadaan Linux, operasi seperti di atas persis seperti yang dilaksanakan oleh pengguna Linux. Namun, dengan semakin berkembangnya Linux, berbagai individu dan kelompok berusaha membuat pekerjaan tersebut lebih mudah dengan cara menyediakan sebuah set bingkisan yang standar dan sudah dikompilasi terlebih dahulu supaya dapat diinstall secara mudah.

Koleksi atau distribusi ini, tidak hanya terdiri dari sistem Linux dasar tetapi juga mengandung instalasi sistem ekstra dan utilitas manajemen, bahkan paket yang sudah dikompilasi dan siap diinstall dari banyak alat UNIX yang biasa, seperti news servers, web browsers, text-processing dan alat mengedit, termasuk juga games.

Distribusi pertama mengatur paket-paket ini secara sederhana, menyediakan sebuah sarana untuk memindahkan seluruh file ke tempat yang sesuai. Salah satu kontribusi yang penting dari distribusi modern adalah manajemen/pengaturan paket-paket yang lebih baik. Distribusi Linux pada saat ini melibatkan database packet tracking yang memperbolehkan suatu paket agar dapat diinstall, di-upgrade, atau dihilangkan tanpa susah payah.

Distribusi SLS (Soft Landing System) adalah koleksi pertama dari bingkisan Linux yang dikenal sebagai distribusi komplit. Walaupun SLS dapat diinstall sebagai entitas tersendiri, dia tidak memiliki alat-alat manajemen bingkisan yang sekarang diharapkan dari distribusi Linux. Distribusi Slackware adalah peningkatan yang besar dalam kualitas keseluruhan (walaupun masih memiliki manajemen bingkisan yang buruk); Slackware masih merupakan salah satu distribusi yang paling sering diinstall dalam komunitas Linux.

Sejak dirilisnya Slackware, sejumlah besar distribusi komersial dan non-komersial Linux telah tersedia. Red Hat dan Debian adalah distribusi yang terkenal dari perusahaan pendukung Linux

komersial dan perangkat lunak bebas komunitas Linux. Pendukung Linux komersial lainnya termasuk distribusi dari Caldera, Craftworks, dan Work-Group Solutions. Contoh distribusi lain adalah SuSE dan Unifix yang berasal dari Jerman.

9.3. Lisensi Linux

Kernel Linux terdistribusi di bawah Lisensi Publik Umum GNU (GPL), dimana peraturannya disusun oleh Free Software Foundation. Linux bukanlah perangkat lunak domain publik: Public Domain berarti bahwa pengarang telah memberikan copyright terhadap perangkat lunak mereka, tetapi copyright terhadap kode Linux masih dipegang oleh pengarang-pengarang kode tersebut. Linux adalah perangkat lunak bebas, namun: bebas dalam arti bahwa siapa saja dapat mengkopi, modifikasi, memakainya dengan cara apa pun, dan memberikan kopi mereka kepada siapa pun tanpa larangan atau halangan.

Implikasi utama peraturan lisensi Linux adalah bahwa siapa saja yang menggunakan Linux, atau membuat modifikasi dari Linux, tidak boleh membuatnya menjadi hak milik sendiri. Jika sebuah perangkat lunak dirilis berdasarkan lisensi GPL, produk tersebut tidak boleh didistribusi hanya sebagai produk biner (*binary-only*). Perangkat lunak yang dirilis atau akan dirilis tersebut harus disediakan sumber kodenya bersamaan dengan distribusi binernya.

9.4. Linux Saat Ini

Saat ini, Linux merupakan salah satu sistem operasi yang perkembangannya paling cepat. Kehadiran sejumlah kelompok pengembang, tersebar di seluruh dunia, yang selalu memperbaiki segala fiturnya, ikut membantu kemajuan sistem operasi Linux. Bersamaan dengan itu, banyak pengembang yang sedang bekerja untuk memindahkan berbagai aplikasi ke Linux (dapat berjalan di Linux).

Masalah utama yang dihadapi Linux dahulu adalah *interface* yang berupa teks (text based interface). Ini membuat orang awam tidak tertarik menggunakan Linux karena harus dipelajari terlebih dahulu untuk dapat dimengerti cara penggunaannya (tidak user-friendly). Tetapi keadaan ini sudah mulai berubah dengan kehadiran KDE dan GNOME. Keduanya memiliki tampilan desktop yang menarik sehingga mengubah persepsi dunia tentang Linux.

Linux di negara-negara berkembang mengalami kemajuan yang sangat pesat. Harga perangkat lunak (misalkan sebuah sistem operasi) bisa mencapai US $100 atau lebih. Di negara yang rata-rata penghasilan per tahun adalah US $200-300, US $100 sangatlah besar. Dengan adanya Linux, semua berubah. Karena Linux dapat digunakan pada komputer yang kuno, dia menjadi alternatif cocok bagi komputer beranggaran kecil. Di negara-negara Asia, Afrika, dan Amerika Latin, Linux adalah jalan keluar bagi penggemar komputer.

Pemanfaatan Linux juga sudah diterapkan pada *supercomputer*. Diberikan beberapa contoh:

- **The Tetragrid.** Sebuah *mega computer* dari Amerika yang dapat menghitung lebih dari 13 trilyun kalkulasi per detik (13.6 TeraFLOPS – *FLoating Operations Per Second*). Tetragrid dapat dimanfaatkan untuk mencari solusi dari masalah matematika kompleks dan simulasi, dari astronomi dan riset kanker hingga ramalan cuaca.
- **Evolocity.** Juga dari Amerika, dapat berjalan dengan kecepatan maksimum 9.2 TeraFLOPS, menjadikannya sebagai salah satu dari lima *supercomputer* tercepat di dunia.

Jika melihat ke depan, kemungkinan Linux akan menjadi sistem operasi yang paling dominan bukanlah suatu hal yang mustahil. Karena semua kelebihan yang dimilikinya, setiap hari semakin banyak orang di dunia yang mulai berpaling ke Linux.

Gambar 9.1. Logo Linux.



Logo Linux adalah sebuah pinguin. Tidak seperti produk komersial sistem operasi lainnya, Linux tidak memiliki simbol yang terlihat hebat. Melainkan Tux, nama pinguin tersebut, memperlihatkan sikap santai dari gerakan Linux. Logo yang lucu ini memiliki sejarah yang unik. Awalnya, tidak ada logo yang dipilih untuk Linux, namun pada waktu Linus (pencetus Linux) berlibur, ia pergi ke daerah selatan. Disanalah dia bertemu seekor pinguin yang pendek cerita menggigit jarinya. Kejadian yang lucu ini merupakan awal terpilihnya pinguin sebagai logo Linux.

Tux adalah hasil karya seniman Larry Ewing pada waktu para pengembang merasa bahwa Linux sudah memerlukan sebuah logo (1996), dan nama yang terpilih adalah dari usulan James Hughes yaitu "(T)orvalds (U)ni(X) -- TUX!". Lengkaplah sudah logo dari Linux, yaitu seekor pinguin bernama Tux.

Hingga sekarang logo Linux yaitu Tux sudah terkenal ke berbagai penjuru dunia. Orang lebih mudah mengenal segala produk yang berbau Linux hanya dengan melihat logo yang unik nan lucu hasil kerjasama seluruh komunitas Linux di seluruh dunia.

9.5. Prinsip Rancangan Linux

Dalam rancangan keseluruhan, Linux menyerupai implementasi UNIX nonmicrokernel yang lain. Ia adalah sistem yang *multiuser*, *multitasking* dengan seperangkat lengkap alat-alat yang kompatibel dengan UNIX. Sistem berkas Linux mengikuti semantik tradisional UNIX, dan model jaringan standar UNIX diimplementasikan secara keseluruhan. Ciri internal rancangan Linux telah dipengaruhi oleh sejarah perkembangan sistem operasi ini.

Walaupun Linux dapat berjalan pada berbagai macam platform, pada awalnya dia dikembangkan secara eksklusif pada arsitektur PC. Sebagian besar dari pengembangan awal tersebut dilakukan oleh peminat individual, bukan oleh fasilitas riset yang memiliki dana besar, sehingga dari awal Linux berusaha untuk memasukkan fungsionalitas sebanyak mungkin dengan dana yang sangat terbatas. Saat ini, Linux dapat berjalan baik pada mesin *multiprocessor* dengan *main memory* yang sangat besar dan ukuran *disk space* yang juga sangat besar, namun tetap mampu beroperasi dengan baik dengan jumlah RAM yang lebih kecil dari 4 MB.

Akibat dari semakin berkembangnya teknologi PC, kernel Linux juga semakin lengkap dalam mengimplementasikan fungsi UNIX. Tujuan utama perancangan Linux adalah cepat dan efisien, tetapi akhir-akhir ini konsentrasi perkembangan Linux lebih pada tujuan rancangan yang ketiga yaitu standarisasi. Standar POSIX terdiri dari kumpulan spesifikasi dari beberapa aspek yang berbeda kelakuan sistem operasi. Ada dokumen POSIX untuk fungsi sistem operasi biasa dan untuk ekstensi seperti proses untuk thread dan operasi *real-time*. Linux dirancang agar sesuai dengan dokumen POSIX yang relevan. Sedikitnya ada dua distribusi Linux yang sudah memperoleh sertifikasi ofisial POSIX.

Karena Linux memberikan antarmuka standar ke programer dan pengguna, Linux tidak membuat banyak kejutan kepada siapa pun yang sudah terbiasa dengan UNIX. Namun interface pemrograman Linux merujuk pada semantik SVR4 UNIX daripada kelakuan BSD. Kumpulan pustaka yang berbeda tersedia untuk mengimplementasi semantik BSD di tempat dimana kedua kelakuan sangat

berbeda.

Ada banyak standar lain di dunia UNIX, tetapi sertifikasi penuh dari Linux terhadap standar lain UNIX terkadang menjadi lambat karena lebih sering tersedia dengan harga tertentu (tidak secara bebas), dan ada harga yang harus dibayar jika melibatkan sertifikasi persetujuan atau kecocokan sebuah sistem operasi terhadap kebanyakan standar. Bagaimana pun juga mendukung aplikasi yang luas adalah penting untuk suatu sistem operasi, sehingga sehingga standar implementasi merupakan tujuan utama pengembangan Linux, walaupun implementasinya tidak sah secara formal. Selain standar POSIX, Linux saat ini mendukung ekstensi thread POSIX dan subset dari ekstensi untuk kontrol proses *real-time* POSIX.

Sistem Linux terdiri dari tiga bagian kode penting:

- **Kernel.** Bertanggung-jawab memelihara semua abstraksi penting dari sistem operasi, termasuk hal-hal seperti memori virtual dan proses-proses.
- **Pustaka sistem.** Menentukan kumpulan fungsi standar dimana aplikasi dapat berinteraksi dengan kernel, dan mengimplementasi hampir semua fungsi sistem operasi yang tidak memerlukan hak penuh atas kernel.
- **Utilitas sistem.** Program yang melakukan pekerjaan manajemen secara individual.

Kernel

Walaupun berbagai sistem operasi modern telah mengadopsi suatu arsitektur message-passing untuk kernel internal mereka, Linux tetap memakai model historis UNIX: kernel diciptakan sebagai biner yang tunggal dan monolitis. Alasan utamanya adalah untuk meningkatkan kinerja, karena semua struktur data dan kode kernel disimpan dalam satu address space, alih konteks tidak diperlukan ketika sebuah proses memanggil sebuah fungsi sistem operasi atau ketika interupsi perangkat keras dikirim. Tidak hanya penjadwalan inti dan kode memori virtual yang menempati address space ini, tetapi juga semua kode kernel, termasuk semua *device drivers*, sistem berkas, dan kode jaringan, hadir dalam satu *address space* yang sama.

Kernel Linux membentuk inti dari sistem operasi Linux. Dia menyediakan semua fungsi yang diperlukan untuk menjalankan proses, dan menyediakan layanan sistem untuk memberikan pengaturan dan proteksi akses ke sumber daya perangkat keras. Kernel mengimplementasi semua fitur yang diperlukan supaya dapat bekerja sebagai sistem operasi. Namun, jika sendiri, sistem operasi yang disediakan oleh kernel Linux sama sekali tidak mirip dengan sistem UNIX. Dia tidak memiliki banyak fitur ekstra UNIX, dan fitur yang disediakan tidak selalu dalam format yang diharapkan oleh aplikasi UNIX. Interface dari sistem operasi yang terlihat oleh aplikasi yang sedang berjalan tidak ditangani langsung oleh kernel, akan tetapi aplikasi membuat panggilan (calls) ke perpustakaan sistem, yang kemudian memanggil layanan sistem operasi yang dibutuhkan.

Pustaka Sistem

Pustaka sistem menyediakan berbagai tipe fungsi. Pada level yang paling sederhana, mereka membolehkan aplikasi melakukan permintaan pada layanan sistem kernel. Membuat suatu system call melibatkan transfer kontrol dari mode pengguna yang tidak penting ke mode kernel yang penting; rincian dari transfer ini berbeda pada masing-masing arsitektur. Pustaka bertugas untuk mengumpulkan argumen system-call dan, jika perlu, mengatur argumen tersebut dalam bentuk khusus yang diperlukan untuk melakukan system call.

Pustaka juga dapat menyediakan versi lebih kompleks dari system call dasar. Contohnya, fungsi buffered file-handling dari bahasa C semuanya diimplementasikan dalam pustaka sistem, yang memberikan kontrol lebih baik terhadap berkas M/K daripada system call kernel dasar. pustaka juga menyediakan rutin yang tidak ada hubungan dengan system call, seperti algoritma penyusunan (sorting), fungsi matematika, dan rutin manipulasi string (string manipulation). Semua fungsi yang diperlukan untuk mendukung jalannya aplikasi UNIX atau POSIX diimplementasikan dalam pustaka sistem.

Utilitas Sistem

Sistem Linux mengandung banyak program-program *pengguna-mode*: utilitas sistem dan utilitas pengguna. Utilitas sistem termasuk semua program yang diperlukan untuk menginisialisasi sistem, seperti program untuk konfigurasi alat jaringan (*network device*) atau untuk load modul kernel. Program server yang berjalan secara kontinu juga termasuk sebagai utilitas sistem; program semacam ini mengatur permintaan pengguna login, koneksi jaringan yang masuk, dan antrian printer.

Tidak semua utilitas standar melakukan fungsi administrasi sistem yang penting. Lingkungan pengguna UNIX mengandung utilitas standar dalam jumlah besar untuk melakukan pekerjaan sehari-hari, seperti membuat daftar direktori, memindahkan dan menghapus file, atau menunjukkan isi dari sebuah file. Utilitas yang lebih kompleks dapat melakukan fungsi text-processing, seperti menyusun data tekstual atau melakukan pattern searches pada input teks. Jika digabung, utilitas-utilitas tersebut membentuk kumpulan alat standar yang diharapkan oleh pengguna pada sistem UNIX mana saja; walaupun tidak melakukan fungsi sistem operasi apa pun, utilitas tetap merupakan bagian penting dari sistem Linux dasar.

9.6. Modul Kernel Linux

Pengertian Modul Kernel Linux

Modul kernel Linux adalah bagian dari kernel Linux yang dapat dikompilasi, dipanggil dan dihapus secara terpisah dari bagian kernel lainnya saat dibutuhkan. Modul kernel dapat menambah fungsionalitas kernel tanpa perlu me-reboot sistem. Secara teori tidak ada yang dapat membatasi apa yang dapat dilakukan oleh modul kernel. Kernel modul dapat mengimplementasikan antara lain *device driver*, sistem berkas, protokol jaringan.

Modul kernel Linux memudahkan pihak lain untuk meningkatkan fungsionalitas kernel tanpa harus membuat sebuah kernel monolitik dan menambahkan fungsi yang mereka butuhkan langsung ke dalam image dari kernel. Selain hal tersebut akan membuat ukuran kernel menjadi lebih besar, kekurangan lainnya adalah mereka harus membangun dan me-reboot kernel setiap saat hendak menambah fungsi baru. Dengan adanya modul maka setiap pihak dapat dengan mudah menulis fungsi-fungsi baru dan bahkan mendistribusikannya sendiri, di luar GPL.

Kernel modul juga memberikan keuntungan lain yaitu membuat sistem Linux dapat dinyalakan dengan kernel standar yang minimal, tanpa tambahan *device driver* yang ikut dipanggil. Device driver yang dibutuhkan dapat dipanggil kemudian secara eksplisit maupun secara otomatis saat dibutuhkan.

Terdapat tiga komponen untuk menunjang modul kernel Linux. Ketiga komponen tersebut adalah manajemen modul, registrasi driver, dan mekanisme penyelesaian konflik. Berikut akan dibahas ketiga komponen pendukung tersebut.

Manajemen Modul Kernel Linux

Manajemen modul akan mengatur pemanggilan modul ke dalam memori dan berkomunikasi dengan bagian lainnya dari kernel. Memanggil sebuah modul tidak hanya memasukkan isi binarinya ke dalam memori kernel, namun juga harus dipastikan bahwa setiap rujukan yang dibuat oleh modul ke simbol kernel atau pun titik masukan diperbaharui untuk menunjuk ke lokasi yang benar di alamat kernel. Linux membuat tabel simbol internal di kernel. Tabel ini tidak memuat semua simbol yang didefinisikan di kernel saat kompilasi, namun simbol-simbol tersebut harus diekspor secara eksplisit oleh kernel. Semua hal ini diperlukan untuk penanganan rujukan yang dilakukan oleh modul terhadap simbol-simbol.

Pemanggilan modul dilakukan dalam dua tahap. Pertama, utilitas pemanggil modul akan meminta kernel untuk mereservasi tempat di memori virtual kernel untuk modul tersebut. Kernel akan memberikan alamat memori yang dialokasikan dan utilitas tersebut dapat menggunakannya untuk memasukkan kode mesin dari modul tersebut ke alamat pemanggilan yang tepat. Berikutnya system calls akan membawa modul, berikut setiap tabel simbol yang hendak diekspor, ke kernel. Dengan

demikian modul tersebut akan berada di alamat yang telah dialokasikan dan tabel simbol milik kernel akan diperbaharui.

Komponen manajemen modul yang lain adalah peminta modul. Kernel mendefinisikan antarmuka komunikasi yang dapat dihubungi oleh program manajemen modul. Saat hubungan tercipta, kernel akan menginformasikan proses manajemen kapan pun sebuah proses meminta *device driver*, sistem berkas, atau layanan jaringan yang belum terpanggil dan memberikan manajer kesempatan untuk memanggil layanan tersebut. Permintaan layanan akan selesai saat modul telah terpanggil. Manajer proses akan memeriksa secara berkala apakah modul tersebut masih digunakan, dan akan menghapusnya saat tidak diperlukan lagi.

Registrasi Driver

Untuk membuat modul kernel yang baru dipanggil berfungsi, bagian dari kernel yang lain harus mengetahui keberadaan dan fungsi baru tersebut. Kernel membuat tabel dinamis yang berisi semua driver yang telah diketahuinya dan menyediakan serangkaian routines untuk menambah dan menghapus driver dari tabel tersebut. Routines ini yang bertanggung-jawab untuk mendaftarkan fungsi modul baru tersebut.

Hal-hal yang masuk dalam tabel registrasi adalah:
- *device driver*
- sistem berkas
- protokol jaringan
- format binari

Resolusi Konflik

Keanekaragaman konfigurasi perangkat keras komputer serta driver yang mungkin terdapat pada sebuah komputer pribadi telah menjadi suatu masalah tersendiri. Masalah pengaturan konfigurasi perangkat keras tersebut menjadi semakin kompleks akibat dukungan terhadap *device driver* yang modular, karena *device* yang aktif pada suatu saat bervariasi.

Linux menyediakan sebuah mekanisme penyelesaian masalah untuk membantu arbitrasi akses terhadap perangkat keras tertentu. Tujuan mekanisme tersebut adalah untuk mencegah modul berebut akses terhadap suatu perangkat keras, mencegah autoprobes mengusik keberadaan driver yang telah ada, menyelesaikan konflik di antara sejumlah driver yang berusaha mengakses perangkat keras yang sama.

Kernel membuat daftar alokasi sumber daya perangkat keras. Ketika suatu driver hendak mengakses sumber daya melalui M/K port, jalur interrupt, atau pun kanal DMA, maka driver tersebut diharapkan mereservasi sumber daya tersebut pada basis data kernel terlebih dahulu. Jika reservasinya ditolak akibat ketidaktersediaan sumber daya yang diminta, maka modul harus memutuskan apa yang hendak dilakukan selanjutnya. Jika tidak dapat melanjutkan, maka modul tersebut dapat dihapus.

9.7. Rangkuman

Linux adalah sebuah sistem operasi yang sangat mirip dengan sistem-sistem UNIX, karena memang tujuan utama desain dari proyek Linux adalah UNIX compatible. Sejarah Linux dimulai pada tahun 1991, ketika mahasiswa Universitas Helsinki, Finlandia bernama Linus Benedict Torvalds menulis Linux, sebuah kernel untuk prosesor 80386, prosesor 32-bit pertama dalam kumpulan CPU intel yang cocok untuk PC. Dalam rancangan keseluruhan, Linux menyerupai implementasi UNIX nonmicrokernel yang lain. Ia adalah sistem yang multiuser, multitasking dengan seperangkat lengkap alat-alat yang compatible dengan UNIX. Sistem berkas Linux mengikuti semantik tradisional UNIX, dan model jaringan standar UNIX diimplementasikan secara keseluruhan. Ciri internal desain Linux telahdipengaruhi oleh sejarah perkembangan sistem operasi ini.

Kernel Linux terdistribusi di bawah Lisensi Publik Umum GNU (GPL), di mana peraturannya disusun oleh Free Software Foundation (FSF). Implikasi utama terhadap peraturan ini adalah bahwa siapa saja boleh menggunakan Linux atau membuat modifikasi, namun tidak boleh membuatnya

menjadi milik sendiri.

Perkembangan sistem operasi Linux sangat cepat karena didukung pengembang di seluruh dunia yang akan selalu memperbaiki segala fiturnya. Di negara-negara berkembang, Linux mengalami kemajuan yang sangat pesat karena dengan menggunakan Linux mereka dapat menghemat anggaran. Linux juga telah diterapkan pada supercomputer.

Prinsip rancangan Linux merujuk pada implementasi agar kompatibel dengan UNIX yang merupakan sistem multiuser dan multitasking. Sistem Linux terdiri dari tiga bagian penting, yaitu kernel, pustaka, dan utilitas. Kernel merupakan inti dari sistem operasi Linux. Pustaka sistem Linux menyediakan berbagai fungsi yang diperlukan untuk menjalankan aplikasi UNIX atau POSIX.

Modul kernel Linux adalah bagian dari kernel Linux yang dapat dikompilasi, dipanggil dan dihapus secara terpisah dari bagian kernel lainnya. Terdapat tiga komponen yang menunjang kernel Linux, di antaranya adalah Manajemen Modul Kernel Linux, Registrasi Driver, dan Resolusi Konflik.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[WEBJones2003] Dave Jones. 2003. *The post-halloween Document v0.48 (aka, 2.6 - what to expect) – http://zenii.linux.org.uk/ ~davej/ docs/ post-halloween-2.6.txt* . Diakses 29 Mei 2006.

Bagian III. Proses dan Penjadwalan

Proses, Penjadwalan, dan Sinkronisasi merupakan trio yang saling berhubungan, sehingga seharusnya tidak dipisahkan. Bagian ini akan membahas Proses dan Penjadwalannya, kemudian bagian berikutnya akan membahas Proses dan Sinkronisasinya.

Bab 10. Konsep Proses

10.1. Pendahuluan

Proses didefinisikan sebagai program yang sedang dieksekusi. Menurut Silberschatz proses tidak hanya sekedar suatu kode program (*text section*), melainkan meliputi beberapa aktivitas yang bersangkutan seperti *program counter* dan *stack*. Sebuah proses juga melibatkan *stack* yang berisi data sementara (parameter fungsi/metode, *return address*, dan variabel lokal) dan data section yang menyimpan variabel-variabel global. Tanenbaum juga berpendapat bahwa proses adalah sebuah program yang dieksekusi yang mencakup *program counter*, register, dan variabel di dalamnya.

Keterkaitan hubungan antara proses dengan Sistem Operasi terlihat dari cara Sistem Operasi menjalankan/mengeksekusi proses. Sistem Operasi mengeksekusi proses dengan dua cara yaitu *Batch System* yang mengeksekusi *jobs* dan *Time-shared System* yang mengatur pengeksekusian program pengguna (*user*) atau *tasks*. Bahkan pada sistem pengguna tunggal (*single user*) pun seperti *Microsoft Windows* dan *Mac OS*, seorang pengguna mampu menjalankan beberapa program pada saat yang sama, seperti *Spread Sheet*, *Web Browser*, dan *Web Email*. Bahkan jika pengguna hanya menggunakan satu program saja pada satu waktu, sistem operasi perlu mendukung program internalnya sendiri, seperti manajemen memori. Dengan kata lain, semua aktivitas tersebut adalah identik sehingga kita menyebutnya "Proses".

Program itu sendiri bukanlah sebuah proses. Program merupakan sebuah entitas pasif; serupa isi dari sebuah berkas didalam disket. Sedangkan sebuah proses dalam suatu entitas aktif, dengan sebuah program counter yang menyimpan alamat instruksi selanjut yang akan dieksekusi dan seperangkat sumber daya (*resource*) yang dibutuhkan agar sebuah proses dapat dieksekusi.

Tanenbaum memberikan sebuah analogi untuk membantu membedakan antara program dan proses. Misalkan seorang tukang kue ingin membuat kue ulang tahun untuk anaknya. Tentunya tukang kue tersebut memiliki resep untuk membuat kue tersebut beserta daftar bahan-bahan yang diperlukan untuk membuat kue ulang tahun seperti tepung, gula, bubuk vanilla, dan bahan-bahan lainnya. Dalam analogi ini, resep kue ulang tahun tadi adalah sebuah program, tukang kue ini adalah *CPU* (prosesor), dan bahan-bahan yang diperlukan untuk membuat kue adalah data input. Sedangkan proses adalah kegiatan yang dilakukan oleh si tukang kue mulai dari membaca resep, mengolah bahan, dan memanggang kue hingga akhirnya selesai.

Dua atau lebih buah proses dapat dihubungkan oleh sebuah program yang sama, tetapi tetap saja proses tersebut akan dianggap dua atau lebih proses yang sekuensi/urutan eksekusinya dilakukan secara terpisah. Sebagai contoh, beberapa pengguna dapat menjalankan salinan yang berbeda pada mail program atau pengguna yang sama dapat meminta salinan yang sama dari editor program. Tiap proses ini adalah proses yang terpisah, dan walaupun bagian text section adalah sama, data section-nya bervariasi. Sudah umum jika terdapat suatu proses yang menghasilkan beberapa proses lainnnya ketika proses bekerja. Hal ini akan dijelaskan pada bagian berikutnya.

10.2. Pembentukan Proses

Saat komputer berjalan, terdapat banyak proses yang berjalan secara bersamaan. Sebuah proses dibuat melalui *system call* `create-process` membentuk proses turunan (*child process*) yang dilakukan oleh proses induk *parent process*. Proses turunan tersebut juga mampu membuat proses baru sehingga kesemua proses-proses ini pada akhirnya membentuk pohon proses.

Ketika sebuah proses dibuat maka proses tersebut dapat memperoleh sumber-daya seperti "waktu CPU", "memori", "berkas" atau perangkat "M/K". Sumber daya ini dapat diperoleh langsung dari Sistem Operasi, dari Proses Induk yang membagi-bagikan sumber daya kepada setiap proses turunannnya, atau proses turunan dan proses induk berbagi sumber-daya yang diberikan Sistem Operasi.

Ada dua kemungkinan bagaimana jalannya (*running*) proses induk dan turunan berjalan (*running*). Proses-proses tersebut berjalan secara konkuren atau proses induk menunggu sampai beberapa/seluruh proses turunannya selesai berjalan. Juga terdapat dua kemungkinan dalam

pemberian ruang alamat (*address space*) proses yang baru. Proses turunan dapat merupakan duplikasi.

Sistem operasi *UNIX* mempunyai *system call fork* yang berfungsi untuk membuat proses baru. Proses yang memanggil *system call fork* ini akan dibagi jadi dua, proses induk dan proses turunan yang identik. Analoginya seperti pembelahan sel, dimana satu sel membelah jadi dua sel yang identik. Proses induk dan turunan independen satu sama lain dan berjalan bersamaan. *Return code* dari *system call* ini adalah suatu *integer*. Untuk proses anak *return code*-nya adalah 0 sementara untuk proses induk *return code*-nya adalah nomor identifikasi proses (*PID*) dari turunannya. Ada juga *system call exec* yang berguna untuk membuat proses turunan yang terbentuk memiliki instruksi yang berbeda dengan proses induknya. Dengan kata lain, proses induk dan proses turunan tidak lagi identik tapi masing-masing punya instruksi berbeda.

Bila *UNIX* menggunakan kemungkinan pertama (proses baru merupakan duplikasi induknya) maka sistem operasi *DEC VMS* menggunakan kemungkinan kedua dalam pembuatan proses baru yaitu setiap proses baru memiliki program yang diload ke ruang alamatnya dan melaksanakan program tersebut. Sedangkan sistem operasi *Microsoft Windows NT* mendukung dua kemungkinan tersebut. Ruang alamat proses induk dapat diduplikasi atau proses induk meminta sistem operasi untuk me-*load* program yang akan dijalankan proses baru ke ruang alamatnya.

10.3. Terminasi Proses

Suatu proses diterminasi ketika proses tersebut telah selesai mengeksekusi perintah terakhir serta meminta sistem operasi untuk menghapus perintah tersebut dengan menggunakan *system call* `exit`. Pada saat itu, proses dapat mengembalikan data keluaran kepada proses induk-nya melalui *system call* `wait`. Semua sumber-daya yang digunakan oleh proses akan dialokasikan kembali oleh sistem operasi agar dapat dimanfaatkan oleh proses lain.

Suatu proses juga dapat diterminasi dengan sengaja oleh proses lain melalui *system call* `abort`. Biasanya proses induk melakukan hal ini pada turunannya. Alasan terminasi tersebut seperti:

- Turunan melampaui penggunaan sumber-daya yang telah dialokasikan. Dalam keadaan ini, proses induk perlu mempunyai mekanisme untuk memeriksa status turunannya-nya.
- Task yang ditugaskan kepada turunan tidak lagi diperlukan.
- Proses induk selesai, dan sistem operasi tidak mengizinkan proses turunan untuk tetap berjalan. Jadi, semua proses turunan akan berakhir pula. Hal ini yang disebut *cascading termination*.

10.4. Status Proses

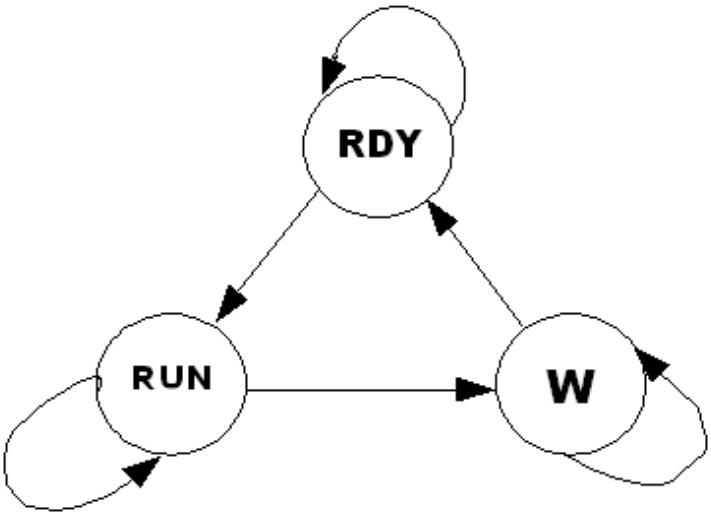
Sebuah proses dapat memiliki tiga status utama yaitu:

- *Running*: status yang dimiliki pada saat instruksi-instruksi dari sebuah proses dieksekusi.
- *Waiting*: status yang dimiliki pada saat proses menunggu suatu sebuah *event* seperti proses M/K.
- *Ready*: status yang dimiliki pada saat proses siap untuk dieksekusi oleh prosesor.

Terdapat dua status tambahan, yaitu saat pembentukan dan terminasi:

- *New*: status yang dimiliki pada saat proses baru saja dibuat.
- *Terminated*: status yang dimiliki pada saat proses telah selesai dieksekusi.

Gambar 10.1. Status Utama Proses



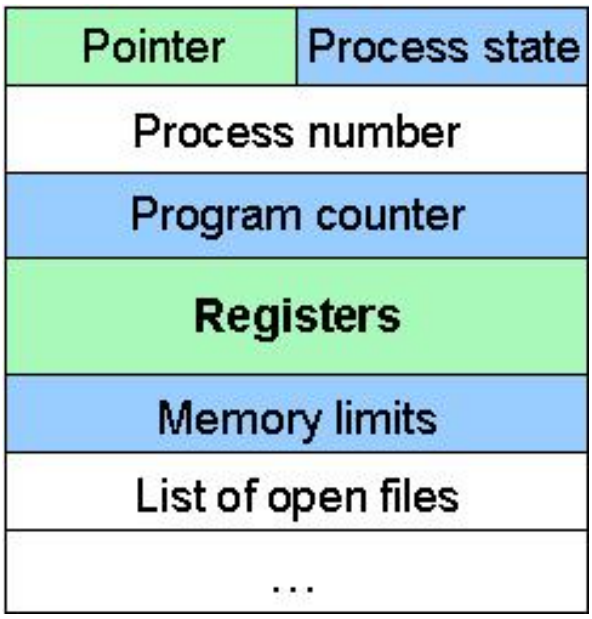
RDY (Ready), RUN (Running), W (Wait).

Hanya satu proses yang dapat berjalan pada prosesor mana pun pada satu waktu. Namun, banyak proses yang dapat berstatus *Ready* atau *Waiting*. Ada tiga kemungkinan bila sebuah proses memiliki status *Running*:

- Jika program telah selesai dieksekusi maka status dari proses tersebut akan berubah menjadi *Terminated*.
- Jika waktu yang disediakan oleh OS untuk proses tersebut sudah habis maka akan terjadi *interrupt* dan proses tersebut kini berstatus *Ready*.
- Jika suatu event terjadi pada saat proses dieksekusi (seperti ada permintaan M/K) maka proses tersebut akan menunggu *event* tersebut selesai dan proses berstatus *Waiting*.

10.5. *Process Control Block*

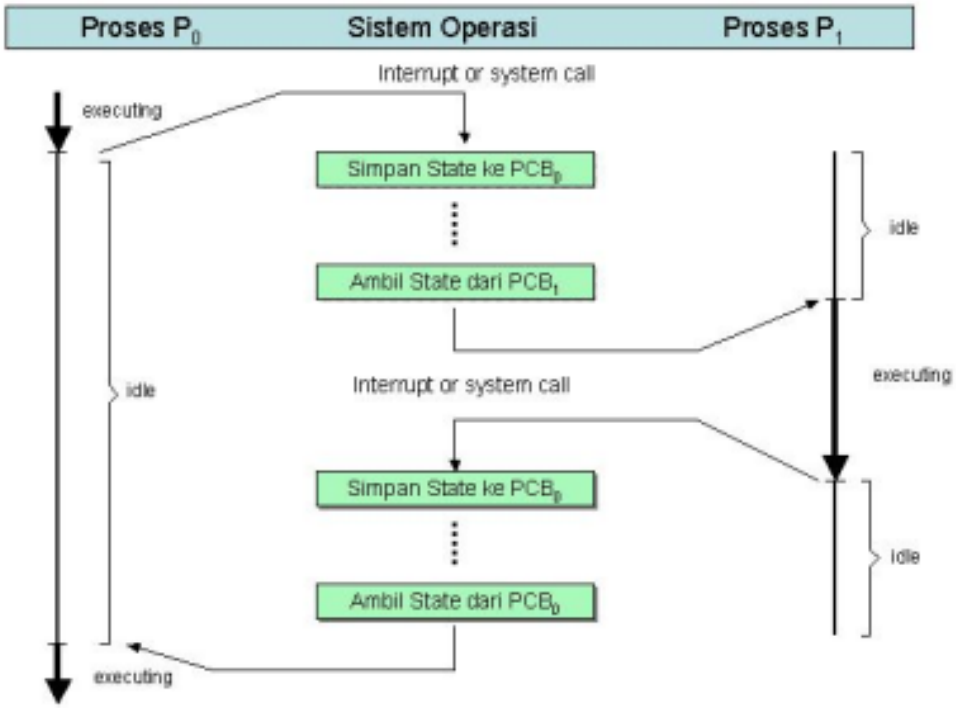
Gambar 10.2. *Process Control Block*



Setiap proses digambarkan dalam sistem operasi oleh sebuah ***process control block*** (PCB) - juga disebut sebuah *control block*. Sebuah PCB ditunjukkan dalam Gambar 10.2, "*Process Control Block*". PCB berisikan banyak bagian dari informasi yang berhubungan dengan sebuah proses yang spesifik, termasuk hal-hal di bawah ini:

- Status proses: status mungkin, *new*, *ready*, *running*, *waiting*, *halted*, dan juga banyak lagi.
- *Program counter*: suatu *stack* yang berisi alamat dari instruksi selanjutnya untuk dieksekusi untuk proses ini.
- *CPU register*: Register bervariasi dalam jumlah dan jenis, tergantung pada rancangan komputer. Register tersebut termasuk *accumulator*, register indeks, *stack pointer*, *general-purposes register*, ditambah *code information* pada kondisi apa pun. Beserta dengan *program counter*, keadaan/status informasi harus disimpan ketika gangguan terjadi, untuk memungkinkan proses tersebut berjalan/bekerja dengan benar setelahnya (lihat Gambar 10.3, "Status Proses").
- Informasi manajemen memori: Informasi ini dapat termasuk suatu informasi sebagai nilai dari dasar dan batas register, tabel page/halaman, atau tabel segmen tergantung pada sistem memori yang digunakan oleh sistem operasi (lihat Bagian V, "Memori").
- Informasi pencatatan: Informasi ini termasuk jumlah dari CPU dan waktu riil yang digunakan, batas waktu, jumlah akun jumlah *job* atau proses, dan banyak lagi.
- Informasi status M/K: Informasi termasuk daftar dari perangkat M/K yang di gunakan pada proses ini, suatu daftar berkas-berkas yang sedang diakses dan banyak lagi.
- PCB hanya berfungsi sebagai tempat penyimpanan informasi yang dapat bervariasi dari proses yang satu dengan yang lain.

Gambar 10.3. Status Proses



10.6. Hirarki Proses

Sistem operasi yang mendukung konsep proses harus menyediakan beberapa cara untuk membuat seluruh proses yang dibutuhkan. Pada sistem yang simple atau sistem yang didisain untuk menjalankan applikasi sederhana, sangat mungkin untuk mendapatkan seluruh proses yang akan dibutuhkan itu, terjadi pada waktu sistem dimulai. Pada kebanyakan system bagaimanapun juga beberapa cara dibutuhkan untuk membuat dan menghacurkan selama operasi.

Hieraki proses biasanya tidak sangat deep (lebih dari tiga tingkatan adalah tidak wajar), dimana

hierarki berkas umumnya empat atau lima. Hierarki proses typically short-lived, kebanyakan umumnya cuma beberapa menit saja, tapi hierarki direktorinya dapat exist sampai bertahun-tahun. Pemilikan dan perlindungan juga membedakan antara proses dan berkas-berkas. Biasanya hanya proses induk yang dapat mengendalikan atau bahkan mengakses sebuah proses turunan, tapi mekanismenya membolehkan berkas-berkas dan direktori dibaca oleh grup daripada hanya pemilik.

Pada UNIX, proses-proses dibuat dengan FORK system call, yang membuat salinan identik dari calling proses. Setelah fork di panggil, induk meneruskan prosesnya dan pararel dengan proses anak. UNIX menyebutnya "proses grup".

10.7. Rangkuman

Sebuah proses adalah suatu program yang sedang dieksekusi. Proses lebih dari sebuah kode program tetapi juga mencakup **program counter**, **stack**, dan sebuah **data section**. Dalam pengeksekusiannya sebuah proses juga memiliki status yang mencerminkan keadaan dari proses tersebut. Status dari proses dapat berubah-ubah setiap saat sesuai dengan kondisinya. Status tersebut mungkin menjadi satu dari lima status berikut: *new*, *ready*, *running*, *waiting*, atau *terminated*. Setiap proses juga direpresentasikan oleh *Proces Control Block* (PCB) yang menyimpan segala informasi yang berkaitan dengan proses tersebut.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

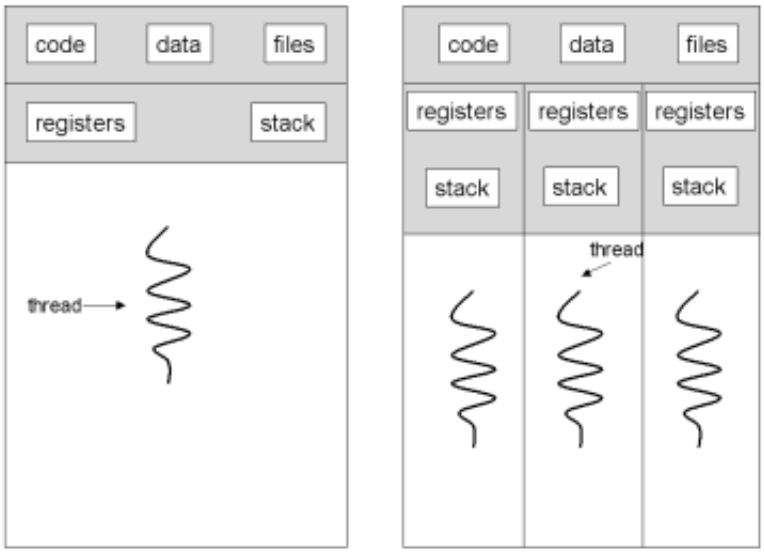
Bab 11. Konsep *Thread*

11.1. Pendahuluan

Sejauh ini, proses merupakan sebuah program yang mengeksekusi *thread* tunggal. Kendali *thread* tunggal ini hanya memungkinkan proses untuk menjalankan satu tugas pada satu waktu. Banyak sistem operasi modern telah memiliki konsep yang dikembangkan agar memungkinkan sebuah proses untuk mengeksekusi *multi-threads*. Umpamanya, secara bersamaan mengetik dan menjalankan pemeriksaan ejaan didalam proses yang sama.

Thread merupakan unit dasar dari penggunaan CPU, yang terdiri dari Thread_ID, *program counter*, *register set*, dan *stack*. Sebuah *thread* berbagi *code section*, *data section*, dan sumber daya sistem operasi dengan Thread lain yang dimiliki oleh proses yang sama. Thread juga sering disebut *lightweight process*. Sebuah proses tradisional atau *heavyweight process* mempunyai *thread* tunggal yang berfungsi sebagai pengendali. Perbedaannya ialah proses dengan *thread* yang banyak – mengerjakan lebih dari satu tugas pada satu satuan waktu.

Gambar 11.1. *Thread*



Pada umumnya, perangkat lunak yang berjalan pada komputer modern dirancang secara *multi-threading*. Sebuah aplikasi biasanya diimplementasi sebagai proses yang terpisah dengan beberapa *thread* yang berfungsi sebagai pengendali. Contohnya sebuah *web browser* mempunyai *thread* untuk menampilkan gambar atau tulisan sedangkan *thread* yang lain berfungsi sebagai penerima data dari network.

Terkadang ada sebuah aplikasi yang perlu menjalankan beberapa tugas yang serupa. Sebagai contohnya sebuah *web server* dapat mempunyai ratusan klien yang mengaksesnya secara *concurrent*. Kalau *web server* berjalan sebagai proses yang hanya mempunyai *thread* tunggal maka ia hanya dapat melayani satu klien pada pada satu satuan waktu. Bila ada klien lain yang ingin mengajukan permintaan maka ia harus menunggu sampai klien sebelumnya selesai dilayani. Solusinya adalah dengan membuat *web server* menjadi *multi-threading*. Dengan ini maka sebuah *web server* akan membuat *thread* yang akan mendengar permintaan klien, ketika permintaan lain diajukan maka *web server* akan menciptakan *thread* lain yang akan melayani permintaan tersebut.

Dewasa ini (2006), banyak sistem operasi yang telah mendukung proses *multithreading*. Setiap sistem operasi memiliki konsep tersendiri dalam pengimplementasiannya. Sistem operasi dapat mendukung *thread* pada tingkatan kernel maupun tingkatan pengguna.

11.2. Keuntungan *Thread*

Keuntungan dari program yang *multithreading* terbagi menjadi empat kategori:

1. **Responsif.** Aplikasi interaktif menjadi tetap responsif meski pun sebagian dari program sedang diblok atau melakukan operasi yang panjang kepada pengguna. Umpamanya, sebuah *thread* dari *web browser* dapat melayani permintaan pengguna sementara *thread* lain berusaha menampilkan gambar.
2. **Berbagi sumber daya.** *thread* berbagi memori dan sumber daya dengan *thread* lain yang dimiliki oleh proses yang sama. Keuntungan dari berbagi kode adalah mengizinkan sebuah aplikasi untuk mempunyai beberapa *thread* yang berbeda dalam lokasi memori yang sama.
3. **Ekonomis.** Pembuatan sebuah proses memerlukan dibutuhkan pengalokasian memori dan sumber daya. Alternatifnya adalah dengan penggunaan *thread*, karena *thread* berbagi memori dan sumber daya proses yang memilikinya maka akan lebih ekonomis untuk membuat dan *context switch thread*. Akan susah untuk mengukur perbedaan waktu antara proses dan *thread* dalam hal pembuatan dan pengaturan, tetapi secara umum pembuatan dan pengaturan proses lebih lama dibandingkan *thread*. Pada Solaris, pembuatan proses lebih lama 30 kali dibandingkan pembuatan *thread*, dan *context switch* proses 5 kali lebih lama dibandingkan *context switch thread*.
4. **Utilisasi arsitektur *multiprocessor*.** Keuntungan dari multithreading dapat sangat meningkat pada arsitektur *multiprocessor*, dimana setiap *thread* dapat berjalan secara pararel di atas processor yang berbeda. Pada arsitektur processor tunggal, CPU menjalankan setiap *thread* secara bergantian tetapi hal ini berlangsung sangat cepat sehingga menciptakan ilusi pararel, tetapi pada kenyataannya hanya satu *thread* yang dijalankan CPU pada satu-satuan waktu (satu-satuan waktu pada CPU biasa disebut *time slice* atau *quantum*).

11.3. *Thread* Pengguna dan Kernel

Thread Pengguna

Thread pengguna didukung kernel serta diimplementasikan dengan pustaka *thread* pada tingkatan pengguna. Pustaka menyediakan fasilitas untuk pembuatan *thread*, penjadwalan *thread*, dan manajemen *thread* tanpa dukungan dari kernel. Karena kernel tidak menyadari user-level *thread* maka semua pembuatan dan penjadwalan *thread* dilakukan dalam ruang pengguna tanpa campur tangan kernel. Oleh karena itu, *thread* pengguna biasanya dapa cepat dibuat dan dikendalikan. Tetapi *thread* pengguna mempunyai kelemahan untuk kernel *thread* tunggal. Salah satu *thread* tingkatan pengguna menjalankan *blocking system call* maka akan mengakibatkan seluruh proses diblok walau pun ada *thread* lain yang dapat jalan dalam aplikasi tersebut. Contoh pustaka *thread* pengguna ialah POSIX Pthreads, Mach C-threads, dan Solaris threads.

Thread Kernel

Thread kernel didukung langsung oleh sistem operasi. Pembuatan, penjadwalan, dan manajemen *thread* dilakukan oleh kernel pada *kernel space*. Karena pengaturan *thread* dilakukan oleh sistem operasi maka pembuatan dan pengaturan kernel *thread* lebih lambat dibandingkan user *thread*. Keuntungannya adalah *thread* diatur oleh kernel, karena itu jika sebuah *thread* menjalankan *blocking system call* maka kernel dapat menjadwalkan *thread* lain di aplikasi untuk melakukan eksekusi. Keuntungan lainnya adalah pada lingkungan *multiprocessor*, kernel dapat menjadwal thread-thread pada processor yang berbeda. Contoh sistem operasi yang mendukung kernel *thread* adalah Windows NT, Solaris, Digital UNIX.

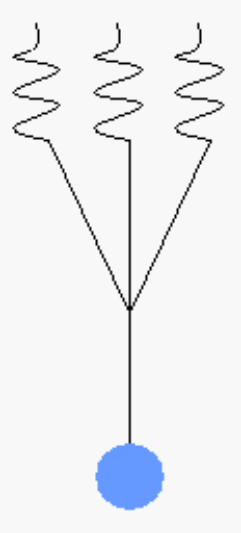
11.4. Model *Multithreading*

Model *Many-to-One*

Model *Many-to-One* memetakan beberapa *thread* tingkatan pengguna ke sebuah *thread* tingkatan kernel. Pengaturan *thread* dilakukan dalam ruang pengguna, sehingga efisien. Hanya satu *thread* pengguna yang dapat mengakses *thread* kernel pada satu saat. Jadi, *multiple thread* tidak dapat berjalan secara pararel pada *multiprocessor*. *Thread* tingkat pengguna yang diimplementasi pada

sistem operasi yang tidak mendukung *thread* kernel menggunakan model *Many-to-One*.

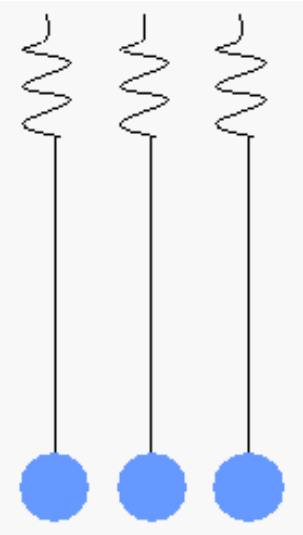
Gambar 11.2. *Many-To-One*



Model *One-to-One*

Model *One-to-One* memetakan setiap *thread* tingkatan pengguna ke *thread* kernel. Ia menyediakan lebih banyak *concurrency* dibandingkan model *Many-to-One*. Keuntungannya sama dengan keuntungan *thread* kernel. Kelemahannya model ini ialah setiap pembuatan *thread* pengguna memerlukan pembuatan *thread* kernel. Karena pembuatan *thread* dapat menurunkan kinerja dari sebuah aplikasi maka implmentasi dari model ini, jumlah *thread* dibatasi oleh sistem. Contoh sistem operasi yang mendukung model *One-to-One* ialah *Windows NT* dan *OS/2*.

Gambar 11.3. *One-To-One*

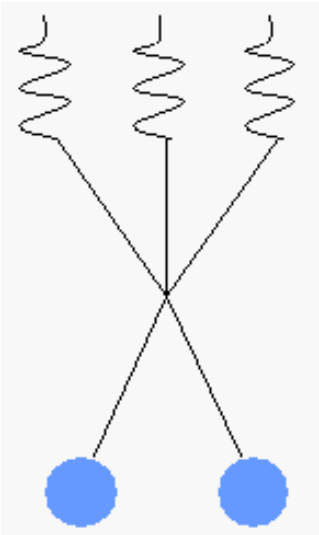


Model *Many-to-Many*

Model *Many-to-Many* memultipleks banyak *thread* tingkatan pengguna ke *thread* kernel yang jumlahnya lebih sedikit atau sama dengan tingkatan pengguna. *thread*. Jumlah *thread* kernel dapat

spesifik untuk sebagian aplikasi atau sebagian mesin. Many-to-One model mengizinkan developer ntuk membuat user *thread* sebanyak yang ia mau tetapi *concurrency* tidak dapat diperoleh karena hanya satu *thread* yang dapat dijadwal oleh kernel pada suatu waktu. One-to-One menghasilkan concurrency yang lebih tetapi developer harus hati-hati untuk tidak menciptakan terlalu banyak *thread* dalam suatu aplikasi (dalam beberapa hal, developer hanya dapat membuat *thread* dalam jumlah yang terbatas). Model *Many-to-Many* tidak menderita kelemahan dari dua model di atas. Developer dapat membuat user *thread* sebanyak yang diperlukan, dan kernel *thread* yang bersangkutan dapat bejalan secara pararel pada *multiprocessor*. Dan juga ketika suatu *thread* menjalankan *blocking system call* maka kernel dapat menjadwalkan *thread* lain untuk melakukan eksekusi. Contoh sistem operasi yang mendukung model ini adalah Solaris, IRIX, dan Digital UNIX.

Gambar 11.4. *Many-To-Many*



11.5. *Fork* dan *Exec System Call*

Terdapat dua kemungkinan dalam sistem UNIX jika *fork* dipanggil oleh salah satu *thread* dalam proses:
1. Semua *thread* diduplikasi.
2. Hanya *thread* yang memanggil *fork*.

Kalau *thread* memanggil *exec System Call* maka program yang dispesifikasi di parameter `exec` akan mengganti keseluruhan proses termasuk *thread* dan LWP. Penggunaan dua versi dari `fork` di atas tergantung dari aplikasi. Kalau `exec` dipanggil seketika sesudah `fork`, maka duplikasi seluruh *thread* tidak dibutuhkan, karena program yang dispesifikasi di parameter `exec` akan mengganti seluruh proses. Pada kasus ini cukup hanya mengganti *thread* yang memanggil *fork*. Tetapi jika proses yang terpisah tidak memanggil *exec* sesudah fork maka proses yang terpisah tersebut hendaknya menduplikasi seluruh *thread*.

11.6. *Cancellation*

Thread cancellation ialah pemberhentian *thread* sebelum tugasnya selesai. Umpama, jika dalam program Java hendak mematikan *Java Virtual Machine* (JVM). Sebelum JVM dimatikan, maka seluruh *thread* yang berjalan harus dihentikan terlebih dahulu. *Thread* yang akan diberhentikan biasa disebut target *thread*.

Pemberhentian target *thread* dapat terjadi melalui dua cara yang berbeda:
1. *Asynchronous cancellation*: suatu *thread* seketika itu juga memberhentikan target *thread*.
2. *Defered cancellation*: target *thread* secara perodik memeriksa apakah dia harus berhenti, cara ini memperbolehkan target *thread* untuk memberhentikan dirinya sendiri secara terurut.

Hal yang sulit dari pemberhentian *thread* ini adalah ketika terjadi situasi dimana sumber daya sudah dialokasikan untuk *thread* yang akan diberhentikan. Selain itu kesulitan lain adalah ketika *thread* yang diberhentikan sedang meng-*update* data yang ia bagi dengan *thread* lain. Hal ini akan menjadi masalah yang sulit apabila digunakan *asynchronous cancellation*. Sistem operasi akan mengambil kembali sumber daya dari *thread* yang diberhentikan tetapi seringkali sistem operasi tidak mengambil kembali semua sumber daya dari *thread* yang diberhentikan.

Alternatifnya adalah dengan menggunakan *deffered cancellation*. Cara kerja dari *deffered cancellation* adalah dengan menggunakan satu *thread* yang berfungsi sebagai pengindikasi bahwa target *thread* hendak diberhentikan. Tetapi pemberhentian hanya akan terjadi jika target *thread* memeriksa apakah ia harus berhenti atau tidak. Hal ini memperbolehkan *thread* untuk memeriksa apakah ia harus berhenti pada waktu dimana ia dapat diberhentikan secara aman yang aman. *Pthread* merujuk tersebut sebagai *cancellation points*.

Pada umumnya sistem operasi memperbolehkan proses atau *thread* untuk diberhentikan secara *asynchronous*. Tetapi Pthread API menyediakan *deferred cancellation*. Hal ini berarti sistem operasi yang mengimplementasikan Pthread API akan mengizinkan *deferred cancellation*.

11.7. Penanganan Sinyal

Sebuah sinyal digunakan di sistem UNIX untuk *notify* sebuah proses kalau suatu peristiwa telah terjadi. Sebuah sinyal dapat diterima secara *synchronous* atau *asynchronous* tergantung dari sumber dan alasan kenapa peristiwa itu memberi sinyal.

Semua sinyal (*asynchronous* dan *synchronous*) mengikuti pola yang sama:
1. Sebuah sinyal dimunculkan oleh kejadian dari suatu persitiwa.
2. Sinyal yang dimunculkan tersebut dikirim ke proses.
3. Sesudah dikirim, sinyal tersebut harus ditangani.

Contoh dari sinyal *synchronous* adalah ketika suatu proses melakukan pengaksesan memori secara ilegal atau pembagian dengan nol, sinyal dimunculkan dan dikirim ke proses yang melakukan operasi tersebut. Contoh dari sinyal *asynchronous* misalnya kita mengirimkan sinyal untuk mematikan proses dengan keyboard (ALT-F4) maka sinyal *asynchronous* dikirim ke proses tersebut. Jadi ketika suatu sinyal dimunculkan oleh peristiwa diluar proses yang sedang berjalan maka proses tersebut menerima sinyal tersebut secara *asynchronous*.

Setiap sinyal dapat ditangani oleh salah satu dari dua penerima sinyal:
1. Penerima sinyal yang merupakan set awal dari sistem operasi.
2. Penerima sinyal yang didefinisikan sendiri oleh user.

Penanganan sinyal pada program yang hanya memakai *thread* tunggal cukup mudah yaitu hanya dengan mengirimkan sinyal ke prosesnya. Tetapi mengirimkan sinyal lebih rumit pada program yang *multithreading*, karena sebuah proses dapat memiliki beberapa *thread*. Secara umum ada empat pilihan kemana sinyal harus dikirim:
1. Mengirimkan sinyal ke *thread* yang dituju oleh sinyal tersebut.
2. Mengirimkan sinyal ke setiap *thread* pada proses tersebut.
3. Mengirimkan sinyal ke *thread* tertentu dalam proses.
4. Menugaskan *thread* khusus untuk menerima semua sinyal yang ditujukan pada proses.

Cara untuk mengirimkan sebuah sinyal tergantung dari jenis sinyal yang dimunculkan. Sebagai contoh sinyal *synchronous* perlu dikirimkan ke *thread* yang memunculkan sinyal tersebut bukan *thread* lain pada proses tersebut. Tetapi situasi dengan sinyal *asynchronous* menjadi tidak jelas. Beberapa sinyal asynchronous seperti sinyal yang berfungsi untuk mematikan proses (contoh: alt-f4) harus dikirim ke semua *thread*. Beberapa versi UNIX yang multithreading mengizinkan *thread* menerima sinyal yang akan ia terima dan menolak sinyal yang akan ia tolak. Karena itu sinyal asynchronouns hanya dikirimkan ke *thread* yang tidak memblok sinyal tersebut. Solaris 2 mengimplementasikan pilihan ke-4 untuk menangani sinyal. Windows 2000 tidak menyediakan fasilitas untuk mendukung sinyal, sebagai gantinya Windows 2000 menggunakan *asynchronous procedure calls* (APCs). Fasilitas APC memperbolehkan user *thread* untuk memanggil fungsi tertentu ketika user *thread* menerima notifikasi peristiwa tertentu.

11.8. *Thread Pools*

Pada *web server* yang *multithreading* ada dua masalah yang timbul:
1. Ukuran waktu yang diperlukan untuk menciptakan *thread* untuk melayani permintaan yang diajukan terlebih pada kenyataannya *thread* dibuang ketika ia seketika sesudah ia menyelesaikan tugasnya.
2. Pembuatan *thread* yang tidak terbatas jumlahnya dapat menurunkan performa dari sistem.

Solusinya adalah dengan penggunaan Thread Pools, cara kerjanya adalah dengan membuat beberapa *thread* pada proses startup dan menempatkan mereka ke *pools*, dimana mereka duduk diam dan menunggu untuk bekerja. Jadi ketika server menerima permintaan maka maka ia akan membangunkan *thread* dari *pool* dan jika *thread* tersedia maka permintaan tersebut akan dilayani. Ketika *thread* sudah selesai mengerjakan tugasnya maka ia kembali ke *pool* dan menunggu pekerjaan lainnya. Bila tidak *thread* yang tersedia pada saat dibutuhkan maka server menunggu sampai ada satu *thread* yang bebas.

Keuntungan *thread* pool:
1. Biasanya lebih cepat untuk melayani permintaan dengan *thread* yang ada dibanding dengan menunggu *thread* baru dibuat.
2. Thread pool membatasi jumlah *thread* yang ada pada suatu waktu. Hal ini pentingpada sistem yang tidak dapat mendukung banyak *thread* yang berjalan secara *concurrent*.

Jumlah *thread* dalam *pool* dapat tergantung dari jumlah CPU dalam sistem, jumlah memori fisik, dan jumlah permintaan klien yang *concurrent*.

11.9. *Thread Specific Data*

Thread yang dimiliki oleh suatu proses memang berbagi data tetapi setiap *thread* mungkin membutuhkan duplikat dari data tertentu untuk dirinya sendiri dalam keadaan tertentu. Data ini disebut *thread-specific* data.

11.10. *Pthreads*

Pthreads merujuk kepada POSIX standard (IEEE 1003.1 c) mendefinisikan sebuah API untuk pembuatan *thread* dan sinkronisasi. Pthreads adalah spesifikasi untuk *thread* dan bukan merupakan suatu implementasi. Desainer sistem operasi boleh mengimplementasikan spesifikasi tersebut dalam berbagai cara yang mereka inginkan. Secara umum pustaka yang mengimplementasikan Pthreads dilarang pada sistem berbasis UNIX seperti Solaris 2. Sistem operasi Windows secara umum belum mendukung Pthreads, walaupun versi *shareware*-nya sudah ada di domain publik.

11.11. Rangkuman

Thread adalah sebuah alur kontrol dari sebuah proses. Suatu proses yang multithreaded mengandung beberapa perbedaan alur kontrol dengan ruang alamat yang sama. Keuntungan dari multithreaded meliputi peningkatan respon dari pengguna, pembagian sumber daya proses, ekonomis, dan kemampuan untuk mengambil keuntungan dari arsitektur multiprosesor. *Thread* tingkat pengguna adalah thread yang tampak oleh programer dan tidak diketahui oleh kernel. *Thread* tingkat pengguna secara tipikal dikelola oleh sebuah pustaka *thread* di ruang pengguna. *Thread* tingkat *kernel* didukung dan dikelola oleh *kernel* sistem operasi. Secara umum, *thread* tingkat pengguna lebih cepat dalam pembuatan dan pengelolaan dari pada *kernel thread*. Ada tiga perbedaan tipe dari model yang berhubungan dengan pengguna dan *kernel thread* yaitu One-to One model, Many-to-One model, Many-to-Many model.
- *Model Many-to-One*: memetakan beberapa pengguna level *thread* hanya ke satu buah *kernel thread*.
- *Model One-to-One*: memetakan setiap *thread* pengguna ke dalam satu *kernel thread* berakhir.
- *Model Many-to-Many*: mengizinkan pengembang untuk membuat *thread* pengguna sebanyak mungkin, konkurensi tidak dapat tercapai karena hanya satu *thread* yang dapat dijadwalkan oleh kernel dalam satu waktu.

Thread cancellation adalah tugas untuk memberhentikan *thread* sebelum ia menyelesaikan tugasnya. *Thread* yang akan diberhentikan disebut target *thread*

Pemberhentian target *thread* dapat terjadi melalui dua cara yang berbeda:
- *Asynchronous cancellation*: suatu *thread* seketika itu juga memberhentikan target thread.
- *Deffered cancellation*: target *thread* secara periodik memeriksa apakah dia harus berhenti, cara ini memperbolehkan target *thread* untuk memberhentikan dirinya sendiri secara terurut.

Thread Pools adalah cara kerja dengan membuat beberapa *thread* pada proses startup dan menempatkan mereka ke pools.

Keuntungan *Thread Pools*
- Biasanya lebih cepat untuk melayani permintaan dengan *thread* yang ada dibanding dengan menunggu *thread* baru dibuat.
- *Thread pool* membatasi jumlah *thread* yang ada pada suatu waktu. Hal ini penting pada sistem yang tidak dapat mendukung banyak *thread* yang berjalan secara *concurrent*

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

Bab 12. *Thread* Java

12.1. Pendahuluan

Java merupakan salah satu dari sedikit bahasa pemerograman yang mendukung *thread* pada tingkatan bahasa untuk pembuatan dan manajemen *thread*. *Thread* dalam Java diatur oleh *Java Virtual Machine* (JVM), sehingga sulit menentukan apakah ada pada tingkatan pengguna atau pun tingkatan kernel.

Setiap program dalam Java memiliki minimal sebuah *thread*, yaitu *main thread* yang merupakan *single-thread* tersendiri di JVM. Java juga menyediakan perintah untuk membuat dan memodifikasi *thread* tambahan sesuai kebutuhan di program.

Java mempunyai pengunaan lain dari *thread* karena Java tidak mempunyai konsep *asynchronous*. Kalau program Java mencoba untuk melakukan koneksi ke server maka ia akan berada dalam keadaan *block state* hingga terbentuk koneksi. Penanggulangan hal ini ialah dengan *thread* yang melakukan koneksi ke server. Kemudian *thread* lain menunggu (tidur) selama beberapa waktu (misalnya 60 detik) kemudian bangun. Ketika waktu tidurnya habis maka ia akan bangun dan memeriksa apakah *thread* yang melakukan koneksi ke server masih mencoba untuk melakukan koneksi ke server, kalau *thread* tersebut masih dalam keadaan mencoba untuk melakukan koneksi ke server maka ia akan melakukan interupsi dan mencegah *thread* tersebut untuk mencoba melakukan koneksi ke server.

12.2. Pembuatan *Thread*

Cara yang lazim digunakan dalam pembuatan *thread* Java ialah dengan `extends class` yang ingin dijadikan *thread*. Cara lainnya ialah dengan cara meng-*implements Interface Runnable*. `Class` yang terbentuk akan menulis ulang `method run()` dari `Interface/class` tersebut, karena mengikuti hukum pewarisan *Interface*.

Perbedaan dari meng-extends *thread* class dan meng-implements Interface Runnable adalah pada hukum pewarisannya. Di Java, suatu class hanya dapat meng-extends satu class saja (pewarisan tunggal). Jika class yang ingin kita jadikan *thread* harus mewarisi (meng-extends) sebuah class, maka kita tidak dapat meng-extends class tersebut dengan class *thread* lagi. Maka tersedia cara lainnya, yaitu dengan meng-implements Interface Runnable. Sehingga dapat disimpulkan di sini, yaitu untuk mendapatkan pewarisan jamak, maka kita dapat menggunakan Interface Runnable.

Contoh pembuatan *thread* dengan membuat obyek baru dari class yang meng-*extends class Thread* di atas. Sebuah obyek yang berasal dari subkelas *Thread* dapat dijalankan sebagai *thread* pengontrol yang terpisah dalam JVM. Membuat obyek dari *class Thread* tidak akan membuat *thread* baru. Hanya dengan *method start thread* baru akan terbentuk. Memanggil *method start* untuk membuat obyek baru akan mengakibatkan dua hal, yaitu:

- Pengalokasian memori dan menginisialisasikan sebuah *thread* baru dalam JVM.
- Memanggil *method run*, yang sudah di-*override*, membuat *thread* dapat dijalankan oleh JVM.

Method run dijalankan jika *method start* dipanggil. Memanggil *method run* secara langsung hanya menghasilkan sebuah *single-thread* tambahan selain *main thread*. Dalam hierarki class Java itu sendiri, sebenarnya class *thread* adalah implementasi langsung dari Interface Runnable. Sehingga, secara tidak langsung, sebuah class yang meng-extends class *thread*, sudah meng-implements Interface Runnable.

Contoh 12.1. Thread

```
import java.util.Random;

class ThreadTest extends Thread {
   ThreadTest(String input) {
      toBePrinted = input;
   }

   public void run() {
      try {
         for (int a=0; a<10; a++) {
            System.out.println(toBePrinted + " " + a);
            sleep(delayTime);
         }
      }
      catch(InterruptedException e) {return;}
   }

   public static void main(String[] args) {
      ThreadTest a = new ThreadTest("Halo");
      ThreadTest2 b = new ThreadTest2("Goodbye");
      Thread c = new Thread(b);
      a.start();
      c.start();
   }

   private int delayTime = ((new Random()).nextInt(3)*700) + 500;
   private String toBePrinted;
}

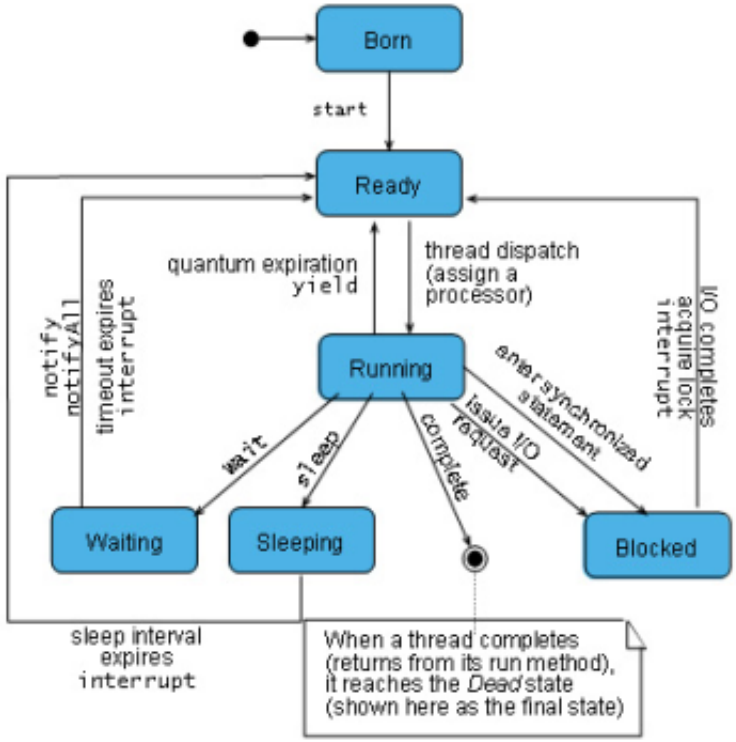
class ThreadTest2 implements Runnable {
   ThreadTest2(String input) {
      toBePrinted = input;
   }

   public void run() {
      try {
         for (int a=0; a<10; a++) {
            System.out.println(toBePrinted + " " + a);
            Thread.sleep(delayTime);
         }
      }
      catch(InterruptedException e) {return;}
   }


   private int delayTime = ((new Random()).nextInt(3)*700) + 500;
   private String toBePrinted;
}
```

12.3. Status *Thread*

Gambar 12.1. Bagan Thread



Sumber : Slide Pemrograman Lanjut 2005

Thread pada Java dapat terdiri dari beberapa status, yaitu:

1. **Baru (*Born*).** Sebuah thread berstatus baru berarti thread tersebut adalah sebuah objek thread yang kosong , belum ada sumber daya sistem yang dialokasikan kepada thread tersebut, oleh karena itu method thread yang dapat dipanggil hanyalah start(), apabila dipanggil method thread yang lain akan menyebabkan IllegalThreadStateException.
2. **Siap (*Ready*).** Sebuah thread dapat memasuki status dapat dijalankan apabila method thread start() telah dijalankan. Method start() mengalokasikan sumber daya sistem yang dibutuhkan oleh thread, menentukan penjadwalan thread tersebut, serta menjalankan method run(), akan tetapi thread tersebut benar-benar berjalan apabila sesuai dengan jadwalnya.
3. **Terhalang (*Blocked*).** Sebuah thread memasuki status tidak dapat dijalankan apabila:
   - Method sleep() dari thread tersebut dijalankan.
   - Method wait() dari thread tersebut dijalankan.
   - Thread tersebut tersangkut dalam proses tunggu M/K sumber daya sistem untuk melakukan operasi input atau pun output.

   Thread tersebut dapat berada pada status dapat dijalankan lagi apabila:
   - Waktu yang ditetapkan oleh method sleep() telah berlalu.
   - Kondisi menunggu dari thread tersebut telah berubah dan telah menerima pesan notify() atau pun notifyAll().
   - Sumber daya sistem yang dibutuhkan telah tersedia dan proses M/K-nya telah selesai dilakukan.

   Perubahan status sebuah thread antara dapat dijalakan dan tidak dapat dijalankan dapat dijelaskan secara garis besar sebagai akibat dari penjadwalan ataupun programmer control.
4. **Mati (*Dead*).** Sebuah thread berada dalam status mati apabila:
   - Method `run()` telah selesai dieksekusi.
   - Method `destroy()` dijalankan, namun method ini tidak membuat thread tersebut melepaskan objek-objeknya yang telah dikunci.

Sebuah proses interupsi tidak membuat sebuah thread berada dalam status mati.

5. **Tidur (*Sleep*).** Thread yang Running dapat memanggil method sleep() di *thread* untuk transisi ke Sleeping untuk suatu periode milisecond yang ditentukan sebagai argumen sleep(). Thread yang Sleeping transisi ke Ready ketika waktu tidur yang ditentukan berlalu. Jika program memanggil method interrupt() di *thread*, interrupt flag di *thread* terpasang dan bergantung state dari *thread*, maka InterruptedException dilempar. Contohnya, jika *thread* sedang Sleeping dan method interrupt() dipanggil di *thread*, method sleep() akan melempar InterruptedException. Dalam kasus ini, *thread* keluar dari Sleeping dan transisi ke Ready sehingga dapat di-dispatch dan mengolah Exception.

12.4. Penggabungan *Thread*

Contoh 12.2. *Thread* Sederhana

```
class JavaThread extends Thread {
   public void run() {
   }
   public JavaThread (String namaThread, ThreadGroup group) {
      super(group, namaThread);
   }
}

public class JavaThreadMain {
   public static void main (String args[]) {
      ThreadGroup grup = new ThreadGroup("Thread Group 1");
      JavaThread t1 = new JavaThread("Thread1", grup);
      JavaThread t2 = new JavaThread("Thread2", grup);
      JavaThread t3 = new JavaThread("Thread3", grup);
      JavaThread t4 = new JavaThread("Thread4", grup);
      t1.start();
      t2.start();
      t3.start();
      t4.start();

      grup.interrupt();// dapat dilakukan bersama-sama
   }
}
```

Multithread programming menjadi suatu masalah besar bagi sebagian programmer, apalagi jika jumlah thread begitu banyak. Solusi sederhana untuk mengatasi masalah ini ialah dengan menggunakan penggabungan thread.

Keuntungan yang dapat diperoleh dari penggunaan *thread* yaitu kita dapat memadukan antara keamanan dan kenyamanan. Sebuah *thread* tidak dapat mengakses induk *threads* dari grup tersebut, sehingga *thread* dapat diisolasi dan dapat dicegah dalam sebuah group dari saling mengubah satu sama lainnya.

Pada Java untuk mengorganisasikan thread kedalam bentuk groups, diwakili dengan *ThreadGroup class*. Sebuah *thread group* terdiri dari beberapa *thread* individual atau *thread group* yang lain, untuk membentuk hirarki *thread*. Hirarki yang terbentuk ialah induk dan turunannya.

Pada Java untuk mengorganisasikan thread kedalam bentuk groups, diwakili dengan ThreadGroup class. Sebuah ThreadGroup terdiri dari beberapa individual threads, atau thread groups yang lain, untuk membentuk sebuah hirarki *thread*. Hirarki yang terbentuk yaitu induk dan turunan.

Pada program Java jika kita ingin mencetak seluruh thread yang ada maka dapat digunakan method getThreadGroup() dari java.lang.Thread. jika kita ingin melihat level paling atas pada hirarcy dari sebuah Threads groups maka kita dapat menggunakan method getParent() dari

java.lang.ThreadGroup. kita juga dapat mendaftar seluruh thread yang ada di thread induk beserta turunannya dengan menggunakan enumerate.

12.5. Terminasi *Thread*

Seperti yang terlihat pada status *thread*, suatu *thread* akan dinyatakan Dead atau sudah di terminasi, ketika *thread* tersebut telah keluar dari run() method nya. Terdapat berbagai macam cara yang disediakan Java untuk men-terminasi *thread*, di antaranya adalah interrupt(), destroy(),dan stop().

interrupt()

Jika sebuah thread di-interrupt(), seharusnya, thread tersebut langsung keluar dari method run() dan melepaskan semua sumberdaya yang dimiliki oleh *thread* tersebut, agar bisa digunakan oleh Thread-*thread* yang lain. Mekanisme untuk keluar dari method run() (mengakhiri method run()) adalah implementasi dari si pembuat program di dalam method run().

destroy() deprecated

Jika sebuat thread di-destroy(), maka thread tersebut LANGSUNG dihancurkan, dan segala lock yang *thread* tersebut pegang, akan hilang. Hal ini akan berakibat kepada Deadlock karena lock yang mengunci sebuah object yang dimiliki *thread* tersebut sudah hilang permanen.

stop() deprecated

Jika sebuat thread di-stop(), maka thread tersebut LANGSUNG berhenti, dan seketika itu juga, akan melepas semua lock yang dimiliki oleh thread tersebut. Keadaan ini akan mengakibatkan ketidakkonsistenan data, karena bisa saja, thread tersebut sedang dalam memperbaharui suatu nilai dari sebuah variabel ketika thread tersebut di-stop().

12.6. JVM dan *Host Operating System*

Implementasi umum dari JVM adalah di atas sebuah *host operating system*. Hal ini memungkinkan JVM untuk menyembunyikan implementasi detail dari sistem operasi tempat JVM dijalankan dan menyediakan lingkungan abstrak dan konsisten yang memungkinkan program-program Java untuk beroperasi di atas *platform* apa pun yang mendukung JVM. Spesifikasi untuk JVM tidak mengindikasikan bagaimana *thread-thread* Java dipetakan ke sistem operasi tempat JVM dijalankan, melainkan menyerahkan keputusan tersebut kepada implementasi tertentu dari JVM. Windows 95/98/NT/2000 menggunakan model *One-to-One*, sehingga setiap *thread* Java untuk JVM pada sistem operasi tersebut dipetakan kepada sebuah *kernel thread*. Solaris 2 awalnya mengimplementasikan JVM menggunakan model *Many-to-One* (disebut *Green Threads*). Akan tetapi, sejak JVM versi 1.1 dengan Solaris 2.6, mulai diimplementasikan menggunakan model *Many-to-Many*.

Implementasi umum dari JVM adalah di atas sebuah host sistem operasi. Hal ini memungkinkan JVM untuk menyembunyikan implementasi detail dari sistem operasi tempat JVM dijalankan dan menyediakan lingkungan abstrak dan konsisten yang memungkinkan program-program Java untuk beroperasi di atas platform apa pun yang mendukung JVM. Spesifikasi untuk JVM tidak mengindikasikan bagaimana thread-thread Java dipetakan ke sistem operasi tempat JVM dijalankan, melainkan menyerahkan keputusan tersebut kepada implementasi tertentu dari JVM. Windows 95/98/NT/2000 menggunakan model One-to-One, sehingga setiap thread Java untuk JVM pada sistem operasi tersebut dipetakan kepada sebuah kernel thread. Solaris 2 awalnya mengimplementasikan JVM menggunakan model Many-to-One (disebut Green *thread*). Akan tetapi, sejak JVM versi 1.1 dengan Solaris 2.6, mulai diimplementasikan menggunakan model Many-to-Many.

12.7. Solusi *Multi-Threading*

Seperti yang sudah diketahui, *thread* tidak selalu berada pada proses running. Ini akan mengakibatkan kekosongan pada proses yang berjalan sehingga penggunaannya menjadi tidak efektif. Karena itu, ketika thread sedang tidak berada pada proses running dapat dibuat thread lain

yang bisa mengisi kekosongan ini. Proses berjalannya thread-thread ini dinamakan MultiThreading.

Untuk menentukan proses mana yang harus berjalan pada multithreading, maka dibuatlah scheduling yanga akan membuat *thread* berjalan sesuai dengan proses yang diinginkan. Untuk lebih jelas tentang scheduling ini, di bahas pada bab lain.

Thread-*thread* yang berada dalam MultiThreading berjalan dengan cara secara bergantian berada di tahap running. Dengan mekanisme scheduling secara bergantian ini, maka proses berjalannya *thread-thread* dapat membentuk effect Pseudo-paralellisme. Yaitu efek ketika *thread* terlihat seperti berjalan secara pararel.

12.8. Rangkuman

Thread di Linux dianggap sebagai *task*. *System call* yang dipakai antara lain fork dan clone. Perbedaan antara keduanya adalah clone selain dapat membuat duplikat dari proses induknya seperti fork, juga dapat berbagi ruang alamat yang sama antara proses induk dengan proses anak. Seberapa besar kedua proses tersebut dapat berbagi tergantung banyaknya flag yang ditandai.

Java adalah unik karena telah mendukung *thread* didalam tingkatan bahasanya. Semua program Java sedikitnya terdiri dari kontrol sebuah *thread* tunggal dan mempermudah membuat kontrol untuk *multiple thread* dengan program yang sama. JAVA juga menyediakan *library* berupa API untuk membuat *thread*, termasuk method untuk suspend dan resume suatu *thread*, agar *thread* tidur untuk jangka waktu tertentu dan menghentikan *thread* yang berjalan. Sebuah java *thread* juga mempunyai empat kemungkinan keadaan, diantaranya: *New*, *Runnable*, *Blocked* dan *Dead*. Perbedaan API untuk mengelola *thread* seringkali mengganti keadaan *thread* itu sendiri.

Salah satu ciri yang membuat Java menjadi bahasa pemrograman yang ampuh adalah dukungannya terhadap pengembangan aplikasi multithread sebagai bagian yang terpadu dari bahasa. Java merupakan bahasa pemrograman yang pertama kali memiliki dukungan intrensif terhadap operasi-operasi *thread* didalam bahasanya.

Thread di Java dapat dibuat melalui 2 cara, dengan meng-extends class *thread* atau dengan meng-implements Runnable interface. Siklus hidupnya dapat melewati 6 tahap, Born(baru dibuat), Ready(method start() dipanggil), Running(diproses), Blocked, Wait (method wait() dipanggil, sampai menunggu notify() atau ntotifyAll()), Sleep (sleep selama milisecond yang ditentukan), dan Dead(diterminasi).

Untuk beberapa kemudahan dan alasan keamanan. Beberapa *thread* dapat digabungkan kedalam sebuah ThreadGroup. Dalam ThreadGroup yang sama sebuah *thread* dapat memanipulasi *thread* lain dalam Group tersebut. Thread-thread dalam ThreadGroup tersebut dapat dimanage dengan mudah.

Untuk menterminasi *thread*, terdapat berbagai cara, antara lain dengan memanggil Thread.interrupt(), Thread.stop(), dan Thread.destroy(). Jika sebuah *thread* di interrupt, maka seharusnya *thread* tersebut melepas segala sumberdaya yang dimilikinya. Kedua cara lainnya tidak aman digunakan karena berpotensi menyebabkan deadlock.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

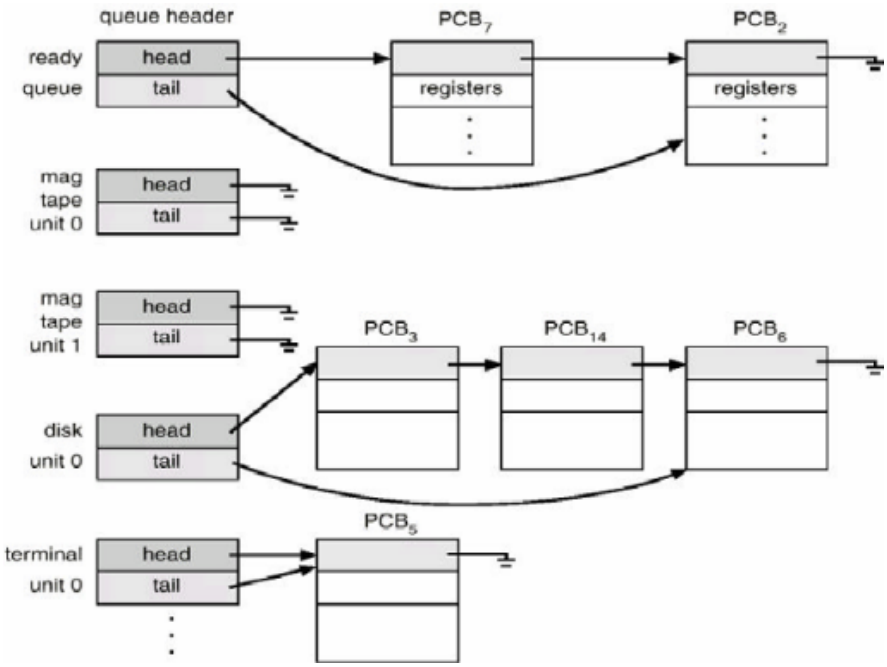
Bab 13. Konsep Penjadwalan

13.1. Pendahuluan

Kita mengenal yang namanya *multiprograming*, yang bertujuan untuk memaksimalkan penggunaan CPU dengan cara mengatur alokasi waktu yang digunakan oleh CPU, sehingga proses berjalan sepanjang waktu dan memperkecil waktu *idle*. Oleh karena itu perlu adanya penjadwalan proses-proses yang ada pada sistem. Untuk sistem yang hanya mempunyai prosesor tunggal (uniprosesor), hanya ada satu proses yang dapat berjalan setiap waktunya. Jika ada proses lebih dari satu maka proses yang lain harus menunggu sampai CPU bebas dan siap untuk dijadwalkan kembali.

13.2. Penjadwalan Antrian

Gambar 13.1. ***Device Queue***



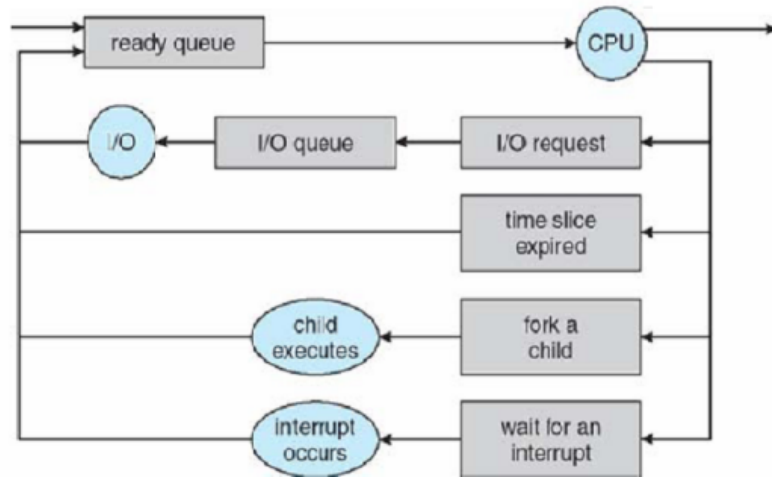
Ketika sebuah proses memasuki sistem, proses itu diletakkan di dalam ***job queue***. Pada antrian ini terdapat seluruh proses yang berada dalam sistem. Sedangkan proses yang berada pada memori utama, siap dan menunggu untuk mengeksekusi disimpan dalam sebuah daftar yang bernama ***ready queue***. Antrian ini biasanya disimpan sebagai *linked list*. *Header* dari *ready queue* berisi *pointer* untuk PCB pertama dan PCB terakhir pada list. Setiap PCB memiliki *pointer field* yang menunjuk kepada PCB untuk proses selanjutnya dalam *ready queue*.

Sistem operasi juga memiliki antrian lain. Ketika sebuah proses dialokasikan ke CPU, proses tersebut berjalan sebentar lalu berhenti, di-interupsi, atau menunggu suatu hal tertentu, seperti selesainya suatu permintaan M/K. Dalam permintaan M/K, dapat saja yang diminta itu adalah *tape drive*, atau peralatan yang di-*share* secara bersama-sama, seperti disk. Karena ada banyak proses dalam sistem, disk dapat saja sibuk dengan permintaan M/K dari proses lainnya. Untuk itu proses tersebut mungkin harus menunggu disk tersebut. Daftar dari proses-proses yang menunggu peralatan M/K tertentu disebut dengan ***device queue***. Tiap peralatan memiliki *device queue*-nya masing-masing (Gambar 14.1, "Siklus *Burst*").

Penjadwalan proses dapat direpresentasikan secara umum dalam bentuk diagram antrian, seperti

yang ditunjukkan oleh Gambar 14.2, “Diagram *Burst*”. Setiap kotak segi empat menunjukkan sebuah antrian. Dua tipe antrian menunjukan antrian yang siap dan seperangkat *device queues*. Lingkaran menunjukkan sumber daya yang melayani antrian, dan tanda panah mengindikasikan alur dari proses-proses yang ada dalam sistem.

Gambar 13.2. Diagram Antrian



Umumnya proses-proses yang ada pada sistem akan ada dalam beberapa tahap antrian yaitu job queue, ready queue, dan device queue.

Job queue, menyimpan seluruh proses yang berada pada sistem. Ketika sebuah proses memasuki sebuah sistem, proses tersebut akan diletakkan di dalam job queue.

Ready queue merupakan sebuah daftar proses-proses yang berada pada memori utama (main memori), siap dan menunggu untuk dieksekusi dan dialokasikan ke CPU. Antrian ini biasanya disimpan sebagai linked-list. Header dari ready queue ini berisi pointer untuk PCB pertama dan PCB terakhir pada linked-list tersebut. Dan setiap PCB memiliki pointer field yang menunjuk kepada PCB untuk proses selanjutnya pada ready queue.

Ketika sebuah proses dialokasikan ke CPU, proses tersebut berjalan sebentar lalu berhenti karena ada interrupt atau menunggu permintaan M/K. Dalam permintaan M/K, dapat saja yang diminta itu adalah peralatan yang di share secara bersama-sama seperti disk. Karena ada banyak proses dalam sistem, disk tersebut dapat saja sibuk dengan permintaan M/K dari proses lainnya. Untuk itu proses tersebut mungkin harus menununggu disk tersebut sampai siap untuk memenuhi permintaan M/K tersebut. Daftar dari proses-proses yang menunggu peralatan M/K tertentu disbut dengan device queue. Tiap peralatan mempunyai device queue-nya masing masing.

Penjadwalan proses dapat direpresentasikan secara umum dalam bentuk diagram antrian, yang ditunjukkan oleh Gambar 13.2, “Diagram Antrian”. Setiap kotak segi empat menunjukkan sebuah antrian. Dua antrian diantaranya menunjukkan ready queue dan seperangkat device queue. Lingkaran menunjukkan sumber daya yang melayani antrian dan tanda panah mengindikasikan alur dari proses-proses yang ada dalam sistem.

Proses-proses yang ada menunggu di dalam ready queue sampai dia dipilih untuk eksekusi, atau di-dispatched. Begitu proses tersebut dipilih lalu dialokasikan ke CPU dan sedang berjalan, satu dari beberapa kemungkinan di bawah ini dapat terjadi.

1. Proses tersebut mengeluarkan permintaan M/K, lalu ditempatakan dalam sebuah M/K device queue.
2. Proses tersebut dapat membuat sub-proses baru dan menunggu untuk di-terminasi.
3. Proses tersebut dikeluarkan (di-remove) secara paksa dari CPU, sebagai hasil dari suatu interrupt dan diletakkan kembali ke dalam ready queue.

Pada dua kemungkinan pertama (1 dan 2) proses berganti status dari waiting state menjadi ready

state, lalu diletakkan kembali ke dalam ready queue. Siklus ini akan terus terjadi pada proses sampai dia di-terminasi, yaitu dimana proses tersebut dikeluarkan dari seluruh antrian yang ada dan memiliki PCB-nya sendiri dan seluruh sumber daya yang dia gunakan dialokasikan kembali.

13.3. *Penjadwal*

Gambar 13.3. ***Medium-term Scheduler***



Sebuah proses berpindah-pindah di antara berbagai penjadwalan antrian seumur hidupnya. Sistem operasi harus memilih dan memproses antrian-antrian ini berdasarkan kategorinya dengan cara tertentu. Oleh karena itu, proses seleksi ini harus dilakukan oleh scheduler yang tepat.

Terdapat dua jenis scheduler pada CPU yang umum dipakai, yaitu:
i. Long-Term Scheduler atau Job Scheduler yang bertugas memilih proses dari tempat ini dan mengisinya ke dalam memori.
ii. Short-Term Scheduler atau CPU scheduler yang bertugas memilih proses yang sudah siap untuk melakukan eksekusi,dan dialokasikan di CPU untuk proses tersebut.

Perbedaan signifikan pada kedua scheduler ini adalah frekuensi dari eksekusinya. Short-Term Scheduler harus sering memilih proses baru untuk CPU dan eksekusinya paling sedikit 1 kali dalam 100 milidetik. Jadi,sudah seharusnya scheduler ini berjalan cepat. Sedangkan pada Long-Term Scheduler melakukan eksekusi lebih sedikit dengan selang waktu pada kisaran menit untuk tiap eksekusinya.

Long-Term Scheduler mengatur degree of multiprogramming, yaitu jumlah proses dalam memori. Jadi jika degree of multi programming stabil, maka rata-rata jumlah proses baru sama dengan jumlah proses yang telah selesai. Oleh karena itu,long-term scheduler dipanggil bila sudah ada proses yang selesai atau telah meninggalkan sistem sehingga eksekusinya jauh lebih jarang dibandingkan short-term scheduler.

Secara umum, proses pada Long-Term Scheduler dapat dibagi menjadi dua, yaitu:
i. M/K Bound yaitu proses yang lebih banyak mengerjakan permintaan M/K dibandinkan komputasi.
ii. CPU Bound yaitu proses yang lebih banyak mengerjakan komputasi dibandingkan permintaan M/K.

M/K Bound dan CPU bound haruslah seimbang. Jika M/K bound terlalu banyak maka ready queue akan selalu hampir kosong dan CPU scheduler memiliki tugas sedikit. Sedangkan jika CPU bound

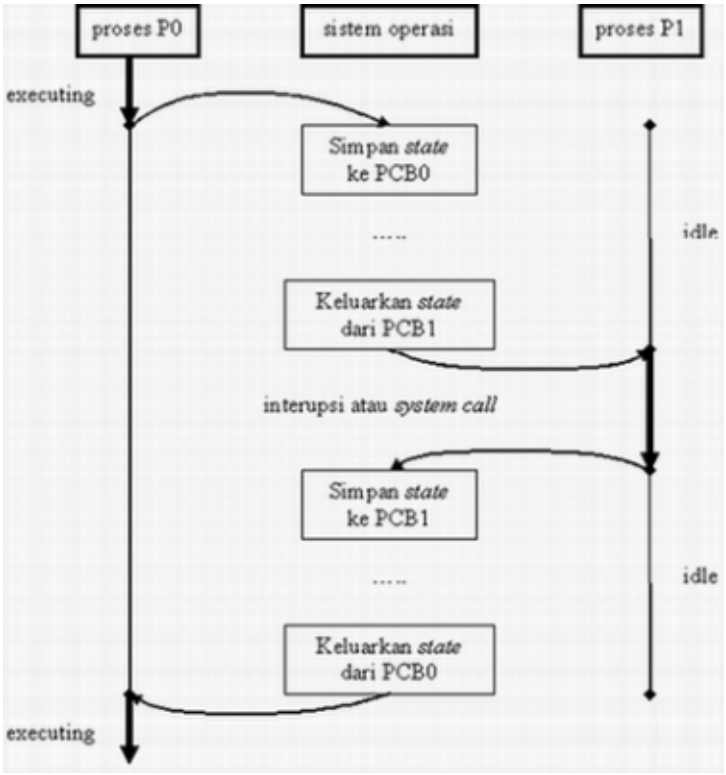
terlalu banyak maka M/K waiting queue akan selalu hampir kosong dan sistem tidak seimbang. Dengan kata lain, jika kedua hal ini tidak seimbang maka prosesor akan lebih banyak dalam kondisi *idle*.

Pada sistem operasi dengan time-sharing system, dikenal suatu scheduler dengan nama Medium-Term Scheduler yaitu suatu scheduler dengan tehnik swapping. Jadi, suatu proses bila belum selesai dalam jatah waktu yang diberikan akan dikeluarkan (swap out) dari CPU dan dikembalikan ke ready queue (swap in) untuk bergantian dengan proses berikutnya. Untuk lebih jelasnya mengenai swapping dapat dilihat pada Bab 29, Alokasi memori.

13.4. *Context Switch*

Mengganti CPU ke proses lain memerlukan penyimpanan keadaan dari proses lama dan mengambil keadaan dari proses yang baru. Hal ini dikenal dengan sebutan ***context switch***. *Context switch* sebuah proses direpresentasikan dalam PCB dari suatu proses; termasuk nilai dari CPU register, status proses (dapat dilihat pada Gambar 14.1, "Siklus *Burst*") dan informasi manajemen memori.

Gambar 13.4. *Context Switch*



Ketika *context switch* terjadi, *kernel* menyimpan data dari proses lama ke dalam PCB nya dan mengambil data dari proses baru yang telah terjadwal untuk berjalan. Waktu *context switch* adalah murni *overhead*, karena sistem melakukan pekerjaan yang tidak begitu berarti selama melakukan pengalihan. Kecepatannya bervariasi dari mesin ke mesin, bergantung pada kecepatan memori, jumlah register yang harus di-*copy*, dan ada tidaknya instruksi khusus (seperti instruksi tunggal untuk mengisi atau menyimpan seluruh register). Tingkat kecepatan umumnya berkisar antara 1 sampai 1000 mikro detik.

CPU, dalam mengganti dari sebuah proses ke proses lain memerlukan penyimpanan keadaan dari proses lama dan mengambil keadaan dari proses yang baru. Inilah yang dikenal dengan sebutan context switch.

Dalam context switch sebuah proses direpresentasikan dalam PCB dari suatu proses, termasuk nilai

dari CPU register, status proses dan informasi manajemen memori. Ketika context switch terjadi, kernel menyimpan data dari proses lama ke dalam PCB nya dan mengambil data dari proses baru yang telah terjadwal untuk berjalan. Waktu context switch adalah murni overhead (sangat cepat), karena sistem melakukan pekerjaan yang tidak begitu berarti selama melakukan pengalihan.

Sebagai contoh, prosesor seperti UltraSPARC menyediakan beberapa set register. Sebuah proses context switch hanya memasukkan perubahan pointer ke set register yang ada saat itu. Tentu saja, jika proses aktif yang ada lebih banyak daripada proses yang ada pada set register, sistem menggunakan bantuan untuk meng-copy data register dari dan ke memori, sebagaimana sebelumnya. Semakin kompleks suatu sistem operasi, semakin banyak pekerjaan yang harus dilakukan selama context switch.

Dapat dilihat pada Bagian V, Memori, teknik manajemen memori tingkat lanjut dapat mensyaratkan data tambahan untuk diganti dengan tiap data. Sebagai contoh, ruang alamat dari proses yang ada saat itu harus dijaga sebagai ruang alamat untuk proses yang akan dikerjakan berikutnya. Bagaimana ruang alamat dijaga, berapa banyak pekerjaan dibutuhkan untuk menjaganya, tergantung pada metoda manajemen memori dari sistem operasi. Akan kita lihat pada Bagian V, Memori, context switch terkadang dapat menyebabkan bottleneck, dan programer menggunakan struktur baru (threads) untuk menghindarinya kapan pun memungkinkan.

13.5. Rangkuman

Sebuah proses, ketika sedang tidak dieksekusi, ditempatkan pada antrian yang sama. Disini ada dua kelas besar dari antrian dalam sebuah sistem operasi: permintaan antrian M/K dan *ready queue*. *Ready queue* memuat semua proses yang siap untuk dieksekusi dan yang sedang menunggu untuk dijalankan pada CPU. PCB dapat digunakan untuk mencatat sebuah ready queue. Penjadwalan Long-term adalah pilihan dari proses-proses untuk diberi izin menjalankan CPU. Normalnya, penjadwalan *long-term* memiliki pengaruh yang sangat besar bagi penempatan sumber daya, terutama manajemen memori. Penjadwalan *short-term* adalah pilihan dari satu proses dari ready queue.

Sebuah proses, ketika sedang tidak dieksekusi, ditempatkan pada antrian yang sama. Disini ada dua kelas besar dari antrian dalam sebuah sistem operasi: permintaan antrian M/K dan ready queue. Ready queue memuat semua proses yang siap untuk dieksekusi dan yang sedang menunggu untuk dijalankan pada CPU. PCB dapat digunakan untuk mencatat sebuah ready queue. Penjadwalan Long-term adalah pilihan dari proses-proses untuk diberi izin menjalankan CPU. Normalnya, penjadwalan long-term memiliki pengaruh yang sangat besar bagi penempatan sumber daya, terutama manajemen memori. Penjadwalan short-term adalah pilihan dari satu proses dari ready queue.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

Bab 14. Penjadwal CPU

14.1. Pendahuluan

Penjadwalan CPU adalah basis dari multi-programming sistem operasi. Dengan men-*switch* CPU diantara proses. Akibatnya sistem operasi dapat membuat komputer produktif. Dalam bab ini kami akan memperkenalkan tentang dasar dari konsep penjadwalan dan beberapa algoritma penjadwalan. Dan kita juga memaparkan masalah dalam memilih algoritma dalam suatu sistem.

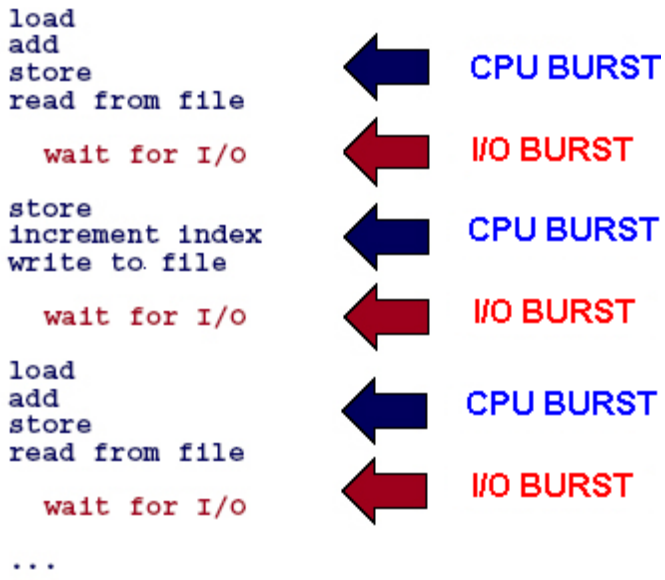
Penjadwalan CPU adalah suatu proses pengaturan atau penjadwalan proses - proses yang ada di dalam komputer. CPU scheduling sangat penting dalam menentukan performance sebuah komputer karena mengatur alokasi resource dari CPU untuk menjalankan proses-proses di dalam komputer. CPU scheduling merupakan suatu konsep dasar dari multiprogramming, karena dengan adanya penjadwalan dari CPU itu sendiri maka proses-proses tersebut akan mendapatkan alokasi resource dari CPU.

Multiprogramming adalah suatu proses menjalankan proses-proses di dalam komputer secara bersamaan (yang disebut paralel). Multiprogramming dapat meningkatkan produktivitas dari sebuah komputer. Tujuan dari multiprogramming adalah menjalankan banyak proses secara bersamaan, untuk meningkatkan performance dari komputer.

14.2. Siklus *Burst* CPU-M/K

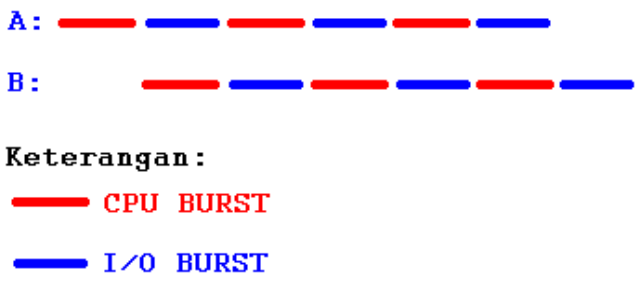
Keberhasilan dari penjadwalan CPU tergantung dari beberapa properti prosesor. Pengeksekusian dari proses tersebut terdiri atas siklus CPU ekskusi dan M/K Wait. Proses hanya akan bolak-balik dari dua state ini. Pengeksekusian proses dimulai dengan CPU Burst, setelah itu diikuti oleh M/K burst, kemudian CPU Burst lagi lalu M/K Burst lagi begitu seterusnya dan dilakukan secara bergiliran. Dan, CPU Burst terakhir, akan berakhir dengan permintaan sistem untuk mengakhiri pengeksekusian daripada melalui M/K Burst lagi. Kejadian siklus Burst akan dijelaskan pada Gambar 14.1, “Siklus *Burst*”.

Gambar 14.1. Siklus *Burst*

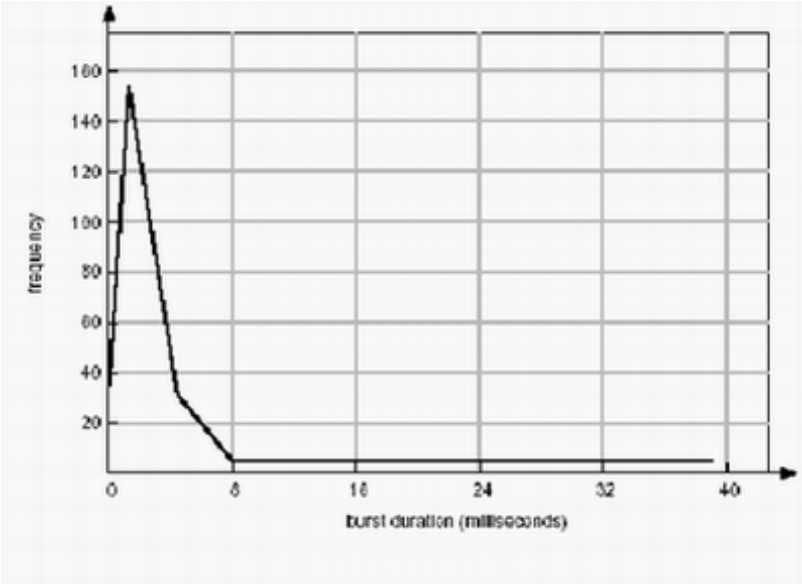


Durasi dari CPU bust ini telah diukur secara ekstensif, walau pun mereka sangat berbeda dari proses ke prose. Mereka mempunyai frekeunsi kurva yang sama seperti yang diperlihatkan pada Gambar 14.2, “Diagram *Burst*”.

Gambar 14.2. Diagram *Burst*



Gambar 14.3. *Burst*



14.3. *Dispatcher*

Komponen yang lain yang terlibat dalam penjadwalan CPU adalan *dispatcher*. *Dispatcher* adalah modul yang memberikan kontrol CPU kepada proses yang fungsinya adalah:

1. *Switching context*
2. *Switching to user mode*
3. Lompat dari suatu bagian di progam user untuk mengulang progam.

Dispatcher seharusnya secepat mungkin. Dispatch Latency adalah waktu yang diperlukan dispatcher untuk menghentikan suatu proses dan memulai proses yang lain.

14.4. Penjadwalan CPU

Keberhasilan dari suatu penjadwalan tergantung kepada properti dari proses yang telah diteliti berikut ini: Suatu proses mengandung suatu cycle atau siklus dari pengeksekusian CPU dan M/K wait.

Suatu proses yang dieksekusi oleh CPU terjadi diantara dua cycle tersebut. Suatu proses pengeksekusian akan dimulai dengan CPU Burst dan diikuti oleh M/K Burst, dan kembali ke M/K Burst seterusnya dalam 2 siklus itu sampai selesai. Pada saat pengeksekusian suatu proses akan menjalankan instruksi nya pada CPU Burst dan akan mengalami suatu M/K Burst pada saat menunggu proses M/K (M/K Burst) Suatu proses dalam komputer itu pada umumnya berada di dua siklus tersebut, tapi ini hanya secara umum. Dapat saja pada suatu proses itu memiliki waktu CPU Burst yang sangat lama disebut dengan CPU Bound, contohnya dalam aplikasi aritmatika.

Proses yang memiliki suatu M/K Burst yang sangat lama disebut dengan M/K Bound, contohnya dalam proses pengeksekusian graphic game, dimana proses tersebut banyak menunggu masukan dan keluaran dari M/K. CPU burst berhak untuk menterminate sebuah proses atau program yang diminta oleh sistem.

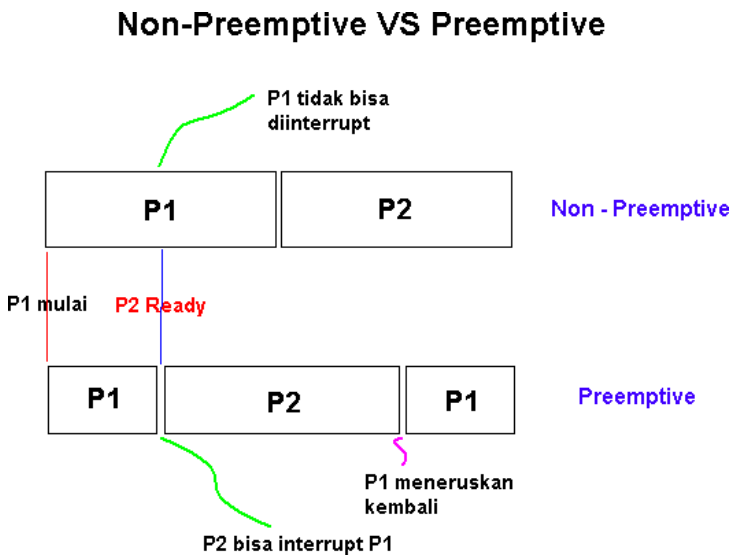
Kapan pun CPU menjadi *idle*, sistem operasi harus memilih salah satu proses untuk masuk kedalam antrian *ready* (siap) untuk dieksekusi. Pemilihan tersebut dilakukan oleh penjadwal *short term*. Penjadwalan memilih dari sekian proses yang ada di memori yang sudah siap dieksekusi, den mengalokasikan CPU untuk mengeksekusinya.

Penjadwalan CPU mungkin akan dijalankan ketika proses dalam keadaan:

1. Berubah dari running ke waiting state.
2. Berubah dari running ke ready state.
3. Berubah dari waiting ke ready.
4. Terminates.

Penjadwalan nomor 1 dan 4 bersifat *non-preemptive* sedangkan lainnya *preemptive*. Dalam penjadwalan *non-preemptive* sekali CPU telah dialokasikan untuk sebuah proses, maka tidak dapat di ganggu, penjadwalan model seperti ini digunakan oleh Windows 3.X; Windows 95 telah menggunakan penjadwalan *preemptive*.

Gambar 14.4. *Preemptive vs Non Preemptive*

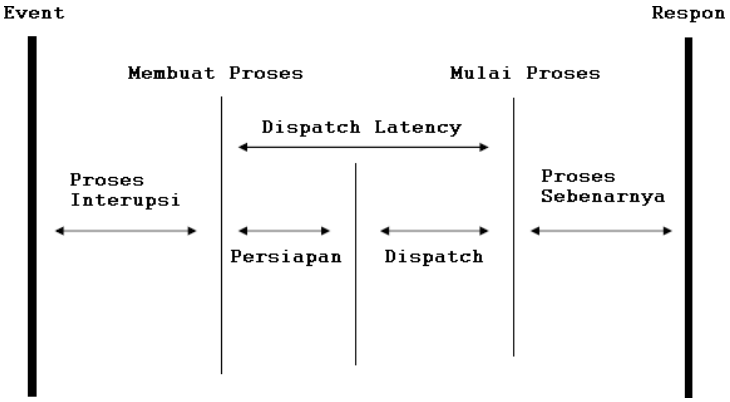


Penjadwalan *Preemptive*

Semua proses yang ada dalam antrian ready (siap) harus dikerjakan oleh CPU. CPU sendiri harus memilih salah satu proses yang akan dikerjakan dalam satu waktu. Pemilihan tersebut dilakukan oleh short term scheduler dan proses yang dipilih akan dieksekusi dalam CPU dalam satu waktu

yang telah ditentukan oleh scheduler.

Gambar 14.5. ***Dispatch Latency***



Penjadwalan *Non-Preemptive*

Penjadwalan non-preemptive terjadi ketika proses hanya:
i. berjalan dari running state sampai waiting state.
ii. dihentikan.
Ini berarti cpu menjaga proses sampai proses itu pindah ke waiting state ataupun dihentikan (proses tidak diinterupt). Metode ini digunakan oleh Microsoft Windows 3.1 dan Macintosh. Ini adalah metode yang dapat digunakan untuk platforms hardware tertentu, karena tidak memerlukan perangkat keras khusus (misalnya timer yang digunakan untuk menginterupt pada metode penjadwalan preemptive).

14.5. Kriteria Penjadwalan

Suatu Algoritma penjadwalan CPU yang berbeda dapat mempunyai nilai yang berbeda untuk sistem yang berbeda. Banyak kriteria yang bisa dipakai untuk menilai algoritma penjadwalan CPU.

Kriteria yang digunakan dalam menilai adalah:

1. **CPU Utilization.** Kita ingin menjaga CPU sesibuk mungkin. CPU utilization akan mempunyai range dari 0 sampai 100 persen. Di sistem yang sebenarnya ia mempunyai range dari 40 sampai 100 persen.

2. **Throughput.** Salah satu ukuran kerja adalah banyaknya proses yang diselesaikan per satuan waktu.jika kita mempunyai beberapa proses yang sama dan memiliki beberapa algoritma penjadwalan yang berbeda, throughput bisa menjadi salah satu kriteria penilaian, dimana algoritma yang menyelesaikan proses terbanyak mungkin yang terbaik.

3. **Turnaroud Time.** Dari sudut pandang proses tertentu, kriteria yang penting adalah berapa lama untuk mengeksekusi proses tersebut. Memang, lama pengeksekusian sebuah proses sangat tergantung dari hardware yang dipakai, namun kontribusi algoritma penjadwalan tetap ada dalam lama waktu yang dipakai untuk menyelesaikan sebuah proses. Misal, kita memilki sistem komputer yang identik dan proses-proses yang identik pula, namun kita memakai algoritma yang berbeda, algoritma yang mampu menyelesaikan proses yang sama dengan waktu yang lebih singkat mungkin lebih baik dari algoritma yang lain. interval waktu yang diijinkan dengan waktu yang dibutuhkan untuk menyelesaikan sebuah proses disebut turnaround time.Turanaround time adalah jumlah periode untuk menunggu untuk dapat ke memori, menunggu di ready queue, eksekusi CPU, dan melakukan operasi M/K.

4. **Waiting Time.** Algoritma penjadwalan CPU tidak mempengaruhi waktu untuk melaksankan proses tersebut atau M/K, itu hanya mempengaruhi jumlah waktu yang dibutuhkan proses di

antrian raedy. Waiting time adalah jumlah waktu yang dbutuhkan proses di antrian ready.

5. **Response time.** Di sistem yang interaktif, turnaround time mungkin bukan waktu yang terbaik untuk kriteria.sering sebuah proses dapat memproduksi output di awal, dan dapat meneruskan hasil yang baru sementara hasil yang sebelumnya telah diberikan ke pengguna. ukuran lain adalah waktu dari pengiriman permintaan sampai respon yang pertama diberikan. Ini disebut respon time, yaitu waktu untuk memulai memberika respon, tetapi bukan waktu yang dipakai output untuk respon tersebut.

Sebaiknya ketika kita akan membuat algoritma penjadwalan yang dilakukan adalah memaksimalkan CPU utilization dan throughput, dan meminimalkan turnaround time, waiting time, dan response time.

14.6. Rangkuman

Penjadwalan CPU adalah pemilihan proses dari antrian ready untuk dapat dieksekusi. Penjadulan CPU merupakan konsep dari multiprogramming, dimana CPU digunakan secara bergantian untuk proses yang berbeda.suatu proses terdiri dari dua siklus yaitu M/K burst dan CPU burst yang dilakukan bergantian hingga proses selesai. Penjadulan CPU mungkin diajalankan ketika proses:
i. running -> waiting time.
ii. running -> ready state.
iii. waiting -> ready state.
iv. terminates.

Proses 1 dan 4 adalah proses non-preemptive, dimana proses tersebut tidak bisa di interrupt, seangkan 2 dan 3 adalah preemptive, dimana proses boleh di interrupt. Komponen yang lain dalam penjadwalan CPU adalah dispatcher, dispatcher adalah modul yang memberikan kendali CPU kepada proses.dalam menilai baik atau buruknya suatau algoritma penjadwalan kita bisa memakai beberapa kriteria, diantaranya CPU utilization, throughput, turnaround time,waiting time, dan response time. algoritma yang baik adalah yang mampu memaksimalkan CPU utilization dan throughput, dan mampu meminimalkan turnaroud time, waiting time, dan response time.

Waktu yang diperlukan oleh dispatcher untuk menghentikan suatu proses dan memulai proses yang lain disebut dengan dispatch latency.

Jika dalam suatu proses CPU Burst jauh lebih besar daripada M/K Burst maka disebut CPU Bound. Demikian unutk sebaliknya disebut dengan M/K Bound.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

Bab 15. Algoritma Penjadwalan I

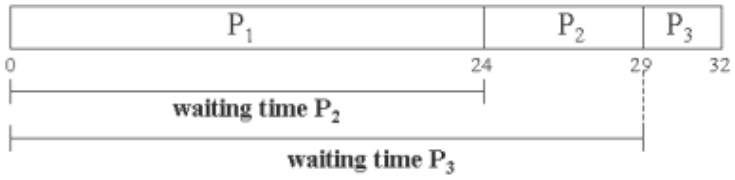
15.1. Pendahuluan

Proses yang belum mendapat jatah alokasi dari CPU akan mengantri di *ready queue*. Di sini algoritma diperlukan untuk mengatur giliran proses-proses tersebut. Berikut ini adalah algoritmanya. Algoritma penjadwalan berfungsi untuk menentukan proses manakah yang ada di ready queue yang akan dieksekusi oleh CPU. Bagian berikut ini akan memberikan ilustrasi beberapa algoritma penjadwalan.

15.2. FCFS: *First-Come, First-Served*

Algoritma ini merupakan algoritma penjadwalan yang paling sederhana yang digunakan CPU. Dengan menggunakan algoritma ini seiap proses yang berada pada status ready dimasukkan ke dalam FIFO queue sesuai dengan waktu kedatangannya. Proses yang tiba terlebih dahulu yang akan dieksekusi.

Gambar 15.1. FCFS



Contoh: Ada tiga buah proses yang datang secara bersamaan yaitu pada 0 ms, P1 memiliki burst time 24 ms, P2 memiliki burst time 5 ms, P3 memiliki burst time 3 ms. Hitunglah wating time rata-rata dan turnaround time (burst time + waiting time) dari ketiga proses tersebut dengan menggunakan algoritma FCFS.

Waiting time untuk p1 adalah 0 ms (P1 tidak perlu menunggu), sedangkan untuk p2 adalah sebesar 24 ms (menunggu P1 selesai) dan untuk p3 sebesar 29 ms (menunggu P1 dan P2 selesai). Waiting time rata-ratanya adalah sebesar (0+24+29)/3 = 17,6 ms.

Turnaround time untuk P1 sebesar 24 ms, sedangkan untuk P2 sebesar 29 ms (dihitung dari awal kedatangan P2 hingga selesai dieksekusi), untuk p3 sebesar 32 ms. Turnaround time rata-rata untuk ketiga proses tersebut adalah (24+29+32)/3 = 28,3 ms.

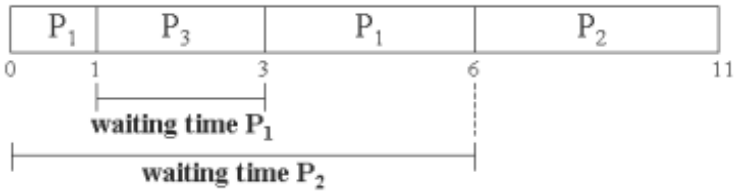
Kelemahan dari algoritma ini:
1. Waiting time rata-ratanya cukup lama.
2. Terjadinya convoy effect, yaitu proses-proses menunggu lama untuk menunggu 1 proses besar yang sedang dieksekusi oleh CPU.

Algoritma ini juga menerapkan konsep non-preemptive, yaitu setiap proses yang sedang dieksekusi oleh CPU tidak dapat di-interrupt oleh proses yang lain.

15.3. SJF: *Shortest-Job First*

Algoritma ini mempunyai cara penjadwalan yang berbeda dengan FCFS. Dengan algoritma ini maka setiap proses yang ada di ready queue akan dieksekusi berdasarkan burst time terkecil. Hal ini mengakibatkan waiting time yang pendek untuk setiap proses dan karena hal tersebut maka waiting time rata-ratanya juga menjadi pendek, sehingga dapat dikatakan bahwa algoritma ini adalah algoritma yang optimal.

Gambar 15.2. SJF *Preemptive*



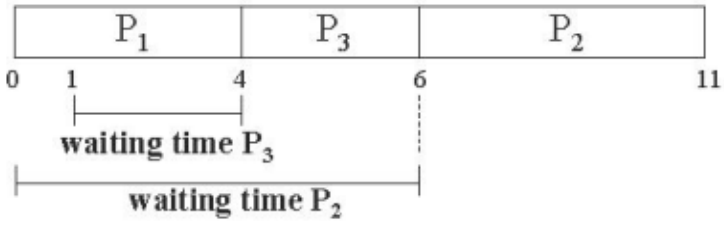
Ada beberapa kekurangan dari algoritma ini yaitu:
1. Susahnya untuk memprediksi burst time proses yang akan dieksekusi selanjutnya.
2. Proses yang mempunyai burst time yang besar akan memiliki waiting time yang besar pula karena yang dieksekusi terlebih dahulu adalah proses dengan burst time yang lebih kecil.

Algoritma ini dapat dibagi menjadi dua bagian yaitu:
1. **Preemptive.** Jika ada proses yang sedang dieksekusi oleh CPU dan terdapat proses di ready queue dengan burst time yang lebih kecil daripada proses yang sedang dieksekusi tersebut, maka proses yang sedang dieksekusi oleh CPU akan digantikan oleh proses yang berada di ready queue tersebut. Preemptive SJF sering disebut juga Shortest-Remaining- Time-First scheduling.
2. **Non-preemptive.** CPU tidak memperbolehkan proses yang ada di ready queue untuk menggeser proses yang sedang dieksekusi oleh CPU meskipun proses yang baru tersebut mempunyai burst time yang lebih kecil.

Contoh: Ada 3 buah proses yang datang berurutan yaitu p1 dengan arrival time pada 0 ms dengan burst time 4 ms, p2 dengan arrival time pada 0 ms dengan burst time 5 ms, p3 dengan arrival time pada 1 ms dengan burst time 2 ms. Hitunglah waiting time rata-rata dan turnaround time dari ketiga proses tersebut dengan mengunakan algoritma SJF.

Waiting time rata-rata untuk ketiga proses tersebut adalah sebesar ((3-1)+(6-0)+(1-1))/3 = 2,6 ms. Turnaround time dari ketiga proses tersebut adalah sebesar (6+11+(3-1))/3 = 6,3 ms.

Gambar 15.3. SJF Non *Preemptive*



Waiting time rata-rata untuk ketiga prses tersebut adalah sebesar (0+6+(4-1))/3 = 3 ms Turnaround time dari ketiga proses tersebut adalah sebesar (4+11+(6-1))/3 = 6,6 ms

15.4. Prioritas

Priority Scheduling merupakan algoritma penjadwalan yang mendahulukan proses yang memiliki prioritas tertinggi. Setiap proses memiliki prioritasnya masing-masing.

Prioritas suatu proses dapat ditentukan melalui beberapa karakteristik antara lain:
1. Time limit.
2. Memory requirement.
3. Akses file.
4. Perbandingan antara M/K Burst dengan CPU Burst.
5. Tingkat kepentingan proses.

Priority Scheduling juga dapat dijalankan secara preemptive maupun non-preemptive. Pada preemptive, jika ada suatu proses yang baru datang memiliki prioritas yang lebih tinggi daripada proses yang sedang dijalankan, maka proses yang sedang berjalan tersebut dihentikan, lalu CPU dialihkan untuk proses yang baru datang tersebut.

Sementara itu, pada non-preemptive, proses yang baru datang tidak dapat menganggu proses yang sedang berjalan, tetapi hanya diletakkan di depan queue.

Kelemahan pada priority scheduling adalah dapat terjadinya indefinite blocking (starvation). Suatu proses dengan prioritas yang rendah memiliki kemungkinan untuk tidak dieksekusi jika terdapat proses lain yang memiliki prioritas lebih tinggi darinya.

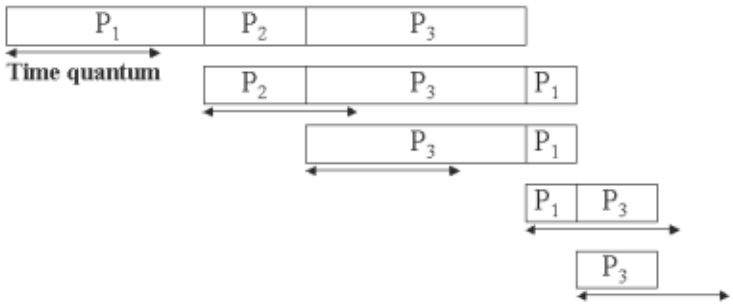
Solusi dari permasalahan ini adalah aging, yaitu meningkatkan prioritas dari setiap proses yang menunggu dalam queue secara bertahap.

Contoh: Setiap 10 menit, prioritas dari masing-masing proses yang menunggu dalam queue dinaikkan satu tingkat. Maka, suatu proses yang memiliki prioritas 127, setidaknya dalam 21 jam 20 menit, proses tersebut akan memiliki prioritas 0, yaitu prioritas yang tertinggi. (semakin kecil angka menunjukkan bahwa prioritasnya semakin tinggi)

15.5. *Round-Robin*

Algoritma ini menggilir proses yang ada di antrian. Proses akan mendapat jatah sebesar *time quantum*. Jika *time quantum*-nya habis atau proses sudah selesai CPU akan dialokasikan ke proses berikutnya. Tentu proses ini cukup adil karena tak ada proses yang diprioritaskan, semua proses mendapat jatah waktu yang sama dari CPU (1/n), dan tak akan menunggu lebih lama dari (n-1)/q.

Gambar 15.4. Round Robin



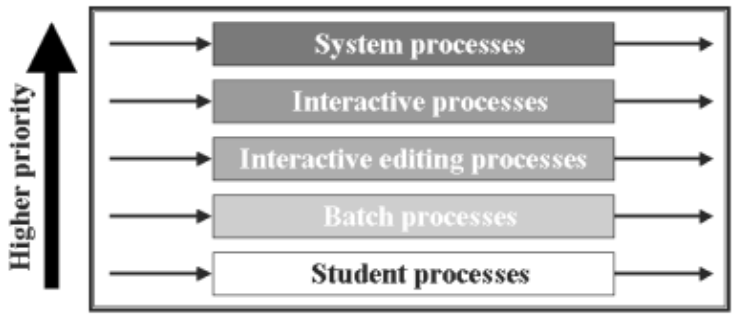
Algoritma ini sepenuhnya bergantung besarnya *time quantum*. Jika terlalu besar, algoritma ini akan sama saja dengan algoritma *first-come first-served*. Jika terlalu kecil, akan semakin banyak peralihan proses sehingga banyak waktu terbuang.

Permasalahan utama pada Round Robin adalah menentukan besarnya time quantum. Jika time quantum yang ditentukan terlalu kecil, maka sebagian besar proses tidak akan selesai dalam 1 time quantum. Hal ini tidak baik karena akan terjadi banyak switch, padahal CPU memerlukan waktu untuk beralih dari suatu proses ke proses lain (disebut dengan context switches time). Sebaliknya, jika time quantum terlalu besar, algoritma Round Robin akan berjalan seperti algoritma First Come First Served. Time quantum yang ideal adalah jika 80% dari total proses memiliki CPU burst time yang lebih kecil dari 1 time quantum.

15.6. *Multilevel Queue*

Ide dasar dari algoritma ini adalah berdasarkan pada sistem prioritas proses. Prinsipnya adalah, jika setiap proses dapat dikelompokkan berdasarkan prioritasnya, maka akan didapati queue seperti pada gambar berikut:

Gambar 15.5. Multilevel Queue



Dari gambar tersebut terlihat bahwa akan terjadi pengelompokan- pengelompokan proses-proses berdasarkan prioritasnya. Kemudian muncul ide untuk menganggap kelompok-kelompok tersbut sebagai sebuah antrian-antrian kecil yang merupakan bagian dari antrian keseluruhan proses, yang sering disebut dengan algoritma multilevel queue.

Dalam hal ini dapat dilihat bahwa seolah-olah algoritma dengan prioritas yang dasar adalah algoritma multilevel queue dimana setiap queue akan berjalan dengan algoritma FCFS dan dapat diketahui bahwa algoritma FCFS memiliki banyak kelemahan, dan oleh karena itu maka dalam prakteknya, algoritma multilevel queue memungkinkan adanya penerapan algoritma internal dalam masing-masing sub- antriannya untuk meningkatkan kinerjanya, dimana setiap sub-antrian bisa memiliki algoritma internal yang berbeda.

Berawal dari priority scheduling, algoritma ini pun memiliki kelemahan yang sama dengan priority scheduling, yaitu sangat mungkin bahwa suatu proses pada queue dengan prioritas rendah bisa saja tidak mendapat jatah CPU. Untuk mengatasi hal tersebut, salah satu caranya adalah dengan memodifikasi algoritma ini dengan adanya jatah waktu maksimal untuk tiap antrian, sehingga jika suatu antrian memakan terlalu banyak waktu, maka prosesnya akan dihentikan dan digantikan oleh antrian dibawahnya, dan tentu saja batas waktu untuk tiap antrian bisa saja sangat berbeda tergantung pada prioritas masing-masing antrian.

15.7. *Multilevel Feedback Queue*

Algoritma ini mirip sekali dengan algoritma *Multilevel Queue*. Perbedaannya ialah algoritma ini mengizinkan proses untuk pindah antrian. Jika suatu proses menyita CPU terlalu lama, maka proses itu akan dipindahkan ke antrian yang lebih rendah. Ini menguntungkan proses interaksi, karena proses ini hanya memakai waktu CPU yang sedikit. Demikian pula dengan proses yang menunggu terlalu lama. Proses ini akan dinaikkan tingkatannya.

Biasanya prioritas tertinggi diberikan kepada proses dengan *CPU burst* terkecil, dengan begitu CPU akan terutilisasi penuh dan *I/O* dapat terus sibuk. Semakin rendah tingkatannya, panjang *CPU burst* proses juga semakin besar.

Algoritma ini didefinisikan melalui beberapa parameter, antara lain:
- Jumlah antrian
- Algoritma penjadwalan tiap antrian
- Kapan menaikkan proses ke antrian yang lebih tinggi
- Kapan menurunkan proses ke antrian yang lebih rendah
- Antrian mana yang akan dimasuki proses yang membutuhkan

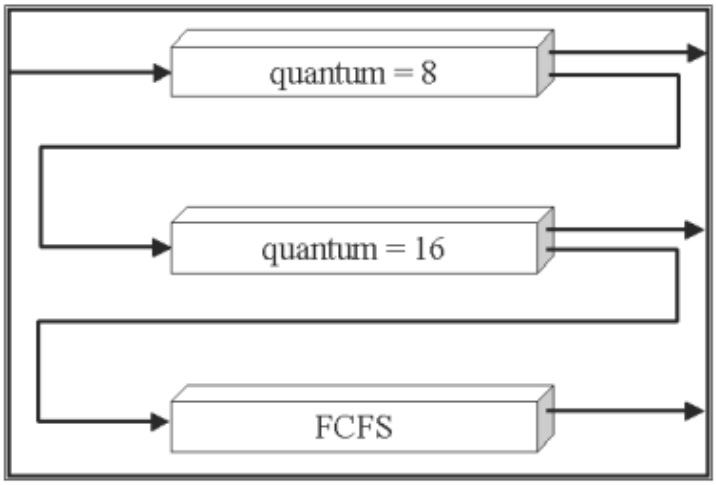
Dengan pendefinisian seperti tadi membuat algoritma ini sering dipakai. Karena algoritma ini mudah dikonfigurasi ulang supaya cocok dengan sistem. Tapi untuk mengatahui mana penjadwal terbaik, kita harus mengetahui nilai parameter tersebut.

Multilevel feedback queue adalah salah satu algoritma yang berdasar pada algoritma mulilevel queue. Perbedaan mendasar yang membedakan multilevel feedback queue dengan multilevel queue

biasa adalah terletak pada adanya kemungkinan suatu proses berpindah dari satu antrian ke antrian lainnya, entah dengan prioritas yang lebih rendah ataupun lebih tinggi, misalnya pada contoh berikut.

1. Semua proses yang baru datang akan diletakkan pada queue 0 (quantum = 8 ms)
2. Jika suatu proses tidak dapat diselesaikan dalam 8 ms, maka proses tersebut akan dihentikan dan dipindahkan ke queue 1 (quantum = 16 ms)
3. Queue 1 hanya akan dikerjakan jika tidak ada lagi proses di queue 0, dan jika suatu proses di queue 1 tidak selesai dalam 16 ms, maka proses tersebut akan dipindahkan ke queue 2
4. Queue 2 akan dikerjakan bila queue 0 dan 1 kosong, dan akan berjalan dengan algoritma FCFS

Gambar 15.6. Multilevel Feedback Queue



Disini terlihat bahwa ada kemungkinan terjadinya perpindahan proses antar queue, dalam hal ini ditentukan oleh time quantum, namun dalam prakteknya penerapan algoritma multilevel feedback queue akan diterapkan dengan mendefinisikan terlebih dahulu parameter- parameternya, yaitu:

1. Jumlah antrian.
2. Algoritma internal tiap queue.
3. Aturan sebuah proses naik ke antrian yang lebih tinggi.
4. Aturan sebuah proses turun ke antrian yang lebih rendah.
5. Antrian yang akan dimasuki tiap proses yang baru datang.

Berdasarkan hal-hal di atas maka algoritma ini dapat digunakan secara fleksibel dan diterapkan sesuai dengan kebutuhan sistem. Pada zaman sekarang ini algoritma multilevel feedback queue adalah salah satu yang paling banyak digunakan.

15.8. Rangkuman

Algoritma diperlukan untuk mengatur giliran proses-proses diready queue yang mengantri untuk dialokasikan ke CPU. Terdapat berbagai macam algoritma,antara lain:

- **First come first serve.** Algoritma ini mendahulukan proses yang lebih dulu datang. Kelemahannya,waiting time rata-rata cukup lama.
- **Shortest job first.** Algoritma ini mendahulukan proses dengan CPU burst terkecil yang akan mengurangi waiting time rata-rata.
- **Priority.** Algoritma ini mendahulukan prioritas terbesar. Kelemahannya, prioritas kecil tidak mendapat jatah CPU. Hal ini dapat diatasi dengan aging,yaitu semakin lama menunggu,prioritas semakin tinggi.
- **Round Robin.** Algoritma ini menggilir proses-proses yang ada diantrian dengan jatah time quantum yang sama. Jika waktu habis,CPU dialokasikan keproses selanjutnya.
- **Multilevel Queue.** Algoritma ini membagi beberapa antrian yang akan diberi prioritas berdasarkan tingkatan. Tingkatan lebih tinggi menjadi prioritas utama.
- **Multilevel Feedback Queue.** Pada dasarnya sama dengan Multilevel Queue,bedanya pada

algoritma ini diizinkan untuk pindah antrian.

Keenam algoritma penjadualan memiliki karakteristik beserta keunggulan dan kelemahannya masing-masing. Namun, Multilevel Feedback Queue Scheduling sebagai algoritma yang paling kompleks, adalah algoritma yang paling banyak digunakan saat ini.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

Bab 16. Algoritma Penjadwalan II

16.1. Pendahuluan

Penjadwalan pada prosesor jamak jelas lebih kompleks, karena kemungkinan masalah yang timbul jauh lebih banyak daripada prosesor tunggal. Prioritas adalah suatu istilah yang digunakan untuk menentukan tingkat urutan atau hirarki suatu proses yang sedang masuk dalam *ready queue*. Konsep prioritas sangat penting dalam penjadwalan prosesor jamak. Prioritas adalah tingkat kepentingan suatu proses yang ada di *ready queue*. Bab ini akan membahas algoritma penjadwalan pada prosesor jamak. Penjadwalan pada prosesor jamak lebih rumit daripada prosesor tunggal. Hal ini sesuai dengan kemampuan prosesor jamak yang lebih baik daripada prosesor tunggal.

16.2. Prosesor Jamak

Ada dua jenis susunan prosesor yang digunakan dalam sistem prosesor jamak, yaitu prosesor-prosesor yang fungsinya identik (homogen) dan prosesor-prosesor yang fungsinya berbeda (heterogen). Bab ini akan memfokuskan pembahasan pada penjadwalan sistem prosesor jamak yang homogen.

Sistem prosesor jamak yang homogen memungkinkan terjadinya pembagian pekerjaan yang tidak merata antar prosesor. Pada waktu yang bersamaan, satu prosesor bisa sangat sibuk sementara prosesor lain tidak mengerjakan proses apapun. Hal ini mengakibatkan sistem tidak bekerja dengan optimal dan efisien. Untuk menghindari hal ini, sistem menggunakan *ready queue* yang berfungsi menampung semua proses yang masuk dan menjadwalkannya ke setiap prosesor yang tersedia.

Ada dua pendekatan penjadwalan CPU pada sistem prosesor jamak:

1. **Proses Jamak Asimetris.** Proses ini mengalokasikan sebuah prosesor yang berfungsi mengatur proses M/K, menjadwal prosesor-prosesor lain dan melakukan pekerjaan-pekerjaan sistem lainnya. Prosesor-prosesor lain dialokasikan untuk menjalankan perintah-perintah dari pengguna. Proses ini sederhana karena hanya ada satu prosesor yang mengakses sistem struktur data sehingga mengurangi kebutuhan untuk pembagian data. Namun, proses ini kurang efisien karena dapat terjadi M/K bound, yaitu suatu interupsi yang dilakukan oleh proses M/K. Proses M/K membutuhkan waktu yang sangat lama sehingga dapat menyebabkan *bottleneck* pada CPU. Untuk mengatasi hal tersebut, proses jamak asimetris dikembangkan menjadi proses jamak simetris.
2. **Proses Jamak Simetris.** Dalam proses jamak simetris, setiap prosessor menjadwalkan dirinya sendiri. Setiap proses dapat berada di *ready queue* yang dipakai bersama atau berada di *queue* yang ada di setiap prosesor. Jenis *queue* apapun yang dipakai, penjadwal pada setiap prosesor akan memeriksa setiap *ready queue* dan memilih proses yang akan dieksekusi. Semua sistem operasi modern mendukung proses jamak simetris, termasuk Windows XP, Windows 2000, Solaris, Linux, dan Mac OS X.

16.3. Sistem Waktu Nyata

Bagian ini akan menjelaskan fasilitas penjadwalan yang dibutuhkan untuk mendukung komputasi waktu nyata dengan bantuan sistem komputer. Penjadwalan waktu nyata digunakan untuk mendukung proses-proses waktu nyata yang ada di sistem komputer. Proses waktu nyata tidak memiliki banyak perbedaan dengan proses-proses lain. Proses waktu nyata adalah proses yang memiliki batasan waktu.

Proses-proses waktu nyata sangat penting dan banyak diterapkan pada bidang industri militer, ilmiah, kesehatan, alat rumah tangga, alat elektronik, praktisnya hampir semua bidang yang terkait dengan teknologi informasi.

Ciri-ciri proses waktu nyata:

1. **Deterministik.** Suatu proses waktu nyata harus memiliki batasan waktu yang jelas.
2. **Bersifat Primitif.** Proses waktu nyata (khususnya hard real-time) biasanya mengerjakan hal-hal yang sederhana dan spesifik. Contoh kasus: pada hulu ledak nuklir, proses waktu nyata yang

bertugas untuk melakukan koordinasi temperatur ruang penyimpanan tidak dapat digabung dengan proses untuk menyalakan AC di ruang direktur utama militer karena sangat berbahaya.

3. **Responsif.** Suatu proses waktu nyata harus dapat diketahui secara pasti kapan prosesnya akan mulai dan berakhir. Misalnya, saat mengetik dengan *keyboard*. Saat menekan tuts *keyboard*, di layar akan langsung terlihat perubahan yang sesuai dengan tuts yang ditekan.
4. **Sulit berkolaborasi dengan *virtual machine* dan sistem prosesor jamak.** Sesuai dengan sifat proses waktu nyata (khususnya *hard real time*) yang deterministik dan responsif, proses waktu nyata sulit berkolaborasi dengan *virtual machine* dan sistem prosesor jamak. Hal itu disebabkan sistem *virtual machine* dan prosesor jamak bersifat *preemptive* sehingga sulit diperkirakan batasan waktunya.

Sistem waktu-nyata dapat dibagi menjadi dua:

1. *Hard Real-Time*
2. *Soft Real-Time*

16.4. Sistem *Hard Real-Time*

Sistem *hard real-time* dibutuhkan untuk menyelesaikan suatu tugas yang sifatnya sangat kritis dengan batasan waktu yang tegas. Proses *hard real-time* tidak dapat mentoleransi keterlambatan yang lebih daripada 100 mikro detik.

Untuk mengerjakan sebuah proses waktu nyata, proses tersebut dikirim kepada penjadwal dengan membawa pernyataan jumlah waktu yang diperlukan untuk menyelesaikan proses tersebut. Kemudian penjadwal dapat menerima dan menjamin kelangsungan proses dengan mengalokasikan sumber daya atau langsung menolaknya. Mekanisme ini disebut *resource allocation*.

Pemberian jaminan ini tidak dapat dilakukan dalam sistem dengan penyimpanan sekunder atau *virtual memori* yang tidak dapat diramalkan waktu yang dibutuhkannya untuk mengeksekusi suatu proses.

Sistem *hard real-time* banyak dipakai dalam sistem-sistem yang memiliki tingkat kefatalan cukup tinggi, seperti sistem pengendali pesawat, sistem hulu ledak nuklir, dll.

Misalnya penggunaan sistem hard real time dalam sistem pengendali pesawat terbang. Batasan waktu pada sistem pengendali pesawat terbang harus tegas karena penyimpangan terhadap batasan waktu dapat berakibat fatal, yaitu kematian penumpang.

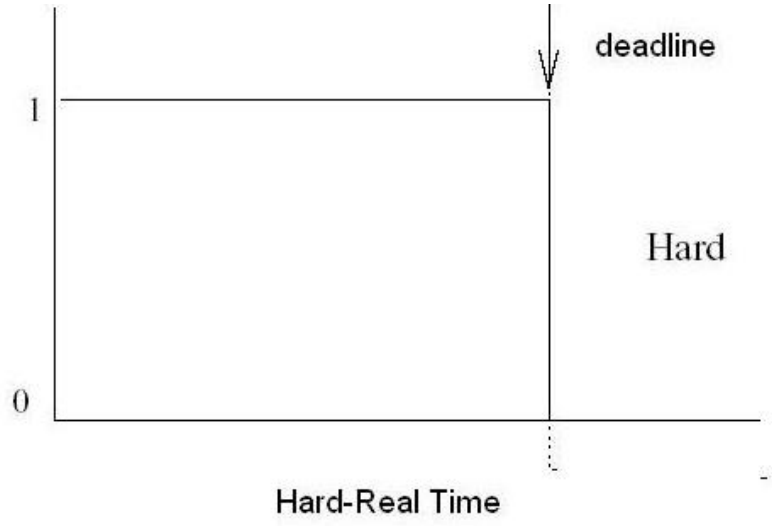
16.5. Sistem *Soft Real-Time*

Komputasi **soft real-time** memiliki sedikit kelonggaran. Dalam sistem ini, proses yang kritis menerima prioritas lebih daripada yang lain. Walaupun menambah fungsi soft real-time ke sistem time sharing mungkin akan mengakibatkan ketidakadilan pembagian sumber daya dan mengakibatkan delay yang lebih lama, atau mungkin menyebabkan *starvation*, hasilnya adalah tujuan secara umum sistem yang dapat mendukung multimedia, grafik berkecepatan tinggi, dan variasi tugas yang tidak dapat diterima di lingkungan yang tidak mendukunng komputasi soft real-time.

Contoh penerapan sistem ini dalam kehidupan sehari-hari adalah pada alat penjual/pelayan otomatis. Jika mesin yang menggunakan sistem ini telah lama digunakan, maka mesin tersebut dapat mengalami penurunan kualitas, misalnya waktu pelayanannya menjadi lebih lambat dibandingkan ketika masih baru. Keterlambatan pada sistem ini tidak menyebabkan kecelakaan atau akibat fatal lainnya, melainkan hanya menyebabkan kerugian keuangan saja. Jika pelayanan mesin menjadi lambat, maka para pengguna dapat saja merasa tidak puas dan akhirnya dapat menurunkan pendapatan pemilik mesin.

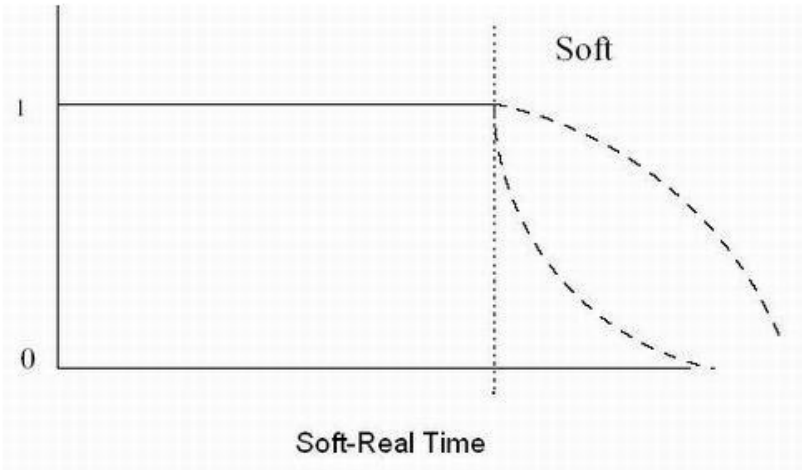
Untuk lebih memahami tentang perbedaan kedua sistem ini dapat diperhatikan dari diagram dibawah ini.

Gambar 16.1. Grafik *Hard Real-Time*



Setelah batas waktu yang diberikan telah habis, pada sistem *hard real-time*, aplikasi yang dijalankan langsung dihentikan. Akan tetapi, pada sistem *soft real-time*, aplikasi yang telah habis masa waktu pengerjaan tugasnya, dihentikan secara bertahap atau dengan kata lain masih diberikan toleransi waktu.

Gambar 16.2. Grafik *Soft Real-Time*



Mengimplementasikan fungsi soft real time membutuhkan design yang hati-hati dan aspek yang berkaitan dengan sistem operasi. Pertama, sistem harus punya prioritas penjadwalan, dan proses real-time harus memiliki prioritas tertinggi, tidak melampaui waktu, walaupun prioritas non real time dapat terjadi. Kedua, *dispatch latency* harus lebih kecil. Semakin kecil latency, semakin cepat real time proses mengeksekusi.

Untuk menjaga *dispatch* tetap rendah, kita butuh agar *system call* untuk *preemptible*. Ada beberapa cara untuk mencapai tujuan ini. Pertama adalah dengan memasukkan *preemption points* di durasi *system call* yang lama, yang memeriksa apakah prioritas utama butuh untuk dieksekusi. Jika sudah, maka *contex switch* mengambil alih, ketika *high priority* proses selesai, proses yang diinterupsi

meneruskan dengan *system call*. Points *premption* dapat diganti hanya di lokasi yang aman di kernel dimana kernel struktur tidak dapat dimodifikasi.

Metoda yang lain adalah dengan membuat semua kernel *preemptible*. Karena operasi yang benar dapat dijamin, semua struktur data kernel harus diproteksi dengan mekanisme sinkronisasi. Dengan metode ini, kernel dapat selalu di *preemptible*, karena setiap data kernel yang sedang di *update* diproteksi dengan pemberian prioritas yang tinggi. Jika ada proses dengan prioritas tinggi ingin membaca atau memodifikasi data kernel yang sedang dijalankan, prioritas yang tinggi harus menunggu sampai proses dengan prioritas rendah tersebut selesai. Situasi seperti ini dikenal dengan **priority inversion**. Kenyataanya, serangkaian proses dapat saja mengakses sumber daya yang sedang dibutuhkan oleh proses yang lebih tinggi prioritasnya. Masalah ini dapat diatasi dengan **priority-inheritance protocol**, yaitu semua proses yang sedang mengakses sumber daya mendapat prioritas tinggi sampai selesai menggunakan sumber daya. Setelah selesai, prioritas proses ini dikembalikan menjadi seperti semula.

Selain program penjadwal yang baik, pengimplementasian *soft real-time* harus memenuhi beberapa kriteria yang berkaitan dengan sistem operasi, yaitu:
1. Sistem harus mempunyai prioritas penjadwalan Proses real time harus memiliki prioritas tertinggi.
2. *Dispatch Latency* harus kecil Semakin kecil *latency* semakin cepat proses waktu nyata mulai berjalan.

Dispatch latency bisa menjadi besar jika terjadi suatu *system call*yang *non-preemptive*. Karena *system call* yang *non-preemptive* itu tidak dapat diinterrupt, waktu untuk beralih dari *system call* yang *non-preemptive* ke proses *soft real-time* menjadi besar.

Untuk mempertahankan *dispatch latency* yang kecil, *system call* harus dibuat *preemptible*, yaitu dengan menempatkan *preemption point* di dalam *system call* yang ada. *Preemption point* berfungsi untuk memeriksa apakah ada proses-proses berprioritas tinggi yang harus berjalan. Jadi, saat ada *system call* yang berjalan dan ada proses waktu nyata dengan prioritas tinggi yang hendak dijalankan, *system call* yang sedang berjalan dapat diinterupsi sehingga *dispatch latency* menjadi kecil.

Namun, *preemption point* hanya dapat diletakkan pada lokasi-lokasi kernel yang tidak mengandung struktur data yang sedang dimodifikasi. Oleh karena itu, hanya beberapa *preemption point* saja yang dapat ditempatkan pada kernel.

Solusi lain adalah dengan memberlakukan kernel yang *preemptible* seluruhnya. Dalam kernel yang *preemptible*, setiap proses dalam kernel dapat disuruh menunggu atau diinterupsi tanpa mempengaruhi datanya. Contohnya kernel linux 2.6.

Akan tetapi, ada kemungkinan proses yang mempunyai prioritas tinggi, memerlukan data yang sedang diakses oleh proses yang mempunyai prioritas lebih rendah. Maka, proses yang berprioritas tinggi harus menunggu dan membiarkan proses yang berprioritas rendah menyelesaikan eksekusinya. Situasi ini disebut pembalikan prioritas (*priority inversion*).

Namun, pembalikan prioritas memungkinkan proses yang prioritasnya lebih rendah dapat terus menerus mengakses sumber daya yang sedang dibutuhkan oleh proses yang prioritasnya lebih tinggi.

Solusi untuk masalah ini adalah priority inheritance protocol, yaitu semua proses yang sedang mengakses sumber daya mendapat prioritas tinggi sampai selesai menggunakan sumber daya. Setelah selesai, prioritas proses ini dikembalikan menjadi seperti semula.

16.6. Penjadwalan Thread

Ada beberapa variabel yang mempengaruhi penjadwalan thread, yaitu:
1. priority
2. scheduler and queue position
3. scheduling policy
4. contention scope
5. processor affinity
6. default

7. process level control
8. thread level control

Begitu dibuat, thread baru dapat dijalankan dengan berbagai macam penjadwalan. Kebijakan penjadwalanlah yang menentukan untuk setiap proses, di mana proses tersebut akan ditaruh dalam daftar proses sesuai dengan proritasnya, dan bagaimana ia bergerak dalam daftar proses tersebut.

Untuk menjadwalkan thread, sistem dengan model mulithreading many to many atau many to one menggunakan:

1. *Process Contention Scope* (PCS) Perpustakaan *thread* menjadwalkan *user thread* untuk berjalan pada LWP (*lightweight process*) yang tersedia.
2. *System Contention Scope* (SCS) SCS berfungsi untuk memilih satu dari banyak *thread* , kemudian menjadwalkannya ke satu *thread* tertentu (CPU / Kernel).

Sistem dengan model *multithreading one to one* hanya menggunakan SCS. SCHED_FIFO adalah penjadwalan *First In First out real time threads*. Peraturan-peraturan yang berlaku dalam SCHED_FIFO adalah:

1. Sistem tidak akan diinterupsi, kecuali:
   - Ada FIFO *thread* lain yang memiliki prioritas lebih tinggi telah siap.
   - Ada FIFO *thread* yang sedang berjalan diblok untuk menunggu suatu *event*, misalnya M/K.
2. Jika FIFO *thread* yang sedang berjalan diinterupsi, *thread* tersebut akan diletakkan pada *queue* yang sesuai dengan prioritasnya.
3. Jika FIFO *thread* yang memiliki prioritas lebih tinggi daripada *thread* yang sedang berjalan telah siap maka FIFO *thread* tersebut akan dijalankan. Jika ada lebih dari satu FIFO *thread* yang siap, *thread* yang paling lama menunggu akan dijalankan.

SCHED_RR adalah penjadwalan *Round Robin real time threads* . Contoh kasus untuk membedakan SCHED_FIFO dengan SCHED_RR:

ready queue

Proses	Proritas
A	minimal
B	menengah
C	menengah
D	maksimal

Urutan pengerjaan *thread* pada SCHED_FIFO adalah D-B-C-A. Urutan pengerjaan *thread* pada SCHED_RR adalah D-B-C-B-C-A. Pada SCHED_RR terjadi *time slicing* yaitu saat perpindahan *thread* dari B ke C ke B ke C. Hal itu memungkinkan karena SCHED_RR menggunakan waktu kuota untuk *thread-thread*nya.

B dan C memiliki prioritas yang sama tetapi B dijalankan lebih dahulu karena B telah lebih lama menunggu. Setelah waktu kuota B habis, C akan dijalankan. Setelah waktu kuota C habis,B dijalankan kembali. Kemudian C dijalankan dan A yang memiliki prioritas paling kecil dijalankan paling akhir.

SCHED_OTHER adalah penjadwalan untuk *non real time threads* . Jika tidak ada *thread* waktu nyata yang siap, maka *thread-thread* dalam SCHED_OTHER akan dijalankan.

16.7. Penjadwalan Java

Prioritas suatu thread dalam Java biasanya didapat dari prioritas thread yang menciptakannya (parent thread), yang secara default bernilai 5. Namun kita dapat menentukan secara manual prioritas dari thread yang sedang berjalan. Java telah menyediakan method setPriority yang menerima argumen integer dalam jangkauan 1-10.

Java menggunakan konsep round robin dalam menjadwalkan thread-thread yang berjalan. Setiap

thread mendapat jatah waktu tertentu untuk berjalan yang disebut quantum atau time slice. Pelaksanaannya dimulai dari thread dengan prioritas tertinggi. Bila jatah waktu thread tersebut habis maka thread lain dengan prioritas yang sama (bila ada) yang akan dijalankan (dengan metode round robin) meskipun thread tersebut belum selesai melaksanakan tugasnya. Hal ini dilakukan sampai thread dengan prioritas lebih tinggi sudah selesai semua melaksanakan tugasnya baru thread dengan prioritas lebih rendah dilaksanakan dengan metode yang sama.

Contoh Program yang menggunakan *method* setPriority()

Contoh 16.1. DialogThread

```
/*
 *   Program DialogThread.java
 *   Author   : Anthony Steven, 120300017X
 *   Tanggal  : 22 September, 2005
 *   CopyLeft : Silahkan memakai program ini untuk kepentingan
 *              akademis
 */

  // Membuat kelas DialogThread
  class DialogThread extends Thread
  {
        String kata;
        int prioritas;

        public DialogThread(String kata, int prioritas){
          this.kata       = kata;
          this.prioritas  = prioritas;
        }

        public synchronized void run(){
          setPriority(prioritas); // setPriority Awal
          for (int i = 0 ; i < 10 ; i ++ )
          {
                System.out.println(kata + " prioritas saya = " +
                getPriority());
                 /*
                     Awal Bagian Pengubahan Prioritas
                */

                if(i == 5 && prioritas == 10){
                       System.out.println("prioritas dikurangi");
                       setPriority(prioritas - 9);
                }
                else if(i == 5 && prioritas == 1){
                       System.out.println("prioritas ditambah");
                       setPriority(prioritas + 9);
                }
                try{Thread.sleep(200);}
                catch(InterruptedException ie){};
                /*
                     Akhir Bagian Pengubahan Prioritas
                */
          }
        }
  }
```

```
// Kelas Aplikasi
public class DialogApplication
{
       public static void main(String [] args){
         DialogThread dt1 = new DialogThread("Dialog Thread No 1,"+
        "sedang berbicara", 10);
         DialogThread dt2 = new DialogThread("Dialog Thread No 2,"+
        "Lagi SPEAK-SPEAK", 10);
         DialogThread dt3 = new DialogThread("Dialog Thread No 3,"+
        "Ngomong Loh", 1);
         DialogThread dt4 = new DialogThread("Dialog Thread No 4,"+
        "Santai AAH", 1);
         dt1.start();
         dt2.start();
         dt3.start();
         dt4.start();
       }
}
```

Penjadwalan di Java berkaitan dengan konsep *multithreading*. Java mendukung konsep thread. Java juga sudah mengatur penjadwalan *thread-thread* tersebut secara internal. Secara umum hanya terdapat satu *thread* yang berjalan di Java (*Thread Main*).

Banyak method yang disediakan Java untuk mengatur perilaku sebuah *thread*, diantaranya untuk menjalankan suatu *thread*, membuat suatu *thread* untuk menunggu, ataupun untuk menghentikan *thread* yang sedang berjalan. Method-method ini dapat dilihat di Java API.

Dalam Java, sebuah thread dapat berada dalam status New (biasa juga disebut Born), Runing/Runable, Blocked (termasuk juga disini status Sleeping dan Waiting) dan Dead atau Terminated.

Setiap thread yang berjalan memiliki prioritas, dengan prioritas tertinggi bernilai 10 dan prioritas terendah bernilai 1. Semakin tinggi prioritas suatu thread maka semakin diutamakan thread tersebut dalam pelaksanaannya. Bagaimanapun tingginya prioritas suatu thread hal itu tidak menjamin urutan eksekusi thread tersebut.

Output program, jika bagian pengubahan prioritas dihilangkan:

Contoh 16.2. DialogThread

```
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 3, Ngomong Loh prioritas saya = 1
Dialog Thread No 4, Santai AAH prioritas saya = 1
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 3, Ngomong Loh prioritas saya = 1
Dialog Thread No 4, Santai AAH prioritas saya = 1
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 3, Ngomong Loh prioritas saya = 1
Dialog Thread No 4, Santai AAH prioritas saya = 1
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 4, Santai AAH prioritas saya = 1
Dialog Thread No 3, Ngomong Loh prioritas saya = 1
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 3, Ngomong Loh prioritas saya = 1
Dialog Thread No 4, Santai AAH prioritas saya = 1
```

```
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 4, Santai AAH prioritas saya = 1
Dialog Thread No 3, Ngomong Loh prioritas saya = 1
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 3, Ngomong Loh prioritas saya = 1
Dialog Thread No 4, Santai AAH prioritas saya = 1
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 4, Santai AAH prioritas saya = 1
Dialog Thread No 3, Ngomong Loh prioritas saya = 1
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 4, Santai AAH prioritas saya = 1
Dialog Thread No 3, Ngomong Loh prioritas saya = 1
Dialog Thread No 2, Lagi SPEAK-SPEAK prioritas saya = 10
Dialog Thread No 1, sedang berbicara prioritas saya = 10
Dialog Thread No 4, Santai AAH prioritas saya = 1
```

Dialog *thread* yang bernomor 1 dan 2 akan terus berada di atas no 3 dan 4, karena Dialog *thread* yang bernomor 1 dan 2 lebih diutamakan.

Output program, jika bagian pengubahan prioritas diadakan:

Dialog Thread yang bernomor 1 dan 2 akan terus berada di atas no 3 dan 4, karena lebih diutamakan, akan tetapi setelah ada pengubahan prioritas, Dialog Thread no 3 dan 4 lebih diutamakan.

Java menggunakan konsep round-robin dalam menjadwalkan *thread*nya, setiap *thread* mendapatkan jatah waktu, jika jatah waktunya habis, maka *thread* lain dengan prioritas yang sama akan dijalankan, meskipun *thread* tersebut belum selesai tugasnya. Jika ada dua *thread* dengan prioritas yang sama hendak berjalan maka Java menggunakan konsep FIFO.

16.8. Kinerja

Ada banyak algoritma penjadwalan, yang mana masing-masing memiliki parameter tersendiri sebagai ukuran dari kinerjanya, sehingga cukup sulit untuk memilih diantara algoritma-algoritma tersebut yang kinerjanya paling baik.

Kinerja dari berbagai macam algoritma merupakan faktor yang penting dalam memilih sebuah algoritma penjadwalan. Bagaimanpun, hampir tidak mungkin untuk membuat perbandingan yang sempurna karena kinerja suatu algoritma relatif bergantung pada banyak faktor, diantaranya termasuk faktor distribusi probabilitas waktu kerja dari berbagai proses, efisiensi algoritma dan mekanisme *context switching* , dan kinerja dari I/O. Walaupun begitu, kita tetap akan membahas evaluasi algoritma dalam kondisi umum.

Cara yang paling akurat dan lengkap untuk mengevaluasi algoritma penjadwalan adalah dengan cara membuat kodenya, meletakkannya di sistem operasi dan melihat bagiamana kode itu bekerja. Pendekatan seperti ini dilakukan dengan menempatkan algoritma yang sebenarnya pada real time system untuk dievalusi di bawah kondisi yang juga real.

Kesulitan terbesar dari cara pendekatan seperti ini adalah pada cost atau biaya. Cost yang tinggi tidak hanya pada saat membuat kode algoritma dan memodifikasi sistem operasi untuk mensupport algoritma tersebut sesuai dengan keperluan struktur datanya, tetapi cost yang tinggi juga ada pada reaksi user terhadap modifikasi sistem operasi.

Peningkatan kinerja dari sistem dengan prosesor jamak adalah adanya efisiensi waktu, cost dan resource dari penggunaan prosesor yang lebih dari satu. Untuk model asynchronous adalah mengalokasikan penjadwalan, pemrosesan M/K, dan kegiatan sistem lainnya kepada satu prosesor tertentu kepada master. Sedangkan prosesor yang lain hanya bertugas mengeksekusi user code.

Permodelan Deterministik (*Deterministic Modelling*)

Salah satu metoda evaluasi algoritma yang disebut *analysis evaluation*, menggunakan algoritma yang diberikan dan waktu kerja sistem untuk menghasilkan sebuah nilai yang dapat menentukan kinerja algoritma tersebut. Salah satu jenis dari metoda ini adalah permodelan deterministik. Metoda ini mengambil suatu sistem dan mendefinisikan kinerja setiap algoritma untuk sistem tersebut.

Untuk lebih jelasnya, mari kita bersama mengevaluasi lima proses dengan menggunakan algoritma FCFS (*First-Come-Fist- Served*), SJF(*Short Jobs First*) *non-preemptive*, dan RR (*Round-Robin*).

Contoh Soal:

Proses	Burst time
A	12
B	27
C	4
D	9
E	15

Dari proses antria tersebut, dapat kita peroleh berbagai informasi. Salah satunya dapat dituangkan dalam bentuk Little's Formula:

n = A X W

```
Keterangan:
n = rata-rata panjang antrian proses.
A = rata-rata waktu tiba arrival rate dari proses mengantri.
W = rata-rata waiting time dalam antrian.
```

Bandingkan rata-rata *waiting time* dari masing-masing algoritma?

FCFS (*First-Come-Fist-Served*)

rata-rata *waiting time* = (0+12+39+53+62) / 5 = 33,2 satuan waktu

SJF (*Short Jobs First*) *non-preemptive*

rata-rata *waiting time* = (13+40+0+4+25) / 5 = 16,4 satuan waktu

RR (*Round-Robin*) (q = 10)

rata-rata *waiting time* = (33+40+20+24+45) / 5 = 32,4 satuan waktu

Dari perhitungan di atas, kita dapati algoritma SJF sebagai algoritma yang memiliki waiting time yang terpendek. Bila kita perhatikan lebih lanjut, dengan metoda ini hampir semua hasil evaluasi menunjukan algoritma SJF sebagai pemenangnya. Tetapi itu hanyalah perhitungan sedehana saja, sehingga belum bisa dijadikan patokan untuk menentukan kinerja algoritma yang baik.

Metoda ini memang sangat sederhana dan cepat, serta memberikan suatu nilai yang eksak. Sayangnya dalam perhitungan kita membutuhkan nilai input yang juga eksak, sehingga perhitungan ini terlalu spesifik atau kurang umum dalam menentukan kinerja algoritma.

Metoda ini lebih dalam dapat anda baca dalam buku *Operating Systems: Internal and Design Principles* Edisi Keempat karangan William Stallings.

16.9. Rangkuman

Telah dibahas mengenai penjadwalan prosesor jamak, penjadwalan waktu nyata, penjadwalan thread, penjadwalan Java, serta evaluasi algoritma. Penjadwalan CPU pada prosesor jamak dibagi menjadi dua, yaitu: proses jamak asimetris dan proses jamak simetris. Semua sistem operasi modern mendukung proses jamak simetris, termasuk Windows XP, Windows 2000, Solaris, Linux, dan Mac OS X.

Proses waktu nyata adalah proses yang memiliki batasan waktu. Ciri-ciri proses waktu nyata, yaitu: deterministik, bersifat primitif, responsif, dan sulit berkolaborasi dengan *virtual machine* dan sistem prosesor jamak. Sistem waktu nyata dapat dibagi dua, yaitu sistem *Hard Real Time* dan sistem *Soft Real Time* .

Ada dua jenis thread, yaitu kernel thread dan user thread. Kernel thread dijadwalkan oleh sistem operasi sedangkan user thread dijadwalkan oleh perpustakaan thread. Perpustakaan thread menyediakan 3 jenis penjadwalan, yaitu: SCHED_FIFO, SCHED_RR, dan SCHED_OTHER.

Java mendukung konsep *multithreading*, dimana penjadwalan *thread* pada Java mengunakan konsep Round-Robin dan FIFO.

Kriteria kinerja algoritma yang baik adalah:
i. Maksimalisasi utilisasi CPU dengan response time maksimal = 1 detik.
ii. Maksimalisasi throughput sehingga rata-rata turnaround time menjadi sebanding terhadap total waktu eksekusi.

Ada empat metoda evaluasi untuk memenuhi kriteria tersebut, yaitu: permodelan deterministik, model antrian, simulasi, dan implementasi.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[WEBIBMNY] IBM Corporation. NY. *General Programming Concepts – Writing and Debugging Programs – http://publib16.boulder.ibm.com/ pseries/ en_US/ aixprggd/ genprogc/ ls_sched_subr.htm* . Diakses 1 Juni 2006.

[WEBIBM1997] IBM Coorporation. 1997. *General Programming Concepts: Writing and Debugging Programs – Threads Scheduling http://www.unet.univie.ac.at/ aix/ aixprggd/ genprogc/ threads_sched.htm* . Diakses 1 Juni 2006.

[WEBLindsey2003] Clark S Lindsey. 2003. *Physics Simulation with Java – Thread Scheduling and Priority – http://www.particle.kth.se/ ~fmi/ kurs/ PhysicsSimulation/ Lectures/ 10A/ schedulePriority.html* . Diakses 1 Juni 2006.

[WEBMooreDrakos1999] Ross Moore dan Nikos Drakos. 1999. *Converse Programming Manual – Thread Scheduling Hooks – http://charm.cs.uiuc.edu/ manuals/ html/ converse/ 3_3Thread_ Scheduling_Hooks.html* . Diakses 1 Juni 2006.

[WEBVolz2003] Richard A Volz. 2003. *Real Time Computing – Thread and Scheduling Basics – http://linserver.cs.tamu.edu/ ~ravolz/ 456/ Chapter-3.pdf* . Diakses 1 Juni 2006.

Bab 17. Manajemen Proses Linux

17.1. Pendahuluan

Setiap aplikasi yang dijalankan di Linux mempunyai pengenal yang disebut sebagai *Process Identification Number* (PID). Hingga kernel versi 2.4, PID disimpan dalam angka 16 bit dengan kisaran dari 0-32767 untuk menjamin kompatibilitas dengan unix. Dari nomor PID inilah linux dapat mengawasi dan mengatur proses-proses yang terjadi didalam system. Proses yang dijalankan ataupun yang baru dibuat mempunyai struktur data yang disimpan di `task_struct`.

Linux mengatur semua proses di dalam sistem melalui pemeriksaan dan perubahan terhadap setiap struktur data task_struct yang dimiliki setiap proses. Sebuah daftar pointer ke semua struktur data `task_struct` disimpan dalam task vector. Jumlah maksimum proses dalam sistem dibatasi oleh ukuran dari task vector. Linux umumnya memiliki task vector dengan ukuran 512 entries. Saat proses dibuat, task_struct baru dialokasikan dari memori sistem dan ditambahkan ke task vector. Linux juga mendukung proses secara real time. Proses semacam ini harus bereaksi sangat cepat terhadap event eksternal dan diperlakukan berbeda dari proses biasa lainnya oleh penjadwal.

Proses akan berakhir ketika ia memanggil `exit()`. Kernel akan menentukan waktu pelepasan sumber daya yang dimiliki oleh proses yang telah selesai tersebut. Fungsi `do_exit()` akan dipanggil saat terminasi yang kemudian memanggil `__exit_mm/files/fs/sighand()` yang akan membebaskan sumber daya. Fungsi `exit_notify()` akan memperbarui hubungan antara proses induk dan proses anak, semua proses anak yang induknya berakhir akan menjadi anak dari proses init. Terakhir akan dipanggil scheduler untuk menjalankan proses baru.

17.2. Deskriptor Proses

Guna keperluan manajemen proses, kernel memelihara informasi tentang setiap proses di sebuah deskriptor proses dengan tipe task_struct. Setiap deskriptor proses mengandung informasi antara lain status proses, ruang alamat, daftar berkas yang dibuka, prioritas proses, dan sebagainya. Berikut gambaran isinya:

Contoh 17.1. Isi Deskriptor Proses

```
struct task_struct{
    volatile long state; /* -1 unrunnable,
                            0 runnable,
                           >0 stopped       */
    unsigned long flags;
                /* 1 untuk setiap flag proses */
    mm_segment_t_addr_limit;
                /* ruang alamat untuk thread  */
    struct exec_domain *exec_domain;
    long need_resched;
    long counter;
    long priority; /* SMP and runqueue state  */
    struct task_struct *next_task, *prev_task;
    struct task_struct *next_run, *prev_run;
    ...
    /* task state                              */
    /* limits                                  */
    /* file system info                        */
    /* ipc stuff                               */
    /* tss for this task                       */
    /* filesystem information                  */
    /* open file information                   */
    /* memory manajement info                  */
    /* signal handlers                         */
};
```

Setiap proses di Linux memiliki status. Status proses merupakan array dari flag yang mutually exclusive. Setiap proses memiliki tepat satu keadaan (status) pada suatu waktu. Status tersebut adalah:

- **TASK_RUNNING.** Pada status ini, proses sedang atau pun siap dieksekusi oleh CPU.
- **TASK_INTERRUPTIBLE.** Pada status ini, proses sedang menunggu sebuah kondisi. Interupsi, sinyal, atau pun pelepasan sumber daya akan membangunkan proses.
- **TASK_UNINTERRUPTIBLE.** Pada status ini, proses sedang tidur dan tidak dapat dibangunkan oleh suatu sinyal.
- **TASK_STOPPED.** Pada status ini proses sedang dihentikan, misalnya oleh sebuah debugger.
- **TASK_ZOMBIE.** Pada status ini proses telah berhenti, namun masih memiliki struktur data task_struct di task vector dan masih memegang sumber daya yang sudah tidak digunakan lagi.

Setiap proses atau pun eksekusi yang terjadwal secara independen memiliki deskriptor prosesnya sendiri. Alamat dari deskriptor proses digunakan untuk mengindentifikasi proses. Selain itu, nomor ID proses (PIDs) juga digunakan untuk keperluan tersebut. PIDs adalah 16-bit bilangan yang mengidentifikasikan setiap proses dengan unik. Linux membatasi PIDs berkisar 0-32767 untuk menjamin kompatibilitas dengan sistem UNIX tradisional.

Karena proses merupakan sesuatu yang dinamis, maka deskriptor proses disimpan dalam memori yang dinamis pula. Untuk itu dialokasikan juga memori sebesar 8KB untuk setiap proses untuk menyimpan proses deskriptornya dan stack proses dari modus kernel. Keuntungan dari dal ini adalah pointer dari deskriptor proses dari proses yang sedang berjalan (running) dapat diakses dengan cepat menggunakan stack pointer. Selain itu, 8KB (EXTRA_TASK_STRUCT) dari memori akan di-cache untuk mem-bypass pengalokasi memori kernel ketika sebuah proses dihapus dan sebuah proses baru dibuat. Kedua perintah free_task_struct() dan alloc_task_struct() akan digunakan untuk melepaskan atau mengalokasikan memori seukuran 8KB sebagai cache.

Deskriptor proses juga membangun sebuah daftar proses dari semua proses yang ada di sistem. Daftar proses tersebut merupakan sebuah doubly-linked list yang dibangun oleh bagian next_task dan prev_task dari deskriptor proses. Deskriptor init_task(mis:swapper) berada di awal daftar tersebut dengan prev_task-nya menunjuk ke deskriptor proses yang paling akhir masuk dalam daftar. Sedangkan makro for_each_task() digunakan untuk memindai seluruh daftar.

Proses yang dijadwalkan untuk dieksekusi dari doubly-linked list dari proses dengan status TASK_RUNNING disebut runqueue. Bagian prev_run dan next_run dari deskriptor proses digunakan untuk membangun runqueue, dengan init_task mengawali daftar tersebut. Sedangkan untuk memanipulasi daftar di deskriptor proses tersebut, digunakan fungsi-fungsi: add_to_runqueue(), del_from_runqueue(), move_first_runqueue(), move_last_runqueue(). Makro NR_RUNNING digunakan untuk menyimpan jumlah proses yang dapat dijalankan, sedangkan fungsi wake_up_process membuat sebuah proses menjadi dapat dijalankan.

Untuk menjamin akurasinya, array task akan diperbarui setiap kali ada proses baru dibuat atau pun dihapus. Sebuah daftar terpisah akan melacak elemen bebas dalam array task itu. Ketika suatu proses dihapus, entrinya ditambahkan di bagian awal dari daftar tersebut.

Proses dengan status task_interruptible dibagi ke dalam kelas-kelas yang terkait dengan suatu event tertentu. Event yang dimaksud misalnya: waktu kadaluarsa, ketersediaan sumber daya. Untuk setiap event atau pun kelas terdapat antrian tunggu yang terpisah. Proses akan diberi sinyal bangun ketika event yang ditunggunya terjadi. Berikut contoh dari antrian tunggu tersebut:

Terdapat tiga pilihan kebijakan penjadwalan thread, yaitu:

1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR: Round Robin Scheduling

SCHED_OTHER

SCHED_OTHER adalah penjadwalan standar linux, dan kebijakan penjadwalan default yang digunakan pada kebanyakan proses. Proses yang berjalan dipilih dari daftar prioritas static 0 yang ditentukan berdasarkan pada prioritas dinamik, dan prioritas dinamik ditentukan di dalam daftar prioritas. Prioritas dinamik ditentukan berdasarkan pada level nice (diset oleh sistem call nice atau setpriority), dan bertambah setiap kali proses siap dijalankan tetapi dibatalkan oleh scheduler. Ini

menjamin kemajuan yang fair diantara semua proses SCHED_OTHER. Pada linux yang standar, ini adalah satu-satunya scheduler yang user biasa dapat jalankan. Standar linux memerlukan keistimewaan superuser untuk menjalankan proses dengan SCHED_FIFO atau SCHED_RR.

SCHED_FIFO: First In First Out scheduling

Ketika proses SCHED_FIFO dapat dijalankan, ia akan segera mem-preempt semua proses SCHED_OTHER yang sedang berjalan normal. SCHED_FIFO adalah algoritma scheduling sederhana tanpa time slicing. Untuk proses yang berdasarkan pada kebijakan SCHED_FIFO, aturan berikut diberlakukan: Sebuah proses SCHED_FIFO yang telahdi-preempted oleh proses lain akan menempati urutan atas untuk prioritasnya dan akan memulai kembali exekusinya segera setelah proses yang prioritasnya lebih tinggi diblock. Ketika proses SCHED_FIFO dapat dijalankan, ia akan dimasukkan pada urutan akhir dari daftar untuk prioritasnya. Sebuah call untuk sched_setscheduler atau sched_param akan membuat proses SCHED_FIFO diidentifikasi oleh pid pada akhir dari list jika ia runnable. Sebuah proses memanggil sched_yield akan ditaruh diakhir dari list. Tidak ada event yang dapat memindahkan proses yang dijadwalkan berdasarkan kebijakan SCHED_FIFO di daftar tunggu runnable process dengan prioritas statik yang sama. Proses SCHED_FIFO berjalan sampai diblok oleh permintaan I/O, di-preempt oleh proses yang berprioritas lebih tinggi, memanggil sched_yield, atau proses tersebut sudah selesai.

SCHED_RR: Round Robin Scheduling

SCHED_RR adalah peningkatan sederhana dari SCHED_FIFO. Semua aturan yang dijelaskan pada SCHED_FIFO juga digunakan pada SCHED_RR, kecuali proses yang hanya diizinkan berjalan untuk sebuah maksimum time quantum. Jika proses telah berjalan selama waktunya atau bahkan lebih lama dari waktu kuantumnya, ia akan ditaruh di bagian akhir dari daftar prioritas. Panjang kuantum time dapat dipulihkan kembali dengan sched_rr_get_interval.

Contoh 17.2. Antrian Tunggu

```
void sleep_on(struct wait_queue **wqptr) {
    struct wait_queue wait;
    current_state=TASK_UNINTERRUPTIBLE;
    wait.task=current;
    add_wait_queue(wqptr, &wait);
    schedule();
    remove_wait_queue(wqptr, &wait);
}
```

Fungsi sleep_on() akan memasukkan suatu proses ke dalam antrian tunggu yang diinginkan dan memulai penjadwal. Ketika proses itu mendapat sinyal untuk bangun, maka proses tersebut akan dihapus dari antrian tunggu.

Bagian lain konteks eksekusi proses adalah konteks perangkat keras, misalnya: isi register. Konteks dari perangkat keras akan disimpan oleh task state segment dan stack modus kernel. Secara khusus tss akan menyimpan konteks yang tidak secara otomatis disimpan oleh perangkat keras tersebut. Perpindahan antar proses melibatkan penyimpanan konteks dari proses yang sebelumnya dan proses berikutnya. Hal ini harus dapat dilakukan dengan cepat untuk mencegah terbuangnya waktu CPU. Versi baru dari Linux mengganti perpindahan konteks perangkat keras ini menggunakan piranti lunak yang mengimplementasikan sederetan instruksi mov untuk menjamin validasi data yang disimpan serta potensi untuk melakukan optimasi.

Untuk mengubah konteks proses digunakan makro switch_to(). Makro tersebut akan mengganti proses dari proses yang ditunjuk oleh prev_task menjadi next_task. Makro switch_to() dijalankan oleh schedule() dan merupakan salah satu rutin kernel yang sangat tergantung pada perangkat keras (hardware-dependent). Lebih jelas dapat dilihat pada kernel/sched.c dan include/asm-*/system.h.

17.3. Proses dan *Thread*

Dewasa ini (2006), banyak sistem operasi yang telah mendukung proses *multithreading*. Setiap sistem operasi memiliki konsep tersendiri dalam mengimplementasikannya ke dalam sistem.

Linux menggunakan representasi yang sama untuk proses dan thread. Secara sederhana thread dapat dikatakan sebuah proses baru yang berbagi alamat yang sama dengan induknya. Perbedaannnya terletak pada saat pembuatannya. Thread baru dibuat dengan system call `clone` yang membuat proses baru dengan identitas sendiri, namun diizinkan untuk berbagi struktur data dengan induknya.

Secara tradisional, sumber daya yang dimiliki oleh proses induk akan diduplikasi ketika membuat proses anak. Penyalinan ruang alamat ini berjalan lambat, sehingga untuk mengatasinya, salinan hanya dibuat ketika salah satu dari mereka hendak menulis di alamat tersebut. Selain itu, ketika mereka akan berbagi alamat tersebut ketika mereka hanya membaca. Inilah proses ringan yang dikenal juga dengan thread.

Thread dibuat dengan `__clone()`. `__clone()` merupakan rutin dari library system call `clone()`. `__clone` memiliki empat buah argumen yaitu:

1. `fn`

   fungsi yang akan dieksekusi oleh thread baru
2. `arg`

   pointer ke data yang dibawa oleh `fn`
3. `flags`

   sinyal yang dikirim ke induk ketika anak berakhir dan pembagian sumber daya antara anak dan induk.
4. `child_stack`

   pointer stack untuk proses anak.

`clone()` mengambil argumen `flags` dan `child_stack` yang dimiliki oleh `__clone` kemudian menentukan id dari proses anak yang akan mengeksekusi `fn` dengan argumen `arg`.

Pembuatan anak proses dapat dilakukan dengan fungsi `fork()` dan `vfork()`. Implementasi `fork()` sama seperti system call `clone()` dengan sighandler `SIGCHLD` di-set, semua bendera clone di-clear yang berarti tidak ada sharing dan `child_stack` dibuat 0 yang berarti kernel akan membuat stack untuk anak saat hendak menulis. Sedangkan `vfork()` sama seperti `fork()` dengan tambahan bendera `CLONE_VM` dan `CLONE_VFORK` di-set. Dengan `vfork()`, induk dan anak akan berbagi alamat, dan induk akan di-block hingga anak selesai.

Untuk memulai pembuatan proses baru, `clone()` akan memanggil fungsi `do_fork()`. Hal yang dilakukan oleh `do_fork()` antara lain:
- memanggil `alloc_task_struct()` yang akan menyediakan tempat di memori dengan ukuran 8KB untuk deskriptor proses dan stack modus kernel.
- memeriksa ketersediaan sumber daya untuk membuat proses baru.
- `find_empty_procees()` memanggil `get_free_taskslot()` untuk mencari sebuah slot di array task untuk pointer ke deskriptor proses yang baru.
- memanggil `copy_files/fm/sighand/mm()` untuk menyalin sumber daya untuk anak, berdasarkan nilai flags yang ditentukan `clone()`.
- `copy_thread()` akan menginisialisasi stack kernel dari proses anak.
- mendapatkan PID baru untuk anak yang akan diberikan kembali ke induknya ketika `do_fork()` selesai.

Beberapa proses sistem hanya berjalan dalam modus kernel di belakang layar. Untuk proses semacam ini dapat digunakan thread kernel. Thread kernel hanya akan mengeksekusi fungsi kernel, yaitu fungsi yang biasanya dipanggil oleh proses normal melalui system calls. Thread kernel juga hanya dieksekusi dalam modus kernel, berbeda dengan proses biasa. Alamat linier yang digunakan oleh thread kernel lebih besar dari PAGE_OFFSET proses normal yang dapat berukuran hingga 4GB. Thread kernel dibuat sebagai berikut:

Contoh 17.3. Thread Kernel

```
int kernel_thread(int (*fn) (void *), void *arg, unsigned long flags);
flags=CLONE_SIGHAND, CLONE_FILES, etc.
```

Kernel Linux mulai menggunakan *thread* pada versi 2.2. *Thread* dalam Linux dianggap sebagai *task*, seperti halnya proses. Kebanyakan sistem operasi yang mengimplementasikan *multithreading* menjalankan sebuah *thread* terpisah dari proses. Linus Torvalds mendefinisikan bahwa sebuah thread adalah *Context of Execution* (COE), yang berarti bahwa hanya ada sebuah *Process Control Block* (PCB) dan sebuah penjadwal yang diperlukan. Linux tidak mendukung *multithreading*, struktur data yang terpisah, atau pun rutin *kernel*.

Setiap proses memiliki struktur data yang unik. Namun demikian, proses-proses di Linux hanya menyimpan *pointer-pointer* ke struktur data lainnya dimana instruksi disimpan, sehingga tidak harus menyimpan instruksi ke setiap struktur data yang ada. Hal ini menyebabkan *context switch* antar proses di Linux menjadi lebih cepat.

Linux menyediakan dua macam *system call*, yaitu *fork* dan *clone*. *fork* memiliki fungsi untuk menduplikasi proses dimana proses anak yang dihasilkan bersifat *independent*. *clone* memiliki sifat yang mirip dengan *fork* yaitu sama-sama membuat duplikat dari proses induk. Namun demikian, selain membuat proses baru yang terpisah dari proses induk, *clone* juga mengizinkan terjadinya proses berbagi ruang alamat antara proses anak dengan proses induk, sehingga proses anak yang dihasilkan akan sama persis dengan proses induknya.

Ketika *fork* dieksekusi, sebuah proses baru dibuat bersamaan dengan proses penyalinan struktur data dari proses induk. Ketika *clone* dieksekusi, sebuah proses baru juga dibuat, namun proses tersebut tidak menyalin struktur data dari proses induknya. Proses baru tersebut hanya menyimpan *pointer* ke struktur data proses induk. Oleh karena itu, proses anak dapat berbagi ruang alamat dan sumber daya dengan proses induknya. Satu set *flag* digunakan untuk mengindikasikan seberapa banyak kedua proses tersebut dapat berbagi. Jika tidak ada *flag* yang ditandai, maka tidak ada *sharing*, sehingga *clone* berlaku sebagai *fork*. Jika kelima *flag* ditandai, maka proses induk harus berbagi semuanya dengan proses anak.

Tabel 17.1. Tabel Flag dan Fungsinya

Flag	Keterangan
CLONE_VM	Berbagi data dan Stack
CLONE_FS	Berbagi informasi sistem berkas
CLONE_FILES	Berbagi berkas
CLONE_SIGHAND	Berbagi sinyal
CLONE_PID	Berbagi PID dengan proses induk

Identitas Proses

Identitas proses utamanya terbagi dalam tiga hal berikut:

Proses ID (PID)

Setiap proses mempunyai nomor yang unik. PID digunakan untuk melabeli proses sistem operasi ketika sebuah aplikasi memanggil system call

Credentials

setiap proses harus mempunyai user id tertentu dan satu atau lebih group id yang menentukan hak dari proses untuk mengakses resource sistem dan file

Personality

Personality digunakan agar system call kompatibel dengan ciri tertentu dari unix. Berikut contoh penggunaan system call fork.

Contoh 17.4. Contoh pengambilan nilai PID proses

```
#include <sys/type.h>
#include <stdio.h>
int main()
{
   pid_t mypid, myparentpid; /* deklarasi variabel penampung */
   mypid       = getpid();   /* ambil ID proses ini          */
   myparentpid = getppid();  /*ambil ID parent proses ini    */
   printf(PID Saya = %d\n, mypid); /* tampilkan PID          */
   printf(PID PARENT Saya = %d\n, myparentpid); /* PPID      */
 }
```

Contoh 17.5. Fork Tanpa Menggunakan Wait

```
/* Contoh fork tanpa menggunakan wait sehingga proses
   berjalan paralel */
#include <sys/type.h>
#include <stdio.h>
int main()
{
   pid_t pid;
   pid = fork();
   if(pid < 0)
   {
      printf(Fork gagal\n);
      exit(EXIT_FAILURE);
   }
   if(pid == 0)
   {
      printf(CHILD: Ini proses Child\n);
   }
   else
   {
      printf(PARENT: Ini proses Parent\n);
   }
}
```

Contoh 17.6. Penggunaan Fork Dengan Wait

```
/* Contoh penggunaan fork dengan wait sehingga proses
   menunggu anaknya terlebih dahulu */
#include <sys/type.h>
#include <stdio.h>
int main()
{
   pid_t pid;
   pid = fork();
   if(pid < 0)
   {
      printf(Fork gagal\n);
      exit(EXIT_FAILURE);
   }
   if(pid == 0)
   {
      printf(CHILD: Ini proses Child\n);
   }
   else
   {
      wait(NULL);
      printf(PARENT: Ini proses Parent\n);
   }
}
```

Dalam Linux kita dapat melihat proses yang sedang berlangsung dengan perintah top. Dengan perintah top kita akan menemui serangkaian proses yang berlangsung lengkap dengan PID, Load Average, Time, dan proses-proses yang lain. Berikut ini adalah salah satu contoh keluaran ketika dilakukan perintah top dalam Linux:

Contoh 17.7. TOP

```
top - 06:47:01 up 23 min, 3 users, load average: 0.88, 0.40, 0.41
Task: 68 total, 2 running, 66 sleeping, 0 stopped, 0 zombie
CPU(s):12.1% us, 4.3% sy, 0.3% ni, 70.6% id, 12.4% wa, 0.2% hi, o.o% si
Mem: 117796k total, 110388k used, 7408k free, 4908k buffers
Swap: 256032k total, 33304k used, 231728k free, 31800k cached

PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM  TIME+  COMMAND

7637 akhmad   16  0   2076  940  740  R   3.9  0.8  0:00.03  top
1 root        16  0   1548  468  428  S   0.0  0.4  0:00.80  init
2 root        34  19    0    0    0   S   0.0  0.0  0:00.02  ksoftirqd/0
3 root       5   -10   0    0    0   S   0.0  0.0  0:00.06  events/0
4 root        6   -10   0    0    0   S   0.0  0.0  0:00.06  khelper
   .          .    .    .    .    .   .    .    .      .        .
   .          .    .    .    .    .   .    .    .      .        .
   .          .    .    .    .    .   .    .    .      .        .
   .          .    .    .    .    .   .    .    .      .        .
   .          .    .    .    .    .   .    .    .      .        .
22 root       15   -10  0    0    0   S   0.0  0.0  0:00.00  kacpid
82 root       5    -10  0    0    0   S   0.0  0.0  0:00.39  kblockd/D
118 root      15   -10  0    0    0   S   0.0  0.0  0:00.09  pdflush
120 root      11   -10  0    0    0   S   0.0  0.0  0:00.00  aio/O
```

Proses ini berjalan pada pukul 06:47:01 menurut komputer yang digunakan menjalankan perintah ini. Komputer telah berjalan selama 23 menit dengan beban CPU pada waktu itu adalah 0.88, 5 menit yang lalu 0.40, dan 15 menit yang lalu adalah 0.41. Dalam komputer ini ada 68 proses (2 running, 66 sleeping, 0 stopped,dan 0 zombie) Sebagian besar proses sedang idle dengan persentase 70.6% sedangkan proses untuk pengguna dan system masing- masing 12.1% dan 4.3%. Dalam proses ini memori yang terpakai adalah 110388 k, untuk buffer 4908k dan yang tak terpakai 7408k.

Dalam proses Linux, duplikasi proses menggunakan fork. Proses ini memiliki perbedaan dengan *thread* yang pada umumnya digunakan. Diantara perbedaan itu adalah:

Contoh 17.8. Tread vs. Fork

Thread	Fork
1. Diperbolehkan adanya pembagian struktur data	Secara keseluruhan tidak
2. Dibuat dengan method clone()	Dibuat dengan method fork ()
3. LigthWeigth proses	Heavyweigth proses

Dalam duplikasi proses, fork lebih cepat dibandingkan thread karena fork hanya clone dirinya sendiri dan memanggil method exce untuk mengganti proses yang identik dengan induk dengan sebuah proses baru.

17.4. Penjadwalan

Penjadwal adalah suatu pekerjaan yang dilakukan untuk mengalokasikan CPU time untuk tasks yang berbeda-beda dalam sistem operasi. Pada umumnya, kita berfikir penjadwalan sebagai menjalankan dan menginterupsi suatu proses, untuk linux ada aspek lain yang penting dalam penjadwalan: seperti menjalankan dengan berbagai kernel tasks. Kernel tasks meliputi task yang diminta oleh proses yang sedang dijalankan dan tasks yand dieksekusi internal menyangkut device driver yang berkepentingan.

Ketika kernel telah mencapai titik penjadwalan ulang, entah karena terjadi interupsi penjadwalan ulang mau pun karena proses kernel yang sedang berjalan telah diblokir untuk menunggu beberapa signal bangun, harus memutuskan proses selanjutnya yang akan dijalankan. Linux telah memiliki dua algoritma penjadwalan proses yang terpisah satu sama lain. Algoritma yang pertama adalah algoritma time-sharing untuk penjadwalan preemptive yang adil diantara sekian banyak proses. Sedangkan algoritma yang kedua didesain untuk tugas real-time dimana proritas mutlak lebih utama daripada keadilan mendapatkan suatu pelayanan.

Bagian dari tiap identitas proses adalah kelas penjadwalan, yang akan menentukan algoritma yang digunakan untuk tiap proses. Kelas penjadwalan yang digunakan oleh Linux, terdapat dalam standar perluasan POSIX untuk sistem komputer waktu nyata.

Untuk proses time-sharing, Linux menggunakan teknik prioritas, sebuah algoritma yang berdasarkan pada kupon. Tiap proses memiliki sejumlah kupon penjadwalan; dimana ketika ada kesempatan untuk menjalankan sebuah tugas, maka proses dengan kupon terbanyaklah yang mendapat giliran. Setiap kali terjadi interupsi waktu, proses yang sedang berjalan akan kehilangan satu kupon; dan ketika kupon yang dimiliki sudah habis maka proses itu akan ditunda dan proses yang lain akan diberikan kesempatan untuk masuk.

Jika proses yang sedang berjalan tidak meiliki kupon sama sekali, linux akan melakukan operasi pemberian kupon, memberikan kupon kepada tiap proses dalam sistem, dengan aturan main:

```
kupon = kupon / 2 + prioritas
```

Algoritma ini cenderung untuk menggabungkan dua faktor yang ada: sejarah proses dan prioritas dari proses itu sendiri. Satu setengah dari kupon yang dimiliki sejak operasi pembagian kupon terakhir akan tetap dijaga setelah algoritma telah dijalankan, menjaga beberapa sejarah sikap proses. Proses yang berjalan sepanjang waktu akan cenderung untuk menghabiskan kupon yang dimilikinya dengan cepat, tapi proses yang lebih banyak menunggu dapat mengakumulasi kuponnya dari. Sistem pembagian kupon ini, akan secara otomatis memberikan proritas yang tinggi ke proses M/K bound atau pun interaktif, dimana respon yang cepat sangat diperlukan.

Kegunaan dari proses pemberian prioritas dalam menghitung kupon baru, membuat prioritas dari suatu proses dapat ditingkatkan. Pekerjaan background batch dapat diberikan prioritas yang rendah; proses tersebut akan secara otomatis menerima kupon yang lebih sedikit dibandingkan dengan pekerjaan yang interaktif, dan juga akan menerima persentase waktu CPU yang lebih sedikit dibandingan dengan tugas yang sama dengan prioritas yang lebih tinggi. Linux menggunakan sistem prioritas ini untuk menerapkan mekanisme standar pembagian prioritas proses yang lebih baik.

Penjadwalan waktu nyata Linux masih tetap lebih sederhana. Linux, menerapkan dua kelas penjadwalan waktu nyata yang dibutuhkan oleh POSIX 1.b: First In First Out dan round-robin. Pada keduanya, tiap proses memiliki prioritas sebagai tambahan kelas penjadwalannya. Dalam penjadwalan time-sharing, bagaimana pun juga proses dengan prioritas yang berbeda dapat bersaing dengan beberapa pelebaran; dalam penjadwalan waktu nyata, si pembuat jadwal selalu menjalankan proses dengan prioritas yang tinggi. Diantara proses dengan prioritas yang sama, maka proses yang sudah menunggu lama, akan dijalankan. Perbedaan satu-satunya antara penjadwalan FIFO dan round-robin adalah proses FIFO akan melanjutkan prosesnya sampai keluar atau pun diblokir, sedangkan proses round-robin akan di-preemptive-kan setelah beberapa saat dan akan dipindahkan ke akhir antrian, jadi proses round-robin dengan prioritas yang sama akan secara otomatis membagi waktu jalan antar mereka sendiri.

Perlu diingat bahwa penjadwalan waktu nyata di Linux memiliki sifat yang lunak. Pembuat jadwal Linux menawarkan jaminan yang tegas mengenai prioritas relatif dari proses waktu nyata, tapi kernel tidak menjamin seberapa cepat penjadwalan proses waktu-nyata akan dijalankan pada saat proses siap dijalankan. Ingat bahwa kode kernel Linux tidak akan pernah bisa dipreemptive oleh kode mode pengguna. Apabila terjadi interupsi yang membangunkan proses waktu nyata, sementara kernel siap untuk mengeksekusi sebuah sistem call sebagai bagian proses lain, proses waktu nyata harus menunggu sampai sistem call yang sedang dijalankan selesai atau diblokir.

17.5. *Symmetric Multiprocessing*

Kernel Linux 2.0 adalah kernel Linux pertama yang stabil untuk mendukung perangkat keras symmetric multiprocessor (SMP). Proses mau pun thread yang berbeda dapat dieksekusi secara paralel dengan processor yang berbeda. Tapi bagaimana pun juga untuk menjaga kelangsungan kebutuhan sinkronisasi yang tidak dapat di-preemptive dari kernel, penerapan SMP ini menerapkan aturan dimana hanya satu processor yang dapat dieksekusi dengan kode mode kernel pada suatu saat. SMP menggunakan kernel spinlock tunggal untuk menjalankan aturan ini. Spinlock ini tidak memunculkan permasalahan untuk pekerjaan yang banyak menghabiskan waktu untuk menunggu proses komputasi, tapi untuk pekerjaan yang melibatkan banyak aktifitas kernel, spinlock dapat menjadi sangat mengkhawatirkan.

Sebuah proyek yang besar dalam pengembangan kernel Linux 2.1 adalah untuk menciptakan penerapan SMP yang lebih masuk akal, dengan membagi kernel spinlock tunggal menjadi banyak kunci yang masing-masing melindungi terhadap masuknya kembali sebagian kecil data struktur kernel. Dengan menggunakan teknik ini, pengembangan kernel yang terbaru mengizinkan banyak processor untuk dieksekusi oleh kode mode kernel secara bersamaan.

17.6. Rangkuman

Kernel Linux mengawasi dan mengatur proses berdasarkan nomor process identification number (PID), sedangkan informasi tentang setiap proses disimpan pada deskriptor proses. Deskriptor proses merupakan struktur data yang memiliki beberapa status, yaitu: TASK_RUNNING,

TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_STOPPED, dan TASK_ZOMBIE.

Linux menyediakan dua system call, yaitu fork dan clone. Fork memiliki fungsi untuk menduplikasi proses di mana proses anak yang dihasilkan bersifat independen. Sedangkan fungsi clone, selain menduplikasi dari proses induk, juga mengizinkan terjadinya proses berbagi ruang alamat antara proses anak dengan proses induk.

Sinkronisasi kernel Linux dilakukan dengan dua solusi, yaitu:
- Dengan membuat normal kernel yang bersifat code nonpreemptible. Akan muncul salah satu dari tiga aksi, yaitu: interupsi, page fault, atau kernel code memanggil penjadwalannya sendiri.
- Dengan meniadakan interupsi pada saat critical section muncul.

Penjadwalan proses Linux dilakukan dengan menggunakan teknik prioritas, sebuah alogritma yang berdasarkan pada kupon, dengan aturan bahwa setiap proses akan memiliki kupon, di mana pada suatu saat kupon akan habis dan mengakibatkan tertundanya proses tersebut.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[WEBDrake96] Donald G Drake. April 1996. *Introduction to Java threads – A quick tutorial on how to implement threads in Java – http://www.javaworld.com/ javaworld/ jw-04-1996/ jw-04-threads.html* . Diakses 29 Mei 2006.

[WEBFasilkom2003] Fakultas Ilmu Komputer Universitas Indonesia . 2003. *Sistem Terdistribusi – http://telaga.cs.ui.ac.id/ WebKuliah/ sisdis2003/* . Diakses 29 Mei 2006.

[WEBHarris2003] Kenneth Harris. 2003. *Cooperation: Interprocess Communication – Concurrent Processing – http://people.cs.uchicago.edu/ ~mark/ 51081/ LabFAQ/ lab5/ IPC.html* . Diakses 2 Juni 2006.

[WEBWalton1996] Sean Walton. 1996. *Linux Threads Frequently Asked Questions (FAQ) – http://linas.org/ linux/ threads-faq.html* . Diakses 29 Mei 2006.

[WEBWiki2006a] From Wikipedia, the free encyclopedia. 2006. *Title – http://en.wikipedia.org/wiki/Zombie_process* . Diakses 2 Juni 2006.

Bagian IV. Proses dan Sinkronisasi

Proses, Penjadualan, dan Sinkronisasi merupakan trio yang saling berhubungan, sehingga seharusnya tidak dipisahkan. Bagian yang lalu telah membahas Proses dan Penjadualannya, sehingga bagian ini akan membahas Proses dan Sinkronisasinya.

Bab 18. Konsep Interaksi

18.1. Pendahuluan

Sebelumnya telah diketahui seluk beluk dari suatu proses mulai dari pengertiannya, cara kerjanya, sampai operasi-operasinya seperti proses pembentukannya dan proses pemberhentiannya setelah selesai melakukan eksekusi. Kali ini kita akan mengulas bagaimana hubungan antar proses dapat berlangsung, misalnya bagaimana beberapa proses dapat saling berkomunikasi dan bekerjasama.

Selain itu pada bab ini kita akan menyingung sedikit mengenai client/server proses. Beberapa topik yang akan dibahas adalah Java *Remote Method Invocation* (RMI) dan *Remote Procedure Call* (RPC). Keduanya menggunakan konsep komunikasi IPC, namun menggunakan sistem terdistribusi yang melibatkan jaringan. Pada bab ini juga akan dibahas mengenai infrastruktur dasar jaringan yaitu *socket*.

18.2. Proses yang Kooperatif

Proses yang bersifat simultan (*concurrent*) yang dijalankan pada sistem operasi dapat dibedakan menjadi proses independen dan proses kooperatif. Suatu proses dikatakan independen apabila proses tersebut tidak dapat terpengaruh atau dipengaruhi oleh proses lain yang sedang dijalankan pada sistem. Berarti, semua proses yang tidak membagi data apa pun (baik sementara/tetap) dengan proses lain adalah independent. Sedangkan proses kooperatif adalah proses yang dapat dipengaruhi atau pun terpengaruhi oleh proses lain yang sedang dijalankan dalam sistem. Dengan kata lain, proses dikatakan kooperatif bila proses dapat membagi datanya dengan proses lain.

Ada empat alasan untuk penyediaan sebuah lingkungan yang memperbolehkan terjadinya proses kooperatif:

1. **Pembagian informasi.** Apabila beberapa pengguna dapat tertarik pada bagian informasi yang sama (sebagai contoh, sebuah berkas bersama), kita harus menyediakan sebuah lingkungan yang mengizinkan akses secara terus menerus ke tipe dari sumber-sumber tersebut.

2. **Kecepatan penghitungan/komputasi.** Jika kita menginginkan sebuah tugas khusus untuk menjalankan lebih cepat, kita harus membagi hal tersebut ke dalam subtask, setiap bagian dari subtask akan dijalankan secara paralel dengan yang lainnya. Peningkatan kecepatan dapat dilakukan hanya jika komputer tersebut memiliki elemen-elemen pemrosesan ganda (seperti CPU atau jalur M/K).

3. **Modularitas.** Kita mungkin ingin untuk membangun sebuah sistem pada sebuah model modular-modular, membagi fungsi sistem menjadi beberapa proses atau *thread*.

4. **Kenyamanan.** Bahkan seorang pengguna mungkin memiliki banyak tugas untuk dikerjakan secara bersamaan pada satu waktu. Sebagai contoh, seorang pengguna dapat mengedit, mencetak, dan mengkompilasi secara paralel.

Kerja sama antar proses membutuhkan suatu mekanisme yang memperbolehkan proses-proses untuk mengkomunikasikan data dengan yang lain dan meng-*synchronize* kerja mereka sehingga tidak ada yang saling menghalangi. Salah satu cara proses dapat saling berkomunikasi adalah *Interprocess Communication* (IPC) yang akan dijelaskan lebih lanjut di bagian berikut.

18.3. Hubungan Antara Proses

Ada dua model yang fundamental dalam hubungan antar proses yaitu:
1. **Shared Memori.** Dalam model ini, proses saling berbagi memori. Untuk menjaga konsistensi data, perlu diatur proses mana yang dapt mengakses memori pada suatu waktu.
2. **Message Passing.** Pada model ini proses berkomunikasi lewat saling mengirimkan pesan.

Pembahasan lebih lanjut ada dibahas pada bagian berikutnya. Contoh di bawah ini dapat

dianalogikan seperti Shared Memori.

Pada program di atas terdapat dua proses, yaitu Produsen dan Konsumen. Keduanya saling berbagi memori yaitu buffer. Sebuah produsen proses membentuk informasi yang dapat digunakan oleh konsumen proses. Sebagai contoh sebuah cetakan program yang membuat banyak karakter yang diterima oleh *driver* pencetak. Untuk memperbolehkan produsen dan konsumen proses agar dapat berjalan secara terus menerus, kita harus menyediakan sebuah item buffer yang dapat diisi dengan proses produsen dan dikosongkan oleh proses konsumen. Proses produsen dapat memproduksi sebuah item ketika konsumen sedang mengkonsumsi item yang lain. Produsen dan konsumen harus dapat selaras. Konsumen harus menunggu hingga sebuah item telah diproduksi.

Contoh 18.1. *Bounded Buffer*

```
import java.util.*;

public class BoundedBuffer {
    public BoundedBuffer() {
        // buffer diinisialisasikan kosong
        count   = 0;
        in      = 0;
        out     = 0;

        buffer = new Object[BUFFER_SIZE];
     }

     // produsen memanggil method ini
     public void enter( Object item ) {
        while ( count == BUFFER_SIZE )
            ; // do nothing
        // menambahkan suatu item ke dalam buffer
        ++count;
        buffer[in]  = item;
        in  = ( in + 1 ) % BUFFER_SIZE;
        if ( count == BUFFER_SIZE )
            System.out.println( "Producer Entered " +
                item + " Buffer FULL" );
        else
            System.out.println( "Producer Entered " +
                item + " Buffer Size = " + count );
    }

    // consumer memanggil method ini
    public Object remove() {
        Object item ;
        while ( count == 0 )
            ; // do nothing
        // menyingkirkan suatu item dari buffer
        --count;
        item  = buffer[out];
        out   = ( out + 1 ) % BUFFER_SIZE;
        if ( count == 0 )
            System.out.println( "Consumer consumed " +
              item + " Buffer EMPTY" );
        else
            System.out.println( "Consumer consumed " +
              item + " Buffer Size = " +count );
        return item;
    }
    public static final int NAP_TIME = 5;
    private static final int BUFFER_SIZE = 5;
    private volatile int count;
    private int in;     // arahkan ke posisi kosong selanjutnya
    private int out;    // arahkan ke posisi penuh  selanjutnya
    private Object[] buffer;
}
```

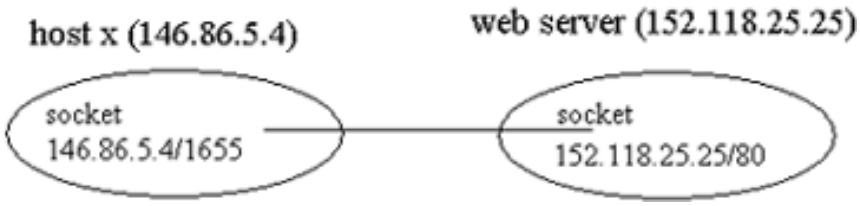
18.4. Komunikasi Proses Dalam Sistem

Cara lain yang bila dilakukan oleh sistem operasi untuk memungkinkan komunikasi antar proses yaitu dengan menyediakan alat-alat proses kooperatif untuk berkomunikasi lewat *Inter-Process Communication* (IPC). IPC menyediakan sebuah mekanisme untuk mengizinkan proses-proses untuk berkomunikasi dan menyelaraskan aksi-aksi mereka tanpa berbagi ruang alamat yang sama. IPC khusus digunakan dalam sebuah lingkungan yang terdistribusi dimana proses komunikasi tersebut mungkin saja tetap ada dalam komputer-komputer yang berbeda yang tersambung dalam sebuah jaringan. IPC adalah penyedia layanan terbaik dengan menggunakan sebuah sistem penyampaian pesan, dan sistem-sistem pesan dapat diberikan dalam banyak cara.

Fungsi dari sebuah sistem pesan adalah untuk memperbolehkan komunikasi satu dengan yang lain tanpa perlu menggunakan pembagian data. Sebuah fasilitas IPC menyediakan paling sedikit dua operasi yaitu "kirim" dan "terima". Pesan dikirim dengan sebuah proses yang dapat dilakukan pada ukuran pasti atau variabel. Jika hanya pesan dengan ukuran pasti dapat dikirimkan, level sistem implementasi adalah sistem yang sederhana. Pesan berukuran variabel menyediakan sistem implementasi level yang lebih kompleks.

Jika dua buah proses ingin berkomunikasi, misalnya proses P dan proses Q, mereka harus mengirim pesan atau menerima pesan dari satu ke yang lainnya. Jalur ini dapat diimplementasikan dengan banyak cara, namun kita hanya akan memfokuskan pada implementasi logiknya saja, bukan implementasi fisik (seperti *shared memory*, *hardware bus*, atau jaringan). Berikut ini ada beberapa metode untuk mengimplementasikan sebuah jaringan dan operasi pengiriman/penerimaan secara logika:
- Komunikasi langsung atau tidak langsung.
- Komunikasi secara simetris/asimetris.
- Buffer otomatis atau eksplisit.
- Pengiriman berdasarkan salinan atau referensi.
- Pesan berukuran pasti dan variabel.

Gambar 18.1. Client Server



18.5. Komunikasi Langsung

Proses-proses yang ingin dikomunikasikan harus memiliki sebuah cara untuk memilih satu dengan yang lain. Mereka dapat menggunakan komunikasi langsung/tidak langsung.

Setiap proses yang ingin berkomunikasi harus memiliki nama yang bersifat eksplisit baik penerimaan atau pengirim dari komunikasi tersebut. Dalam konteks ini, pengiriman dan penerimaan pesan secara primitive dapat dijabarkan sebagai:
- Send (P, message) - mengirim sebuah pesan ke proses P.
- Receive (Q, message) - menerima sebuah pesan dari proses Q.

Sifat-sifat yang dimiliki oleh jaringan komunikasi pada pembataasan ini ialah:
- Sebuah jaringan harus didirikan antara kedua proses yang ingin berkomunikasi. Selain itu, dibutuhkan pengenal-pengenal khusus yang dapat digunakan sebagai pengidentifikasi dalam proses komunikasi tersebut.

- Sebuah jaringan terdiri atas tepat dua proses.
- Setiap pesan yang dikirim atau diterima oleh proses harus melalui tepat sebuah jaringan.

Pembahasan ini memperlihatkan sebuah cara simetris dalam pemberian alamat. Oleh karena itu, baik keduanya yaitu pengirim dan penerima proses harus memberi nama bagi yang lain untuk berkomunikasi, hanya pengirim yang memberikan nama bagi penerima sedangkan penerima tidak menyediakan nama bagi pengirim. Dalam konteks ini, pengirim dan penerima secara sederhana dapat dijabarkan sebagai:
- Send (P, message) - mengirim sebuah pesan kepada proses P.
- Receive (ID, message) - menerima sebuah pesan dari semua proses. Variabel ID diatur sebagai nama dari proses dengan komunikasi.

Kerugian dari sistem komunikasi tidak langsung adalah keterbatasan modularitas, disebabkan oleh hanya ada paling banyak dua proses yang berkomunikasi dalam satu jaringan. Selain itu, perubahan pengenal-pengenal dari suatu proses yang digunakan dalam komunikasi dapat mengakibatkan jaringan yang telah terbentuk harus di-set ulang, karena seluruh referensi ke pengenal yang lama harus di-set ulang agar komunikasi dapat berjalan dengan baik.

18.6. Komunikasi Tidak Langsung

Dengan komunikasi tidak langsung, pesan akan dikirimkan pada dan diterima dari/melalui *mailbox* (Kotak Surat) atau terminal-terminal, sebuah *mailbox* dapat dilihat secara abstrak sebagai sebuah obyek didalam setiap pesan yang dapat ditempatkan dari proses dan dari setiap pesan yang bias dipindahkan. Setiap kotak surat memiliki sebuah identifikasi (identitas) yang unik, sebuah proses dapat berkomunikasi dengan beberapa proses lain melalui sebuah nomor dari *mailbox* yang berbeda. Dua proses dapat saling berkomunikasi apabila kedua proses tersebut sharing *mailbox*. Pengirim dan penerima dapat dijabarkan sebagai:
- Send (A, *message*) - mengirim pesan ke *mailbox* A.
- Receive (A, *message*) - menerima pesan dari *mailbox* A.

Dalam masalah ini, link komunikasi mempunyai sifat sebagai berikut:
- Sebuah link dibangun diantara sepasang proses dimana kedua proses tersebut membagi *mailbox*.
- Sebuah link mungkin dapat berasosiasi dengan lebih dari dua proses.
- Diantara setiap pasang proses komunikasi, mungkin terdapat link yang berbeda-beda, dimana setiap link berhubungan pada satu *mailbox*.

Misalkan terdapat proses P1, P2 dan P3 yang semuanya *share mailbox*. Proses P1 mengirim pesan ke A, ketika P2 dan P3 masing-masing mengeksekusi sebuah kiriman dari A. Proses mana yang akan menerima pesan yang dikirim P1? Jawabannya tergantung dari jalur yang kita pilih:
- Mengizinkan sebuah link berasosiasi dengan paling banyak dua proses.
- Mengizinkan paling banyak satu proses pada suatu waktu untuk mengeksekusi hasil kiriman (*receive operation*).
- Mengizinkan sistem untuk memilih secara mutlak proses mana yang akan menerima pesan (apakah itu P2 atau P3 tetapi tidak keduanya, tidak akan menerima pesan). Sistem mungkin mengidentifikasi penerima kepada pengirim.

Mailbox mungkin dapat dimiliki oleh sebuah proses atau sistem operasi. Jika *mailbox* dimiliki oleh proses, maka kita mendefinisikan antara pemilik (yang hanya dapat menerima pesan melalui *mailbox*) dan pengguna dari *mailbox* (yang hanya dapat mengirim pesan ke *mailbox*). Selama setiap *mailbox* mempunyai kepemilikan yang unik, maka tidak akan ada kebingungan tentang siapa yang harus menerima pesan dari *mailbox*. Ketika proses yang memiliki *mailbox* tersebut diterminasi, *mailbox* akan hilang. Semua proses yang mengirim pesan ke *mailbox* ini diberi pesan bahwa *mailbox* tersebut tidak lagi ada.

Dengan kata lain, mempunyai *mailbox* sendiri yang independent, dan tidak melibatkan proses yang lain. Maka sistem operasi harus memiliki mekanisme yang mengizinkan proses untuk melakukan hal-hal di bawah ini:
- Membuat *mailbox* baru.
- Mengirim dan menerima pesan melalui *mailbox*.
- Menghapus *mailbox*.

Proses yang pertama kali membuat sebuah *mailbox* secara default akan memiliki *mailbox* tersebut. Untuk pertama kali, hanya proses yang membuat *mailbox* yang dapat berkomunikasi melalui

mailbox tersebut. Meskipun demikian, sebuah *mailbox* dapat saja di-assign ke proses lain, sehingga beberapa proses dapat berkomunikasi melalui *mailbox* tersebut.

Proses komunikasi proses dengan menggunakan mailbox dapat dianalogikan sebagai kegiatan komunikasi dengan 'papan pesan'. beberapa proses dapat memiliki sebuah mailbox bersama-sama, di mana mailbox ini berfungsi sebagai 'tempat penitipan pesan' bagi proses-proses yang berkomunikasi. Dalam komunikasi jenis ini, proses-proses hanya dapat berkomunikasi dengan proses lain yang berbagi mailbox yang sama dengan proses tersebut.

18.7. Sinkronisasi

Komunikasi antara proses membutuhkan *place by calls* untuk mengirim dan menerima data *primitive*. Terdapat design yang berbeda-beda dalam implementasi setiap primitive. Pengiriman pesan mungkin dapat diblok (*blocking*) atau tidak dapat dibloking (*nonblocking*) – juga dikenal dengan nama sinkron atau asinkron.

- Pengiriman yang diblok: Proses pengiriman di blok sampai pesan diterima oleh proses penerima (*receiving process*) atau oleh *mailbox*.
- Pengiriman yang tidak diblok: Proses pengiriman pesan dan mengkalkulasi operasi.
- Penerimaan yang diblok: Penerima memblok sampai pesan tersedia.
- Penerimaan yang tidak diblok: Penerima mengembalikan pesan valid atau null.

18.8. Buffering

Apa pun jenis komunikasinya, langsung atau tidak langsung, penukaran pesan oleh proses memerlukan antrian sementara. Pada dasarnya, terdapat tiga cara untuk mengimplementasikan antrian tersebut:

- **Kapasitas Nol.** Antrian mempunyai panjang maksimum 0, sehingga tidak ada penungguan pesan (*message waiting*). Dalam kasus ini, pengirim harus memblok sampai penerima menerima pesan.
- **Kapasitas Terbatas.** Antrian mempunyai panjang yang telah ditentukan, paling banyak n pesan dapat dimasukkan. Jika antrian tidak penuh ketika pesan dikirimkan, pesan yang baru akan menimpa, dan pengirim pengirim dapat melanjutkan eksekusi tanpa menunggu. Link mempunyai kapasitas terbatas. Jika link penuh, pengirim harus memblok sampai terdapat ruang pada antrian.
- **Kapasitas Tak Terbatas.** Antrian mempunyai panjang yang tak terhingga, sehingga semua pesan dapat menunggu disini. Pengirim tidak akan pernah di blok.

18.9. *Mailbox*

Contoh 18.2. *Mailbox*

```
import java.util.*;
public class MessageQueue {
    private Vector q;
    public MessageQueue() { q = new Vector(); }
    // Mengimplementasikan pengiriman nonblocking
    public void send( Object item ) {
        q.addElement( item );
    }
    // Mengimplementasikan penerimaan nonblocking
    public Object receive() {
        Object item;
        if ( q.size() == 0 ) { return null; }
        else {
            item = q.firstElement();
            q.removeElementAt(0);
            return item;
        }
    }
}
```

Program contoh ini ditunjukan bahwa mailbox sering kali diimplementasikan dengan menggunakan antrian (queue). Mengirim dan menerima pesan dalam mailbox merupakan operasi yang fleksibel. Bila ketika mengirim queue dalam keadaan kosong, pesan dapat langsung dicopy. Sedangkan bila queue dalam penuh, pengirim mempunyai 4 pilihan dalam menanggapi hal itu, yaitu:

i. Menunggu sampai batas waktu yang tidak dapat ditentukan sampai terdapat ruang kosong pada mailbox.
ii. Menunggu paling banyak n milidetik.
iii. Tidak menunggu, tetapi kembali (return) secepatnya.
iv. Satu pesan dapat diberikan kepada sistem operasi untuk disimpan, walaupun mailbox yang dituju penuh. Ketika pesan dapat disimpan pada mailbox, pesan akan dikembalikan kepada pengirim (sender). Hanya satu pesan kepada mailbox yang penuh yang dapat diundur (pending) pada suatu waktu untuk diberikan kepada thread pengirim.

18.10. *Socket Client/Server System*

Dengan makin berkembangnya teknologi jaringan komputer, sekarang ini ada kecenderungan sebuah sistem yang menggunakan jaringan untuk saling berhubungan. Bagian berikut ini akan kita bahas beberapa metoda komunikasi antar proses yang melibatkan jaringan komputer.

Socket adalah sebuah *endpoint* untuk komunikasi didalam jaringan. Sepasang proses atau *thread* berkomunikasi dengan membangun sepasang socket, yang masing-masing proses memilikinya. Socket dibuat dengan menyambungkan dua buah alamat IP melalui port tertentu. Secara umum socket digunakan dalam *client/server system*, dimana sebuah server akan menunggu client pada port tertentu. Begitu ada client yang mengkontak server maka server akan menyetujui komunikasi dengan client melalui socket yang dibangun.

Sebagai contoh sebuah program web browser pada host x (IP 146.86.5.4) ingin berkomunikasi dengan web server (IP 152.118.25.15) yang sedang menunggu pada port 80. Host x akan menunjuk sebuah port. Dalam hal ini port yang digunakan ialah port 1655. Sehingga terjadi sebuah hubungan dengan sepasang socket (146.86.5.4:1655) dengan (152.118.25.15:80).

18.11. *Server* dan *Thread*

Pada umumnya sebuah server melayani client secara konkuren, oleh sebab itu dibutuhkan thread yang masing-masing thread melayani clientnya masing-masing. Jadi server akan membentuk thread baru begitu ada koneksi dari client yang diterima (accept). Server menggunakan thread apabila client melakukan koneksi, sehingga server memiliki tingkat reabilitas yang tinggi. Pada sistem yang memiliki banyak pemakai sekaligus thread mutlak dibutuhkan, karena setiap pemakai sistem pasti menginginkan respon yang baik dari server.

Salah satu contoh pengunaan server thread ialah pada aplikasi database. Server Database seringkali harus menerima permintaan dari ratusan pengguna. Maka agar client tidak terlalu lama menunggu permintaan nya dijawab oleh aplikasi server, masing-masing client mendapat sebuah thread yang akan melayani permintaan yang diminta client.

Java Socket

Java menyediakan dua buah tipe socket yang berbeda dan sebuah socket spesial. Semua soket ini tersedia dalam paket jaringan, yang merupakan paket standar java. Berikut ini soket yang disediakan oleh java:

- Connection-Oriented (TCP) socket, yang diimplementasikan pada kelas *java.net.Socket*
- Connectionless Socket (UDP), yang diimplentasikan oleh kelas *java.net.DatagramSocket*
- Dan yang terakhir adalah *java.net.MulticastSocket*, yang merupakan perluasan (extended) dari Socket UDP. Tipe socket ini memiliki kemampuan untuk mengirim pesan kebanyak client sekaligus (Multicast), sehingga baik digunakan pada sistem yang memiliki jenis layanan yang sama.

Potongan kode berikut ini memperlihatkan teknik yang digunakan oleh java untuk membuka socket (pada kasus ini server socket). Selanjutnya server dapat berkomunikasi dengan clientnya menggunakan InputStream untuk menerima pesan dan OutputStream untuk mengirim pesan.

Contoh 18.3. *WebServer*

```
import java.net.*;
import java.io.*;

public class Server1
{
  public static void main(String []args)
  {
      try{
        //membuka socket
        ServerSocket server= new ServerSocket( 12345, 100 );
        Socket connection= server.accept();
        //membuka stream
        ObjectOutputStream output = new ObjectOutputStream(
             connection.getOutputStream());
        output.flush();
        ObjectInputStream input = new ObjectInputStream(
             connection.getInputStream() );

        //membaca pesan
        try{
          String message = ( String ) input.readObject();
          System.out.println(message);
        }
        catch(ClassNotFoundException e)
        {
          System.out.println("kelas tidak ditemukan");
        }
        //mengirim pesan
        output.writeObject( "OS" );
      }
      catch(IOException e)
      {
        System.out.println("terjadi IOEXception");
      }
  }
}
```

Contoh 18.4. ***Client***

```
import java.net.*;
import java.io.*;

public class Client1
{
  public static void main(String []args)
  {
      try{
        //membuka socket
        Socket client= client = new Socket(
             InetAddress.getByName( "localhost" ), 12345 );

        //membuka stream
        ObjectOutputStream output = new ObjectOutputStream(
             client.getOutputStream());
        output.flush();
        ObjectInputStream input = new ObjectInputStream(
             client.getInputStream() );

        //mengirim pesan
        output.writeObject( "pesan" );
        //membaca pesan
        try{
          String message = ( String ) input.readObject();
          System.out.println(message);
        }
        catch(ClassNotFoundException e)
        {
          System.out.println("kelas tidak ditemukan");
        }

      }
      catch(IOException e)
      {
        System.out.println("terjadi IOEXception");
      }

  }
}
```

Remote Procedure Call

Remote Procedure Call (RPC) adalah sebuah metoda yang memungkinkan kita untuk mengakses sebuah prosedur yang berada di komputer lain. Untuk dapat melakukan ini sebuah komputer (server) harus menyediakan layanan remote prosedur. Pendekatan yang dilakuan adalah, sebuah server membuka socket, menunggu client yang meminta proseduryang disediakan oleh server. Bila client tidak tahu harus menghubungi port yang mana, client bisa merequest kepada sebuah matchmaker pada sebuah RPC port yang tetap. Matchmaker akan memberikan port apa yang digunakan oleh procedure yang diminta client.

RPC masih menggunakan cara primitive dalam pemrograman, yaitu menggunakan paradigma procedural programming. Hal itu membuat kita sulit ketika menyediakan banyak remote procedure.

RPC menggunakan soket untuk berkomunikasi dengan proses lainnya. Pada sistem seperti SUN, RPC secara default sudah terinstall kedalam sistemnya, biasanya RPC ini digunakan untuk administrasi sistem. Sehingga seorang administrator jaringan dapat mengakses sistemnya dan mengelola sistemnya dari mana saja, selama sistemnya terhubung ke jaringan.

Masalah dapat timbul jika antara *client* dan *server RPC* mempunyai perbedaan dalam representasi data. Ada beberapa komputer yang menggunakan sistem *big-endian* (*most significant byte* diletakan pada *high memori address*) dan lainnya menggunakan *little Endian*. Solusi untuk permasalahan ini ialah dengan mengunakan XDR (*external data representation*). *Client* sebelum mengirim data nya harus mengubah format data kedalam XDR. Server yang menerimanya mengubah XDR menjadi format data yang dimilikinya.

Pembuatan Obyek *Remote*

Pendekatan kedua yang akan kita bahas adalah Remote Method Invocation (RMI), sebuah teknik paradigma pemrograman berorientasi obyek (OOP). RMI merupakan RPC versi java. Dengan RMI memungkinkan kita untuk mengirim obyek sebagai parameter dari remote method. Dengan dibolehkannya program java memanggil method pada remote obyek, RMI membuat pengguna dapat mengembangkan aplikasi java yang terdistribusi pada jaringan

Untuk membuat remote method dapat diakses RMI mengimplementasikan remote object menggukan stub dan skleton. Stub bertindak sebagai proxy disisi client, yaitu yang menghubungkan client dengan skleton yang berada disisi server. Stub yang ada disisi client bertanggung-jawab untuk membungkus nama method yang akan diakses, dan parameternya, hal ini biasa dikenal dengan marshalling. Stub mengirim paket yang sudah dibungkus ini ke server dan akan di buka (unmarshalling) oleh skleton. Skleton akan menerima hasil keluaran yang telah diproses oleh method yang dituju, lalu akan kembali dibungkus (marshal) dan dikirim kembali ke client yang akan diterima oleh stub dan kembali dibuka paketnya (unmarshall).

Untuk membuat remote obyek kita harus mendefinisikan semua method yang akan kita sediakan pada jaringan, setelah itu dapat digunakan RMI kompilator untuk membuat stub dan skleton. Setelah itu kita harus mem-binding remote obyek yang kita sediakan kedalam sebuah RMI registry. Setelah itu client dapat mengakses semua remotemethod yang telah kita sediakan menggunkan stub yang telah dikompilasi menggunakan RMI kompilator terebut.

Akses ke Obyek *Remote*

Sekali obyek didaftarkan ke server, client dapat mengakses remote object dengan menjalankan Naming.lookup() method. RMI menyediakan URL untuk pengaksesan ke remote obyek yaitu rmi://host/obyek, dimana host adalah nama server tempat kita mendaftarkan remote obyek dan obyek adalah parameter yang kita gunakan ketika kita memanggil method Naming.rebind(). Client juga harus menginstall RMISecurityManager untuk memastikan keamanan client ketika membuka soket ke jaringan.

Java memiliki sistem security yang baik sehingga pengguna dapat lebih nyaman dalam melakukan komunikasi pada jaringan. Selain itu java sudah mendukung pemorograman berorientasi object, sehingga pengembangan software berskala besar sangat dimungkinkan dilakukan oleh java. RMI sendiri merupakan sistem terdistribusi yang dirancang oleh SUN pada platfrom yang spesifik yaitu Java, pabila anda tertarik untuk mengembangkan sistem terdistribusi yang lebih portable dapat digunakan CORBA sebagai solusi alternatifnya.

18.12. Rangkuman

Proses-proses pada sistem dapat dieksekusi secara berkelanjutan. Ada beberapa alasan bahwa proses-proses tersebut dieksekusi secara berkelanjutan: pembagian informasi, penambahan kecepatan komputasi, modularitas, dan kenyamanan atau kemudahan.

Proses komunikasi antar proses dapat terjadi dalam dua cara: langsung atau tidak langsung. pada komunikasi langsung, hanya ada dua proses yang dapat berkomunikasi. sedangkan pada komunikasi tidak langsung, dapat lebih dari dua proses yang saling berkomunikasi.Sinkronisasi dan buffering merupakan jembatan dari komunikasi proses, yang menentukan apakah suatu proses harus ter-blok atau tidak ketika berkomunikasi

Beberapa contoh komunikasi antarproses yang sering terjadi adalah dalam Socket Client-Server System. Dalam kegiatan ini dilakukan komunikasi antarproses dalam sebuah jaringan. Contoh lain

adalah Remote Procedure Call (RPC) dan Remote Method Invocation (RMI), yang dilakukan oleh Java.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

Bab 19. Sinkronisasi

19.1. Pendahuluan

Apakah sinkronisasi itu sebenarnya? Dan mengapa kita memerlukan sinkronisasi tersebut? Marilah kita pelajari lebih lanjut mengenai sinkronisasi. Seperti yang telah kita ketahui bahwa proses dapat bekerja sendiri (*independent process*) dan juga dapat bekerja bersama proses-proses yang lain (*cooperating process*). Pada umumnya ketika proses saling bekerjasama (*cooperating process*) maka proses-proses tersebut akan saling berbagi data. Pada saat proses-proses berbagi data, ada kemungkinan bahwa data yang dibagi secara bersama itu akan menjadi tidak konsisten dikarenakan adanya kemungkinan proses-proses tersebut melakukan akses secara bersamaan yang menyebabkan data tersebut berubah, hal ini dikenal dengan istilah *Race Condition*.

19.2. *Race Condition*

Dalam sebuah sistem yang terdiri dari beberapa cooperating sequential process yang berjalan secara asynchronous dan berbagi data, dapat terjadi seperti program berikut ini. Untuk lebih jelasnya marilah kita lihat contoh program java berikut yang memperlihatkan timbulnya *Race Condition*.

Contoh 19.1. Produser/Konsumer

```
01. int counter = 0;
02.
03. //Proses yang dilakukan oleh produsen
04. item nextProduced;
05.
06. while (1)
07. {
08.    while (counter == BUFFER_SIZE) { ... do nothing ... }
09.
10.    buffer[in] = nextProduced;
11.    in = (in + 1) % BUFFER_SIZE;
12.    counter++;
13. }
14.
15. //Proses yang dilakukan oleh konsumen
16. item nextConsumed;
17.
18. while (1)
19. {
20.     while (counter == 0) { ... do nothing ... }
21.     nextConsumed = buffer[out] ;
22.     out = (out + 1) % BUFFER_SIZE;
23.     counter--;
24. }
```

Pada program produser/konsumer tersebut dapat kita lihat pada baris 12 dan baris 23 terdapat perintah `counter++` dan `counter--` yang dapat diimplementasikan dengan bahasa mesin sebagai berikut:

Contoh 19.2. *Counter* (1)

```
01.  //counter++(nilai counter bertambah 1 setiap dieksekusi)
02.  register1 = counter
03.  register1 = register1 + 1
04.  counter = register1
05.  //counter--(nilai counter berkurang 1 setiap dieksekusi)
06.  register2 = counter
07.  register2 = register2 - 1
08.  counter = register2
```

Dapat dilihat jika perintah dari `counter++` dan `counter--` dieksekusi secara bersama maka akan sulit untuk mengetahui nilai dari *counter* sebenarnya sehingga nilai dari *counter* itu akan menjadi tidak konsisten. Marilah kita lihat contoh berikut ini:

Contoh 19.3. *Counter* (2)

```
01. //misalkan nilai awal counter adalah 2
02. produsen: register1 = counter (register1 = 2)
03. produsen: register1 = register1 + 1 (register1 = 3)
04. konsumen: register2 = counter (register2 = 2)
05. konsumen: register2 = register2 - 1 (register2 = 1)
06. konsumen: counter = register2 (counter = 1)
07. produsen: counter = register1 (counter = 3)
```

Pada contoh tersebut dapat kita lihat bahwa *counter* memiliki dua buah nilai yaitu bernilai tiga (pada saat `counter++` dieksekusi) dan bernilai satu (pada saat `counter--` dieksekusi). Hal ini menyebabkan nilai dari *counter* tersebut menjadi tidak konsisten. Perhatikan bahwa nilai dari *counter* akan bergantung dari perintah terakhir yang dieksekusi. Oleh karena itu maka kita membutuhkan sinkronisasi yang merupakan suatu upaya yang dilakukan agar proses-proses yang saling bekerja bersama-sama dieksekusi secara beraturan demi mencegah timbulnya suatu keadaan yang disebut dengan *Race Condition*.

19.3. Problem *Critical Section*

Pada sub pokok bahasan sebelumnya, kita telah mengenal *race condition* sebagai masalah yang dapat terjadi pada beberapa proses yang memanipulasi suatu data secara konkruen, sehingga data tersebut tidak sinkron lagi. Nilai akhirnya akan tergantung pada proses mana yang terakhir dieksekusi. Kini, kita akan membahas masalah Critical Section. Apa itu Critical Section? Untuk memahaminya, sebelumnya marilah kita meninjau masalah Race Condition lebih dalam dari sisi programming.

Dalam kenyataannya, suatu proses akan lebih sering melakukan perhitungan internal dan hal-hal teknis lainnya tanpa ada bahaya Race Condition di sebagian besar waktu. Akan tetapi, beberapa proses memiliki suatu segmen kode di mana jika segmen tersebut dieksekusi maka proses-proses tersebut dapat saling mengubah variabel, meng-update suatu tabel, menulis ke dalam file, dan lain sebagainya. Hal-hal seperti yang sudah disebutkan inilah yang dapat membawa proses ke dalam bahaya Race Condition, dan segmen kode yang dapat menyebabkan hal ini disebut sebagai Critical Section.

Maka bagaimana cara menghindari *race condition* ini serta situasi-situasi lain yang melibatkan memori bersama, berkas bersama atau sumber daya yang digunakan bersama-sama? Kuncinya adalah menemukan jalan untuk mencegah lebih dari satu proses melakukan proses tulis atau baca kepada data atau berkas pada saat yang bersamaan. Dengan kata lain, kita membutuhkan *Mutual Exclusion*. *Mutual Exclusion* adalah suatu cara yang menjamin jika ada sebuah proses yang menggunakan variabel atau berkas yang sama (digunakan juga oleh proses lain), maka proses lain akan dikeluarkan dari pekerjaan yang sama.

Sekarang kita akan membahas masalah *race condition* ini dari sisi teknis programming. Biasanya sebuah proses akan sibuk melakukan perhitungan internal dan hal-hal lainnya tanpa ada bahaya yang menuju ke *race condition* pada sebagian besar waktu. Akan tetapi, beberapa proses memiliki suatu segmen kode dimana jika segmen itu dieksekusi, maka proses-proses itu dapat saling mengubah variabel, mengupdate suatu tabel, menulis ke suatu file, dan lain sebagainya, dan hal ini dapat membawa proses tersebut ke dalam bahaya *race condition*. Segmen kode yang seperti inilah yang disebut *Critical Section*.

19.4. Persyaratan

Solusi untuk memecahkan masalah *critical section* adalah dengan merancang sebuah protokol di mana proses-proses dapat menggunakannya secara bersama-sama. Setiap proses harus 'meminta izin' untuk memasuki *critical section*-nya. Bagian dari kode yang mengimplementasikan izin ini disebut *entry section*. Akhir dari *critical section* itu disebut *exit section*. Bagian kode selanjutnya disebut *remainder section*.

Struktur umum dari proses Pi yang memiliki segmen *critical section* adalah:

Contoh 19.4. *Critical Section (1)*

```
do {
   entry section
     <emphasis role="strong">critical section</emphasis>
   exit section
   remainder section
} while (1);
```

Dari kode di atas, dapat kita lihat bahwa untuk bisa memasuki *critical section* sebuah proses harus melalui *entry section*.

Solusi dari masalah *critical section* harus memenuhi tiga syarat berikut [Silbeschatz 2004]:

1. ***Mutual Exclusion*.** Jika suatu proses sedang menjalankan *critical section*-nya, maka proses-proses lain tidak dapat menjalankan *critical section mereka*. Dengan kata lain, tidak ada dua proses yang berada di *critical section* pada saat yang bersamaan.
2. **Terjadi kemajuan (*progress*).** Jika tidak ada proses yang sedang menjalankan *critical section*-nya dan ada proses-proses lain yang ingin masuk ke *critical section*, maka hanya proses-proses yang yang sedang berada dalam *entry section* saja yang dapat berkompetisi untuk mengerjakan critical section.
3. **Ada batas waktu tunggu (*bounded waiting*).** Jika seandainya ada proses yang sedang menjalankan *critical section*, maka proses lain memiliki waktu tunggu yang ada batasnya untuk menjalankan *critical section*-nya, sehingga dapat dipastikan bahwa proses tersebut dapat mengakses *critical section*-nya (tidak mengalami *starvation*: proses seolah-olah berhenti, menunggu request akses ke *critical section* diperbolehkan).

19.5. Rangkuman

Suatu proses yang bekerja bersama-sama dan saling berbagi data dapat mengakibatkan race

condition atau pengaksesan data secara bersama-sama. Critical section adalah suatu segmen kode dari proses-proses itu yang yang memungkinkan terjadinya critical section. Untuk mengatasi masalah critical section ini, suatu data yang sedang diproses tidak boleh diganggu proses lain. Solusi selengkapnya akan dibahas di bab selanjutnya.

Rujukan

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

Bab 20. Masalah *Critical Section*

20.1. Pendahuluan

Pemecahan masalah *critical section* digunakan untuk mengatasi Race Condition pada segmen code *critical section*. Race Condition adalah peristiwa ketidakvalidan data karena proses-proses menggunakan data secara bersamaan. Data yang didapat akhirnya adalah data yang terakhir kali dieksekusi. Untuk itu perlu dicegah terjadinya hal tersebut.

1. **Mutual Exclusion.** Pengertian Mutual Exclusion adalah ketika suatu proses (P0) sedang menggunakan *critical section*, maka tidak boleh ada proses lain (P1) yang menggunakan *critical section* di saat bersamaan.
2. **Progress.** Artinya ketika tidak ada proses yang menggunakan *critical section* dan ada proses-proses yang ingin menggunakan *critical section* tersebut, maka harus ada proses yang menggunakan *critical section* tersebut.
3. **Bounded Waiting.** Maksud dari Bounded Waiting adalah setiap proses yang menunggu menggunakan *critical section*, maka proses-proses yang menunggu tersebut dijamin suatu saat akan menggunakan *critical section*. Dijamin tidak ada thread yang mengalami starvation.

Berikut akan di perlihatkan algoritma-algoritma untuk pemecahan masalah *critical section*. Setiap algoritma menggunakan dua proses dan dianggap berjalan secara concurrent/bersamaan.

Ada dua jenis solusi masalah *critical section*, yaitu:

1. **Solusi Perangkat Lunak.** Dengan menggunakan algoritma-alogoritma yang nilai kebenarannya tidak tergantung pada asumsi-asumsi lain, selain bahwa setiap proses berjalan pada kecepatan yang bukan nol.
2. **Solusi Perangkat Keras.** Tergantung pada beberapa instruksi mesin tertentu, misalnya dengan me-non-aktifkan interupsi atau dengan mengunci suatu variabel tertentu (Lihat: Bab 25, *Bounded Buffer*).

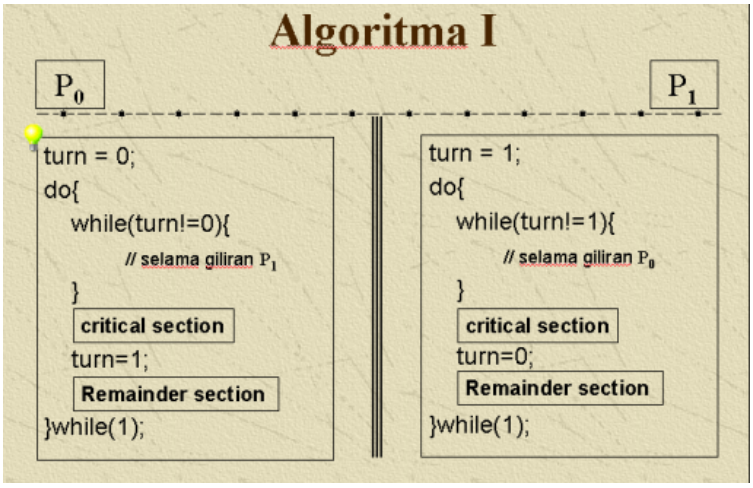
20.2. Algoritma I

Ide Algoritma I ialah memberikan giliran kepada setiap proses untuk memasuki *critical section*-nya. Asumsi yang digunakan disini setiap proses secara bergantian memasuki *critical section*-nya.

Sebenarnya kedua proses berjalan secara bersamaan, dan kita tidak dapat memprediksi proses manakah yang akan berjalan duluan. Misalkan, yang pertama kali dieksekusi adalah baris pertama pada proses P0, maka turn akan diset menjadi 0, artinya sekarang giliran P0 untuk menggunakan *critical section*. Lalu, seandainya proses P1 yang dieksekusi, maka ia akan merubah turn menjadi 1, artinya sekarang giliran P1 menggunakan *critical section*. Lalu P0 akan menunggu karena turn sudah berubah menjadi 1. Lalu P1 akan memasuki *critical section* karena turn=1. Setelah P1 selesai, maka ia akan merubah turn menjadi 0, artinya P1 memberikan giliran P0 untuk menggunakan *critical section*. Selanjutnya P1 akan menunggu di while turn!=1 jika ingin menggunakan *critical section* lagi. Lalu P0 keluar dari loop, karena turn sudah menjadi 0, saatnya giliran P0 memasuki *critical section*. Setelah P0 selesai, P0 akan merubah turn=1, artinya P0 memberikan giliran P1 untuk menggunakan *critical section*. Kejadian ini terus terjadi berulang-ulang.

Permasalahannya adalah, jika suatu proses, misalkan P0 mendapat giliran, artinya turn=0, setelah selesai menggunakan *critical section*, P0 akan merubah turn menjadi 1, nah seandainya P1 tidak membutuhkan *critical section*, turn akan tetap menjadi 1. Lalu pada saat P0 membutuhkan akses ke *critical section* lagi, ia harus menuggu P1 menggunakan *critical section* dan merubah turn menjadi 0. Akibatnya *critical section* dalam keadaan tidak dipakai oleh proses manapun, sedangkan ada proses (P0) yang membutuhkannya. Sampai kapanpun P1 tidak dapat mengakses *critical section* hingga P0 menggunakan *critical section*, lalu merubah turn menjadi 0. Kondisi kekosongan dari *critical section* ini tidak memenuhi syarat solusi *critical section* yang kedua yaitu progress. Analogi Algoritma I seperti kamar mandi dua pintu, ada dua orang yang bergiliran memakai kamar mandi yaitu A dan B. Kamar mandi adalah *critical section*. A tidak dapat menggunakan kamar mandi hingga B menggunakan kamar mandi lalu memberikan giliran kepada A. Begitu juga sebaliknya.

Gambar 20.1. Algoritma I

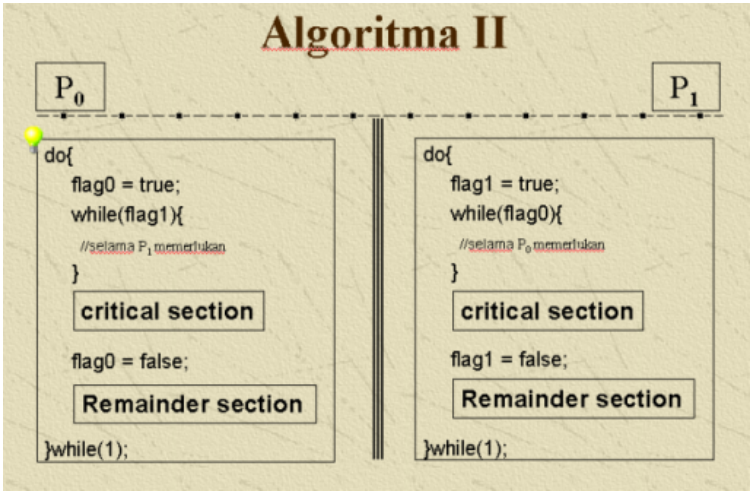


Algoritma 1 menggunakan yield() yang menjaga supaya thread di kondisi Runnable tetapi juga mengizinkan JVM untuk memilih thread Runnable lainnya.

20.3. Algoritma 2

Masalah yang terjadi pada algoritma 1 ialah ketika di entry section terdapat sebuah proses yang ingin masuk ke *critical section*, sementara di *critical section* sendiri tidak ada proses yang sedang berjalan, tetapi proses yang ada di entry section tadi tidak bisa masuk ke *critical section*. Hal ini terjadi karena giliran untuk memasuki *critical section* adalah giliran proses yg lain sementara proses tersebut masih berada di remainder section.

Gambar 20.2. Algoritma II



Untuk mengatasi masalah ini maka dapat diatasi dengan merubah variabel trun pada algoritma pertama dengan variabel flag, seperti pada algoritma kedua. Pada Algoritma ini digunakan boolean flag untuk melihat proses mana yang boleh masuk *critical section*. Proses yang butuh mengakses *critical section* akan memberikan nilai flagnya true. Sedangkan proses yang tidak sedang membutuhkan *critical section* akan memberikan nilai flagnya bernilai false. Suatu proses akan menggunakan *critical section*, jika tidak ada proses lain yang sedang membutuhkan *critical section*, dengan kata lain flag proses lainnya sedang bernilai false.

Pada algoritma ini terlihat bahwa setiap proses melihat proses lain, apakah proses lain itu menginginkan *critical section* atau tidak. Jika ya, maka proses tersebut akan menunggu proses lain selesai memakai *critical section*.

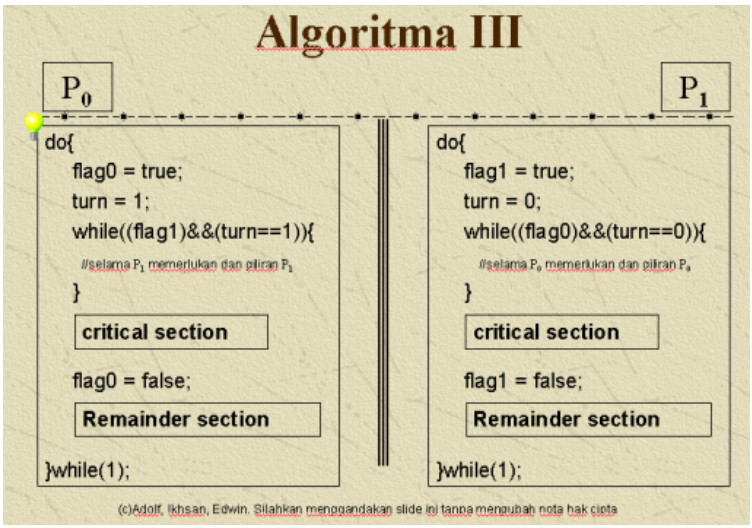
Pertama-tama, masing-masing flag di set false. Kemudian flag0 diset true, menandakan bahwa proses P0 ingin menggunakan *critical section*, lalu P0 melihat apakah flag1 true atau false (proses P1 menginginkan *critical section* atau tidak). Jika ya, maka proses P0 akan mengijinkan proses P1 menggunakannya dahulu dan P0 menunggu. Setelah selesai, maka proses P0 gantian menggunakan *critical section* tersebut, ditandai dengan P1 mengeset flag1=false.

Permasalahan muncul ketika, kedua proses secara bersamaan menginginkan *critical section*, maka kedua proses akan menset flag masing-masing menjadi true. P0 menset flag0=true, P1 menset flag1=true. Lalu P0 akan menunggu P1 selesai menggunakan *critical section* dengan melihat flag1 apakah masih true. Sedangkan P1 juga akan menunggu P0 selesai menggunakan *critical section* dengan melihat apakah flag0 masih true. Akibatnya tidak ada yang mengakses *critical section*, sedangkan ada proses(P0 dan P1) yang ingin mengaksesnya. Sehingga syarat progress tidak terpenuhi. Kondisi ini akan terus bertahan.

20.4. Algoritma 3

Idenya berasal dari algoritma 1 dan 2. Algoritma 3 mengatasi kelemahan pada algoritma 1 dan 2 sehingga progres yang diperlukan untuk mengatasi *critical section* terpenuhi. Berikut ini code dari algoritma 3:

Gambar 20.3. Algoritma III



Pertama-tama masing flag diset false, menandakan bahwa proses tersebut tidak ingin menggunakan *critical section*. Kemudian ketika ada proses yang menginkan *critical section*, maka ia akan mengubah flag(memberikan tanda bahwa ia butuh *critical section*) lalu proses tersebut memberikan turn kepada lawannya. Jika lawannya menginginkan *critical section* dan turn adalah turn lawannya maka ia akan menunggu.

Misalkan, proses-proses dieksekusi sesuai langkah-langkah berikut, Ketika proses P0 menginginkan *critical section*, maka ia menset flag0=true, lalu P1 juga menginginkan *critical section*, maka ia akan menset flag1=true. Selanjutnya P0 memberikan turn pada P1(turn=1), lalu P1 juga memberikan turn pada P0(turn=0). Sehingga P0 dapat memasuki *critical section*, sedangkan P1 menunggu P0 selesai mengakses *critical section* lalu P0 merubah flag0=false. P1 sekarang dapat memasuki *critical section*, setelah selesai P1 akan merubah flag1=false, sedangkan turn terakhir bernilai 0. Jika P0 atau P1 selesai mengakses remainder section dan menginginkan *critical section* maka proses ini akan mirip seperti diatas tentunya yang lebih dulu memasuki *critical section* adalah yang lebih dulu memberikan turn. Jika hanya ada satu proses yang ingin memasuki *critical section*,

maka ia dapat langsung masuk karena tidak perlu menunggu(karena flag lawannya bernilai false, tidak peduli berapapun nilai turnnya).

Apabila kedua proses P0 dan P1 berjalan secara bersamaan, maka proses yang akan memasuki *critical section* lebih dulu adalah proses yang lebih dulu mempersilahkan proses lainnya (yang lebih dulu mengubah turn menjadi turn untuk proses lawan). Karena turn akan diberikan oleh proses yang terakhir kali memberikan turn, artinya proses yang terakhir akan menentukan nilai turn (merubah turn menjadi turn untuk lawannya).

Syarat progress terpenuhi karena ketika *critical section* tidak ada yang menggunakan dan ada proses-proses yang ingin menggunakannya maka dipastikan akan ada yang menggunakan *critical section* tersebut.

20.5. Algoritma Tukang Roti

Algoritma ini didasarkan pada algoritma penjadwalan yang biasanya digunakan oleh tukang roti, di mana urutan pelayanan ditentukan dalam situasi yang sangat sibuk.

Contoh 20.1. *Algoritma Tukang Roti*

```
do {
   choosing[i] = true;
   number[i] = max(number[0], number [1], ..., number [n+1])+1;
   choosing[i] = false;
   for (j=0; j < n; j++) {
      while (choosing[j]);
      while ((number[j]!=0) && ((number[j],j) < number[i],i)));
   }
      critical section
   number[i] = 0;
      remainder section
} while (1);
```

Algoritma ini dapat digunakan untuk memecahkan masalah *critical section* untuk n buah proses, yang diilustrasikan dengan n buah pelanggan. Ketika memasuki toko, setiap pelanggan menerima sebuah nomor. Sayangnya, algoritma tukang roti ini tidak dapat menjamin bahwa dua proses (dua pelanggan) tidak akan menerima nomor yang sama. Dalam kasus di mana dua proses menerima nomor yang sama, maka proses dengan nomor ID terkecil yang akan dilayani dahulu. Jadi, jika `Pi` dan `Pj` menerima nomor yang sama dan `i < j,` maka `Pi` dilayani dahulu. Karena setiap nama proses adalah unik dan berurut, maka algoritma ini dapat digunakan untuk memecahkan masalah *critical section* untuk n buah proses.

Struktur data umum algoritma ini adalah

```
boolean choosing[n];
int number [n];
```

Awalnya, struktur data ini diinisialisasi masing-masing ke `false` dan `0,` dan menggunakan notasi berikut:

```
- (a, b) &lt; (c, d) jika a &lt; c atau jika a= c dan b &lt; d
- max(a0, ...,  an-1) adalah sebuah bilangan k, sedemikian
  sehingga k &gt;= ai untuk setiap i= 0, ..., n - 1
```

20.6. Rangkuman

Solusi dari *critical section* harus memenuhi tiga syarat, yaitu:
1. *mutual exclusion*
2. terjadi kemajuan (*progress*)
3. ada batas waktu tunggu (*bounded waiting*)

Solusi dari *critical section* dibagi menjadi dua jenis, yaitu solusi perangkat lunak dan solusi perangkat keras. Solusi dengan perangkat lunak yaitu dengan menggunakan algoritma 1, algoritma 2 dan algoritma 3 seperti yang telah dijelaskan. Dari ketiga algoritma itu, hanya algoritma 3 yang memenuhi ketiga syarat solusi *critical section*. Untuk menyelesaikan masalah *critical section* untuk lebih dari dua proses, maka dapat digunakan algoritma tukang roti.

Rujukan

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

Bab 21. Perangkat Sinkronisasi I

21.1. Pendahuluan

Salah satu syarat untuk tercapainya sinkronisasi ialah harus tercipta suatu kondisi yang *mutual exclusive*, yaitu suatu kondisi dimana hanya ada sebuah proses yang sedang dieksekusi. Pada pendekatan perangkat keras ini ditekankan bagaimana caranya agar kondisi *mutual exclusive* itu tercapai. Pendekatan dari sisi perangkat keras dapat dibagi menjadi dua:
1. *Processor Synchronous*
2. *Memory Synchronous*

Processor Synchronous

Central Processing Unit (CPU) mempunyai suatu mekanisme yang dinamakan interupsi. Di dalam sistem operasi, mekanisme ini digunakan secara intensif, atau dengan kata lain, banyak Konsep sistem operasi yang menggunakan mekanisme ini. Sebagai contoh: *system call*, *process scheduling*, dan sebagainya.

Berbicara mengenai sinkronisasi berarti kita mengasumsikan bahwa akan ada dua atau lebih proses yang sedang berjalan di komputer secara *concurrent*, atau dengan kata lain konsep *time-shared* sudah diimplementasikan di sistem operasi.

Sistem *time-shared* yang sering diimplementasikan dengan algoritma RR (*Round Robin*), memanfaatkan mekanisme interupsi di CPU. Jadi di dalam RR ada suatu satuan waktu yang dinamakan kuantum yang mana setiap kuantum dibatasi oleh satu interupsi perangkat lunak.

Teknisnya, akan ada suatu interupsi – yang biasanya adalah timer interupsi – yang secara berkala akan menginterupsi sistem. Pada saat interupsi dilakukan sistem operasi akan segera melakukan proses pergantian dari proses yang satu ke proses yang lainnya sesuai dengan algoritma.

Seperti yang telah diketahui bahwa untuk menghentikan instruksi tersebut kita memerlukan suatu mekanisme yang terdapat pada sistem operasi (baca mengenai *process scheduling*). Dan mekanisme tersebut sangat bergantung kepada mekanisme interupsi dari perangkat keras. Sehingga, jika kita dapat menon-aktifkan interupsi pada saat sebuah proses berada di dalam *critical section* maka permasalahan dari sinkronisasi dapat diselesaikan.

Contoh 21.1. ***Critical Section***

```
00  mainModul:
01          CLI          ' masuk ke Critical Section dengan cara
02                       ' men-disable interupsi
03          ADD r1,r2    ' Critical Section
04          ....         ' Critical Section
05          SBI          ' pergi dari Critical Section dengan cara
06                       ' meng-enable interupsi
07          ....         ' Remainder Section
```

Ternyata para perancang komputer melihat celah ini, sehingga saat ini hampir semua komputer yang ada telah mengimplementasikan instruksi mesin yang akan menon-aktifkan serfis interupsi, dan terdapat instruksi mesin lain yang kemudian akan mengaktifkan interupsi tersebut.

Sebagai contoh sederhana, kita akan melihat contoh program dari prosesor Atmel ARM [tm] (contoh ini diambil karena prosesor ini mudah didapatkan dan harganya tidak terlalu mahal, serta memiliki dev-kit.

Pada baris ke 01, prosesor akan menon-aktifkan interupsi, yang menyebabkan instruksi-instruksi berikutnya tidak akan terganggu oleh interupsi. Kemudian setelah setelah baris 03 dieksekusi maka proses akan keluar dari *critical section*, yang menyebabkan prosesor mengaktifkan kembali interupsi dan mekanisme *scheduling* di sistem operasi dapat berjalan kembali.

Terlihat bahwa dengan mekanisme ini kita sudah cukup mengatasi isu yang ada. Tetapi ternyata mekanisme ini tidak dapat diterapkan dengan baik di lingkungan *multiprocessor*. Hal ini disebabkan jika kita menon-aktifkan interupsi, maka yang akan dinon-aktifkan hanyalah **satu** prosesor saja, sehingga dapat mengakibatkan terjadinya hal-hal yang tidak diinginkan.

Memory Synchronous

Dilihat dari nama mekanismenya, maka kita sudah dapat memprediksi bahwa mekanisme ini menggunakan jasa dari memori. Hal tersebut benar adanya, mekanisme *memory synchronous* memakai suatu nilai yang disimpan di dalam memori, dan jika suatu proses berhasil mengubah nilai ini, maka proses tersebut akan meneruskan ke instruksi selanjutnya. Tetapi jika tidak, maka proses ini akan berusaha terus untuk mengubah nilai tersebut.

Jika dilihat dari paragraf di atas, mekanisme ini lebih cocok dikategorikan sebagai pendekatan dari perangkat lunak. Tetapi, jika kita perhatikan lebih lanjut, ternyata mekanisme ini memerlukan jasa dari perangkat keras. Mekanisme ini memiliki suatu syarat yang harus dipenuhi agar dapat berjalan sesuai dengan yang diinginkan yaitu perlunya perangkat keras mempunyai kemampuan untuk membuat suatu instruksi dijalankan secara **atomik**. Pengertian dari instruksi atomik adalah satu atau sekelompok instruksi yang tidak dapat diberhentikan sampai instruksi tsb selesai. Detil mengenai hal ini akan dibicarakan di bagian-bagian selanjutnya.

Keunggulan dari *memory synchronous* adalah pada lingkungan *multiprocessor*, semua processor akan terkena dampak ini. Jadi semua proses yang berada di processor, yang ingin mengakses *critical section*, meski pun berada di processor yang berbeda-beda, akan berusaha untuk mengubah nilai yang dimaksud. Sehingga semua processor akan tersinkronisasi.

21.2. Instruksi Atomik

Seperti yang telah dijelaskan pada bagian sebelumnya, instruksi atomik adalah satu atau sekelompok instruksi yang tidak dapat diberhentikan sampai instruksi tersebut selesai. Kita telah memakai instruksi ini pada method testAndSet. Instruksi yang dimaksud di sini adalah instruksi-instruksi pada high-level programming, bukanlah pada tingkat instruksi mesin yang memang sudah bersifat atomik. Sebagai contoh: i++ pada suatu bahasa pemrograman akan diinterpertasikan beberapa instruksi mesin yang bersifat atomik sebagai berikut.

Contoh 21.2. *testANDset*

```
00 boolean testAndSet( boolean  variable[] )
01      {
02          boolean t = variable[0];
03           variable[0] = true;
04          return t;
05       }
         .....
56       while (testAndSet(lock)) { /* do nothing  */ }
57         // Critical Section
58       Lock[0] = false;
59
// Remainder Section
```

Metoda testAndSet haruslah bersifat atomik, sehingga method ini dianggap sebagai satu instruksi

mesin. Perhatikan pada baris 56 dimana method ini dipakai. Pada baris ini proses berusaha untuk mengubah nilai dari variable reference lock. Jikalau ia tidak berhasil maka akan terus mencoba, tapi jika berhasil maka proses akan masuk ke bagian kritis dan setelah ini proses akan mengubah nilai dari lock sehingga memberikan kemungkinan proses lain untuk masuk.

Contoh 21.3. Asembler

```
00  Load  R1,i  ' load nilai i ke register 1
01  Inc   R1    ' tambahkan nilai register 1 dengan angka 1
02  Store i,R1  ' simpan nilai register 1 ke i
```

Instruksi baris 00-02 bersifat atomik, tetapi i++ tidak bersifat atomik, mengapa? Sebagai contoh kasus, katakanlah sekarang processor baru menyelesaikan baris 01, dan ternyata pada saat tersebut interupsi datang, dan menyebabkan processor melayani interupsi terlebih dahulu. Hal ini menyebabkan terhentinya instruksi i++ sebelum instruksi ini selesai. Jikalau instruksi ini (i++) bersifat atomik, maka ketiga instruksi mesin tsb tidak akan diganggu dengan interupsi.

Banyak mesin yang menyediakan instruksi hardware istimewa yang tidak bisa di-Interrupt. Instruksi-instruksi semacam ini dikenal dengan nama Instruksi Atomik. Berikutnya, akan dijelaskan 2 contoh Instruksi Atomik dan penggunaannya dalam penyelesaian masalah critical section.

Instruksi TestAndSet merupakan sebuah instruksi yang melakukan test terhadap suatu nilai data dan memodifikasi nilai tersebut menjadi suatu nilai tertentu. Untuk lebih mudahnya, silahkan lihat pseudocode dari instruksi TestAndSet dalam bahasa pemrograman Java:

Instruksi Swap dan Implementasinya

Contoh 21.4. Asembler

```
void swap (boolean a, boolean b)
{
 boolean temp = a;
 a = b;
 b = temp;
}

while (true)
{
 boolean key = true;
 while (key == true)
 {
  swap (lock, key);
 }
 // critical section
 lock = false;
 // remainder section
}
```

Instruksi Swap merupakan sebuah instruksi yang melakukan pertukaran nilai antara dua buah data. Untuk lebih mudahnya, silahkan lihat pseudocode dari instruksi Swap dalam bahas pemrograman Java.

21.3. Semafor

Pada Bagian 21.2, “Instruksi Atomik” telah dijelaskan bahwa masalah critical section bisa diselesaikan dengan penggunaan Instruksi Atomik. Akan tetapi, cara tersebut tidak mudah untuk diterapkan pada masalah yang lebih kompleks, misalnya ada lebih dari dua proses yang berjalan. Untuk mengatasi hal ini, kita dapat menggunakan alat sinkronisasi yang dinamakan semafor.

Contoh 21.5. Asembler

```
instruksi wait:
void wait (int s)
{
 while (s &lt;= 0)
 {
  // no operation
 }
 s--;
}

instruksi signal:
void signal (int s)
{
 s++;
}
```

Pada tahun 1967, Djikstra mengajukan suatu konsep dimana kita memakai suatu variable integer untuk menghitung banyaknya proses yang sedang aktif atau yang sedang tidur. Jenis variabel ini disebut semafor.

Contoh 21.6. Asembler

```
instruksi wait:
void wait (int s)
{
   while (s &lt;= 0)
   {
      wait();
   }
   s--;
}

instruksi signal:
void signal (int s)
{
   s++;
   notifyAll();
}
```

Satu hal yang perlu diingat adalah subrutin wait dan signal haruslah bersifat atomik. Di sini kita lihat betapa besarnya dukungan perangkat keras dalam proses sinkronisasi.

Nilai awal dari semafor tersebut menunjukkan berapa banyak proses yang boleh memasuki *critical section* dari suatu program. Biasanya untuk mendukung sifat *mutual exclusive*, nilai ini diberi 1.

Sub-rutin wait akan memeriksa apakah nilai dari semafor tersebut di atas 0. Jika ya, maka nilainya akan dikurangi dan akan melanjutkan operasi berikutnya. Jika tidak maka proses yang menjalankan wait akan menunggu sampai ada proses lain yang menjalankan subrutin signal.

Perlu ditekankan di sini, bahwa semafor bukan digunakan untuk menyelesaikan masalah *critical section* saja, melainkan untuk menyelesaikan permasalahan sinkronisasi secara umum.

Konsep Semafor diajukan oleh DJikstra pada tahun 1967. Dalam konsep ini, kita menggunakan sebuah variabel integer untuk menghitung banyaknya proses yang aktif atau yang sedang tidak aktif atau tidur(sleep). Sebuah semafor S adalah sebuah variabel integer yang, selain saat inisiasi, hanya bisa diakses oleh dua buah operasi atomik standar, wait dan signal. Untuk lebih mudahnya, silahkan lihat pseudocode dari method wait dan method signal dalam bahasa pemrograman java:

Implementasi Semafor

Berikut ini adalah implementasi dari semafor dalam penyelesaian masalah critical section(dalam pseudocode java). Bagian yang di-run oleh lebih dari 1 proses:

Contoh 21.7. Asembler

```
inisialisasi awal:
int mutex = 1;

while (true)
{
   wait (mutex);
   // critical section
   signal (mutex);
   // remainder section
}
```

21.4. *Wait* dan *Signal*

Seperti yang telah dikatakan di atas, bahwa di dalam subrutin ini, proses akan memeriksa harga dari semafor, apabila harganya 0 atau kurang maka proses akan menunggu, sebaliknya jika lebih dari 0, maka proses akan mengurangi nilai dari semafor tersebut dan menjalankan operasi yang lain.

Arti dari harga semafor dalam kasus ini adalah hanya boleh satu proses yang dapat melewati subrutin wait pada suatu waktu tertentu, sampai ada salah satu atau proses itu sendiri yang akan memanggil signal.

Bila kita perhatikan lebih kritis lagi, pernyataan "menunggu" sebenarnya masih abstrak. Bagaimanakah cara proses tersebut menunggu, adalah hal yang menarik. Cara proses menunggu dapat dibagi menjadi dua:

1. **Tipe Spinlock.** Dalam tipe Spinlock, proses melakukan Spinlock waiting, yaitu melakukan iterasi(looping) secara terus menerus tanpa menjalankan perintah apapun. Dengan kata lain, proses terus berada di running state. Tipe Spinlock, yang biasa disebut busy waiting, menghabiskan CPU cycle. Pseudocode bisa dilihat di atas. *Spinlock waiting* berarti proses tersebut menunggu dengan cara menjalankan perintah-perintah yang tidak ada artinya. Dengan kata lain proses masih *running state* di dalam *spinlock waiting*. Keuntungan *spinlock* pada lingkungan multiprocessor adalah, tidak diperlukan *context switch*. Tetapi *spinlock* yang biasanya disebut *busy waiting* ini menghabiskan *cpu cycle* karena, daripada proses tersebut melakukan perintah-perintah yang tidak ada gunanya, sebaiknya dialihkan ke proses lain yang mungkin lebih membutuhkan untuk mengeksekusi perintah-perintah yang berguna.
2. **Tipe Non-Spinlock.** Dalam tipe Non-Spinlock, proses akan melakukan Non-Spinlock waiting, yaitu memblock dirinya sendiri dan secara otomatis masuk ke dalam waiting queue. Di dalam

waiting queue, proses tidak aktif dan menunggu sampai ada proses lain yang membangunkannya dan membawanya ke ready queue. Untuk lebih mudahnya, silahkan lihat pseudocode method wait dan method signal yang bertipe Non-Spinlock dalam bahasa pemrograman java. Berbeda dengan *spinlock waiting*, *non-spinlock waiting*, memanfaatkan fasilitas sistem operasi. Proses yang melakukan *non-spinlock waiting* akan memblock dirinya sendiri dan secara otomatis akan membawa proses tersebut ke dalam *waiting queue*. Di dalam *waiting queue* ini proses tidak aktif dan menunggu sampai ada proses lain yang membangunkan dia sehingga membawanya ke *ready queue*.

Maka marilah kita lihat listing subrutin dari kedua versi wait. Perbedaan dari kedua subrutin ini adalah terletak pada aksi dari kondisi nilai semafor kurang atau sama dengan dari 0 (nol). Untuk *spinlock*, disini kita dapat melihat bahwa proses akan berputar-putar di while baris 02 (maka itu disebut *spinlock* atau menunggu dengan berputar). Sedangkan pada *non-spinlock*, proses dengan mudah memanggil perintah wait, setelah itu sistem operasi akan mengurus mekanisme selanjutnya.

Jangan bingung dengan kata *synchronized* pada baris 10. Kata ini ada karena memang konsep dari Java[tm], apabila sebuah proses ingin menunggu, maka proses tersebut harus menunggu di suatu obyek. Pembahasan mengenai hal ini sudah diluar dari konteks buku ini, jadi untuk lebih lanjut silahkan merujuk kepada buku Java[tm] pegangan anda.

Contoh 21.8. *waitSpinLock*

```
00 void waitSpinLock(int semaphore[])
01 {
02    while(semaphore[0] &lt;= 0)
03    { .. Do nothing .. } // spinlock
03    semaphore[0]--;
04 }
05 void synchronized waitNonSpinLock( int semaphore [])
06 {
07    while(semaphore[0]  &lt;= 0)
08    {
09       wait(); // blocks thread
10    }
11    semaphore[0]--;
12 }
```

Karena subrutin wait memiliki dua versi maka hal ini juga berpengaruh terhadap subrutin signal. Subrutin signal akan terdiri dari dua versi sesuai dengan yang ada pada subrutin wait.

Contoh 21.9. *signalSpinLock*

```
00 void signalSpinLock( int  semaphore [])
01 {
02    semaphore[0]++;
03 }
04
05 void synchronizedsignalNonSpinLock( int semaphore [])
06 {
07    semaphore[0]++;
08    notifyAll(); // membawa waiting thread
09                 // ke ready queue
10 }
```

Letak perbedaan dari kedua subrutin di atas adalah pada notifyAll. NotifyAll berarti membangunkan semua proses yang sedang berada di waiting queue dan menunggu semafor yang disignal.

Perlu diketahui di sini bahwa setelah semafor disignal, proses-proses yang sedang menunggu, apakah itu spinlock waiting ataukah *non-spinlock* waiting, akan **berkompetisi** mendapatkan akses semafor tersebut. Jadi memanggil signal bukan berarti membangunkan salah satu proses tetapi memberikan kesempatan proses-proses untuk berkompetisi.

21.5. Jenis Semafor

Ada 2 macam semafor yang cukup umum, yaitu:
1. *Binary semaphore*
2. *Counting semaphore*

Binary semaphore adalah semafor yang bernilai hanya 1 dan 0. Sedangkan *Counting semaphore* adalah semafor yang dapat bernilai 1 dan 0 dan nilai integer yang lainnya.

Banyak sistem operasi yang hanya mengimplementasi *binary semaphore* sebagai primitif, sedangkan *counting semaphore* dibuat dengan memakai primitif ini. Untuk lebih rinci mengenai cara pembuatan *counting semaphore* dapat dilihat pada bagian berikutnya.

Ada beberapa jenis *counting semaphore*, yaitu semafor yang dapat mencapai nilai negatif dan semafor yang tidak dapat mencapai nilai negatif (seperti yang telah dicontohkan pada bagian sebelumnya).

21.6. *Critical Section* Dan Semafor

Kita telah lihat bagaimana penggunaan semafor untuk menyelesaikan masalah sinkronisasi dengan memakai contoh pada masalah *critical section*. Pada bagian ini, kita akan melihat lebih dekat lagi apa dan seberapa besar sebenarnya peran dari semafor itu sendiri sebagai solusi dalam memecahkan masalah *critical section*.

Lihatlah pada kode-kode di bagian demo. Telitilah, bagian manakah yang harus dieksekusi secara *mutual exclusive*, dan bagian manakah yang tidak. Jika diperhatikan lebih lanjut anda akan menyadari bahwa akan selalu ada **satu pasang** instruksi wait dan signal dari suatu semafor.

Perintah wait digunakan sebagai pintu masuk *critical section* dan perintah signal sebagai pintu keluarnya. Mengapa semafor dapat dijadikan seperti ini? Hal ini disebabkan dengan semafor ketiga syarat utama sinkronisasi dapat dipenuhi.

Seperti yang telah dijelaskan pada bagian sebelumnya, agar *critical section* dapat terselesaikan ada tiga syarat yaitu:
1. *Mutual exclusive*
2. *Make progress*
3. *Bounded waiting*

Sekarang marilah melihat lagi listing program yang ada di bagian sebelumnya mengenai wait dan signal. Jika nilai awal dari semafor diberikan 1, maka artinya adalah hanya ada satu proses yang akan dapat melewati pasangan wait-signal. Proses-proses yang lainnya akan menunggu. Dengan kata lain, mekanisme semafor dengan *policy* nilai diberikan 1, dapat menjamin syarat yang pertama, yaitu *mutual exclusive*.

Bagaimana dengan syarat yang kedua, *make progress*? Sebenarnya pada waktu proses yang sedang berada di dalam *critical section* keluar dari bagian tersebut dengan memanggil signal, proses tersebut **tidak** memberikan akses ke *critical section* kepada proses tertentu yang sedang menunggu tetapi, **membuka** kesempatan bagi proses lain untuk berkompetisi untuk mendapatkannya. Lalu bagaimana jika ada 2 proses yang sedang menunggu dan saling mengalah? mekanisme semafor memungkinkan salah satu pasti ada yang masuk, yaitu yang pertama kali yang berhasil mengurangi nilai semafor menjadi 0. Jadi di sini semafor juga berperan dalam memenuhi syarat kedua.

Untuk syarat yang ketiga, jelas tampak bahwa semafor didefinisikan sebagai pasangan wait-signal.

Dengan kata lain, setelah wait, pasti ada signal. Jadi proses yang sedang menunggu pasti akan mendapat giliran, yaitu pada saat proses sedang berada di *critical section* memanggil signal.

21.7. Solusi Masalah Sinkronisasi Antar Proses Dengan Semafor

Kadangkala kita ingin membuat suatu proses untuk menunggu proses yang lain untuk menjalankan suatu perintah. Isu yang ada di sini adalah bagaimana caranya suatu proses mengetahui bahwa proses yang lain telah menyelesaikan instruksi tertentu. Oleh karena itu digunakanlah semafor karena semafor adalah solusi yang cukup baik dan mudah untuk mengatasi hal tersebut.

Nilai semafor diset menjadi 0

```
Proses 1                        Proses 2
56 print "satu"                 17  wait(semaphoreVar)
57 signal(semaphoreVar)         18  print "dua"
```

siapa pun yang berjalan lebih cepat, maka keluarannya pasti "satu" kemudian diikuti oleh "dua". Hal ini disebabkan karena jika proses 2 berjalan terlebih dahulu, maka proses tersebut akan menunggu (nilai semafor = 0) sampai proses 1 memanggil signal. Sebaliknya jika proses 1 berjalan terlebih dahulu, maka proses tersebut akan memanggil signal untuk memberikan jalan terlebih dahulu kepada proses 2.

21.8. Modifikasi *Binary Semaphore*

Pembuatan *counting semaphore* banyak dilakukan para programer untuk memenuhi alat sinkronisasi yang sesuai dengannya. Seperti yang telah dibahas di atas, bahwa *counting semaphore* ada beberapa macam. Pada bagian ini, akan dibahas *counting semaphore* yang memperbolehkan harga negatif.

Listing program di bawah ini diambil dari buku Silberschatz.

```
00 binary-semaphore S1,S2;
01 int C;
02 wait (S1);
03 C--;
04 if ( C < 0 ) {
05     signal (S1);
06     wait (S2);
07 }
08 signal (S1);
09 wait (S1);
10 C++;
11 if (C <= 0)
12   signal (S2);
13 else
14    signal (S1);
```

Kita memerlukan dua *binary semaphore* pada kasus ini, maka pada baris 00 didefinisikan dua *binary semaphore*. Baris 01 mendefinisikan nilai dari semafor tersebut. Perlu diketahui di sini bahwa waitC adalah wait untuk *counting semaphore*, sedangkan wait adalah untuk *binary semaphore*.

Jika diperhatikan pada subrutin waitC dan signalC di awal dan akhir diberikan pasangan wait dan signal dari *binary semaphore*. Fungsi dari *binary semaphore* ini adalah untuk menjamin *critical section* (instruksi wait dan signal dari semafor bersifat atomik, maka begitu pula untuk waitC dan signalC, jadi kegunaan lain semafor adalah untuk membuat suatu subrutin bersifat atomik).

Binary semaphore S2 sendiri digunakan sebagai tempat menunggu giliran proses-proses. Proses-proses tersebut menunggu dengan cara *spinlock* atau *non-spinlock* tergantung dari implementasi *binary semaphore* yang ada.

Perhatikan baris 03 dan 04. Baris ini berbeda dengan apa yang sudah dijabarkan pada bagian sebelumnya. Karena baris ini maka memungkinkan nilai semafor untuk menjadi negatif. Lalu apa artinya bagi kita? Ternyata nilai negatif mengandung informasi tambahan yang cukup berarti bagi kita yaitu bila nilai semafor negatif, maka absolut dari nilai tersebut menunjukkan banyaknya proses yang sedang menunggu atau wait. Jadi arti baris 11 menyatakan bahwa bila ada proses yang menunggu maka semua proses dibangunkan untuk berkompetisi.

Mengapa pada baris 05 dilakukan signal untuk S1? Alasannya karena seperti yang telah kita ketahui bahwa semafor menjamin ketiga sifat dari *critical section*. Tetapi adalah tidak relevan bila pada saat waktu menunggu, waitC masih mempertahankan mutual exclusivenya. Bila hal ini terjadi, proses lain tidak akan dapat masuk, sedangkan proses yang berada di dalam menunggu proses yang lain untuk signal. Dengan kata lain *deadlock* terjadi. Jadi, baris 05 perlu dilakukan untuk menghilangkan sifat *mutual exclusive* pada saat suatu proses menunggu.

Pada baris 12 hanya menyatakan signal untuk S2 saja. Hal ini bukanlah merupakan suatu masalah, karena jika signal S2 dipanggil, maka pasti ada proses yang menunggu akan masuk dan meneruskan ke instruksi 07 kemudian ke instruksi 08 di mana proses ini akan memanggil signal S1 yang akan mewakili kebutuhan di baris 12.

21.9. Pemrograman *Windows*

Win32API (Windows 32 bit *Application Programming Interface*), menyediakan fungsi-fungsi yang berkaitan dengan semafor. Fungsi-fungsi yang ada antara lain adalah membuat semafor dan menambahkan semafor.

Hal yang menarik dari semafor yang terdapat di Windows [tm] adalah tersedianya dua jenis semafor yaitu, *Binary semaphore* dan *counting semaphore*. Pada Windows[tm] selain kita dapat menentukan nilai awal dari semafor, kita juga dapat menentukan nilai maksimal dari semafor. Setiap thread yang menunggu di semafor pada Windows[tm] menggunakan metode antrian FIFO (*First In First Out*.)

21.10. Rangkuman

Hardware merupakan faktor pendukung yang sangat berperan dalam proses sinkronisasi. Banyak dari para perancang prosesor yang membuat fasilitas *atomic* instruction dalam produknya. Ada 2 metode dalam sinkronisasi hardware, yaitu: *Processor Synchronous* dan *Memory Synchronous*. Semafor merupakan konsep yang dibuat oleh Djikstra dengan mengandalkan sebuah variable integer dan fasilitas atomic instruction dari prosesor. Semafor merupakan primitif dalam pembuatan alat sinkronisasi yang lebih tinggi lagi. Semafor dapat menyelesaikan permasalahan seperti: *Critical section*, sinkronisasi baris, *counting semaphore*, *Dining philosopher*, *readers-writers*, dan *producer-consumer*. banyak dipakai oleh para programer, sebagai contoh dapat dilihat di pemrograman Win32API. Tetapi ternyata Java[tm] tidak menggunakan semafor secara explisit namun memakai konsep monitor yang dibangun dari semafor ini.

Banyak mesin yang menyediakan instruksi hardware istimewa yang tidak bisa di-Interrupt. Instruksi-instruksi semacam ini dikenal dengan nama Instruksi Atomik.

Sebuah semafor S adalah sebuah variabel integer yang, selain saat inisiasi, hanya bisa diakses oleh dua buah operasi atomik standar, wait dan signal.

Rujukan

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

Bab 22. Perangkat Sinkronisasi II

22.1. Pendahuluan

Sejauh ini, kita telah mengenal semafor sebagai perangkat keras sinkronisasi yang ampuh, lalu mengapa kita membutuhkan bahasa pemrograman untuk melakukan sinkronisasi? Alasan yang sederhana adalah karena ternyata implementasi semafor memiliki beberapa kelemahan. Kelemahan yang pertama adalah kenyataan bahwa semafor memerlukan implementasi di tingkat rendah, sesuatu hal yang kompleks untuk dilakukan kebanyakan orang. Bahasa pemrograman lebih mudah untuk dipelajari dan diimplementasikan. Kode semafor terdistribusi dalam seluruh program sehingga menyulitkan pemeliharaan. Hal ini merupakan kelemahan lain dari semafor. Sehingga jelas tampaknya, bahwa kita memerlukan konstruksi tingkat tinggi yang dapat mengatasi atau paling tidak mengurangi kelemahan-kelemahan yang terdapat dalam semafor.

22.2. Transaksi Atomik

Yang dimaksud dengan transaksi atomik adalah suatu transaksi yang dilakukan secara keseluruhan atau tidak dilakukan sama sekali. Sebagai ilustrasi adalah ketika dilakukan transfer dari rekening A ke rekening B terjadi kegagalan listrik, maka lebih baik tidak dilakukan perubahan terhadap balance setiap rekening. Disinilah digunakan instruksi atomik.

Keatomikan (atomicity) merupakan komponen penting dalam menghindari bahaya race condition. Operasi atomik dijamin hanya ada dua kemungkinan keluaran (contohnya berhasil atau gagal) dan ketika banyak proses berusaha melakukan operasi atomik dapat dipastikan hanya satu yang akan berhasil (meskipun semuanya dapat gagal).

Secara tipikal, keatomikan diimplementasikan dengan menyediakan mekanisme yang mencatat transaksi mana yang telah dimulai dan selesai atau dengan membuat salinan data sebelum dilakukan perubahan. Sebagai contoh banyak database mendukung mekanisme commit-rollback dalam penerapan transaksi atomik, dimana bila transaksi berhasil maka dilakukan commit tetapi bila transaksi gagal akan dilakukan rollback ke kondisi awal. Metode ini biasanya menyimpan perubahan dalam sebuah log. Bila sebuah perubahan berhasil dilakukan maka akan disimpan dalam log. Bila terjadi kegagalan maka hal tersebut tidak disimpan dalam log. Bila diperlukan kondisi akan diubah ke kondisi terakhir dari transaksi yang berhasil.

Pada tingkat hardware diperlukan instruksi seperti TestAndSet dan operasi increment/decrement. Bila diperlukan maka dapat dilakukan pencegahan pelayanan interupsi yang terjadi ketika transaksi dijalankan dimana transaksi tersebut harus selesai dijalankan barulah interupsi dilayani.

Keatomikan (atomicity) merupakan komponen penting dalam menghindari bahaya race condition. Operasi atomik dijamin hanya ada dua kemungkinan keluaran (contohnya berhasil atau gagal) dan ketika banyak proses berusaha melakukan operasi atomik dapat dipastikan hanya satu yang akan berhasil (meskipun semuanya dapat gagal).

Secara tipikal, keatomikan diimplementasikan dengan menyediakan mekanisme yang mencatat transaksi mana yang telah dimulai dan selesai atau dengan membuat salinan data sebelum dilakukan perubahan. Sebagai contoh banyak database mendukung mekanisme commit-rollback dalam penerapan transaksi atomik, dimana bila transaksi berhasil maka dilakukan commit tetapi bila transaksi gagal akan dilakukan rollback ke kondisi awal. Metoda ini biasanya menyimpan perubahan dalam sebuah log. Bila sebuah perubahan berhasil dilakukan maka akan disimpan dalam log. Bila terjadi kegagalan maka hal tersebut tidak disimpan dalam log. Bila diperlukan kondisi akan diubah ke kondisi terakhir dari transaksi yang berhasil.

Pada tingkat hardware diperlukan instruksi seperti TestAndSet dan operasi increment/decrement. Bila diperlukan maka dapat dilakukan pencegahan pelayanan interupsi yang terjadi ketika transaksi dijalankan dimana transaksi tersebut harus selesai dijalankan barulah interupsi dilayani.

22.3. *Critical Region*

Critical region adalah bagian dari program dan diamanatkan untuk selalu berada dalam keadaan mutual exclusion. Perbedaan critical region ini dengan mutual exclusion biasa yang dibahas sebelumnya adalah critical region diimplementasikan oleh kompilator. Keuntungan menggunakan ini adalah programer tidak perlu lagi mengimplementasikan algoritma yang rumit untuk mendapatkan mutual exclusion.

Critical region memiliki sebuah komponen boolean yang menguji apakah bagian dari program boleh masuk kedalam state critical region atau tidak. Jika nilai boolean ini true maka proses boleh masuk ke critical region. Jika boolean ini bernilai false bagian yang ini akan dimasukan kedalam sebuah antrian sampai nilai boolean ini bernilai true.

Dalam critical region dikenal ada 2 antrian: main queue dan event queue. Main queue berfungsi untuk menampung proses yang akan memasuki critical region hanya saja critical region masih digunakan oleh proses lain. Event queue berguna untuk menampung proses yang tidak dapat memasuki critical region karena nilai boolaen-nya bernilai false.

22.4. Monitor

Konsep monitor diperkenalkan pertama kali oleh Hoare (1974) dan Brinch Hansen (1975) untuk mengatasi beberapa masalah yang timbul ketika memakai semafor.

Contoh 22.1. Sintaks Monitor

```
monitor monitor-name
{
        shared variable declarations

        Procedure body P1 (....)
        {
                ......
        }

        Procedure body P2 (....)
        {
                ......
        }
                .
                .
                .
                .
                .

        Procedure body Pn (....)
        {
                ......
        }
        {
                initialization code
        }
}
```

Monitor merupakan kumpulan dari prosedur, variabel, dan struktur data dalam satu modul. Monitor hanya dapat diakses dengan menjalankan fungsinya. Kita tidak dapat mengambil variabel dari

monitor tanpa melalui prosedurnya. Hal ini dilakukan untuk melindungi variabel dari akses yang tidak sah dan juga mengurangi terjadinya error.

Konstruksi monitor memastikan hanya satu proses yang aktrif pada suatu waktu. Sehingga sebenarnya programmer tidak membutuhkan synchronization codes yang disisipkan secara eksplisit. Akan tetapi konstruksi monitor tidak benar-benar powerfull untuk modelisasi sebuah skema synchronization, sehingga perlu ditambahkan mekanisme sinkronisasi tambahan.

Monitor dapat dianalogikan sebagai sebuah sekretariat dalam fakultas. Dalam hal ini dimisalkan mahasiswa dan dosen sebagai sebuah proses, serta informasi akademik sebagai sebuah variabel. Bila mahasiswa akan mengambil suatu informasi akademik (misal: transkip nilai), maka ia akan meminta kepada petugas sekretariat untuk mengambilkannya, sebab bila ia sendiri yang mengambil, maka besar kemungkinannya untuk terjadi kerusakan. Sehingga error dapat diperkecil kemungkinannya.

Monitor merupakan konsep bahasa pemrograman, sehingga kompilator bertanggung jawab dalam mengkondisikan monitor sebagai mutual eksklusif. Namun, tidak semua kompilator bisa menerapkannya. Sehingga meski bahasa pemrograman yang sama mungkin tidak memiliki semafor, akan tetapi menambahkan semator akan lebih mudah.

22.5. Pemrograman Java[tm]

Java telah menyediakan alat sinkronisasi untuk programer yaitu kata kunci synchronized. Pada sinkronisasi thread, jika menggunakan kata kunci ini, maka program akan berjalan dengan benar, dalam satu waktu hanya satu thread yang dieksekusi, dan pada saat sebuah thread memulai salah satu method maka dipastikan akan menyelesaikan eksekusi tersebut sebelum ada thread lain yang akan mengeksekusi synchronized method pada objek yang sama. Selain itu, untuk menghindari deadlock, thread dapat memanggil wait(), dan notifyAll(). Tetapi perlu diingat bahwa itu tidak selalu menyelesaikan semua deadlock.

Sehingga bisa dilihat bahwa kata kunci sychronized itu sebenarnya juga diilhami oleh konsep monitor dan dalam konsep ini biasanya menggunakan semafor sebagai primitif.

Jadi dengan kata lain, Java secara implisit telah menyediakan semafor bagi kita, namun bersifat transparan, sehingga tidak dapat diraba dan diketahui baik oleh programer, maupun pengguna akhir.

22.6. Masalah Umum Sinkronisasi

Secara garis besar ada tiga masalah umum yang berkaitan dengan sinkronisasi yang dapat diselesaikan dengan menggunakan semafor, ketiga masalah itu adalah:
1. Masalah *Bounded Buffer* (*Producer/Consumer*)
2. Masalah *Readers/Writers*
3. Masalah *Dining Philosophers*

Latar belakang dan solusi dari ketiga permasalahan di atas akan kita pahami lebih lanjut di bab-bab berikutnya.

22.7. Sinkronisasi Kernel Linux

Cara penjadwalan kernel pada operasinya secara mendasar berbeda dengan cara penjadwalan suatu proses. Terdapat dua cara agar sebuah permintaan akan eksekusi kernel-mode dapat terjadi. Sebuah program yang berjalan dapat meminta service sistem operasi, dari system call atau pun secara implisit (untuk contoh: ketika page fault terjadi). Sebagai alternatif, device driver dapat mengirim interupsi perangkat keras yang menyebabkan CPU memulai eksekusi kernel-define handler untuk suatu interupsi.

Problem untuk kernel muncul karena berbagai tasks mungkin mencoba untuk mengakses struktur data internal yang sama. Jika hanya satu kernel task ditengah pengaksesan struktur data ketika interupsi service routine dieksekusi, maka service routine tidak dapat mengakses atau merubah data yang sama tanpa resiko mendapatkan data yang rusak. Fakta ini berkaitan dengan ide dari *critical section* (Bab 24, *Diagram Graf*).

Sebagai hasilnya, sinkronisasi kernel melibatkan lebih banyak dari hanya penjadwalan proses saja. sebuah framework dibutuhkan untuk memperbolehkan kernel's critical sections berjalan tanpa diinterupsi oleh critical section yang lain.

Solusi pertama yang diberikan oleh linux adalah membuat normal kernel code nonpreemptible (Bab 13, *Konsep Penjadwalan*). Biasanya, ketika sebuah timer interrupt diterima oleh kernel, membuat penjadwal proses, kemungkinan besar akan menunda eksekusi proses yang sedang berjalan pada saat itu dan melanjutkan menjalankan proses yang lain. Biar bagaimana pun, ketika timer interrupt diterima ketika sebuah proses mengeksekusi kernel-system service routine, penjadwalan ulang tidak dilakukan secara mendadak; cukup, kernel need_resched flag teraktifkan untuk memberitahu kernel untuk menjalankan penjadwalan kembali setelah system call selesai dan control dikembalikan ke user mode.

Sepotong kernel code mulai dijalankan, akan terjamin bahwa itu adalah satu-satunya kernel code yang dijalankan sampai salah satu dari aksi dibawah ini muncul:
- interupsi
- *page fault*
- kernel code memanggil fungsi penjadwalan sendiri

Interupsi adalah suatu masalah bila mengandung critical section-nya sendiri. Timer interrupt tidak secara langsung menyebabkan terjadinya penjadwalan ulang suatu proses; hanya meminta suatu jadwal untuk dilakukan kemudian, jadi kedatangan suatu interupsi tidak mempengaruhi urutan eksekusi dari noninterrupt kernel code. Sekali interrupt service selesai, eksekusi akan menjadi lebih simpel untuk kembali ke kernel code yang sedang dijalankan ketika interupsi mengambil alih.

Page faults adalah suatu masalah yang potensial; jika sebuah kernel routine mencoba untuk membaca atau menulis ke user memory, akan menyebabkan terjadinya page fault yang membutuhkan M/K disk untuk selesai, dan proses yang berjalan akan di tunda sampai M/K selesai. Pada kasus yang hampir sama, jika system call service routine memanggil penjadwalan ketika sedang berada di mode kernel, mungkin secara eksplisit dengan membuat direct call pada code penjadwalan atau secara implisit dengan memanggil sebuah fungsi untuk menunggu M/K selesai, setelah itu proses akan menunggu dan penjadwalan ulang akan muncul. Ketika proses jalan kembali, proses tersebut akan melanjutkan untuk mengeksekusi dengan mode kernel, melanjutkan intruksi setelah call (pemanggilan) ke penjadwalan.

Kernel code dapat terus berasumsi bahwa ia tidak akan diganggu (pre-empted) oleh proses lainnya dan tidak ada tindakan khusus dilakukan untuk melindungi critical section. Yang diperlukan adalah critical section tidak mengandung referensi ke user memory atau menunggu M/K selesai.

Teknik kedua yang di pakai Linux untuk critical section yang muncul pada saat interrupt service routines. Alat dasarnya adalah perangkat keras interrupt-control pada processor. Dengan meniadakan interupsi pada saat critical section, maka kernel menjamin bahwa ia dapat melakukan proses tanpa resiko terjadinya ketidak-cocokan akses dari struktur data yang di share.

Untuk meniadakan interupsi terdapat sebuah pinalti. Pada arsitektur perangkat keras kebanyakan, pengadaan dan peniadaan suatu interupsi adalah sesuatu yang mahal. Pada prakteknya, saat interupsi ditiadakan, semua M/K ditunda, dan device yang menunggu untuk dilayani akan menunggu sampai interupsi diadakan kembali, sehingga kinerja meningkat. Kernel Linux menggunakan synchronization architecture yang mengizinkan critical section yang panjang dijalankan untuk seluruh durasinya tanpa mendapatkan peniadaan interupsi. Kemampuan secara spesial berguna pada networking code: Sebuah interupsi pada network device driver dapat memberikan sinyal kedatangan dari keseluruhan paket network, dimana akan menghasilkan code yang baik dieksekusi untuk disassemble, route, dan forward paket ditengah interrupt service routine.

Linux mengimplementasikan arsitektur ini dengan memisahkan interrupt service routine menjadi dua seksi: the top half dan the bottom half. The top half adalah interupsi yang normal, dan berjalan dengan interupsi rekursif ditiadakan (interupsi dengan prioritas yang lebih tinggi dapat menginterupsi routine, tetapi interupsi dengan prioritas yang sama atau lebih rendah ditiadakan). The bottom half service routine berjalan dengan semua interupsi diadakan, oleh miniatur penjadwalan yang menjamin bahwa bottom halves tidak akan menginterupsi dirinya sendiri. The bottom half scheduler dilakukan secara otomatis pada saat interupt service routine ada.

Pemisahan itu berarti bahwa kegiatan proses yang kompleks dan harus selesai diberi tanggapan

untuk suatu interupsi dapat diselesaikan oleh kernel tanpa kecemasan tentang diinterupsi oleh interupsi itu sendiri. Jika interupsi lain muncul ketika bottom half dieksekusi, maka interupsi dapat meminta kepada bottom half yang sama untuk dieksekusi, tetapi eksekusinya akan dilakukan setelah proses yang sedang berjalan selesai. Setiap eksekusi dari bottom half dapat di interupsi oleh top half tetapi tidak dapat diinterupsi dengan bottom half yang mirip.

Arsitektur Top-half bottom-half komplit dengan mekanisme untuk meniadakan bottom halver yang dipilih ketika dieksekusi secara normal, foreground kernel code. Kernel dapat meng-codekan critical section secara mudah dengan mengunakan sistem ini: penanganan interupsi dapat mengkodekan *critical section*-nya sebagai bottom halves, dan ketika foreground kernel ingin masuk ke critical section, setiap bottom halves ditiadakan untuk mencegah critical section yang lain diinterupsi. Pada akhir dari critical section, kernel dapat kembali mengadakan bottom halves dan menjalankan bottom half tasks yang telah di masukkan kedalam queue oleh top half interrupt service routine pada saat critical section.

22.8. Rangkuman

Critical Region merupakan bagian kode yang selalu dilaksanakan dalam kondisi mutual eksklusif. Perbedaannya adalah bahwa yang mengkondisikan mutual eksklusif adalah kompilator dan bukan programer sehingga mengurangi resiko kesalahan programer. Monitor merupakan kumpulan dari prosedur, variabel, dan struktur data dalam satu modul. Dengan mempergunakan monitor, sebuah proses dapat memanggil prosedur di dalam monitor, tetapi tidak dapat mengakses struktur data (termasuk variabel- variabel) internal dalam monitor. Dengan karakteristik demikian, monitor dapat mengatasi manipulasi yang tidak sah terhadap variabel yang diakses bersama-sama karena variabel lokal hanya dapat diakses oleh prosedur lokal.

Rujukan

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBMassey2000] Massey University. May 2000. *Monitors & Critical Regions – http://www-ist.massey.ac.nz/ csnotes/ 355/ lectures/ monitors.pdf* . Diakses 29 Mei 2006.

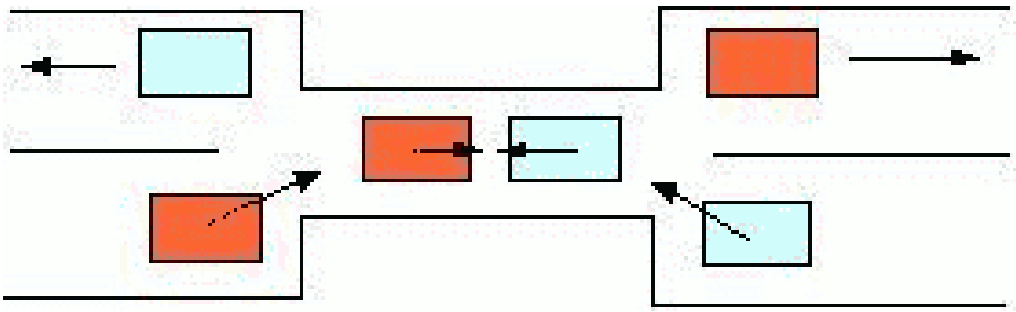
[WEBWiki2006b] From Wikipedia, the free encyclopedia. 2006. *Atomicity– http://en.wikipedia.org/wiki/Atomicity* . Diakses 6 Juni 2006.

Bab 23. *Deadlock*

23.1. Pendahuluan

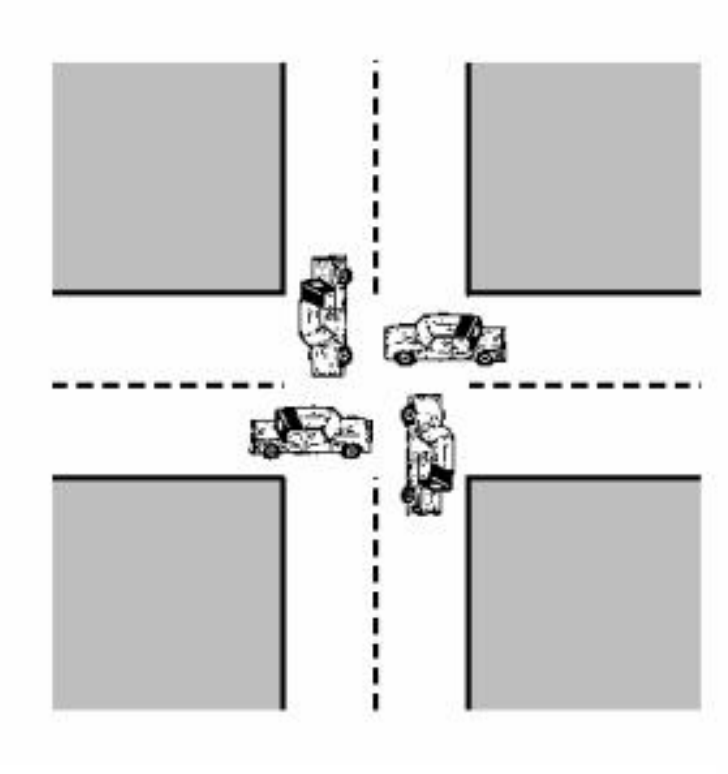
Deadlock dalam arti sebenarnya adalah kebuntuan. Kebuntuan yang dimaksud dalam sistem operasi adalah kebuntuan proses. Jadi *Deadlock* ialah suatu kondisi dimana proses tidak berjalan lagi atau pun tidak ada komunikasi lagi antar proses. *Deadlock* disebabkan karena proses yang satu menunggu sumber daya yang sedang dipegang oleh proses lain yang sedang menunggu sumber daya yang dipegang oleh proses tersebut. Dengan kata lain setiap proses dalam set menunggu untuk sumber yang hanya dapat dikerjakan oleh proses lain dalam set yang sedang menunggu. Contoh sederhananya ialah pada gambar berikut ini.

Gambar 23.1. Contoh Deadlock di Jembatan



Yang berada di sebelah kiri jembatan tidak dapat melaju sebab terjadi *Deadlock* di tengah jembatan (bagian yang dilingkari). Contoh lain ialah di persimpangan jalan berikut ini:

Gambar 23.2. Contoh Deadlock di Persimpangan Jalan



Dalam kasus ini setiap mobil bergerak sesuai nomor yang ditentukan, tetapi tanpa pengaturan yang benar, maka setiap mobil akan bertemu pada satu titik yang permanen (yang dilingkari) atau dapat dikatakan bahwa setiap mobil tidak dapat melanjutkan perjalanan lagi atau dengan kata lain terjadi *Deadlock*. Contoh lain pada proses yang secara umum terdiri dari tiga tahap, yaitu untuk meminta, memakai, dan melepaskan sumber daya yang di mintanya. Contoh kode-nya:

Contoh 23.1. Lalulintas

```
public class Proses {
    public synchronized void getA() {
        //proses untuk mendapat sumber daya a
    }
    public synchronized void getB(){
        //proses untuk mendapat sumber daya b
    }
    public void releaseA(){
        //proses untuk melepaskan sumber daya a
    }
    public void releaseB(){
        //proses untuk melepaskan sumber daya b
    }
}
public class Coba {
    public static void main(String [] args) {
        Proses P = new Proses();
        Proses Q = new Proses();
        P.getA();
        Q.getB();
        P.getB();
        Q.getA();
    }
}
```

Tanpa adanya perintah untuk melepaskan artinya saat P mendapatkan A dan Q mendapatkan B, tetapi tidak dilepaskan, maka saat P minta B dan Q minta A, maka keduanya akan saling menunggu hingga salah satu melepaskan sumber dayanya, sedangkan kebutuhan P ada pada Q dan Q ada pada P, sehingga terjadi *Deadlock*. Secara umum kejadian ini dapat mudah terjadi dalam pemrograman multi-thread. Sebab ada kemungkinan lebih besar untuk menggunakan sumber daya bersama.

23.2. Daur Ulang Sumber Daya

Kejadian *Deadlock* selalu tidak lepas dari sumber daya, seperti kita lihat dari contoh-contoh diatas, bahwa hampir seluruhnya merupakan masalah sumber daya yang digunakan bersama-sama. Oleh karena itu, kita juga perlu tahu tentang jenis sumber daya, yaitu: sumber daya dapat digunakan lagi berulang-ulang dan sumber daya yang dapat digunakan dan habis dipakai atau dapat dikatakan sumber daya sekali pakai.

Sumber daya ini tidak habis dipakai oleh proses mana pun.Tetapi setelah proses berakhir, sumber daya ini dikembalikan untuk dipakai oleh proses lain yang sebelumnya tidak kebagian sumber daya ini. Contohnya prosesor, kanal M/K, disk, semafor. Contoh peran sumber daya jenis ini pada terjadinya *Deadlock* ialah misalnya sebuah proses memakai disk A dan B, maka akan terjadi *Deadlock* jika setiap proses sudah memiliki salah satu disk dan meminta disk yang lain. Masalah ini tidak hanya dirasakan oleh pemrogram tetapi oleh seorang yang merancang sebuah sistem operasi. Cara yang digunakan pada umumnya dengan cara memperhitungkan dahulu sumber daya yang digunakan oleh proses-proses yang akan menggunakan sumber daya tersebut. Contoh lain yang

menyebabkan *Deadlock* dari sumber yang dapat dipakai berulang-ulang ialah berkaitan dengan jumlah proses yang memakai memori utama. Contohnya dapat dilihat dari kode berikut ini:

Contoh 23.2. P-Q

```
//dari kelas proses kita tambahkan method yaitu meminta
public void meminta (int banyakA) {
    //meminta dari sumber daya a
    if ( banyakA &lt; banyak )
         banyak = banyak - banyakA;
    else
        wait();
}

//mengubah kode pada mainnya sebagai berikut
public static void main ( String [] args ) {
    Proses P = new Proses();
    Proses Q = new Proses();
    P.meminta(80);
    Q.meminta(70);
    P.meminta(60);
    Q.meminta(80);
}

private int banyak = 200;
private int banyakA;
```

Setelah proses P dan Q telah melakukan fungsi meminta untuk pertama kali, maka sumber daya yang tersedia dalam banyak ialah 50 (200-70-80). Maka saat P menjalankan fungsi meminta lagi sebanyak 60, maka P tidak akan menemukan sumber daya dari banyak sebanyak 60, maka P akan menunggu hingga sumber daya yang diminta dipenuhi. Demikian juga dengan Q, akan menunggu hingga permintaannya dipenuhi, akhirnya terjadi *Deadlock*. Cara mengatasinya dengan menggunakan memori maya.

23.3. Sumber Daya Sekali Pakai

Dalam kondisi biasa tidak ada batasan untuk memakai sumber daya apa pun, selain itu dengan tidak terbatasnya produksi akan membuat banyak sumber daya yang tersedia. Tetapi dalam kondisi ini juga dapat terjadi *Deadlock*. Contohnya:

Contoh 23.3. Deadlock

```
//menambahkan method receive dan send
public void receive( Proses p ){
    //method untuk menerima sumber daya
}

public void send ( Proses p ){
    //method untuk memberi sumber daya
}
```

Dari kedua fungsi tersebut ada yang bertindak untuk menerima dan memberi sumber daya, tetapi ada kalanya proses tidak mendapat sumber daya yang dibuat sehingga terjadi blok, karena itu terjadi

Deadlock. Tentu saja hal ini sangat jarang terjadi mengingat tidak ada batasan untuk memproduksi dan mengkonsumsi, tetapi ada suatu keadaan seperti ini yang mengakibatkan *Deadlock*. Hal ini mengakibatkan *Deadlock* jenis ini sulit untuk dideteksi. Selain itu *Deadlock* ini dihasilkan oleh beberapa kombinasi yang sangat jarang terjadi.

23.4. Kondisi untuk Terjadinya *Deadlock*

Ada empat kondisi yang dapat menyebabkan terjadinya *deadlock*. Keempat kondisi tersebut tidak dapat berdiri sendiri, saling mendukung.

1. **Mutual Eksklusif.** Hanya ada satu proses yang boleh memakai sumber daya, dan proses lain yang ingin memakai sumber daya tersebut harus menunggu hingga sumber daya tadi dilepaskan atau tidak ada proses yang memakai sumber daya tersebut.
2. **Memegang dan Menunggu.** Proses yang sedang memakai sumber daya boleh meminta sumber daya lagi maksudnya menunggu hingga benar-benar sumber daya yang diminta tidak dipakai oleh proses lain, hal ini dapat menyebabkan kelaparan sumber daya sebab dapat saja sebuah proses tidak mendapat sumber daya dalam waktu yang lama.
3. **Tidak ada *Preemption*.** Sumber daya yang ada pada sebuah proses tidak boleh diambil begitu saja oleh proses lainnya. Untuk mendapatkan sumber daya tersebut, maka harus dilepaskan terlebih dahulu oleh proses yang memegangnya, selain itu seluruh proses menunggu dan mempersilahkan hanya proses yang memiliki sumber daya yang boleh berjalan.
4. ***Circular Wait.*** Kondisi seperti rantai, yaitu sebuah proses membutuhkan sumber daya yang dipegang proses berikutnya.

Banyak cara untuk menanggulangi *Deadlock*

1. Mengabaikan masalah *Deadlock*.
2. Mendeteksi dan memperbaiki
3. Penghindaran yang terus menerus dan pengalokasian yang baik dengan menggunakan protokol untuk memastikan sistem tidak pernah memasuki keadaan *Deadlock*. Yaitu dengan *Deadlock* avoidance sistem untuk mendata informasi tambahan tentang proses mana yang akan meminta dan menggunakan sumber daya.
4. Pencegahan yang secara struktur bertentangan dengan empat kondisi terjadinya *Deadlock* dengan *Deadlock* prevention sistem untuk memasti- kan bahwa salah satu kondisi yang penting tidak dapat menunggu.

23.5. Mengabaikan Masalah *Deadlock*

Untuk memastikan sistem tidak memasuki *deadlock*, sistem dapat menggunakan pencegahan *deadlock* atau penghindaran *deadlock*. Penghindaran *deadlock* membutuhkan informasi tentang sumber daya yang mana yang akan suatu proses meminta dan berapa lama akan digunakan. Dengan informasi tersebut dapat diputuskan apakah suatu proses harus menunggu atau tidak. Hal ini disebabkan oleh keberadaan sumber daya, apakah ia sedang digunakan oleh proses lain atau tidak.

Metode ini lebih dikenal dengan Algoritma Ostrich. Dalam algoritma ini dikatakan bahwa untuk menghadapi *Deadlock* ialah dengan berpura-pura bahwa tidak ada masalah apa pun. Hal ini seakan-akan melakukan suatu hal yang fatal, tetapi sistem operasi Unix menanggulangi *Deadlock* dengan cara ini dengan tidak mendeteksi *Deadlock* dan membiarkannya secara otomatis mematikan program sehingga seakan-akan tidak terjadi apa pun. Jadi jika terjadi *Deadlock*, maka tabel akan penuh, sehingga proses yang menjalankan proses melalui operator harus menunggu pada waktu tertantu dan mencoba lagi.

23.6. Mendeteksi dan Memperbaiki

Caranya ialah dengan cara mendeteksi jika terjadi *Deadlock* pada suatu proses maka dideteksi sistem mana yang terlibat di dalamnya. Setelah diketahui sistem mana saja yang terlibat maka diadakan proses untuk memperbaiki dan menjadikan sistem berjalan kembali.

Jika sebuah sistem tidak memastikan *deadlock* akan terjadi, dan juga tidak didukung dengan pendeteksian *deadlock* serta pencegahannya, maka kita akan sampai pada kondisi *deadlock* yang dapat berpengaruh terhadap performance sistem karena sumber daya tidak dapat digunakan oleh proses sehingga proses-proses yang lain juga terganggu. Akhirnya sistem akan berhenti dan harus

direstart.

Hal-hal yang terjadi dalam mendeteksi adanya *Deadlock* adalah:
1. Permintaan sumber daya dikabulkan selama memungkinkan.
2. Sistem operasi memeriksa adakah kondisi *circular wait* secara periodik.
3. Pemeriksaan adanya *Deadlock* dapat dilakukan setiap ada sumber daya yang hendak digunakan oleh sebuah proses.
4. Memeriksa dengan algoritma tertentu.

Ada beberapa jalan untuk kembali dari *Deadlock*.

Lewat *Preemption*

Dengan cara untuk sementara waktu menjauhkan sumber daya dari pemakainya, dan memberikannya pada proses yang lain. Ide untuk memberi pada proses lain tanpa diketahui oleh pemilik dari sumber daya tersebut tergantung dari sifat sumber daya itu sendiri. Perbaikan dengan cara ini sangat sulit atau dapat dikatakan tidak mungkin. Cara ini dapat dilakukan dengan memilih korban yang akan dikorbankan atau diambil sumber dayanya utuk sementara, tentu saja harus dengan perhitungan yang cukup agar waktu yang dikorbankan seminimal mungkin. Setelah kita melakukan preemption dilakukan pengkondisian proses tersebut dalam kondisi aman. Setelah itu proses dilakukan lagi dalam kondisi aman tersebut.

Lewat Melacak Kembali

Setelah melakukan beberapa langkah *preemption*, maka proses utama yang diambil sumber dayanya akan berhenti dan tidak dapat melanjutkan kegiatannya, oleh karena itu dibutuhkan langkah untuk kembali pada keadaan aman dimana proses masih berjalan dan memulai proses lagi dari situ. Tetapi untuk beberapa keadaan sangat sulit menentukan kondisi aman tersebut, oleh karena itu umumnya dilakukan cara mematikan program tersebut lalu memulai kembali proses. Meski pun sebenarnya lebih efektif jika hanya mundur beberapa langkah saja sampai *Deadlock* tidak terjadi lagi. Untuk beberapa sistem mencoba dengan cara mengadakan pengecekan beberapa kali secara periodik dan menandai tempat terakhir kali menulis ke disk, sehingga saat terjadi *Deadlock* dapat mulai dari tempat terakhir penandaannya berada.

Lewat membunuh proses yang menyebabkan *Deadlock*

Cara yang paling umum ialah membunuh semua proses yang mengalami *Deadlock*. Cara ini paling umum dilakukan dan dilakukan oleh hampir semua sistem operasi. Namun, untuk beberapa sistem, kita juga dapat membunuh beberapa proses saja dalam siklus *Deadlock* untuk menghindari *Deadlock* dan mempersilahkan proses lainnya kembali berjalan. Atau dipilih salah satu korban untuk melepaskan sumber dayanya, dengan cara ini maka masalah pemilihan korban menjadi lebih selektif, sebab telah diperhitungkan beberapa kemungkinan jika si proses harus melepaskan sumber dayanya.

Kriteria seleksi korban ialah:
1. Yang paling jarang memakai prosesor
2. Yang paling sedikit hasil programnya
3. Yang paling banyak memakai sumber daya sampai saat ini
4. Yang alokasi sumber daya totalnya tersedkit
5. Yang memiliki prioritas terkecil

23.7. Menghindari *Deadlock*

Pada sistem kebanyakan permintaan terhadap sumber daya dilakukan sebanyak sekali saja. Sistem sudah harus dapat mengenali bahwa sumber daya itu aman atau tidak (tidak terkena *Deadlock*), setelah itu baru dialokasikan. Ada dua cara yaitu:
1. Jangan memulai proses apa pun jika proses tersebut akan membawa kita pada kondisi *Deadlock*, sehingga tidak mungkin terjadi *Deadlock* karena ketika akan menuju *Deadlock* sudah dicegah.
2. Jangan memberi kesempatan pada suatu proses untuk meminta sumber daya lagi jika penambahan ini akan membawa kita pada suatu keadaan *Deadlock*

Jadi diadakan dua kali penjagaan, yaitu saat pengalokasian awal, dijaga agar tidak *Deadlock* dan ditambah dengan penjagaan kedua saat suatu proses meminta sumber daya, dijaga agar jangan sampai terjadi *Deadlock*. Pada *Deadlock* avoidance sistem dilakukan dengan cara memastikan bahwa program memiliki maksimum permintaan. Dengan kata lain cara sistem ini memastikan terlebih dahulu bahwa sistem akan selalu dalam kondisi aman. Baik mengadakan permintaan awal atau pun saat meminta permintaan sumber daya tambahan, sistem harus selalu berada dalam kondisi aman.

Algoritma Bankir

Menurut Dijkstra (1965) algoritma penjadwalan dapat menghindari *Deadlock* dan algoritma penjadwalan itu lebih dikenal dengan sebutan algoritma bankir. Algoritma ini dapat digambarkan sebagai seorang bankir yang berurusan dengan kelompok orang yang meminta pinjaman. Jadi kepada siapa dia dapat memberikan pinjamannya. Dan setiap pelanggan memberikan batas pinjaman maksimum kepada setiap peminjam dana.

Tentu saja si bankir tahu bahwa si peminjam tidak akan meminjam dana maksimum yang mereka butuhkan dalam waktu yang singkat melainkan bertahap. Jadi dana yang ia punya lebih sedikit dari batas maksimum yang dipinjamkan. Lalu ia memprioritaskan yang meminta dana lebih banyak, sedangkan yang lain disuruh menunggu hingga peminta dana yang lebih besar itu mengembalikan pinjaman berikut bunganya, baru setelah itu ia meminjamkan pada peminjam yang menunggu.

Jadi algoritma bankir ini mempertimbangkan apakah permintaan mereka itu sesuai dengan jumlah dana yang ia miliki, sekaligus memperkirakan jumlah dana yang mungkin diminta lagi. Jangan sampai ia sampai pada kondisi dimana dananya habis dantidak dapat meminjamkan uang lagi. Jika demikian maka akan terjadi kondisi *Deadlock*. Agar kondisi aman, maka asumsi setiap pinjaman harus dikembalikan waktu yang tepat.

Secara umum algoritma bankir dapat dibagi menjadi empat struktur data:
1. **Tersedia.** Jumlah sumber daya/dana yang tersedia.
2. **Maksimum.** Jumlah sumber daya maksimum yang diminta oleh setiap proses.
3. **Alokasi.** Jumlah sumber daya yang dibutuhkan oleh setiap proses.
4. **Kebutuhan.** Maksimum-alokasi, sisa sumber daya yang dibutuhkan oleh proses setelah dikurangi dengan yang dialokasikan.

Kondisi Aman

Sebuah sistem dapat mengalokasikan sumber dayanya dalam kondisi aman kepada setiap proses. Maksud kondisi aman disini yaitu kita menggunakan algoritma aman setelah menggunakan algoritma Bankir.

Gambar 23.3. Tabel 1

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P0	0 1 0	0 0 0	0 0 0
P1	2 0 0	2 0 2	
P2	3 0 3	0 0 0	
P3	2 1 1	1 0 0	
P4	0 0 2	0 0 2	

Algoritma aman merupakan suatu algoritma yang membantu menentukan proses siapa dahulu yang dieksekusi.

```
01   work and finish masing-masing sebuah vektor yang diinisialisasikan:
     work      = tersedia
     finish[i] = FALSE untuk i= 1,2,3,..,n.

02   jika finish[i] == FALSE, kebutuhan i &lt;= work
     else ke tahap 04

03   work = work + Alokasi
     finish [i] = TRUE
     ke tahap 02

04   jika finish[i]==TRUE untuk semua i,
        maka sistem dalam keadaan aman.
```

Gambar 23.4. Tabel 2

	Request
	A B C
P0	0 0 0
P1	2 0 1
P2	0 0 1
P3	1 0 0
P4	0 0 2

23.8. Pencegahan *Deadlock*

Jika pada awal bab ini kita membahas tentang ke-empat hal yang menyebabkan terjadinya *Deadlock*. Maka pada bagian ini, kita akan membahas cara menanggulangi keempat penyebab *Deadlock* itu, sehingga dengan kata lain kita mengadakan pencegahan terhadap *Deadlock*.

1. Masalah Mutual Eksklusif Kondisi ini tidak dapat dilarang, jika aksesnya perlu bersifat spesial untuk satu proses, maka hal ini harus di dukung oleh kemampuan sistem operasi. Jadi diusahakan agar tidak mempergunakan kondisi spesial tersebut sehingga sedapat mungkin *Deadlock* dapat dihindari.
2. Masalah Kondisi Menunggu dan Memegang Penanggulangan *Deadlock* dari kondisi ini lebih baik dan menjanjikan, asalkan kita dapat menahan proses yang memegang sumber daya untuk tidak menunggu sumber daya laun, kita dapat mencegah *Deadlock*. Caranya ialah dengan meminta semua sumber daya yang ia butuhkan sebelum proses berjalan. Tetapi masalahnya sebagian proses tidak mengetahui keperluannya sebelum ia berjalan. Jadi untuk mengatasi hal ini, kita dapat menggunakan algoritma bankir. Yang mengatur hal ini dapat sistem operasi atau pun sebuah protokol. Hasil yang dapat terjadi ialah sumber daya lebih di-spesifikasi dan kelaparan sumber daya, atau proses yang membutuhkan sumber daya yang banyak harus menunggu sekian lama untuk mendapat sumber daya yang dibutuhkan.
3. Masalah tidak ada *Preemption* Hal ketiga ialah jangan sampai ada *preemption* pada sumber daya yang telah dialokasikan. Untuk memastikan hal ini, kita dapat menggunakan protokol. Jadi jika sebuah proses meminta sumber daya yang tidak dapat dipenuhi saat itu juga, maka proses mengalami preempted. Atau dengan kata lain ada sumber daya dilepaskan dan diberikan ke proses yang menunggu, dan proses itu akan menunggu sampai kebutuhan sumber dayanya dipenuhi.

   Atau kita harus mencek sumber daya yang dimaui oleh proses di cek dahulu apakah tersedia. Jika ya maka kita langsung alokasikan, sedangkan jika tidak tersedia maka kita melihat apakah ada proses lain yang menunggu sumber daya juga. Jika ya, maka kita ambil sumber daya dari proses

yang menunggu tersebut dan memberikan pada proses yang meminta tersebut. Jika tidak tersedia juga, maka proses itu harus menunggu. Dalam menunggu, beberapa dari sumber dayanya dapat saja di preempted, tetapi jika ada proses yang memintanya. Cara ini efektif untuk proses yang menyimpan dalam memory atau register.

4. Masalah Circular Wait Masalah ini dapat ditangani oleh sebuah protokol yang menjaga agar sebuah proses tidak membuat lingkaran siklus yang dapat mengakibatkan *Deadlock*.

23.9. Rangkuman

Sebenarnya *deadlock* dapat disebabkan oleh empat hal yaitu:
1. Proses *Mutual Exclusion*
2. Proses memegang dan menunggu
3. Proses *Preemption*
4. Proses Menunggu dengan siklus *deadlock* tertentu

Penanganan *deadlock deadlock*:
1. mengabaikan masalah *deadlock.*
2. mendeteksi dan memperbaiki
3. penghindaran yang terus menerus dan pengalokasian yang baik.
4. pencegahan yang secara struktur bertentangan dengan empat kondisi terjadinya *deadlock.*

Deadlock adalah suatu kondisi dimana sekumpulan proses tidak dapat berjalan kembali akibat kompetisi memperebutkan sumber daya.

Sebuah proses berada dalam keadaan *deadlock* apabila semua proses berada dalam keadaan menunggu (di dalam waiting queue) peristiwa yang hanya bisa dilakukan oleh proses yang berada dalam waiting queue tersebut.

Algoritma Bankir digunakan sebagai salah satu cara untuk menangani *deadlock.* Secara umum algoritma bankir mempunyai struktur data yaitu tersedia, alokasi, maksimum, dan kebutuhan.

Apabila pada sebuah sistem tidak tersedia pencegahan ataupun penghindaran *deadlock*, kemungkinan besar *deadlock* dapat terjadi. Pada keadaan seperti ini, sistem harus menyediakan algoritma pendeteksian *deadlock* algoritma pemulihan *deadlock.*

Rujukan

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

Bab 24. Diagram Graf

24.1. Pendahuluan

Sebuah sistem komputer terdiri dari berbagai macam sumber-daya (*resources*), seperti:
1. Fisik (Perangkat, Memori)
2. Logika (Lock, Database record)
3. Sistem Operasi (PCB Slots)
4. Aplikasi (Berkas)

Diantara sumber-daya tersebut ada yang *preemptable* dan ada juga yang tidak. Sumber-daya ini akan digunakan oleh proses-proses yang membutuhkannya. Mekanisme hubungan dari proses-proses dan sumber-daya yang dibutuhkan/digunakan dapat di diwakilkan dengan graf.

Graf adalah suatu struktur diskrit yang terdiri dari vertex dan sisi, dimana sisi menghubungkan vertex-vertex yang ada. Berdasarkan tingkat kompleksitasnya, graf dibagi menjadi dua bagian, yaitu simple graf dan multigraf. Simpel graf tidak mengandung sisi paralel (lebih dari satu sisi yang menghubungkan dua vertex yang sama). Berdasarkan arahnya graf dapat dibagi menjadi dua bagian yaitu graf berarah dan graf tidak berarah. Graf berarah memperhatikan arah sisi yang menghubungkan dua vertex, sedangkan graf tidak berarah tidak memperhatikan arah sisi yang menghubungkan dua vertex.

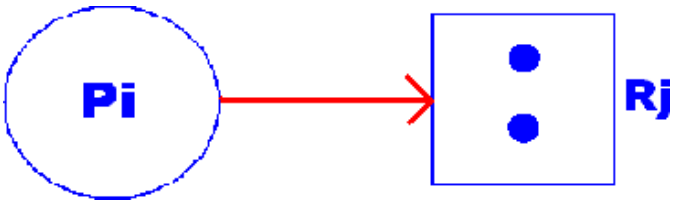
Dalam hal ini akan dibahas mengenai implementasi graf dalam sistem operasi. Salah satunya dalah graf alokasi sumber daya. Graf alokasi sumber daya merupakan graf sederhana dan graf berarah. Graf alokasi sumber daya adalah bentuk visualisasi dalam mendeteksi maupun menyelesaikan masalah *deadlock*.

24.2. Komponen Graf Alokasi Sumber Daya

Graf alokasi sumber daya mempunyai komponen- komponen layaknya graf biasa. Hanya saja dalam graf alokasi sumber daya ini, vertex dibagi menjadi 2 jenis yaitu:
1. Proses P= {P0, P1, P2, P3,..., Pi,..., Pm}. Terdiri dari semua proses yang ada di sistem. Untuk proses, vertexnya digambarkan sebagai lingkaran dengan nama prosesnya.
2. Sumber daya R= {R0, R1, R2, R3,..., Rj,..., Rn}. Terdiri dari semua sumber daya yang ada di sistem. Untuk sumber daya, vertexnya digambarkan sebagai segi empat dengan instans yang dapat dialokasikan serta nama sumber dayanya.

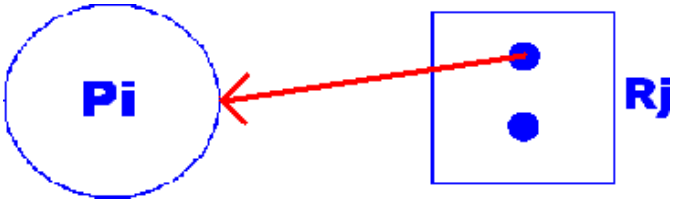
Sisi, E={Pi-> Rj, , Rj-> Pi, } terdiri dari dua jenis, yaitu:
1. Sisi permintaan: Pi -> Rj Sisi permintaan menggambarkan adanya suatu proses Pi yang meminta sumber daya Rj.
2. Sisi alokasi: Rj -> Pi. Sisi alokasi menggambarkan adanya suatu sumber daya Rj yang mengalokasikan salah satu instansnya pada proses Pi.

Gambar 24.1. Proses Pi meminta sumber daya Rj



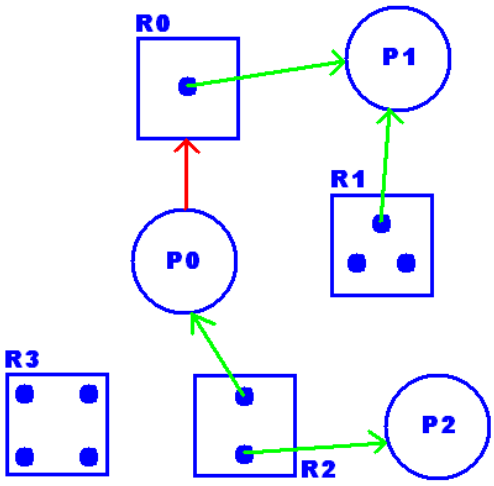
Berikut akan diberikan suatu contoh yang menggambarkan visualisasi dari graf alokasi sumber daya.

Gambar 24.2. Sumber daya Rj yang mengalokasikan salah satu



Pada graf di atas terdiri dari 7 vertex, V={P0, P1, P2, P3, R0, R1, R3} dan 5 sisi, E= {P0->R0, R0->P1, R1->P1, R2->P0, R2->P2}. Gambar 24.3, “Graf Alokasi Sumber Daya” menunjukkan beberapa hal:
1. P0 meminta sumber daya dari R0.
2. R0 memberikan sumber dayanya kepada P1.
3. R1 memberikan salah satu instans sumber dayanya kepada P1.
4. R2 memberikan salah satu instans sumber dayanya kepada P0.
5. R2 memberikan salah satu instans sumber dayanya kepada P2.

Gambar 24.3. Graf Alokasi Sumber Daya



Setelah suatu proses telah mendapatkan semua sumber daya yang diperlukan maka sumber daya tersebut dilepas dan dapat digunakan oleh proses lain.

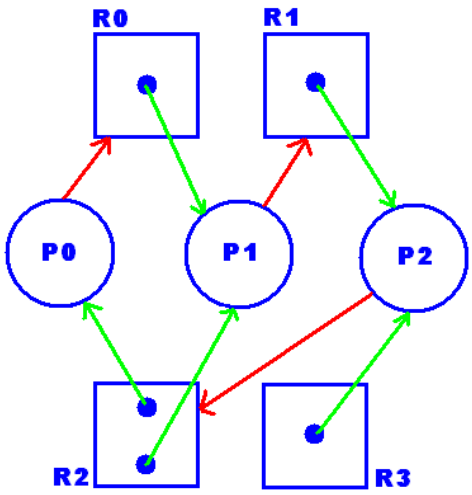
24.3. Pendeteksian *Deadlock*

Untuk detail mengenai pencegahan dan penghindaran dari sebuah deadlock, telah dibahas pada sub-bab sebelumnya. Disini kami hanya akan menunjukan kondisi deadlock tersebut dengan sebuah diagram alokasi sumber daya.

Untuk mengetahui ada atau tidaknya deadlock dalam suatu graf dapat dilihat dari perputaran dan resource yang dimilikinya, yaitu:
1. Jika tidak ada perputaran berarti tidak deadlock.
2. Jika ada perputaran, ada potensi terjadi deadlock.
3. Resource dengan instan tunggal dan perputaran mengakibatkan deadlock.

Pada bagian berikut ini akan ditunjukkan bahwa perputaran tidak selalu mengakibatkan *deadlock*. Pada Gambar 24.4, “Graf dengan *deadlock*” graf memiliki perputaran dan *deadlock* terjadi sedangkan pada Gambar 24.5, “Tanpa *deadlock*” graf memiliki perputaran tetapi tidak terjadi *deadlock*.

Gambar 24.4. Graf dengan *deadlock*

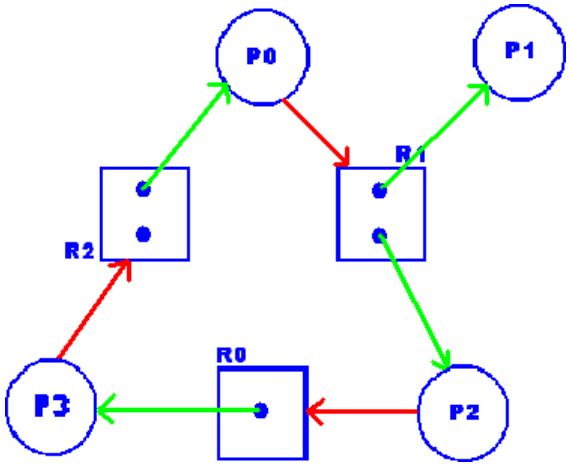


Pada gambar "Graf dengan deadlock", terlihat bahwa ada perputaran yang memungkinkan tejadinya deadlock dan semua sumber daya memiliki satu instans kecuali sumber daya R2. Graf tersebut memiliki minimal dua perputaran, yaitu:
1. R2 -> P0 -> R0 -> P1 -> R1 -> P2 -> R2
2. R2 -> P1 -> R1 -> P2 -> R2

Gambar di atas menunjukkan beberapa hal sebagai berikut:
1. P0 meminta sumber daya R0.
2. R0 mengalokasikan sumber dayanya pada P1.
3. P1 meminta sumber daya R1.
4. R1 mengalokasikan sumber dayanya pada P2.
5. P2 meminta sumber daya R2.
6. R2 mengalokasikan sumber dayanya pada P0 dan P1.
7. R3 mengalokasikan sumber dayanya pada P2.

Hal-hal tersebut dapat mengakibatkan *deadlock* sebab P0 memerlukan sumber daya R0 untuk menyelesaikan prosesnya, sedangkan R0 dialokasikan untuk P1. Di lain pihak P1 memerlukan sumber daya R1 sedangkan R1 dialokasikan untuk P2. P2 memerlukan sumber daya R2 akan tetapi R2 mengalokasikan sumber dayanya pada R3.

Gambar 24.5. Tanpa *deadlock*



Dengan kata lain, tidak ada satu pun dari proses-proses tersebut yang dapat menyelesaikan tugasnya sebab sumber daya yang diperlukan sedang digunakan oleh proses lain. Sedangkan proses lain juga memerlukan sumber daya lain. Semua sumber daya yang diperlukan oleh suatu proses tidak dapat dipenuhi sehingga proses tersebut tidak dapat melepaskan sumber daya yang telah dialokasikan kepadanya. Dan terjadi proses tunggu-menunggu antarproses yang tidak dapat berakhir. Inilah yang dinamakan *deadlock*.

Gambar 24.5, "Tanpa *deadlock*" memiliki perputaran tetapi *deadlock* tidak terjadi. Pada gambar di atas, graf memiliki 1 perputaran yaitu: **P0 –> R1 –> P2 –> R0 –> P3 –> R2 –> P0.**

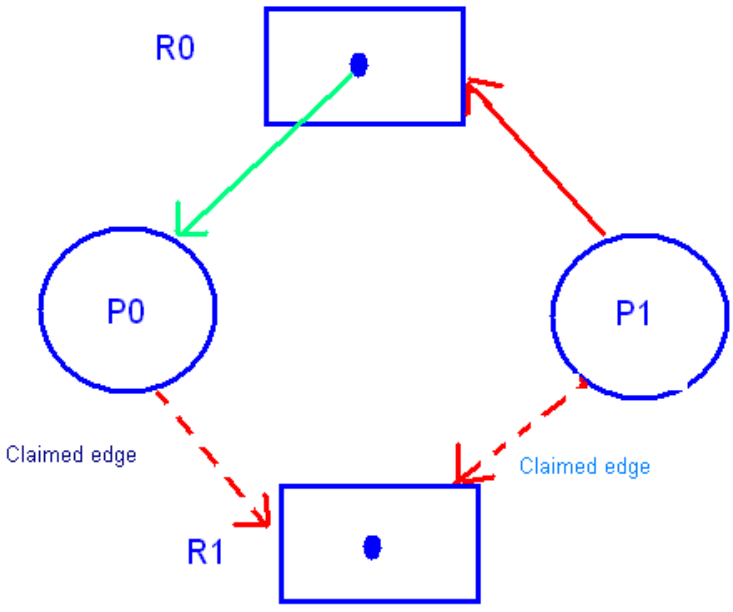
Graf di atas menunjukkan beberapa hal:
1. P0 meminta sumber daya R1.
2. R1 mengalokasikan sumber dayanya pada P2.
3. P2 meminta sumber daya R0.
4. R0 mengalokasikan sumber dayanya pada P3.
5. P3 meminta sumber daya R2.
6. R0 mengalokasikan sumber dayanya pada P3.
7. R1 mengalokasikan sumber dayanya pada P1.

Hal ini tidak menyebabkan *deadlock* walaupun ada perputaran sebab semua sumber-daya yand diperlukan P1 dapat terpenuhi sehingga P1 dapat melepaskan semua sumber-dayanya, yang kemuadian dapat digunakan oleh proses lain.

24.4. Pencegahan *Deadlock*

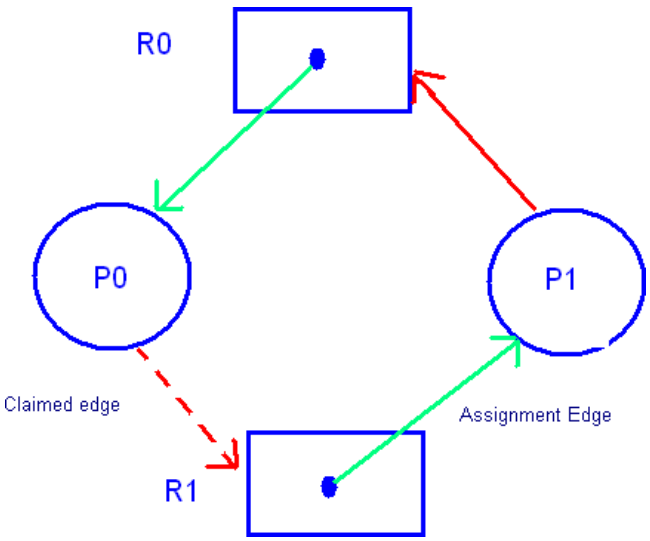
Algoritma ini dapat dipakai untuk mencegah *deadlock* jika sumber daya hanya memiliki satu instans. Pada algoritma ini ada komponen tambahan pada sisi yaitu *claimed edge*. Sama halnya dengan sisi yang lain, claimed edge menghubungkan antara sumber daya dan vertex.

Gambar 24.6. Graf alokasi sumber daya dalam status aman



Claimed edge Pi -> Rj berarti bahwa proses Pi akan meminta sumber daya Rj pada suatu waktu. *Claimed edge* sebenarnya merupakan sisi permintaan yang digamabarkan sebagai garis putus-putus. Ketika proses Pi memerlukan sumber daya Rj, *claimed edge* diubah menjadi sisi permintaan. Dan setelah proses Pi selesai menggunakan Rj, sisi alokasi diubah kembali menjadi *claimed edge*.

Gambar 24.7. Graf alokasi sumber daya dalam status tidak aman

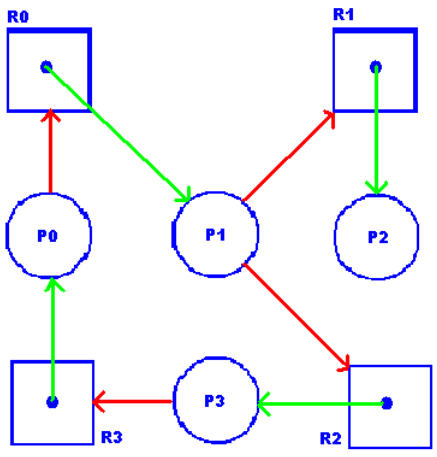


Dengan algoritma ini bentuk perputaran pada graf tidak dapat terjadi. Sebab untuk setiap perubahan yang terjadi akan diperiksa dengan algoritma deteksi perputaran. Algoritma ini memerlukan waktu n^2 dalam mendeteksi perputaran dimana n adalah jumlah proses dalam sistem. Jika tidak ada perputaran dalam graf, maka sistem berada dalam status aman. Tetapi jika perputaran ditemukan maka sistem berada dalam status tidak aman. Pada saat status tidak aman ini, proses Pi harus menunggu sampai permintaan sumber dayanya dipenuhi.

Pada saat ini R1 sedang tidak mengalokasikan sumber dayanya, sehingga P1 dapat memperoleh sumber daya R1. Namun, jika *claimed edge* diubah menjadi sisi permintaan dan kemudian diubah menjadi sisi alokasi, hal ini dapat menyebabkan terjadinya perputaran (Gambar 24.7, “Graf alokasi sumber daya dalam status tidak aman”).

24.5. Pendeteksian dengan Graf Tunggu

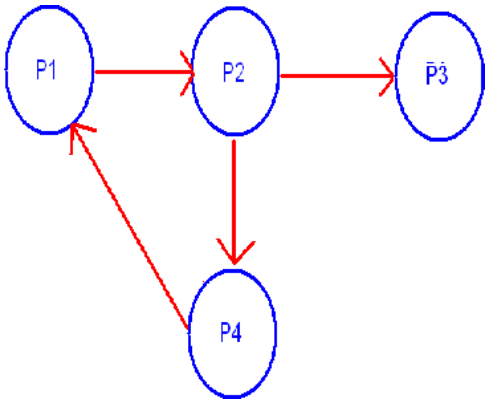
Gambar 24.8. Graf alokasi sumber daya



Jika semua sumber daya hanya memiliki satu instans, *deadlock* dapat dideteksi dengan mengubah graf alokasi sumber daya menjadi graf tunggu. Ada pun caranya sebagai berikut:

1. Cari sumber daya Rm yang memberikan instansnya pada Pi dan Pj yang meminta sumber daya pada Rm.
2. Hilangkan sumber daya Rm dan hubungkan sisi Pi dan Pj dengan arah yang bersesuaian yaitu Pj->Pi.
3. Lihat apakah terdapat perputaran pada graf tunggu? ***Deadlock* terjadi jika dan hanya jika pada graf tunggu terdapat perputaran.**

Gambar 24.9. Graf tunggu



Untuk mendeteksi *deadlock*, sistem perlu membuat graf tunggu dan secara berkala memeriksa apakah ada perputaran atau tidak. Untuk mendeteksi adanya perputaran diperlukan operasi sebanyak n^2, dimana n adalah jumlah vertex dalam graf alokasi sumber daya.

24.6. Rangkuman

Untuk mendeteksi deadlock dan menyelesaikannya dapat digunakan graf sebagai visualisasinya. Jika tidak ada *cycle*, berarti tidak ada *deadlock*. Jika ada *cycle*, ada potensi terjadi *deadlock*. Resource dengan satu instans dan *cycle* mengakibatkan *deadlock*.

Rujukan

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

Bab 25. *Bounded Buffer*

25.1. Pendahuluan

Yang dimaksud dengan bounded buffer adalah suatu struktur data untuk menampung (buffer) suatu nilai dengan kapasitas tertentu (bounded). Bounded buffer mampu menyimpan beberapa nilai dan mengeluarkannya kembali ketika diperlukan. Contoh dari penggunaan bounded buffer adalah pada proses produsen-konsumen. Produsen akan menghasilkan suatu barang dan konsumen akan mengkonsumsi barang yang dihasilkan oleh produsen. Setelah menghasilkan suatu barang, produsen akan menaruh barang itu di bounded buffer. Jika konsumen membutuhkan suatu barang, maka dia akan mengambil dari bounded buffer. Jadi produsen dan konsumen ini akan mengakses bounded buffer yang sama.

Yang menjadi masalah adalah bagaimana jika dua proses berbeda, yaitu produsen dan konsumen, berusaha mengakses buffer tersebut dalam waktu bersamaan. Kedua proses akan berlomba untuk memasuki critical section. Lalu apa yang terjadi bila ketika produsen ingin mengisi sebuah item ke buffer, ternyata buffer sudah penuh (karena kapasitasnya terbatas)? Apakah produsen tetap mengisi buffer tersebut dengan item yang baru sehingga item lama yang belum dibaca konsumen akan hilang? Sebaliknya bagaimana bila konsumen ingin membaca item dari buffer tetapi ternyata tidak ada item di dalam buffer (buffer kosong)?

Contoh 25.1. Class Produsen Konsumen

```
001 // Authors: Greg Gagne, Peter Galvin, Avi Silberschatz
002 // Slightly Modified by: Rahmat M. Samik-Ibrahim
003 // Copyright (c) 2000 by G. Gagne, P. Galvin, A. Silberschatz
004 // Applied Operating Systems Concepts - John Wiley and Sons, Inc.
005 //
006 // Class "Date":
007 //     Allocates a Date object and initializes it so that it
008 //     represents the time at which it was allocated,
009 //     (E.g.): "Wed Apr 09 11:12:34 JAVT 2003"
010 // Class "Object"/ method "notify":
011 //     Wakes up a single thread that is waiting on this object's
    //     monitor.
012 // Class "Thread"/ method "start":
013 //     Begins the thread execution and calls the run method of
    //     the thread.
014 // Class "Thread"/ method "run":
015 //     The Runnable object's run method is called.
016
017 import java.util.*;
018 // main ***************************************************
019 public class BoundedBufferServer
020 {
021     public static void main(String args[])
022     {
023         BoundedBuffer server         = new BoundedBuffer();
024         Producer      producerThread = new Producer(server);
025         Consumer      consumerThread = new Consumer(server);
026         producerThread.start();
027         consumerThread.start();
028     }
029 }
030
```

```
031 // Producer ************************************************
032 class Producer extends Thread
033 {
034     public Producer(BoundedBuffer b)
035     {
036          buffer = b;
037     }
038
039     public void run()
040     {
041         Date message;
042         while (true)
043      {
044             BoundedBuffer.napping();
045
046             message = new Date();
047             System.out.println("P: PRODUCE  " + message);
048             buffer.enter(message);
049         }
050     }
051     private  BoundedBuffer buffer;
052 }
053
054 // Consumer ************************************************
055 class Consumer extends Thread
056 {
057    public Consumer(BoundedBuffer b)
058    {
059       buffer = b;
060    }
061    public void run()
062    {
063          Date message;
064          while (true)
065          {
066              BoundedBuffer.napping();
067              System.out.println("C: CONSUME  START");
068              message = (Date)buffer.remove();
069          }
070     }
071     private  BoundedBuffer buffer;
072 }
073
074 // BoundedBuffer.java ****************************************
075 class BoundedBuffer
076 {
077    public BoundedBuffer()
078    {
079         count  = 0;
080         in     = 0;
081         out    = 0;
082         buffer = new Object[BUFFER_SIZE];
083         mutex  = new Semaphore(1);
084         empty  = new Semaphore(BUFFER_SIZE);
085         full   = new Semaphore(0);
086    }
087    public static void napping()
088    {
089         int sleepTime = (int) (NAP_TIME * Math.random() );
090         try { Thread.sleep(sleepTime*1000); }
091         catch(InterruptedException e) { }
092    }
```

```
093    public void enter(Object item)
094    {
095         empty.P();
096         mutex.P();
097         ++count;
098         buffer[in] = item;
099         in = (in + 1) % BUFFER_SIZE;
100         System.out.println("P: ENTER    " + item);
101         mutex.V();
102         full.V();
103    }
104    public Object remove()
105    {
106         Object item;
107         full.P();
108         mutex.P();
109         --count;
110         item = buffer[out];
111         out = (out + 1) % BUFFER_SIZE;
112         System.out.println("C: CONSUMED " + item);
113         mutex.V();
114         empty.V();
115         return item;
116     }
117     public  static final int NAP_TIME    = 5;
118     private static final int BUFFER_SIZE = 3;
119     private Semaphore        mutex;
120     private Semaphore        empty;
121     private Semaphore        full;
122     private int              count, in, out;
123     private Object[]         buffer;
124 }
125
126 // Semaphore.java ********************************************
127
128 final class Semaphore
129 {
130     public Semaphore()
131     {
132         value = 0;
133     }
134     public Semaphore(int v)
135     {
136        value = v;
137     }
138     public synchronized void P()
139     {
140         while (value <= 0)
141         {
142             try { wait(); }
143             catch (InterruptedException e) { }
144         }
145         value --;
146     }
147     public synchronized void V()
148     {
149         ++value;
150         notify();
151     }
152     private int value;
153 }
```

Contoh 25.2. Keluaran Program

```
C: CONSUME  START
P: PRODUCE  Mon Aug 22 11:19:19 WIT 2005
P: ENTER    Mon Aug 22 11:19:19 WIT 2005
C: CONSUMED Mon Aug 22 11:19:19 WIT 2005
C: CONSUME  START
P: PRODUCE  Mon Aug 22 11:19:20 WIT 2005
P: ENTER    Mon Aug 22 11:19:20 WIT 2005
C: CONSUMED Mon Aug 22 11:19:20 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:21 WIT 2005
P: ENTER    Mon Aug 22 11:19:21 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:22 WIT 2005
P: ENTER    Mon Aug 22 11:19:22 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:23 WIT 2005
P: ENTER    Mon Aug 22 11:19:23 WIT 2005
C: CONSUME  START
C: CONSUMED Mon Aug 22 11:19:21 WIT 2005
C: CONSUME  START
C: CONSUMED Mon Aug 22 11:19:22 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:27 WIT 2005
P: ENTER    Mon Aug 22 11:19:27 WIT 2005
C: CONSUME  START
C: CONSUMED Mon Aug 22 11:19:23 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:29 WIT 2005
P: ENTER    Mon Aug 22 11:19:29 WIT 2005
C: CONSUME  START
C: CONSUMED Mon Aug 22 11:19:27 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:32 WIT 2005
P: ENTER    Mon Aug 22 11:19:32 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:33 WIT 2005
P: ENTER    Mon Aug 22 11:19:33 WIT 2005
```

25.2. Penggunaan Semafor

Kita dapat menerapkan konsep semafor untuk menyelesaikan masalah tersebut. Disini kita menggunakan tiga buah semafor yaitu mutex, full, dan empty. Mutex digunakan untuk menjamin hanya ada satu proses yang boleh berjalan mengakses buffer pada suatu waktu, awalnya diinisialisasi sebesar satu (1). Full digunakan untuk menghitung jumlah buffer yang terisi, awalnya diinisialisasi sebesar nol (0). Sedangkan empty digunakan untuk menghitung jumlah buffer yang kosong, awalnya diinisialisasi sebesar ukuran buffer.

25.3. Penjelasan Program

Kita memiliki dua proses di sini, yaitu produsen dan konsumen. Produsen adalah thread yang menghasilkan waktu(Date) kemudian menyimpannya ke dalam antrian pesan. Produsen juga mencetak waktu tersebut di layar (sebagai umpan balik bagi kita). Konsumen adalah thread yang akan mengakses antrian pesan untuk mendapat kan waktu(Date) itu dan mencetaknya di layar. Kita menginginkan supaya konsumen itu mendapatkan waktu sesuai dengan urutan produsen menyimpan waktu tersebut. Model antrian yang digunakan adalah FIFO (First In First Out).

Pada dasarnya kelas semafor berfungsi untuk melakukan pembatasan-pembatasan terhadap sebuah objek dari thread-thread yang berusaha mengakses objek tersebut. Prinsipnya adalah dengan menggunakan satu buah integer sebagai penanda apakah sebuah thread masih boleh masuk atau tidak, seperti pada potongan *source code* dibawah:

Contoh 25.3. Class Semaphore

```
....
128 final class Semaphore
129 {
130     public Semaphore(int v)
131     {
132         value = v;
133     }
....
152     private int value;
153 }
....
```

Nilai value pada *source code* di atas akan digunakan sebagai pembatas jumlah thread bagi objek yang akan diatur oleh semafor yang bersangkutan, dan nilai v akan berubah-ubah sesuai dengan sisa tempat jika ada thread yang ingin mengakses objek yang bersangkutan. Adapun tata cara penerapan pembatasannya bisa dilihat pada kedua method yang mengatur sebuah semafor, yaitu p dan v:

Contoh 25.4. Semaphore P

```
....
138     public synchronized void P()
139     {
140         while (value <= 0)
141         {
142             try { wait(); }
143             catch (InterruptedException e) { }
144         }
145         value --;
146     }
....
```

Method p akan digunakan apabila ada thread yang berusaha mengakses objek yang bersangkutan, method ini akan mengurangi value dari objek semafor yang menandakan sisa tempat untuk thread yang tersedia berkurang. Jika value dari objek tersebut telah kurang dari atau sama dengan nol, maka seharusnya tidak ada lagi thread yang boleh masuk. Disini terlihat bahwa pada kondisi seperti itu, thread yang memanggil method P tersebut harus menunggu sampai di-notify oleh thread lain yang mengakses method V dan tempat untuknya sudah tersedia.

Contoh 25.5. Semaphore V

```
....
147     public synchronized void V()
148     {
149         ++value;
150         notify();
151     }
....
```

Method V akan digunakan apabila ada thread yang sudah selesai meng akses objek yang bersangkutan. Ia akan menambah nilai value dari semafor yang menandakan bahwa tempat yang

tersedia kini telah bertambah, dan ia akan me-notify thread lain yang tadi mungkin telah menunggu di method P.

Contoh 25.6. Enter

```
....
093    public void enter(Object item)
094    {
095         empty.P();
096         mutex.P();
097         ++count;
098         buffer[in] = item;
099         in = (in + 1) % BUFFER_SIZE;
100         System.out.println("P: ENTER    " + item);
101         mutex.V();
102         full.V();
103    }
....
```

Method enter ini hanya diakses oleh producer karena secara umum method ini berfungsi menambahkan item ke dalam buffer. Pada baris 095 thread memasuki semafor empty untuk mengambil kunci, bila tidak ada kunci yang tersedia maka thread tersebut akan menunggu sampai ada kunci di semafor empty. Pada baris 096 thread memasuki semafor mutex dan mengambil kunci yang ada. Bila dalam semafor tersebut tidak ada kunci, maka thread akan menunggu karena hal ini menandakan bahwa ada pross lain yang berada dalam critical section. Baris 097 berfungsi untuk menambah count yang berfungsi sebagai penanda jumlah item yang ada. Baris 098 berfungsi untuk menunjukkan di bagian mana item harus di simpan. Baris 099 menambah jumlah in yang kemudian di modulo dengan BUFFER_SIZE. Ini berfungsi sebagai pointer untuk item selanjutnya yang akan masuk ke dalam buffer. Baris 101 berfungsi mengembalikan kunci semafor mutex yang sebelumnya telah dipakai. Pada baris 102 kunci di semafor full bertambah atau dengan kata lain, kunci yang sebelumnya dipakai producer (dari semafor empty) di kembalikan ke semafor full.

Contoh 25.7. Remove

```
....
104    public Object remove()
105    {
106         Object item;
107         full.P();
108         mutex.P();
109         --count;
110         item = buffer[out];
111         out = (out + 1) % BUFFER_SIZE;
112         System.out.println("C: CONSUMED " + item);
113         mutex.V();
114         empty.V();
115         return item;
116     }
....
```

Method remove ini hanya diakses oleh konsumen karena secara umum method ini berfungsi mengambil item yang ada di dalam buffer. Pada baris 107 thread memasuki Semaphore full untuk mengambil kunci yang ada, bila tidak ada kunci yang tersedia maka thread tersebut akan menunggu

sampai ada kunci di semafor empty. Pada baris 108 thread memasuki semafor mutex dan mengambil kunci yang ada,bila dalam semafor tesebut tidak ada kunci, maka thread akan menunggu. Baris 109 berfungsi untuk mengurangi count yang ada diprogram ini yang berfungsi menunjukkan jumlah item yang ada. Baris 110 berfungsi untuk menunjukkan dari bagian mana item harus diambil. Baris 111 menambah jumlah out yang kemudian di modulo dengan BUFFER_SIZE, ini berfungsi sebagai pointer untuk item selanjutnya yang akan diambil dari dalam buffer. Baris 113 berfungsi mengembalikan kunci semafor mutex yang sebelumnya telah dipakai. Pada baris 114 kunci di dalam semafor empty bertambah atau dengan kata lain, kunci yang sebelum nya dipakai konsumen (dari semafor empty) di kembalikan ke dalam semafor empty. Baris 115 berfungsi mengembalikan nilai item kepada thread yang memanggilnya.

25.4. Rangkuman

Bounded buffer merupakan salah satu contoh dari masalah sinkronisasi dimana ada beberapa proses secara bersama-sama ingin mengakses *critical section*. Permasalahannya adalah bagaimana mengatur sinkronisasi dari beberapa proses yang secara konkuren ingin mengakses (mengisi/mengosongkan) buffer tersebut. Pengaturan dilakukan dengan menerapkan konsep semafor yang menjamin hanya ada satu proses dalam suatu waktu yang boleh mengakses *buffer* sehingga tidak terjadi *race condition*.

Rujukan

[Deitel2005] Harvey M Deitel dan Paul J Deitel. 2005. *Java How To Program*. Sixth Edition. Prentice Hall.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

Bab 26. *Readers/Writers*

26.1. Pendahuluan

Salah satu dari sekian banyak permasalahan sinkronisasi yang sering terjadi di dunia nyata, yaitu ketika ada dua jenis proses – readers dan writers – yang bisa saling berbagi (shared) data dan mengakses database secara paralel. proses yang pertama (reader) bertugas untuk membaca data, sedangkan proses kedua bertugas untuk menulis data baru/mengupdate nilai dalam data (writer). Solusinya adalah menjadikan proses dapat dijalankan dalam keadaan terpisah/terisolasi dari proses lain. untuk menghindari gangguan dalam perubahan data, writer harus mempunyai kemampuan mengakses data secara eksklusif.

Kondisi berikut harus dipenuhi oleh readers dan writers:
- maksimal hanya ada satu writer yang mengubah data. Jika suatu writer sedang mengubah data, maka tidak ada satupun reader yang diperbolehkan untuk membaca data.
- Dalam suatu waktu diperbolehkan lebih dari satu reader membaca data. Ketika data sedang dibaca, tidak ada writer yang boleh mengubah data.

Dalam memecahkan masalah Readers-Writers haruslah memenuhi kondisi berikut:
- readers yang baru tiba mendapat prioritas yang lebih tinggi daripada writers yang sedang menunggu.
- jika ada lebih dari satu reader yang sedang berada dalam critical section, maka reader yang lain diijinkan untuk memasuki critical section juga.
- writer yang sedang menunggu dapat mengakses data hanya bila tidak ada writers yang sedang berada di dalam sistem.
- ketika writers sedang dijalankan oleh sistem, semua readers yang akan menunggu mempunyai prioritas yang lebih tinggi untuk dijalankan daripada writers yang sedang mengantri.

Akan tetapi, dari solusi ini masih timbul permasalahan baru. yaitu ketika readers terus menerus datang, writers tidak akan mendapatkan giliran untuk mengakses data (starvation).

26.2. Program Java

Contoh 26.1. Class ReaderWriterServer

```
001 // Gabungan ReaderWriterServer.java Reader.java Writer.java
002 //          Semaphore.java Database.java
003 // (c) 2000 Gagne, Galvin, Silberschatz
004
005 public class ReaderWriterServer {
006    public static void main(String args[]) {
007       Database server = new Database();
008       Reader[] readerArray = new Reader[NUM_OF_READERS];
009       Writer[] writerArray = new Writer[NUM_OF_WRITERS];
010       for (int i = 0; i < NUM_OF_READERS; i++) {
011          readerArray[i] = new Reader(i, server);
012          readerArray[i].start();
013       }
014       for (int i = 0; i < NUM_OF_WRITERS; i++) {
015          writerArray[i] = new Writer(i, server);
016          writerArray[i].start();
017       }
018    }
019    private static final int NUM_OF_READERS = 3;
020    private static final int NUM_OF_WRITERS = 2;
021 }
```

```
023 class Reader extends Thread {
024    public Reader(int r, Database db) {
025       readerNum = r;
026       server = db;
027    }
028    public void run() {
029      int c;
030      while (true) {
031        Database.napping();
032        System.out.println("reader " + readerNum + " wants to read.");
033        c = server.startRead();
034        System.out.println("reader " + readerNum +
035             " is reading. Reader Count = " + c);
036        Database.napping();
037        System.out.print("reader " + readerNum + " is done reading. ");
038        c = server.endRead();
039       }
040    }
041    private Database  server;
042    private int       readerNum;
043 }
044
045 class Writer extends Thread {
046    public Writer(int w, Database db) {
047       writerNum = w;
048       server = db;
049    }
050    public void run() {
051      while (true) {
052        System.out.println("writer " + writerNum + " is sleeping.");
053        Database.napping();
054        System.out.println("writer " + writerNum + " wants to write.");
055        server.startWrite();
056        System.out.println("writer " + writerNum + " is writing.");
057        Database.napping();
058        System.out.println("writer " + writerNum + " is done writing.");
059        server.endWrite();
060       }
061    }
062    private Database  server;
063    private int       writerNum;
064 }
065
066 final class Semaphore {
067    public Semaphore() {
068       value = 0;
069    }
070    public Semaphore(int v) {
071       value = v;
072    }
073    public synchronized void P() {
074       while (value <= 0) {
075          try { wait(); }
076          catch (InterruptedException e) { }
077       }
078       value--;
079    }
080    public synchronized void V() {
081       ++value;
082       notify();
083    }
084    private int value;
085 }
```

```
087 class Database {
088    public Database() {
089       readerCount = 0;
090       mutex = new Semaphore(1);
091       db = new Semaphore(1);
092    }
093    public static void napping() {
094      int sleepTime = (int) (NAP_TIME * Math.random() );
095      try { Thread.sleep(sleepTime*1000); }
096      catch(InterruptedException e) {}
097    }
098    public int startRead() {
099       mutex.P();
100       ++readerCount;
101       if (readerCount == 1) {
102          db.P();
103       }
104       mutex.V();
105       return readerCount;
106    }
107    public int endRead() {
108       mutex.P();
109       --readerCount;
110       if (readerCount == 0) {
111          db.V();;
112       }
113       mutex.V();
114       System.out.println("Reader count = " + readerCount);
115       return readerCount;
116    }
117    public void startWrite() {
118       db.P();
119    }
120    public void endWrite() {
121       db.V();
122    }
123    private int readerCount;
124    Semaphore mutex;
125    Semaphore db;
126    private static final int NAP_TIME = 15;
127 }
128
129 // The Class java.lang.Thread
130 // When a thread is created, it is not yet active;
    // it begins to run when method start is called.
131 // Invoking the .tart method causes this thread to begin
132 // execution; by calling the .run. method.
133 // public class Thread implements Runnable {
134 //     ...
135 //     public void run();
136 //     public void start()
137 //        throws IllegalThreadStateException;
138 //     ...
139 // }
```

Contoh 26.2. Keluaran

```
writer 0 is sleeping.
writer 1 is sleeping.
writer 0 wants to write.
writer 0 is writing.
writer 1 wants to write.
reader 1 wants to read.
reader 2 wants to read.
reader 0 wants to read.
writer 0 is done writing.
writer 0 is sleeping.
writer 1 is writing.
writer 0 wants to write.
writer 1 is done writing.
writer 1 is sleeping.
reader 1 is reading. Reader Count = 1
reader 2 is reading. Reader Count = 2
reader 0 is reading. Reader Count = 3
reader 0 is done reading. Reader count = 2
reader 2 is done reading. Reader count = 1
reader 0 wants to read.
reader 0 is reading. Reader Count = 2
writer 1 wants to write.
reader 1 is done reading. Reader count = 1
reader 0 is done reading. Reader count = 0
writer 0 is writing.
reader 0 wants to read.
reader 1 wants to read.
writer 0 is done writing.
writer 0 is sleeping.
writer 1 is writing.
writer 1 is done writing.
writer 1 is sleeping.
reader 0 is reading. Reader Count = 1
reader 1 is reading. Reader Count = 2
reader 2 wants to read.
reader 2 is reading. Reader Count = 3
reader 2 is done reading. Reader count = 2
reader 1 is done reading. Reader count = 1
reader 1 wants to read.
reader 1 is reading. Reader Count = 2
writer 0 wants to write.
```

26.3. Penjelasan Program

Program kami terdiri dari lima kelas. Kelas pertama adalah kelas Semaphore. Kelas ini berfungsi sebagai pengatur agar beberapa *thread* yang sedang berjalan tidak mengakses critical section. Thread yang kami maksud di sini adalah kelas Pembaca dan kelas Penulis. Kelas Semaphore ini mempunyai beberapa method, yang pertama adalah kunci(). Method ini mempunyai fungsi untuk memblok thread dengan memanfaatkan sebuah variabel. Di dalam kelas ini variabel tersebut bernama value. Bila nilai value kurang dari atau sama dengan nol, maka method ini memanggil method wait() yang akan menyebabkan thread-thread yang akan memasuki critical section menunggu. Method kedua adalah buka(). Method ini juga memanfaatkan variabel value. Jika nilai value > 0, maka dia akan memanggil method notify() yang akan membangunkan thread-thread yang sedang menunggu karena memanggil method wait() pada method kunci.

Kelas kedua adalah kelas Penulis. Kelas ini meng-extend *thread* dan berfungsi menjalankan fungsi menulis data pada kelas Database, yang akan dijelaskan lebih lanjut. Kelas ini memanggil method tidur(), mulaiBaca(), dan selesaiBaca() di kelas Database.

Kelas ketiga adalah kelas Pembaca. Kelas ini bertindak sebagai thread yang mempunyai fungsi membaca string yang ada di kelas Database. Kelas ini hanya memanggil method MulaiTulis() dan selesaiTulis().

Kelas keempat adalah kelas Database. Dalam kelas ini terdapat variabel readerCount yang menyatakan jumlah pembaca yang sedang masuk di dalam critical section. Dalam kelas ini terdapat sebuah String yang bernama "nama", yang merupakan bagian yang diakses oleh pembaca dan penulis. Di kelas ini juga terdapat dua objek Semaphore yaitu mutex dan dBase. Semaphore mutex berfungsi untuk mengatur jumlah Pembaca yang sedang mengakses readerCount, penggunaan variabel readerCount yang bersamaan antara beberapa Pembaca menandakan bahwa pengaksesan readerCount ini juga merupakan suatu critical section. Semaphore dBase mempunyai fungsi mengatur thread yang sedang mengakses variabel "nama". Thread yang dimaksud di sini adalah thread-thread Pembaca dan thread-thread Penulis. Variabel "nama" tidak boleh diakses secara bersamaan oleh Pembaca dan Penulis, sehingga variabel nama memerlukan Semaphore dBase ini.

Kelas ini mengimplementasi semua tugas yang dilaksanakan oleh kelas Penulis dan kelas Pembaca. Kelas Database ini mempunyai method-method antara lain tidur(). Method ini berfungsi membuat *thread* yang mengaksesnya akan memanggil method sleep() dengan interval waktu yang random. Dengan adanya hal ini diharapkan jalannya *thread* akan sedikit melambat dan akan berjalan dengan tidak bersamaan, sehingga jalannya *thread* akan dapat diamati.

Method kedua adalah mulaiBaca(). Method ini mereturn sebuah String data yang merepresentasikan data yang sedang dipegang oleh database pada saat itu. Pada baris 93 dapat kita lihat pengaturan yang dilakukan oleh method mulaiBaca() ini, dia hanya mengijinkan hanya satu reader pada satu waktu yang dapat mengakses readerCount dengan mengunci Semaphore mutex. Bila jumlah Pembaca == 1 maka Pembaca tersebut akan menutup Semaphore dBase sehingga Penulis tidak diijinkan masuk untuk mengakses Data.

Method selanjutnya adalah selesaiBaca(). Method ini diakses oleh Pembaca yang telah selesai mengakses data. Untuk mengurangi readerCount maka Pembaca tersebut harus mengakses Semaphore mutex kembali. Jika nilai readerCount == 1, yang berarti sudah tidak ada lagi Pembaca yang sedang mengakses data, maka Pembaca memanggil method buka() untuk membuka Semaphore dBase, sehingga Penulis yang sedang mengantri dapat masuk untuk mengakses data.

Method selanjutnya adalah mulaiTulis(). Method ini hanya memanggil satu method yaitu menutup Semaphore dBase. Hal ini dimaksudkan bila Penulis hendak mengakses data, maka tidak ada Penulis lain atau Pembaca yang diperbolehkan masuk untuk mengakses data. Method selesaiTulis() dipanggil oleh kelas Penulis yang telah selesai mengakses database. Method ini membuka Semaphore dBase sehingga Pembaca atau Penulis lain dapat masuk untuk mengakses data.

Kelas yang terakhir adalah OS_RW. Kelas ini berfungsi menjalankan kelas-kelas di atas. Kelas ini membuat objek Penulis sebanyak tiga dan Pembaca sebanyak dua dan menjalankan semua objek-objek tadi. Jalannya program dapat diamati melalui teks yang dicetak oleh program untuk menandakan suatu objek sedang sampai pada tahap yang mana.

Pada saat program dijalankan kita sulit untuk mengetahui thread manakah yang berjalan lebih dahulu karena waktu sleep() yang random. Yang terpenting di sini adalah pemahaman mengenai proses sinkronisasi yang dijalankan oleh kelas Database untuk mengatur thread-tread Pembaca dan thread-thread Penulis yang mengakses variabel "nama" secara bersamaan. Kaidah pemecahan masalah Readers-Writers harus terpenuhi di sini.

26.4. Rangkuman

Yang dibahas di sini adalah ilustrasi dari beberapa kasus sinkronisasi yang acapkali terjadi di dunia sebenarnya. Benar, bahwa sinkronisasi ini memiliki peran yang penting dalam menjaga validitas data. seperti contoh diatas, bagaimana jika kesalahan data itu terjadi di tempat-tempat penting yang sering melakukan transaksi uang dalam jumlah yang besar... Bagaimana jika kita sebagai nasabah bank yang menyetor uang, jumlah uang kita di bank malah dikurangi? Dan berbagai transaksi

tingkat tinggi lainnya... Solusi yang ditawarkan ada tiga:
1. Solusi Pembaca diprioritaskan
2. Solusi Penulis diprioritaskan
3. Solusi Prioritas Bergantian

Akan tetapi, dari ketiga solusi tersebut hanya solusi ketiga yang tidak mengakibatkan starvation.

Rujukan

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBCornel2005] Cornel Computer Science Department . 2005. *Classic Sync Problems Monitors – http://www.cs.cornell.edu/ Courses/ cs414/ 2005fa/ docs/ cs414-fa05-06-semaphores.pdf* . Diakses 13 Juni 2006.

Bab 27. Sinkronisasi Dua Arah

27.1. Pendahuluan

Sinkronisasi dua arah adalah suatu mekanisme di mana suatu thread dapat mengendalikan sinkronisasi thread lain, begitu pun sebaliknya. Tujuan sinkronisasi dua arah adalah mengendalikan thread-thread (proses) sesuai dengan keinginan kita melalui kendali super proses. Dalam contoh program ini, kami bermaksud mengurutkan proses-proses yang datang secara random. Proses-proses diurutkan dari proses dengan indeks tinggi ke rendah. Hal ini bisa terjadi karena adanya kendali dari Super Proses.

Dalam sinkronisasi dua arah ini, kami juga menggunakan semaphore sebagai alat sinkronisasi yang sudah umum dipakai. Namun, semaphore yang kami gunakan adalah semaphore buatan (class Semafor) dalam bahasa pemrograman Java dengan menggunakan keyword synchronized.Semaphore ini berjenis non-spinlock, artinya jika ada proses yang sedang berjalan, maka proses lain menunggu (wait()). Sedangkan semaphore jenis spinlock akan looping terus-menerus.

27.2. Program Java

Contoh 27.1. Class SuperProses

```
000 /*************************************************************
001  * SuperProses (c) 2005 Rahmat M. Samik-Ibrahim, GPL-like   */
002
003 // ********* SuperProses *
004 public class SuperProses {
005    public static void main(String args[]) {
006       Semafor[] semafor1 = new Semafor[JUMLAH_PROSES];
007       Semafor[] semafor2 = new Semafor[JUMLAH_PROSES];
008       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
009          semafor1[ii] = new Semafor();
010          semafor2[ii] = new Semafor();
011       }
012
013       Thread superp = new
014            Thread(new SuperP(semafor1,semafor2,JUMLAH_PROSES));
015       superp.start();
016
017       Thread[] proses= new Thread[JUMLAH_PROSES];
018       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
019          proses[ii]=new Thread(new Proses(semafor1,semafor2,ii));
020       }
021       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
022          proses[ii].start();
023       }
024    }
025
026    private static final int JUMLAH_PROSES = 16;
027 }
028
```

```
029 // ** SuperP *********************
030 class SuperP implements Runnable {
031    SuperP(Semafor[] sem1, Semafor[] sem2, int jmlh) {
032       semafor1      = sem1;
033       semafor2      = sem2;
034       jumlah_proses = jmlh;
035    }
036
037    public void run() {
038       for (int ii = 0; ii < jumlah_proses; ii++) {
039          semafor1[ii].kunci();
040       }
041       System.out.println("SUPER PROSES siap...");
042       for (int ii = jumlah_proses-1 ; ii >= 0; ii--) {
043          semafor2[ii].buka();
044          semafor1[ii].kunci();
045       }
046    }
047
048    private Semafor[] semafor1, semafor2;
049    private int       jumlah_proses;
050 }
051
052 // ** Proses *********************
053 class Proses implements Runnable {
054    Proses(Semafor[] sem1, Semafor[] sem2, int num) {
055       num_proses = num;
056       semafor1   = sem1;
057       semafor2   = sem2;
058    }
059
060    public void run() {
061       try {Thread.sleep((int)(ISTIROHAT * Math.random()));}
062       catch(InterruptedException e) { }
063       System.out.println("Proses " + num_proses + " hadir...");
064       semafor1[num_proses].buka();
065       semafor2[num_proses].kunci();
066       System.out.println("Proses " + num_proses + " siap...");
067       semafor1[num_proses].buka();
068    }
069    private static final int ISTIROHAT = 16;     // (ms)
070    private Semafor[]        semafor1, semafor2;
071    private int              num_proses;
072 }
073
074 // ** Semafor *
075 class Semafor {
076    public Semafor()        { value = 0;   }
077    public Semafor(int val) { value = val; }
078
079    public synchronized void kunci() {
080       while (value == 0) {
081          try   { wait(); }
082          catch (InterruptedException e) { }
083       }
084       value--;
085    }
086
087    public synchronized void buka() {
088       value++;
089       notify();
090    }
091
092    private int value;
093 }
```

Contoh 27.2. Keluaran Program

```
Proses 12 hadir...
Proses 11 hadir...
Proses 7 hadir...
Proses 6 hadir...
Proses 13 hadir...
Proses 5 hadir...
Proses 10 hadir...
Proses 4 hadir...
Proses 3 hadir...
Proses 8 hadir...
Proses 14 hadir...
Proses 15 hadir...
Proses 2 hadir...
Proses 0 hadir...
Proses 1 hadir...
Proses 9 hadir...
SUPER PROSES siap...
Proses 15 siap...
Proses 14 siap...
Proses 13 siap...
Proses 12 siap...
Proses 11 siap...
Proses 10 siap...
Proses 9 siap...
Proses 8 siap...
Proses 7 siap...
Proses 6 siap...
Proses 5 siap...
Proses 4 siap...
Proses 3 siap...
Proses 2 siap...
Proses 1 siap...
Proses 0 siap...
```

27.3. Alur Program

Di baris 5-10 class SuperProses, kita membuat 2 buah array Semafor berkapasitas 10 (10 proses) yaitu semafor1 dan semafor2. Setiap semafor diisi dengan objek Semafor berparameter kosong, artinya value semafor tersebut diset 0, agar saat pertama kali semafor memanggil method kunci, thread akan terkunci di semafor.

Untuk implementasi sinkronisasi 2 arah, kita membuat 2 jenis Thread, yaitu "SuperP" dan "Proses". Thread SuperP yang kita buat berjumlah 1 (baris 12-13), berfungsi untuk mengendalikan sinkronisasi 10 *thread* Proses. Thread SuperP dapat mengunci diri di semafor1 pada indeks 0-9. Ia juga bisa membuka kunci semafor2 dari indeks 0-9 (baris 16-19). Sebaliknya, ke-10 *thread* Proses juga dapat mengendalikan sinkronisasi dari *thread* SuperP. *thread* proses kita beri nama atau nomor dari proses ke-0 sampai proses ke-9. Proses-proses dapat mengunci diri di semafor2 dan membuka kunci semafor1 pada indeks yang sesuai dengan nomor proses itu. Jadi, kedua jenis *thread* ini dapat saling mengendalikan sinkronisasi satu sama lain dengan cara mengasumsikan semafor1 untuk mekanisme sinkronisasi *thread* SuperP dan semafor2 untuk mekanisme sinkronisasi *thread* Proses.

Ketika *thread* SuperP dijalankan (baris 14), SuperP mengunci dirinya di semafor1 secara bergantian dengan indeks dari 0-9 (baris 35-36). Jika proses ke-0 sampai, maka proses ke-0 akan membuka kunci SuperProses di indeks ke-0 (baris 62), kemudian SuperProses akan mengunci diri di indeks ke-1 sampai proses ke-1 membukakan kuncinya. Demikian seterusnya hingga proses ke-9. Jika ternyata saat SuperProses telah sampai ke array ke-n dan ternyata proses ke-n sudah sampai

sebelumnya, maka hal itu tidak masalah karena saat proses ke-n membuka kunci, ia akan menaikan nilai semafor menjadi 1, sehingga saat SuperProses mengunci diri, ia tidak terkunci, melainkan hanya mengurangkan nilai semafor kembali menjadi 0. Thread SuperProses baru bisa lanjut ke baris 37 setelah semua kunci di semafor1 telah dibuka.

Sebelum thread membuka kunci SuperProses, ia sleep dulu (baris 57-58), waktu sleep random, sehingga tidak dapat ditentukan mana thread proses yang bangun lebih dulu. Saat *thread* proses sudah bangun dan sudah membuka kunci SuperProses (baris ke-62), ia langsung mengunci dirinya di semafor2 pada indeks yang sesuai dengan nomor proses tersebut (baris 63).

Selanjutnya, *thread* SuperProses lanjut ke baris 38-41. Pada looping baris 39-41, *thread* SuperP mengatur agar *thread* Proses berjalan berurutan mulai dari proses 9 sampai proses 0. Caranya adalah membuka kunci *thread* proses satu per satu agar proses siap dengan urutan dari indeks tertinggi ke indeks terendah. Hal ini memungkinkan karena adanya mekanisme seperti yang tertera di baris 40-41 dan baris 63-65. Mekanismenya adalah, SuperProses akan membuka kunci semafor2 dan mengunci diri di semafor1 secara bergantian (baris 40-41). Pertama-tama SuperProses membuka kunci proses ke-9, kemudian ia mengunci diri di semafor1 indeks ke-9 (baris 41). Setelah proses ke-9 mencetak Proses 9 siap, ia akan membuka kunci SuperProses (baris 64-65) dan proses 9 keluar dari program. Selanjutnya SuperProses akan membuka kunci proses selanjutnya. Demikianlah hingga proses ke-0 membuka kunci SuperProses di semafor1 indeks ke-0. Setelah itu thread SuperProses keluar dari program.

Pada output program terlampir, terlihat bahwa 10 proses hadir secara acak. Namun, ke-10 proses tersebut siap dengan urutan dari indeks tertinggi ke indeks terendah. Hal ini dapat terjadi karena adanya mekanisme sinkronisasi 2 arah.

27.4. Rangkuman

Ini merupakan ilustrasi dimana sebuah thread memegang kendali sinkronisasai lainnya.

Rujukan

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

Bagian V. Memori

Pengelolaan memori merupakan komponen penting lainnya dari sebuah Sistem Operasi. Pada bagian ini akan diperkenalkan semua aspek yang berhubungan dengan pengelolaan memori seperti, pengalamatan logika dan fisik, *swap*, halaman (*page*), bingkai (*frame*), memori virtual. segmentasi, serta alokasi memori. Bagian ini akan ditutup dengan penguraian pengelolaan memori Linux.

Bab 28. Manajemen Memori

28.1. Pendahuluan

Memori adalah pusat kegiatan pada sebuah komputer, karena setiap proses yang akan dijalankan harus melalui memori terlebih dahulu. CPU mengambil instruksi dari memori sesuai yang ada pada program counter. Instruksi memerlukan proses memasukkan/menyimpan ke alamat di memori. Tugas sistem operasi adalah mengatur peletakan banyak proses pada suatu memori. Pada bagian ini kita akan membahas berbagai cara untuk mengatur memori. Algoritma untuk manajemen memori ini bervariasi dari yang menggunakan pendekatan primitif pada mesin sampai pemberian halaman dan strategi segmentasi. Memori harus dapat digunakan dengan baik, sehingga dapat memuat banyak proses dalam suatu waktu.

28.2. Address Binding

Biasanya sebuah program ditempatkan dalam disk dalam bentuk berkas biner yang dapat dieksekusi. Sebelum dieksekusi, sebuah program harus ditempatkan di memori terlebih dahulu. Kumpulan proses yang ada pada disk harus menunggu dalam antrian (input queue) sebelum dibawa ke memori dan dieksekusi. Prosedur penempatan yang biasa adalah dengan memilih salah satu proses yang ada di input queue, kemudian proses tersebut ditempatkan ke memori. Sebelum dieksekusi, program akan melalui beberapa tahap dimana dalam setiap tahap alamat sebuah program akan direpresentasikan dengan cara yang berbeda. Alamat di dalam sebuah sumber program biasanya dalam bentuk simbol-simbol. Sebuah kompilator akan memetakan simbol-simbol ini ke alamat relokasi. Linkage editor akan memetakan alamat relokasi ini menjadi alamat absolut. Binding adalah pemetaan dari satu ruang alamat ke alamat yang lain.

Binding instruksi dan data ke memori dapat dapat terjadi dalam tiga cara yang berbeda:

a. **Compilation Time.** Jika kita tahu dimana proses akan ditempatkan di memori pada saat mengkompilasi, maka kode yang absolut dapat dibuat. Kita harus mengkompilasi ulang kode jika lokasi berubah.

b. **Load Time.** Kita harus membuat kode relokasi jika pada saat mengkompilasi kita tidak tahu proses akan ditempatkan dimana dalam memori. Pada kasus ini, binding harus ditunda sampai load time.

c. **Execution Time.** Binding harus ditunda sampai waktu proses berjalan selesai jika pada saat dieksekusi proses dapat dipindah dari satu segmen ke segmen yang lain di dalam memori. Kita butuh perangkat keras khusus untuk melakukan ini.

28.3. Ruang Alamat Logika dan Fisik

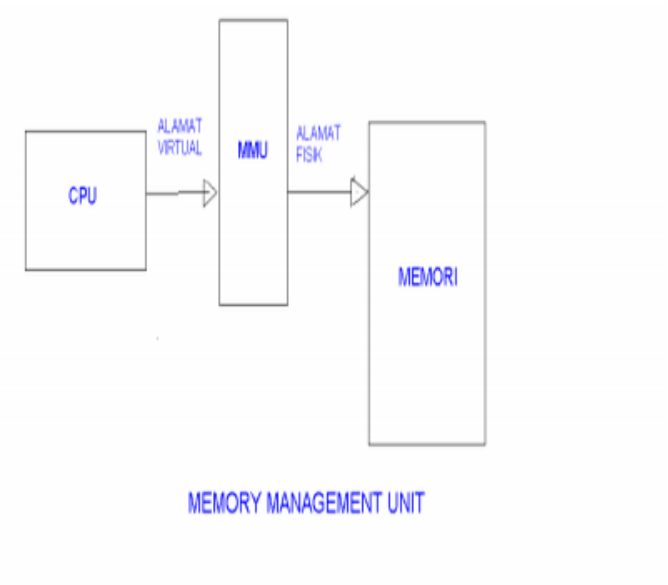
Alamat logika adalah alamat yang dihasilkan oleh CPU, disebut juga alamat virtual. Alamat fisik adalah alamat memori yang sebenarnya. Pada saat waktu kompilasi dan waktu pemanggilan, alamat fisik dan alamat logika adalah sama. Sedangkan pada waktu eksekusi menghasilkan alamat fisik dan alamat virtual yang berbeda.

Kumpulan alamat virtual yang dibuat oleh CPU disebut ruang alamat virtual. Kumpulan alamat fisik yang berkorespondensi dengan alamat virtual disebut ruang alamat fisik. Untuk mengubah alamat virtual ke alamat fisik diperlukan suatu perangkat keras yang bernama Memory Management Unit (MMU).

Register utamanya disebut register relokasi. Nilai pada register relokasi akan bertambah setiap alamat dibuat oleh proses pengguna dan pada waktu yang sama alamat ini dikirimkan ke memori. Ketika ada program yang menunjuk ke alamat memori, kemudian mengoperasikannya, dan menaruh lagi ke memori, akan dilokasikan oleh MMU karena program pengguna hanya berinteraksi dengan

alamat logika. Pengubahan alamat virtual ke alamat fisik merupakan pusat dari manajemen memori.

Gambar 28.1. MMU: *Memory Management Unit*



28.4. Pemanggilan Dinamis

Telah kita ketahui bahwa seluruh proses dan data berada di memori fisik ketika dieksekusi. Ukuran dari memori fisik terbatas. Untuk mendapatkan utilisasi ruang memori yang baik, kita melakukan pemanggilan dinamis. Dengan pemanggilan dinamis, sebuah rutin tidak akan dipanggil sampai diperlukan.

Semua rutin diletakkan dalam disk dengan format yang dapat dialokasikan ulang. Progam utama ditempatkan dalam memori dan dieksekusi. Jika sebuah rutin memanggil rutin lainnya, maka akan dicek terlebih dahulu apakah rutin tersebut ada di dalam memori atau tidak, jika tidak ada maka linkage loader akan dipanggil untuk menempatkan rutin-rutin yang diinginkan ke memori dan memperbaharui tabel alamat program untuk menyesuaikan perubahan. Kemudian kendali diberikan pada rutin yang baru dipanggil.

Keuntungan dari pemanggilan dinamis adalah rutin yang tidak digunakan tidak pernah dipanggil. Metode ini berguna untuk kode dalam jumlah banyak, ketika muncul kasus-kasus yang tidak lazim, seperti rutin yang salah. Dalam kode yang besar, walaupun ukuran kode besar, tapi yang dipanggil dapat jauh lebih kecil.

Pemanggilan Dinamis tidak memerlukan bantuan sistem operasi. Ini adalah tanggung jawab para pengguna untuk merancang program yang mengambil keuntungan dari metode ini. Sistem operasi dapat membantu pembuat program dengan menyediakan kumpulan data rutin untuk mengimplementasi pemanggilan dinamis.

28.5. Link Dinamis dan Pustaka Bersama

Pada proses dengan banyak langkah, ditemukan juga penghubungan-penghubungan pustaka yang dinamis, dimana menghubungkan semua rutin yang ada di pustaka. Beberapa sistem operasi hanya mendukung penghubungan yang statis, dimana seluruh rutin yang ada dihubungkan ke dalam suatu ruang alamat. Setiap program memiliki salinan dari seluruh pustaka. Konsep penghubungan dinamis, serupa dengan konsep pemanggilan dinamis. Pemanggilan lebih banyak ditunda selama waktu eksekusi, dari pada lama penundaan oleh penghubungan dinamis. Keistimewaan ini biasanya digunakan dalam sistem kumpulan pustaka, seperti pustaka bahasa subrutin. Tanpa fasilitas ini, semua program dalam sebuah sistem, harus mempunyai salinan dari pustaka bahasa mereka (atau

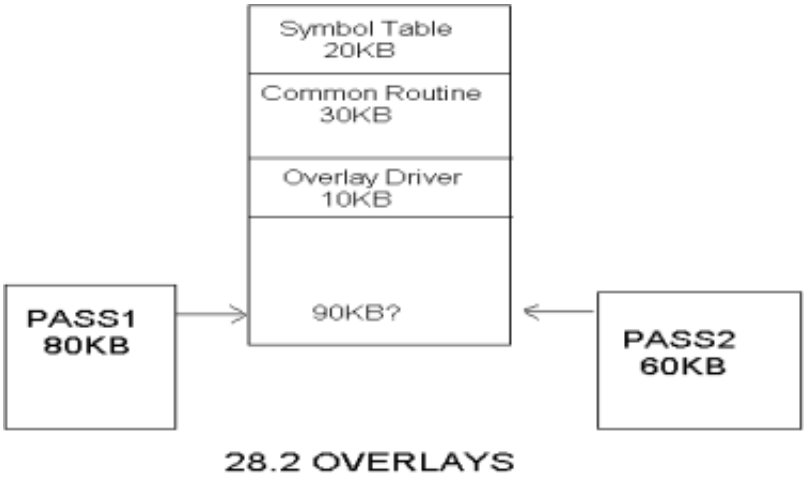
setidaknya referensi rutin oleh program) termasuk dalam tampilan yang dapat dieksekusi. Kebutuhan ini sangat boros baik untuk *disk*, maupun memori utama. Dengan pemanggilan dinamis, sebuah potongan dimasukkan ke dalam tampilan untuk setiap rujukan pustaka subrutin. Potongan ini adalah sebuah bagian kecil dari kode yang menunjukan bagaimana mengalokasikan pustaka rutin di memori dengan tepat, atau bagaimana menempatkan pustaka jika rutin belum ada.

Ketika potongan ini dieksekusi, dia akan memeriksa dan melihat apakah rutin yang dibutuhkan sudah ada di memori. Jika rutin yang dibutuhkan tidak ada di memori, program akan menempatkannya ke memori. Jika rutin yang dibutuhkan ada di memori, maka potongan akan mengganti dirinya dengan alamat dari rutin, dan mengeksekusi rutin. Demikianlah, berikutnya ketika segmentasi kode dicapai, rutin pada pustaka dieksekusi secara langsung, dengan begini tidak ada biaya untuk penghubungan dinamis. Dalam skema ini semua proses yang menggunakan sebuah kumpulan bahasa, mengeksekusi hanya satu dari salinan kode pustaka.

Fasilitas ini dapat diperluas menjadi pembaharuan pustaka. Sebuah kumpulan data dapat ditempatkan lagi dengan versi yang lebih baru dan semua program yang merujuk ke pustaka akan secara otomatis menggunakan versi yang baru. Tanpa pemanggilan dinamis, semua program akan membutuhkan pemanggilan kembali, untuk dapat mengakses pustaka yang baru. Jadi semua program tidak secara sengaja mengeksekusi yang baru, perubahan versi pustaka, informasi versi dapat dimasukkan ke dalam memori, dan setiap program menggunakan informasi versi untuk memutuskan versi mana yang akan digunakan dari salinan pustaka. Sedikit perubahan akan tetap meggunakan nomor versi yang sama, sedangkan perubahan besar akan menambah satu versi sebelumnya. Karenanya program yang dikompilasi dengan versi yang baru akan dipengaruhi dengan perubahan yang terdapat di dalamnya. Program lain yang berhubungan sebelum pustaka baru diinstal, akan terus menggunakan pustaka lama. Sistem ini juga dikenal sebagai berbagi pustaka. Jadi seluruh pustaka yang ada dapat digunakan bersama-sama. Sistem seperti ini membutuhkan bantuan sistem operasi.

28.6. *Overlays*

Gambar 28.2. *Two-Pass Assembler*



Overlays merupakan suatu metoda untuk memungkinkan suatu proses yang membutuhkan memori yang cukup besar menjadi lebih sederhana. Penggunaan overlays ini dapat menghemat memori yang digunakan dalam pengeksekusian instruksi-instruksi. Hal ini sangat berguna terlebih jika suatu program yang ingin dieksekusi mempunyai ukuran yang lebih besar daripada alokasi memori yang tersedia.

Cara kerjanya yaitu pertama-tama membuat beberapa overlays yang didasarkan pada instruksi-instruksi yang dibutuhkan pada satu waktu tertentu. Setelah itu, membuat overlays drivernya yang digunakan sebagai jembatan atau perantara antara overlays yang dibuat.

Proses selanjutnya ialah me-load instruksi yang dibutuhkan pada satu waktu ke dalam absolut

memori dan menunda instruksi lain yang belum di butuhkan pada saat itu. Setelah selesai dieksekusi maka instruksi yang tertunda akan diload menggantikan instruksi yang sudah tidak dibutuhkan lagi. Proses-proses tersebut diatur oleh overlay driver yang dibuat oleh pengguna. Untuk membuat suatu overlays dibutuhkan relokasi dan linking algoritma yang baik oleh pengguna. Untuk lebih jelasnya perhatikan gambar dibawah ini.

Sebagai contoh, ada beberapa instruksi seperti pada gambar diatas. Untuk menempatkan semuanya sekaligus, kita akan membutuhkan 200K memori. Jika hanya 150K yang tersedia, kita tidak dapat menjalankan proses. Perhatikan bahwa pass1 dan pass2 tidak harus berada di memori pada saat yang sama. Kita mendefinisikan dua buah overlays. Overlays A untuk pass1, tabel simbol dan rutin, overlays B untuk pass2, tabel simbol dan rutin. Lalu kita buat sebuah overlay driver (10 Kbytes)sebagai jembatan antara kedua overlays tersebut. Pertama-tama kita mulai dengan me-load overlays A ke memori. Setelah dieksekusi, kemudian pindah ke overlay driver, me-load overlays B ke dalam memori, menimpa overlays A, dan mengirim kendali ke pass2. Overlays A hanya butuh 120 Kb, dan overlays B membutuhkan 150 Kb memori. Nah, sekarang kita dapat menjalankan program dengan memori 150 Kb (karena kita menjalankannya secara bergantian). Seperti dalam pemanggilan dinamis, overlays tidak membutuhkan bantuan dari sistem operasi. Implementasi dapat dilakukan sepenuhnya oleh pengguna, oleh karenanya programmer harus merancang overlays tersebut dengan algoritma yang tepat.

28.7. Rangkuman

Memori adalah pusat kegiatan pada komputer karena setiap proses yang akan dijalankan harus melalui memori terlebih dahulu. CPU mengambil instruksi dari memori sesuai dengan yang ada pada program counter. Alamat-alamat yang dihasilkan oleh CPU merupakan alamat virtual (logic), alamat-alamat ini akan dipetakan ke alamat-alamat fisik (alamat-alamat memori yang sebenarnya). Pemetaan ini dapat dilakukan pada saat kompilasi, saat pemuatan (loading) atau saat eksekusi. Adakalanya ukuran program yang akan dieksekusi melebihi ukuran memori, hal ini dapat diatasi dengan menggunakan dynamic loading dan overlays. Overlays memungkinkan suatu program yang mempunyai ukuran yang besar dapat dieksekusi oleh memori dengan kapasitas yang lebih kecil. Penggunaan overlays ini dikarenakan pada masa lalu memori yang ada sangatlah kecil, sehingga banyak program yang kapasitasnya jauh lebih besar daripada memori yang tersedia. Tetapi dewasa ini overlays sudah tidak digunakan lagi karena sudah ada virtual memori yang memungkinkan memori dapat mengeksekusi program manapun.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBAmirSch2000] Yair Amir dan Theo Schlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni

2006.

Bab 29. Alokasi Memori

29.1. Pendahuluan

Sebuah proses agar bisa dieksekusi bukan hanya membutuhkan sumber daya dari CPU, tetapi juga harus terletak dalam memori. Dalam tahapannya, suatu proses bisa saja ditukar sementara keluar memori ke sebuah penyimpanan sementara dan kemudian dibawa lagi ke memori untuk melanjutkan pengeksekusian. Hal ini dalam sistem operasi disebut swapping. Sebagai contoh, asumsikan sebuah multiprogramming environment dengan algoritma penjadwalan CPU round-robin. Ketika waktu kuantum habis, pengatur memori akan menukar proses yang telah selesai dan memasukkan proses yang lain ke dalam memori yang sudah bebas. Sementara di saat yang bersamaan, penjadwal CPU akan mengalokasikan waktu untuk proses lain di dalam memori. Ketika waktu kuantum setiap proses sudah habis, proses tersebut akan ditukar dengan proses lain. Untuk kondisi yang ideal, manajer memori dapat melakukan penukaran proses dengan cepat sehingga proses akan selalu berada dalam memori dan siap dieksekusi saat penjadwal CPU hendak menjadwal CPU. Hal ini juga berkaitan dengan CPU utilization.

Swaping dapat juga kita lihat dalam algoritma berbasis prioritas. Jika proses dengan prioritas lebih tinggi tiba dan meminta layanan, manajer memori dapat menukar keluar memori proses-proses yang prioritasnya rendah sehingga proses-proses yang prioritasnya lebih tinggi tersebut dapat dieksekusi. Setelah proses-proses yang memiliki prioritas lebih tinggi tersebut selesai dieksekusi, proses-proses dengan prioritas rendah dapat ditukar kembali ke dalam memori dan dilanjutkan eksekusinya. Cara ini disebut juga dengan metoda roll in, roll out.

Ketika proses yang sebelumnya ditukar, akan dikembalikan ke ruang memori. Ada 2 kemungkinan yang terjadi. Pertama, apabila pemberian alamat dilakukan pada waktu pembuatan atau waktu pengambilan, maka proses tersebut pasti akan menempati ruang memori yang sama. Akan tetapi, apabila pemberian alamat diberikan pada waktu eksekusi, ada kemungkinan proses akan dikembalikan ke ruang memori yang berbeda dengan sebelumnya. Dua kemungkinan ini berkaitan dengan penjelasan pada bab sebelumnya yaitu Manajemen Memori.

Penukaran membutuhkan sebuah penyimpanan sementara. Penyimpanan sementara pada umumnya adalah sebuah *fast disk*, dan harus cukup untuk menampung salinan dari seluruh gambaran memori untuk semua pengguna, dan harus mendukung akses langsung terhadap gambaran memori tersebut. Sistem mengatur *ready queue* yang berisikan semua proses yang gambaran memorinya berada di memori dan siap untuk dijalankan. Saat sebuah penjadwal CPU ingin menjalankan sebuah proses, ia akan memeriksa apakah proses yang mengantri di *ready queue* tersebut sudah berada di dalam memori tersebut atau belum. Apabila belum, penjadwal CPU akan melakukan penukaran keluar terhadap proses-proses yang berada di dalam memori sehingga tersedia tempat untuk memasukkan proses yang hendak dieksekusi tersebut. Setelah itu *register* dikembalikan seperti semula dan proses yang diinginkan akan dieksekusi.

Waktu pergantian isi dalam sebuah sistem yang melakukan penukaran pada umumnya cukup tinggi. Untuk mendapatkan gambaran mengenai waktu pergantian isi, akan diilustrasikan sebuah contoh. Misalkan ada sebuah proses sebesar 1 MB, dan media yang digunakan sebagai penyimpanan sementara adalah sebuah *hard disk* dengan kecepatan transfer 5 MBps. Waktu yang dibutuhkan untuk mentransfer proses 1 MB tersebut dari atau ke dalam memori adalah:

1000 KB/5000 KBps = 1/5 detik = 200 milidetik

Apabila diasumsikan *head seek* tidak dibutuhkan dan rata-rata waktu latensi adalah 8 milidetik, satu proses penukaran memakan waktu 208 milidetik. Karena kita harus melakukan proses penukaran sebanyak 2 kali, (memasukkan dan mengeluarkan dari memori), maka keseluruhan waktu yang dibutuhkan adalah 416 milidetik.

Untuk penggunaan CPU yang efisien, kita menginginkan waktu eksekusi kita relatif panjang dibandingkan dengan waktu penukaran. Oleh karena itu, misalnya dalam penjadwalan CPU yang menggunakan metoda round robin, waktu kuantum yang kita tetapkan harus lebih besar dari 416 milidetik. Jika tidak, waktu lebih banyak terbuang pada proses penukaran saja sehingga penggunaan prosesor tidak efisien lagi.

Bagian utama dari waktu penukaran adalah waktu transfer. Besar waktu transfer berhubungan langsung dengan jumlah memori yang di-tukar. Jika kita mempunyai sebuah komputer dengan memori utama 128 MB dan sistem operasi memakan tempat 5 MB, besar proses pengguna maksimal adalah 123 MB. Bagaimana pun juga, proses pengguna pada kenyataannya dapat berukuran jauh lebih kecil dari angka tersebut. Bahkan terkadang hanya berukuran 1 MB. Proses sebesar 1 MB dapat ditukar hanya dalam waktu 208 milidetik, jauh lebih cepat dibandingkan menukar proses sebesar 123 MB yang akan menghabiskan waktu 24.6 detik. Oleh karena itu, sangatlah berguna apabila kita mengetahui dengan baik berapa besar memori yang dipakai oleh proses pengguna, bukan sekedar dengan perkiraan saja. Setelah itu, kita dapat mengurangi besar waktu penukaran dengan cara hanya menukar proses-proses yang benar-benar membutuhkannya. Agar metoda ini bisa dijalankan dengan efektif, pengguna harus menjaga agar sistem selalu memiliki informasi mengenai perubahan kebutuhan memori. Oleh karena itu, proses yang membutuhkan memori dinamis harus melakukan pemanggilan sistem (permintaan memori dan pelepasan memori) untuk memberikan informasi kepada sistem operasi akan perubahan kebutuhan memori.

Penukaran dipengaruhi oleh banyak faktor. Jika kita hendak menukar suatu proses, kita harus yakin bahwa proses tersebut siap. Hal yang perlu diperhatikan adalah kemungkinan proses tersebut sedang menunggu M/K. Apabila M/K secara asinkron mengakses memori pengguna untuk M/K *buffer*, maka proses tersebut tidak dapat ditukar. Bayangkan apabila sebuah operasi M/K berada dalam antrian karena peralatan M/K-nya sedang sibuk. Kemudian kita hendak mengeluarkan proses P1 dan memasukkan proses P2. Operasi M/K mungkin akan berusaha untuk memakai memori yang sekarang seharusnya akan ditempati oleh P2. Cara untuk mengatasi masalah ini adalah:
1. Hindari menukar proses yang sedang menunggu M/K.
2. Lakukan eksekusi operasi M/K hanya di *buffer* sistem operasi.

Hal tersebut akan menjaga agar transfer antara *buffer* sistem operasi dan proses memori hanya terjadi saat si proses ditukar kedalam.

Pada masa sekarang ini, proses penukaran secara dasar hanya digunakan di sedikit sistem. Hal ini dikarenakan penukaran menghabiskan terlalu banyak waktu tukar dan memberikan waktu eksekusi yang terlalu kecil sebagai solusi dari manajemen memori. Akan tetapi, banyak sistem yang menggunakan versi modifikasi dari metoda penukaran ini.

Salah satu sistem operasi yang menggunakan versi modifikasi dari metoda penukaran ini adalah UNIX. Penukaran berada dalam keadaan non-aktif, sampai apabila ada banyak proses yang berjalan yang menggunakan memori yang besar. Penukaran akan berhenti lagi apabila jumlah proses yang berjalan sudah berkurang.

Pada awal pengembangan komputer pribadi, tidak banyak perangkat keras (atau sistem operasi yang memanfaatkan perangkat keras) yang dapat mengimplementasikan memori manajemen yang baik, melainkan digunakan untuk menjalankan banyak proses berukuran besar dengan menggunakan versi modifikasi dari metoda penukaran. Salah satu contoh yang baik adalah Microsoft Windows 3.1, yang mendukung eksekusi proses berkesinambungan. Apabila suatu proses baru hendak dijalankan dan tidak terdapat cukup memori, proses yang lama perlu dimasukkan ke dalam *disk*. Sistem operasi ini, bagaimana pun juga, tidak mendukung penukaran secara keseluruhan karena yang lebih berperan menentukan kapan proses penukaran akan dilakukan adalah pengguna dan bukan penjadwal CPU. Proses-proses yang sudah dikeluarkan akan tetap berada di luar memori sampai pengguna memilih proses yang hendak dijalankan. Sistem-sistem operasi Microsoft selanjutnya, seperti misalnya Windows NT, memanfaatkan fitur Unit Manajemen Memori.

29.2. Proteksi Memori

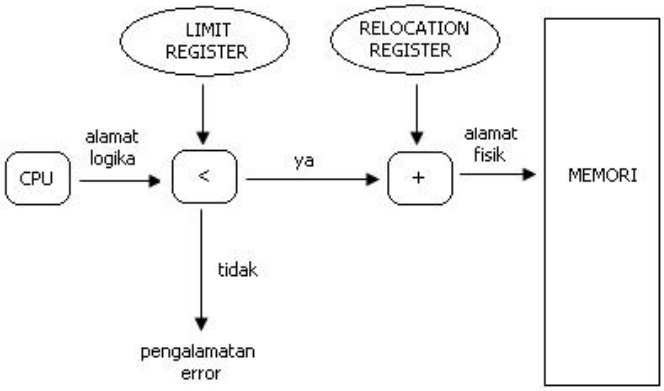
Proteksi memori adalah sebuah sistem yang mencegah sebuah proses dari pengambilan memori proses lain yang sedang berjalan pada komputer yang sama dan pada saat yang sama pula. Proteksi memori selalu mempekerjakan hardware (Memori Manajemen Unit/MMU) dan sistem software untuk mengalokasikan memori yang berbeda untuk proses yang berbeda dan untuk mengatasi exception yang muncul ketika sebuah proses mencoba untuk mengakses memori di luar batas.

Proteksi memori dapat menggunakan Relocation Register dengan Limit Register. Relocation Register berisi nilai terkecil alamat fisik. Limit Register berisi rentang nilai alamat logika. Dengan Relokasi dan Limit Register, tiap alamat logika harus lebih kecil dari Limit Register. MMU

memetakan alamat logika secara dinamis dengan menambahkan nilai di Relocation Register. Alamat pemetaan ini kemudian dikirimkan ke memori.

Untuk lebih jelasnya, dapat dilihat pada gambar di bawah ini:

Gambar 29.1. Proteksi Memori



Efektivitas dari proteksi memori berbeda antara sistem operasi yang satu dengan yang lainnya. Ada beberapa cara yang berbeda untuk mencapai proteksi memori. Segmentasi dan pemberian halaman adalah dua metoda yang paling umum digunakan.

Segmentasi adalah skema manajemen memori dengan cara membagi memori menjadi segmen-segmen. Dengan demikian, sebuah program dibagi menjadi segmen-segmen. Segmen adalah sebuah unit logis, yaitu unit yang terdiri dari beberapa bagian yang berjenis yang sama. Segmen dapat terbagi jika terdapat elemen di tabel segmen yang berasal dari dua proses yang berbeda yang menunjuk pada alamat fisik yang sama. Saling berbagi ini muncul di level segmen dan pada saat ini terjadi semua informasi dapat turut terbagi. Proteksi dapat terjadi karena ada bit proteksi yang berhubungan dengan setiap elemen dari segmen tabel. Bit proteksi ini berguna untuk mencegah akses ilegal ke memori. Caranya menempatkan sebuah array di dalam segmen itu sehingga perangkat keras manajemen memori secara otomatis akan memeriksa indeks arraynya legal atau tidak.

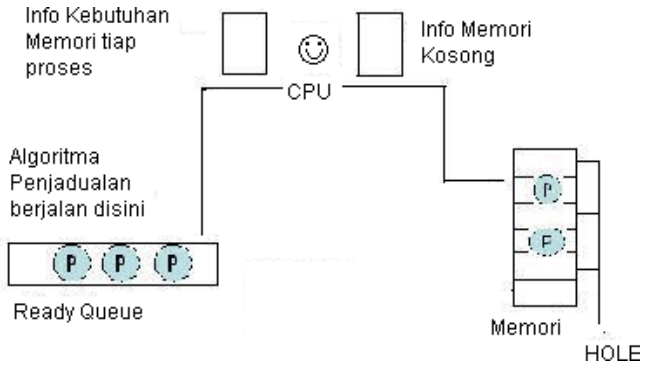
Pemberian halaman merupakan metoda yang paling sering digunakan untuk proteksi memori. Pemberian halaman adalah suatu metoda yang memungkinkan suatu alamat fisik memori yang tersedia dapat tidak berurutan. Proteksi memori di lingkungan halaman bisa dilakukan dengan cara memproteksi bit-bit yang berhubungan dengan setiap bingkai. Biasanya bit-bit ini disimpan di dalam sebuah tabel halaman. Satu bit bisa didefinisikan sebagai baca-tulis atau hanya baca saja. Setiap referensi ke memori menggunakan tabel halaman untuk menemukan nomor bingkai yang benar. Pada saat alamat fisik sedang dihitung, bit proteksi bisa memeriksa bahwa kita tidak bisa menulis ke mode tulis saja. Untuk lebih jelasnya, kedua metoda ini akan dijelaskan pada bab berikutnya.

29.3. Alokasi Memori Berkesinambungan

Memori utama harus dapat melayani baik sistem operasi maupun proses pengguna. Oleh karena itu kita harus mengalokasikan pembagian memori seefisien mungkin. Salah satunya adalah dengan cara **alokasi memori berkesinambungan**. Alokasi memori berkesinambungan berarti alamat memori diberikan kepada proses secara berurutan dari kecil ke besar. Keuntungan menggunakan alokasi memori berkesinambungan dibandingkan menggunakan alokasi memori tidak berkesinambungan adalah:

1. Sederhana
2. Cepat
3. Mendukung proteksi memori

Gambar 29.2. Proses Dalam Memori



Sedangkan kerugian dari menggunakan alokasi memori berkesinambungan adalah apabila tidak semua proses dialokasikan di waktu yang sama, akan menjadi sangat tidak efektif sehingga mempercepat habisnya memori.

Alokasi memori berkesinambungan dapat dilakukan baik menggunakan sistem partisi banyak, maupun menggunakan sistem partisi tunggal. Sistem partisi tunggal berarti alamat memori yang akan dialokasikan untuk proses adalah alamat memori pertama setelah pengalokasian sebelumnya. Sedangkan sistem partisi banyak berarti sistem operasi menyimpan informasi tentang semua bagian memori yang tersedia untuk dapat diisi oleh proses-proses (disebut lubang). Sistem partisi banyak kemudian dibagi lagi menjadi sistem partisi banyak tetap, dan sistem partisi banyak dinamis. Hal yang membedakan keduanya adalah untuk sistem partisi banyak tetap, memori dipartisi menjadi blok-blok yang ukurannya tetap yang ditentukan dari awal. Sedangkan sistem partisi banyak dinamis artinya memori dipartisi menjadi bagian-bagian dengan jumlah dan besar yang tidak tentu. Untuk selanjutnya, kita akan memfokuskan pembahasan pada sistem partisi banyak.

Sistem operasi menyimpan sebuah tabel yang menunjukkan bagian mana dari memori yang memungkinkan untuk menyimpan proses, dan bagian mana yang sudah diisi. Pada intinya, seluruh memori dapat diisi oleh proses pengguna. Saat sebuah proses datang dan membutuhkan memori, CPU akan mencari lubang yang cukup besar untuk menampung proses tersebut. Setelah menemukannya, CPU akan mengalokasikan memori sebanyak yang dibutuhkan oleh proses tersebut, dan mempersiapkan sisanya untuk menampung proses-proses yang akan datang kemudian (seandainya ada).

Saat proses memasuki sistem, proses akan dimasukkan ke dalam antrian masukan. Sistem operasi akan menyimpan besar memori yang dibutuhkan oleh setiap proses dan jumlah memori kosong yang tersedia, untuk menentukan proses mana yang dapat diberikan alokasi memori. Setelah sebuah proses mendapat alokasi memori, proses tersebut akan dimasukkan ke dalam memori. Setelah proses tersebut dimatikan, proses tersebut akan melepas memori tempat dia berada, yang mana dapat diisi kembali oleh proses lain dari antrian masukan.

Sistem operasi setiap saat selalu memiliki catatan jumlah memori yang tersedia dan antrian masukan. Sistem operasi dapat mengatur antrian masukan berdasarkan algoritma penjadwalan yang digunakan. Memori dialokasikan untuk proses sampai akhirnya kebutuhan memori dari proses selanjutnya tidak dapat dipenuhi (tidak ada lubang yang cukup besar untuk menampung proses tersebut). Sistem operasi kemudian dapat menunggu sampai ada blok memori cukup besar yang kosong, atau dapat mencari proses lain di antrian masukan yang kebutuhan memorinya memenuhi jumlah memori yang tersedia.

Pada umumnya, kumpulan lubang-lubang dalam berbagai ukuran tersebar di seluruh memori sepanjang waktu. Apabila ada proses yang datang, sistem operasi akan mencari lubang yang cukup besar untuk menampung memori tersebut. Apabila lubang yang tersedia terlalu besar, akan dipecah menjadi 2. Satu bagian akan dialokasikan untuk menerima proses tersebut, sementara bagian lainnya tidak digunakan dan siap menampung proses lain. Setelah proses selesai, proses tersebut akan melepas memori dan mengembalikannya sebagai lubang-lubang. Apabila ada dua lubang yang kecil yang berdekatan, keduanya akan bergabung untuk membentuk lubang yang lebih besar. Pada saat

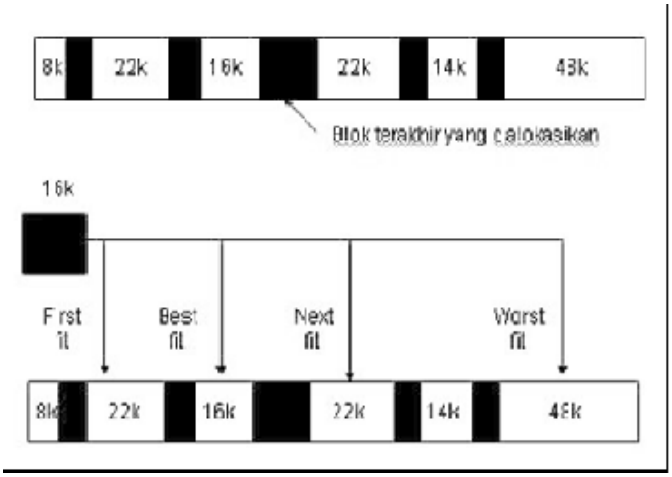
ini, sistem harus memeriksa apakah ada proses yang menunggu yang dapat dimasukkan ke dalam ruang memori yang baru terbentuk tersebut.

Hal ini disebut **Permasalahan alokasi penyimpanan dinamis**, yakni bagaimana memenuhi permintaan sebesar *n* dari kumpulan lubang-lubang yang tersedia. Ada berbagai solusi untuk mengatasi hal ini, yaitu:

1. *First fit*: Mengalokasikan lubang pertama ditemukan yang besarnya mencukupi. Pencarian dimulai dari awal.
2. *Best fit*: Mengalokasikan lubang dengan besar minimum yang mencukupi permintaan.
3. *Next fit*: Mengalokasikan lubang pertama ditemukan yang besarnya mencukupi. Pencarian dimulai dari akhir pencarian sebelumnya.
4. *Worst fit*: Mengalokasikan lubang terbesar yang ada.

Memilih yang terbaik diantara keempat metoda diatas adalah sepenuhnya tergantung kepada pengguna, karena setiap metoda memiliki kelebihan dan kekurangan masing-masing. Menggunakan *best fit* dan *worst fit* berarti kita harus selalu memulai pencarian lubang dari awal, kecuali apabila lubang sudah disusun berdasarkan ukuran. Metode *worst fit* akan menghasilkan sisa lubang yang terbesar, sementara metoda *best fit* akan menghasilkan sisa lubang yang terkecil.

Gambar 29.3. Permasalahan alokasi penyimpanan dinamis



29.4. Fragmentasi

Fragmentasi adalah munculnya lubang-lubang yang tidak cukup besar untuk menampung permintaan dari proses. Fragmentasi dapat berupa fragmentasi internal maupun fragmentasi eksternal.

Fragmentasi ekstern muncul apabila jumlah keseluruhan memori kosong yang tersedia memang mencukupi untuk menampung permintaan tempat dari proses, tetapi letaknya tidak berkesinambungan atau terpecah menjadi beberapa bagian kecil sehingga proses tidak dapat masuk. Umumnya, ini terjadi ketika kita menggunakan sistem partisi banyak dinamis. Pada sistem partisi banyak dinamis, seperti yang diungkapkan sebelumnya, sistem terbagi menjadi blok-blok yang besarnya tidak tetap. Maksud tidak tetap di sini adalah blok tersebut bisa bertambah besar atau bertambah kecil. Misalnya, sebuah proses meminta ruang memori sebesar 17 KB, sedangkan memori dipartisi menjadi blok-blok yang besarnya masing-masing 5 KB. Maka, yang akan diberikan pada proses adalah 3 blok ditambah 2 KB dari sebuah blok. Sisa blok yang besarnya 3 KB akan disiapkan untuk menampung proses lain atau jika ia bertetangga dengan ruang memori yang kosong, ia akan bergabung dengannya. Akibatnya dengan sistem partisi banyak dinamis, bisa tercipta lubang-lubang di memori, yaitu ruang memori yang kosong. Keadaan saat lubang-lubang ini tersebar yang masing-masing lubang tersebut tidak ada yang bisa memenuhi kebutuhan proses padahal jumlah dari besarnya lubang tersebut cukup untuk memenuhi kebutuhan proses disebut sebagai fragmentasi ekstern.

Fragmentasi intern muncul apabila jumlah memori yang diberikan oleh penjadwal CPU untuk ditempati proses lebih besar daripada yang diminta proses karena adanya selisih antara permintaan proses dengan alokasi lubang yang sudah ditetapkan. Hal ini umumnya terjadi ketika kita menggunakan sistem partisi banyak tetap. Kembali ke contoh sebelumnya, di mana ada proses dengan permintaan memori sebesar 17 KB dan memori dipartisi menjadi blok yang masing-masing besarnya 5 KB. Pada sistem partisi banyak tetap, memori yang dialokasikan untuk proses adalah 4 blok, atau sebesar 20 KB. Padahal, yang terpakai hanya 17 KB. Sisa 3 KB tetap diberikan pada proses tersebut, walaupun tidak dipakai oleh proses tersebut. Hal ini berarti pula proses lain tidak dapat memakainya. Perbedaan memori yang dialokasikan dengan yang diminta inilah yang disebut fragmentasi intern.

Algoritma alokasi penyimpanan dinamis mana pun yang digunakan, tetap tidak bisa menutup kemungkinan terjadinya fragmentasi. Bahkan hal ini bisa menjadi fatal. Salah satu kondisi terburuk adalah apabila kita memiliki memori terbuang setiap dua proses. Apabila semua memori terbuang itu digabungkan, bukan tidak mungkin akan cukup untuk menampung sebuah proses. Sebuah contoh statistik menunjukkan bahwa saat menggunakan metoda *first fit*, bahkan setelah dioptimisasi, dari N blok teralokasi, sebanyak 0.5N blok lain akan terbuang karena fragmentasi. Jumlah sebanyak itu berarti kurang lebih setengah dari memori tidak dapat digunakan. Hal ini disebut dengan **aturan 50%**.

Fragmentasi ekstern dapat diatasi dengan beberapa cara, diantaranya adalah:
1. **Pemadatan**, yaitu mengatur kembali isi memori agar memori yang kosong diletakkan bersama di suatu bagian yang besar sehingga proses dapat masuk ke ruang memori kosong tersebut. Pemadatan hanya dapat dilakukan bila pengalamatan program dilakukan pada saat eksekusi. Kebutuhan akan pemadatan akan hilang bila pengalamatan dapat dilakukan secara berurutan, walaupun sebenarnya proses tidak ditempatkan pada lokasi memori yang berurutan. Nantinya, konsep ini diterapkan dengan Paging.
2. **Penghalamanan**.
3. **Segmentasi**.

Fragmentasi internal tidak dapat dihindarkan apabila kita menggunakan sistem partisi banyak berukuran tetap, mengingat besar hole yangdisediakan selalu tetap, kecuali jika kita menggunakan sistem partisi banyak dinamis, yang memungkinkan suatu proses untuk diberikan ruang memori sebesar yang dia minta.

29.5. Rangkuman

Sebuah proses dapat di-swap sementara keluar memori ke sebuah backing store untuk kemudian dibawa masuk lagi ke memori untuk melanjutkan pengeksekusian. Salah satu proses yang memanfaatkan metoda ini adalah roll out, roll in, yang pada intinya adalah proses swapping berdasarkan prioritas.

Agar main memori dapat melayani sistem operasi dan proses dengan baik, dibutuhkan pembagian memori seefisien mungkin. Salah satunya adalah dengan contiguous memori allocation. Artinya alamat memori diberikan secara berurutan dari kecil ke besar. Ruang memori yang masih kosong dan dapat dialokasikan untuk proses disebut hole. Hole ini menciptakan permasalahan alokasi memori dinamis, yaitu bagaimana memenuhi kebutuhan memori suatu proses dari hole yang ada. Metoda untuk menyelesaikan ini diantaranya adalah first fit, next fit, best fit, dan worst fit. Masalah yang sering muncul dalam pengalamatan memori adalah fragmentasi (baik intern maupun ekstern), yaitu munculnya hole-hole yang tidak cukup besar untuk menampung permintaan dari proses.

Rujukan

[SariYansen2005] Riri Fitri Sari dan Yansen. 2005. *Sistem Operasi Modern* . Edisi Pertama. Andi. Yogyakarta.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBAmirSch2000] Yair Amir dan Theo Schlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

Bab 30. Pemberian Halaman

30.1. Pendahuluan

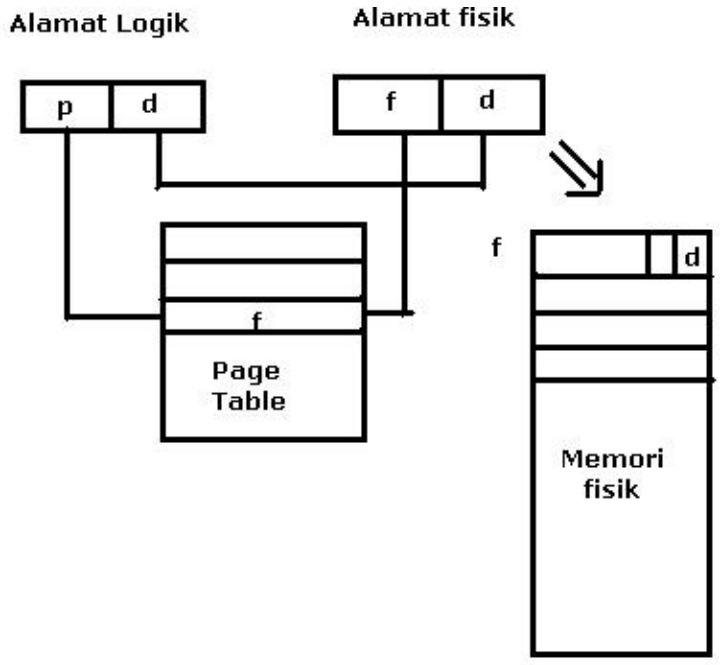
Yang dimaksud dengan pemberian halaman adalah suatu metoda yang memungkinkan suatu alamat fisik memori yang tersedia dapat tidak berurutan. Pemberian halaman bisa menjadi solusi untuk pemecahan masalah luar. Untuk bisa mengimplementasikan solusi ini adalah melalui pengunaan dari skema pemberian halaman. Dengan pemberian halaman bisa mencegah masalah penting dari pengepasan besar ukuran memori yang bervariasi kedalam penyimpanan cadangan. Ketika beberapa pecahan kode dari data yang tersisa di memori utama perlu untuk ditukar keluar, harus ditemukan ruang untuk penyimpanan cadangan. Masalah pemecahan kode didiskusikan dengan kaitan bahwa pengaksesannya lebih lambat. Biasanya bagian yang menunjang untuk pemberian halaman telah ditangani oleh perangkat keras. Bagaimana pun, desain yang ada baru-baru ini telah mengimplementasikan dengan menggabungkan perangkat keras dan sistem operasi, terutama pada prosesormikro 64 bit .

30.2. Metoda Dasar

Jadi metoda dasar yang digunakan adalah dengan memecah memori fisik menjadi blok-blok berukuran tetap yang akan disebut sebagai frame. selanjutnya memori logis juga dipecah menjadi blok-blok dengan ukuran yang sama disebut sebagai halaman. Selanjutnya kita membuat suatu tabel halaman yang akan menterjemahkan memori logis kita kedalam memori fisik. Jika suatu proses ingin dieksekusi maka memori logis akan melihat dimanakah dia akan ditempatkan di memori fisik dengan melihat kedalam tabel halamannya.

Untuk jelasnya bisa dilihat pada Gambar 30.1, “Penerjemahan Halaman”. Kita lihat bahwa setiap alamat yang dihasilkan oleh CPU dibagi-bagi menjadi dua bagian yaitu sebuah nomor halaman (p) dan sebuah offset halaman (d). Nomor halaman ini akan digunakan sebagai indeks untuk tabel halaman. Tabel halaman mengandung basis alamat dari tiap-tiap halaman di memori fisik. Basis ini dikombinasikan dengan offset halaman untuk menentukan alamat memori fisik yang dikirim ke unit memori.

Gambar 30.1. Penerjemahan Halaman



Memori fisik dipecah menjadi beberapa blok berukuran yang tetap yang disebut frame (bingkai), sedangkan memori logis juga dipecah dengan ukuran yang sama yang disebut halaman. Suatu alamat memori yang digenerate oleh CPU terdiri dari 2 bagian yaitu halaman dan offset. Halaman berfungsi sebagai indeks dari suatu halaman table. Isi dari indeks yang ditunjuk pada halaman table (frame) digabungkan dengan offset maka akan membentuk suatu alamat asli dari suatu data pada memori fisik. Offset sendiri berfungsi sebagai penunjuk dari suatu blok data yang berada dalam suatu frame.

30.3. Dukungan Perangkat Keras

Setiap sistem operasi mempunyai caranya tersendiri untuk menyimpan tabel halaman. Biasanya sistem operasi mengalokasikan sebuah tabel halaman untuk setiap proses. sebuah penunjuk ke tabel halaman disimpan dengan nilai register yang lain didalam blok pengontrol proses. Akan tetapi penggunaannya menjadi tidak praktis karena ternyata tabel halaman disimpan pada memori utama yang tentu saja kecepatannya jauh lebih lambat dari dari register.

Translation Lookaside Buffers (TLBs) dibuat untuk mengatasi masalah tersebut. TLBs adalah suatu asosiatif memori berkecepatan tinggi yang berfungsi hampir sama seperti cache memori tapi terjadi pada tabel halaman, TLBs menyimpan sebagian alamat-alamat data dari suatu proses yang berada pada tabel halaman yang sedang digunakan atau sering digunakan. TLBs biasanya terletak pada Memori Manajement Unit (MMU).

Salah satu feature dari TLBs adalah mampu membuat proteksi suatu alamat memori. Feature ini dinamakan address-space identifiers (ASIDs). ASIDs ini membuat suatu alamat memori hanya ada 1 proses yang bisa mengaksesnya. Contohnya adalah JVM. Pada saat program Java berjalan, Java membuat alokasi alamat memori untuk JVM dan yang bisa mengakses alamat tersebut hanya program Java yang sedang berjalan itu saja.

Suatu keadaan dimana pencarian alamat memori berhasil ditemukan pada TLBs disebut TLB hit. Sedangkan sebaliknya jika terjadi pada tabel halaman (dalam hal ini TLB gagal) disebut TLB miss. Effective Address Time adalah waktu yang dibutuhkan untuk mengambil data dalam memori fisik dengan persentase TLB hit (hit ratio) dan TLB miss (miss ratio).

Persentasi dari beberapa kali TLB *hit* adalah disebut *hit ratio*. *hit ratio* 80% berarti menemukan nomor halaman yang ingin kita cari didalam TLB sebesar 80%. Jika waktu akses ke TLB memakan waktu 20 nanodetik dan akses ke memori memakan waktu sebesar 100 nanodetik maka total waktu kita memetakan memori adalah 120 nanodetik jika TLB *hit*. dan jika TLB *miss* maka total waktunya adalah 220 nanodetik. Jadi untuk mendapatkan waktu akses memori yang efektif maka kita harus membagi-bagi tiap kasus berdasarkan kemungkinannya:

```
Effective address time=
hit ratio * (time search TLB + time access memori) +
miss ratio * (time search TLB + time access tabel halaman +
time access memori)
```

Suatu mesin yang efektif apabila memiliki effective address time yang kecil. Banyak cara yang telah dilakukan untuk optimal, salah satunya dengan mereduksi TLBs miss. Jadi sistem operasi memiliki intuisi untuk memprediksi halaman selanjutnya yang akan digunakan kemudian me-loadnya ke dalam TLB.

30.4. Proteksi Memori

Berbeda dengan poteksi memori pada bab sebelumnya, yang menggunakan Limit Register dan Relokasi Register, proteksi memori di lingkungan halaman dikerjakan oleh bit-bit proteksi yang berhubungan dengan tiap frame. Biasanya bit-bit ini disimpan didalam sebuah tabel halaman. Satu bit bisa didefinisikan sebagai baca-tulis atau hanya baca saja suatu halaman. Setiap referensi ke memori menggunakan tabel halaman untuk menemukan nomor frame yang benar. Pada saat alamat fisik sedang dihitung, bit proteksi bisa dicek untuk memastikan tidak ada kegiatan tulis-menulis ke

dalam 'read-only halaman'. Hal ini diperlukan untuk menghindari tejadinya 'memori-protection violation', suatu keadaan dimana terjadi suatu percobaan menulis di 'halaman *read-only*'.

Ketika bit proteksi bernilai 'valid', berarti halaman yang dicari ada di dalam ruang alamat logika. Yang artinya halaman tersebut dapat diakses. Jika bit proteksi bernilai 'invalid', artinya halaman yang dimaksud tidak berada dalam ruang alamat logika. Sehingga halaman tersebut tidak dapat di akses.

30.5. Untung/Rugi Pemberian Halaman

- Jika kita membuat ukuran dari masing-masing halaman menjadi lebih besar.
  - **Keuntungan.** Akses memori akan relatif lebih cepat.
  - **Kerugian.** Kemungkinan terjadinya fragmentasi intern sangat besar.
- Jika kita membuat ukuran dari masing-masing halaman menjadi lebih kecil.
  - **Keuntungan.** Kemungkinan terjadinya internal Framentasi akan menjadi lebih kecil.
  - **Kerugian.** Akses memori akan relatif lebih lambat.

Keuntungan lainnya dari paging adalah, konsep memori virtual bisa diterapkan dengan menuliskan halaman ke disk, dan pembacaan halaman dari disk ketika dibutuhkan. Hal ini dikarenakan jarangnya pengunaan kode-kode dan data suatu program secara keseluruhan pada suatu waktu.

Kerugian lainnya dari paging adalah, paging tidak bisa diterapkan untuk beberapa prosesor tua atau kecil (dalam keluarga Intel x86, sebagai contoh, hanya 80386 dan di atasnya yang punya MMU, yang bisa diterapkan *paging*). Hal ini dikarenakan paging membutuhkan MMU (*Memory Management Unit*).

30.6. Tabel Halaman

Sebagian besar komputer modern memiliki perangkat keras istimewa yaitu **unit manajemen memori** (MMU). Unit tersebut berada diantara CPU dan unit memori. Jika CPU ingin mengakses memori (misalnya untuk memanggil suatu instruksi atau memanggil dan menyimpan suatu data), maka CPU mengirimkan alamat memori yang bersangkutan ke MMU, yang akan menerjemahkannya ke alamat lain sebelum melanjutkannya ke unit memori. Alamat yang dihasilkan oleh CPU, setelah adanya pemberian indeks atau aritmatik ragam pengalamatan lainnya disebut **alamat logis (*virtual address*)**. Sedangkan alamat yang didapatkan fisik membuat sistem operasi lebih mudah pekerjaannya saat mengalokasikan memori. Lebih penting lagi, MMU juga mengizinkan halaman yang tidak sering digunakan bisa disimpan di *disk*. Cara kerjanya adalah sbb: Tabel yang digunakan oleh MMU mempunyai bit sahih untuk setiap halaman di bagian alamat logis. Jika bit tersebut di set, maka penterjemahan oleh alamat logis di halaman itu berjalan normal. Akan tetapi jika dihapus, adanya usaha dari CPU untuk mengakses suatu alamat di halaman tersebut menghasilkan suatu interupsi yang disebut *page fault trap*. Sistem operasi telah mempunyai *interrupt handler* untuk kesalahan halaman, juga bisa digunakan untuk mengatasi interupsi jenis yang lain. *Handler* inilah yang akan bekerja untuk mendapatkan halaman yang diminta ke memori.

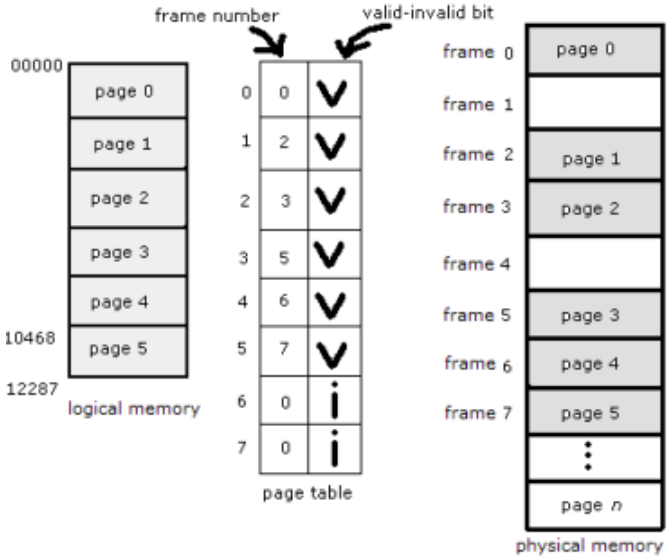
Untuk lebih jelasnya, saat kesalahan halaman dihasilkan untuk halaman ***p1***, ***interrupt handler*** melakukan hal-hal berikut ini:

- Mencari dimana isi dari halaman ***p1*** disimpan di *disk*. Sistem operasi menyimpan informasi ini di dalam tabel. Ada kemungkinan bahwa halaman tersebut tidak ada dimana-mana, misalnya pada kasus saat referensi memori adalah *bug*. Pada kasus tersebut , sistem operasi mengambil beberapa langkah kerja seperti mematikan prosesnya. Dan jika diasumsikan halamannya berada dalam *disk*:
- Mencari halaman lain yaitu ***p2*** yang dipetakan ke *frame* lain ***f*** dari alamat fisik yang tidak banyak dipergunakan.
- Menyalin isi dari *frame* ***f*** keluar dari *disk*.
- Menghapus bit sahih dari halaman ***p2*** sehingga sebagian referensi dari halaman ***p2*** akan menyebabkan kesalahan halaman.
- Menyalin data halaman ***p1*** dari *disk* ke *frame* ***f***.
- *Update* tabel MMU sehingga halaman ***p1*** dipetakan ke *frame* ***f***.
- Kembali dari interupsi dan mengizinkan CPU mengulang instruksi yang menyebabkan interupsi tersebut.

Pada dasarnya MMU terdiri dari tabel halaman yang merupakan sebuah rangkaian *array* dari

masukan-masukan (*entries*) yang mempunyai indeks berupa nomor halaman (*p*). Setiap masukan terdiri dari *flags* (contohnya bit sahih dan nomor *frame*). Alamat fisik dibentuk dengan menggabungkan nomor *frame* dengan ofset, yaitu bit paling rendah dari alamat logis.

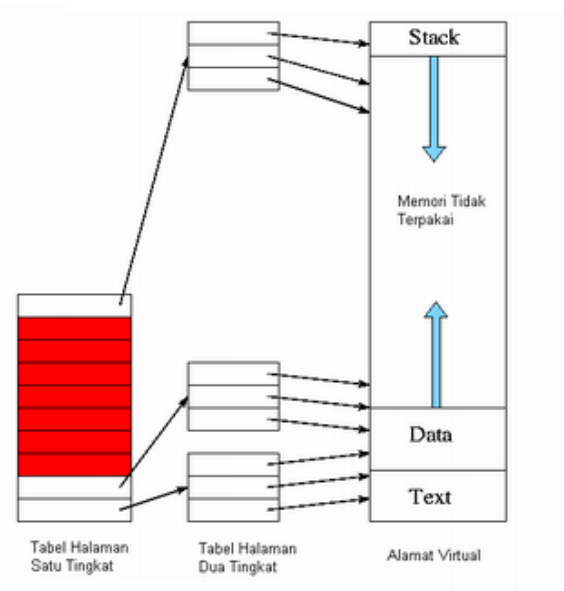
Gambar 30.2. Skema Tabel Halaman Dua tingkat



Setiap sistem operasi mempunyai metodanya sendiri untuk menyimpan tabel halaman. Sebagian besar mengalokasikan tabel halaman untuk setiap proses. Penunjuk ke tabel halaman disimpan dengan nilai register yang lain (seperti pencacah instruksi) di blok kontrol proses. Ketika pelaksana *dispatcher* mengatakan untuk memulai proses, maka harus disimpan kembali register-register pengguna dan mendefinisikan nilai tabel halaman perangkat keras yang benar dari tempat penyimpanan tabel halaman dari pengguna.

30.7. Pemberian Halaman Secara Bertingkat

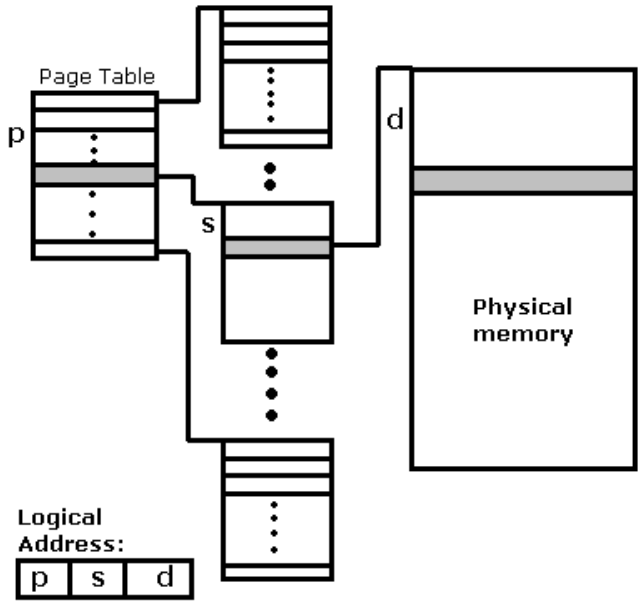
Gambar 30.3. Tabel Halaman secara Bertingkat



Hierarchical paging atau pemberian halaman bertingkat merupakan metoda pemberian halaman secara maju (forward mapped paging). Pemberian halaman dengan cara ini menggunakan pembagian tingkat setiap segmen alamat logikal. Setiap segmen menunjukkan indeks dari tabel halaman, kecuali segmen terakhir yang menunjukkan langsung frame pada memori fisik. Segmen terakhir ini biasa disebut offset(D). Dapat disimpulkan bahwa segmen yang terdapat dalam alamat lojik menentukan berapa lapis paging yang digunakan yaitu banyak segmen-1. Selain itu dalam sistem ini setiap tabel halaman memiliki page table lapis kedua yang berbeda.

Dengan metoda ini, isi pada indeks tabel halaman pertama akan menunjuk pada tabel halaman kedua yang bersesuaian dengan isi dari tabel halaman pertama tersebut. Sedangkan isi dari page table kedua menunjukkan tempat di mana tabel halaman ketiga bermula, sedang segmen alamat lojik kedua adalah indeks ke-n setelah starting point tabel halaman ketiga dan seterusnya sampai dengan segmen terakhir. Sedangkan segmen terakhir menunjukkan langsung indeks setelah alamat yang ditunjukkan oleh tabel halaman terakhir.

Gambar 30.4. ***Hierarchical Paging***

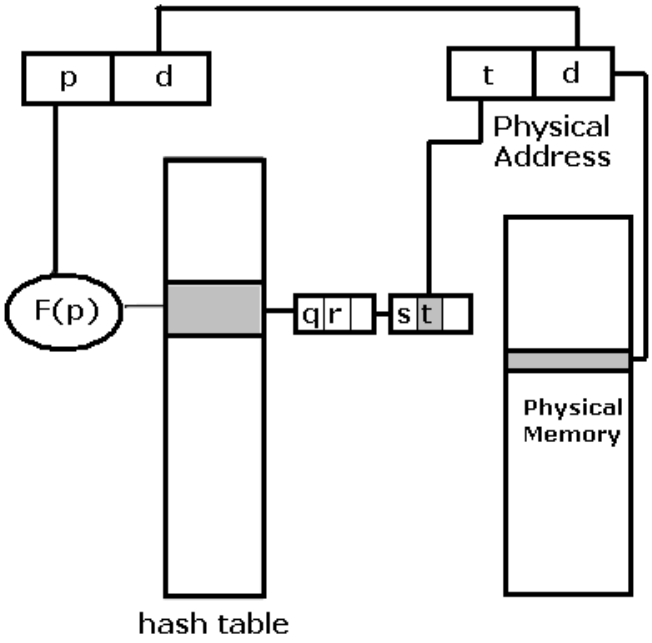


Kekurangan dari metoda ini adalah borosnya memori karena setiap tabel halaman menunjuk pada page table lainnya yang berbeda. Jika segmen pertama berisi p bit, maka besar tabel halaman pertama sebesar 2 pangkat p. Bila segmen kedua berisi s bit, maka setiap tabel halaman pertama menunjuk pada 2 pangkat s banyaknya ruang pada memori. Sehingga sampai dengan tingkat dua ini ruang yang dibutuhkan untuk paging sudah mencapai 2 pangkat s+p. Dapat dikatakan bahwa metoda ini sangat tidak cocok untuk diterapkan pada mapping besar seperti 64-bit walaupun dapat saja dilakukan.

30.8. Tabel Halaman secara *Hashed*

Tabel Halaman secara Hashed cukup cocok untuk paging berukuran besar, seperti 64-bit. Karakteristik dari metoda ini adalah digunakannya sebuah fungsi untuk memanipulasi alamat lojik. Selain sebuah fungsi, tabel halaman secara hashed juga menggunakan tabel hash dan juga pointer untuk menangani linked list. Hasil dari hashing akan dipetakan pada hash tabel halaman yang berisi linked list. Penggunaan linked list adalah untuk pengacakan data yang dikarenakan besar hash table yang sangat terbatas. Pada metoda ini offset masih sangat berperan untuk menunjukkan alamat fisik, yaitu dengan meng-concate isi dari linked list dengan offset tersebut. Sistem ini dapat dikembangkan menjadi Clustered Tabel Halaman yang lebih acak dalam pengalamatan dalam memori fisik.

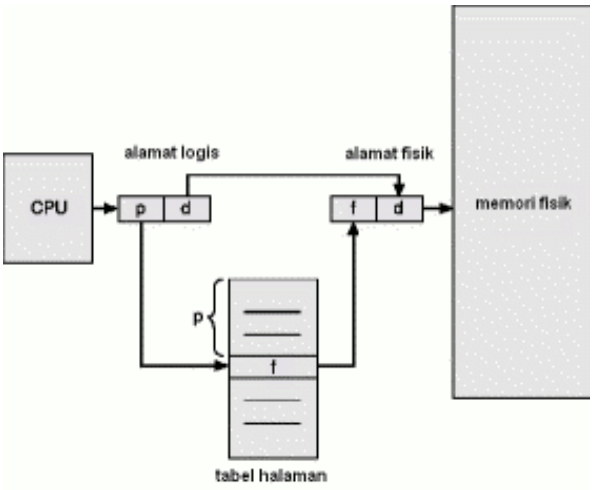
Gambar 30.5. Hashed Page Tabels



30.9. Tabel Halaman secara *Inverted*

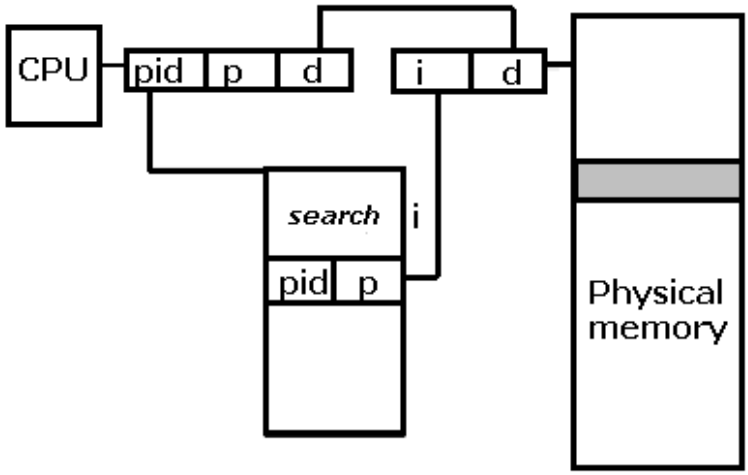
Metoda ini berbeda dengan metode lainnya. Pada Tabel Halaman Inverted, proses pemberian halaman dipusatkan pada proses yang sedang ditangani. Alamat lojik yang menggunakan inverted tabel halaman merepresentasikan proses yang dimiliki. Sehingga tabel halaman pada metoda ini sama besar atau lebih dengan jumlah proses yang dapat ditangani dalam setiap kesempatan. Bila diasumsikan sistem dapat menangani n buah proses maka paling tidak tabel halaman juga sebesar n, sehingga setidaknya satu proses memiliki satu halaman yang bersesuaian dengan page tersebut. Metoda inverted ini bertumpu pada proses pencarian identitas dari proses di dalam tabel halaman tersebut. Jika proses sudah dapat ditemukan di dalam tabel halaman maka letak indeks di tabel halaman yang dikirimkan dan dikonketenasi dengan offset sehingga membentuk alamat fisik yang baru. Karena Inverted paging membatasi diri pada banyaknya proses maka jika dibandingkan dengan hierarchical paging metoda ini membutuhkan memori yang lebih sedikit.

Gambar 30.6. Tabel Halaman secara *Inverted*



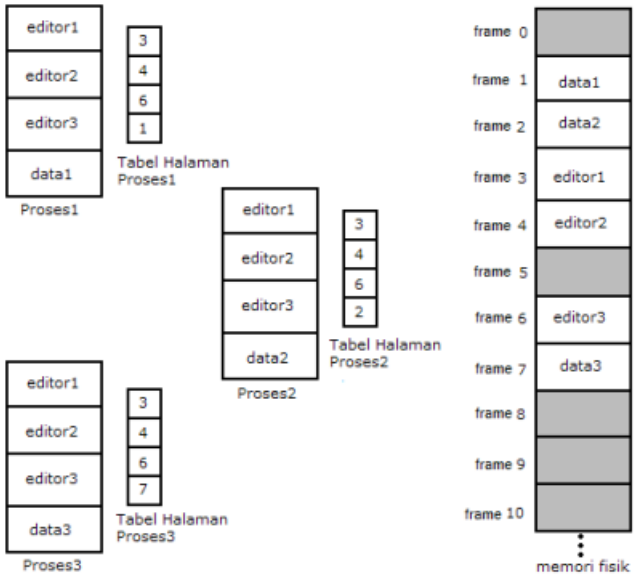
Kekurangan pertama dari inverted paging ini disebabkan oleh fasilitas searching yang dimilikinya. Memori fisik sendiri tersusun secara berurutan di dalam tabel halaman, namun proses yang ingin dicari berasal dari alamat virtual sehingga ada kemungkinan dilakukan pencarian seluruh tabel halaman sampai akhirnya halaman tersebut ditemukan. Hal ini menyebabkan ketidakstabilan metoda ini sendiri. Kekurangan lain dari inverted tabel halaman adalah sulitnya menerapkan memori berbagi (shared). Memori sharing adalah sebuah cara dimana proses yang berbeda dapat mengakses suatu alamat di memori yang sama. Ini bertentangan dengan konsep dari inverted tabel halaman sendiri yaitu, setiap proses memiliki satu atau lebih frame di memori, dengan pasangan proses dan frame unik. Unik dalam arti beberapa frame hanya dapat diakses oleh satu buah proses saja. Karena perbedaan konsep yang sedemikian jauh tersebut maka memori sharing hampir mustahil diterapkan dengan inverted tabel halaman.

Gambar 30.7. *Inverted Page Tables*



30.10. Berbagi Halaman (*Share*)

Gambar 30.8. Berbagi Halaman

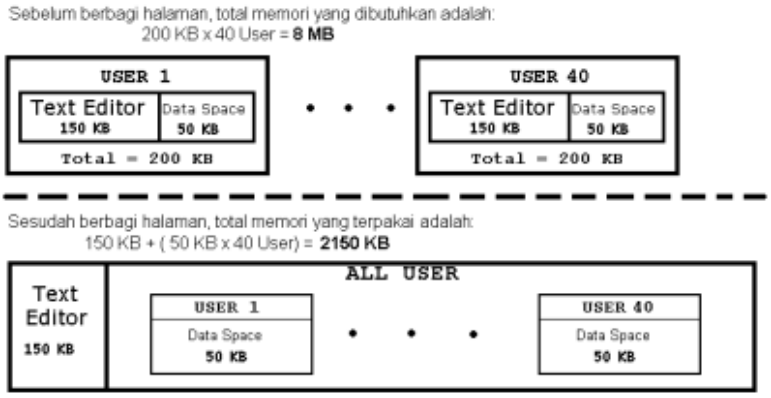


Berbagi kode di lingkungan halaman

Keuntungan lain dari pemberian halaman adalah kemungkinannya untuk berbagi kode yang sama. Pertimbangan ini terutama sekali penting pada lingkungan yang berbagi waktu. Pertimbangkan sebuah sistem yang mendukung 40 pengguna, yang masing-masing menjalankan aplikasi pengedit teks. Jika editor teks tadi terdiri atas 150K kode dan 50K ruang data, kita akan membutuhkan 8000K untuk mendukung 40 pengguna. Jika kodenya dimasukan ulang, bagaimana pun juga dapat dibagi-bagi, seperti pada gambar. Disini kita lihat bahwa tiga halaman editor (masing-masing berukuran 50K; halaman ukuran besar digunakan untuk menyederhanakan gambar) sedang dibagi-bagi diantara tiga proses. Masing-masing proses mempunyai halaman datanya sendiri.

Terlihat jelas selisih penggunaan memori sesudah dan sebelum berbagi halaman adalah sebesar 5850MB. Bingkai yang berisi editor diakses oleh banyak pengguna. Jadi hanya ada satu salinan editor yang ditaruh di memori. Tiap tabel halaman dari tiap proses mengakses editor yang sama, tapi halaman data dari tiap proses tetap ditaruh di frame yang berbeda. Inilah yang dimaksud dengan berbagi halaman.

Gambar 30.9. *Share Page*



30.11. Rangkuman

Paging adalah suatu metoda yang memungkinkan suatu alamat fisik memori yang tersedia dapat tidak berurutan. Prinsipnya adalah memecah memori fisik dan memori logika menjadi blok-blok dengan ukuran sama (disebut *page*). Setelah itu kita membuat *page table* yang akan menerjemahkan memori logika menjadi memori fisik dengan perantara *Memory Management Unit* (MMU), dan pengeksekusian proses akan mencari memori berdasarkan tabel tersebut.

Paging atau pemberian halaman adalah salah satu teknik manajemen memori, dimana suatu memori komputer dibagi menjadi bagian-bagian kecil, yang disebut sebagai frame. Setiap sistem operasi mengimplementasikan paging dengan caranya masing-masing. Paging menjamin keamanan data di memori saat suatu proses sedang berjalan, dan dapat mempercepat akses ke memori, serta menghemat pemakaian memori. Untuk mencapai tujuan tersebut terdapat beberapa metoda pemberian halaman seperti Hierarchical Paging, Inverted Halaman tables, dan Hashed tabel halaman, yang masing-masing memiliki kelebihan maupun kekurangannya.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design*

and Implementation. Second Edition. Prentice-Hall.

[WEBAmirSch2000] YairTheo AmirSchlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

Bab 31. Segmentasi

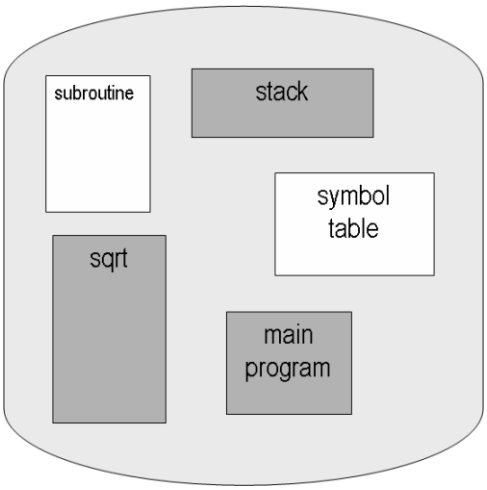
31.1. Pendahuluan

Aspek penting dari memori menajemen yang menjadi tak terhindarkan dengan paging adalah memori dari sudut pandang pengguna dan memori fisk yang sebenarnya. Sudut pandang pengguna terhadap memori tidak sama dengan memori fisik. sudut pandang pengguna ini dipetakan pada memori fisik, dimana dengan pemetaan tersebut mengizinkan perbedaan antara memori lojik dengan memori fisik.

Pernahkah pengguna mebayangkan sebuah memori sebagai sebuah array linier dari suatu byte, beberapanya terdiri dari instruksi dan sebagian yang lain berisi data? Sebagian besar orang akan berkata TIDAK. User lebih suka memandang sebuah memori sebagai sekumpulan variabel-variabel yang berada dalam segment-segmen dalam ukuran tertentu.

Jika kita membayangkan sebuah program yang terdiri dari main program, sub routin, fungsi dan modul-modul. Bisa juga terdapat berbagai struktur data seperti: tabel, array, stack, variabel dan lain-lain. Masing-masing modul atau elemen data mengacu pada sebuah nama. Sehingga jika membicarakan tabel simbol, fungsi sqrt atau main program tanpa mempedulikan alamatdi memori tetapi tetap dapat mendapatkannya. Juga kita tidak mempedulikan apakah fungsi sqrt disimpan setelah atau sebelum main program. Masing-masing segment ini adalah panjang varibael: Panjang ini terdefinisikan secara intrinsik sebagai tujuan dari pensegmenan dari sebuah program. Elemen-elemen dalam sebuah segmen teridentifikasi oleh offset-offsetnya dari awal segmennya.

Gambar 31.1. Alamat Lojik



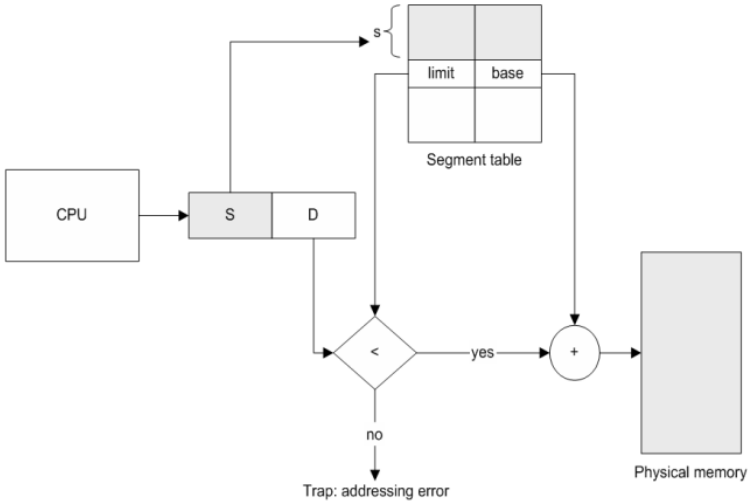
Segmentasi merupakan skema manajemen memori yang mendukung cara pandang seorang programmer terhadap memori. Ruang alamat lojik merupakan sekumpulan dari segmen-segmen. Masing-masing segment mempunyai panjang dan nama. Alamat diartikan sebagai nama segmen dan offset dalam suatu segmen. Jadi jika seorang pengguna ingin menunjuk sebuah alamat dapat dilakukan dengan menunjuk nama segmen dan offsetnya. Untuk lebih menyederhanakan implementasi, segmen-segmen diberi nomor yang digunakan sebagai pengganti nama segment. Sehingga, alamat lojik terdiri dari dua tupple: [segment-number, offset]

31.2. Segmentasi Perangkat Keras

Meskipun seorang pengguna dapat memandang suatu objek dalam suatu program sebagai alamat berdimensi dua, memori fisik yang sebenarnya tentu saja masih satu dimensi barisan byte. Jadi kita harus bisa mendefinisikan pemetaan dari dua dimensi alamat yang didefinisikan oleh pengguna ke satu dimensi alamat fisik. Pemetaan ini disebut sebagai sebuah segment table. Masing-masing

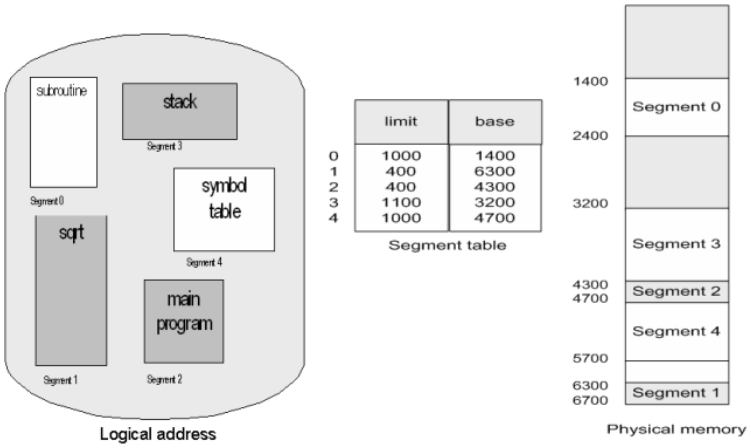
masukan dari mempunyai segment base dan segment limit. Segment base merupakan alamat fisik dan segmen limit diartikan sebagai panjang dari segmen.

Gambar 31.2. Arsitektur Segmentasi



Ilustrasi penggunaan segmen dapat dilihat pada Gambar 31.2, “Arsitektur Segmentasi”. Suatu alamat lojik terdiri dari dua bagian, yaitu nomor segmen(s), dan offset pada segmen(d). Nomor segmen digunakan sebagai indeks dalam segmen table. Offset d alamat lojik harus antara 0 hingga dengan segmen limit. Jika tidak maka diberikan pada sistem operasi. Jika offset ini legal maka akan dijumlahkan dengan segmen base untuk menjadikannya suatu alamat di memori fisik dari byte yang diinginkan. Jadi segmen table ini merupakan suatu array dari pasangan base dan limit register.

Gambar 31.3. Segmentasi



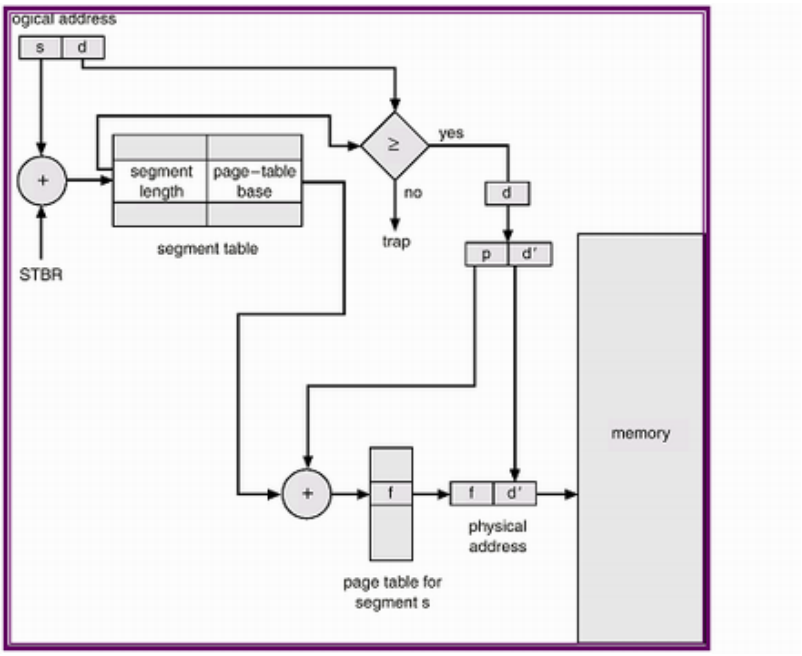
Sebagai contoh, Lihat pada Gambar 31.2, “Arsitektur Segmentasi”. Kita mempunyai nomor segmen dari 0 sampai dengan 4. Segmen-segmen ini disimpan dalam suatu memori fisik. Tabel segmen berisi data untuk masing-masing segmen, yang memberikan informasi tentang awal alamat dari segmen di fisik memori (atau base) dan panjang dari segmen (atau limit). Misalkan, segmen 2 mempunyai panjang 400 dan dimulai pada lokasi 4300. Jadi, referensi di byte 53 dari segmen 2 dipetakan ke lokasi 4300 + 53 = 5353. Suatu referensi ke segmen 3, byte 852, dipetakan ke 3200 (sebagai base dari segmen) + 852 = 4052. Referensi ke byte 1222 dari segmen 0 akan menghasilkan suatu trap ke sistem operasi, karena segmen ini hanya mempunyai panjang 1000 byte.

31.3. Keuntungan Segmentasi

Kelebihan Pemberian Halaman: tidak ada fragmentasi luar-alokasinya cepat.

Kelebihan Segmentasi: saling berbagi-proteksi.

Gambar 31.4. Segmentasi dengan Pemberian Halaman



Keuntungan pemakaian cara segmentasi ini adalah sebagai berikut:

1. Menyederhanakan penanganan struktur data yang berkembang. Seringkali penanganan struktur data menuntut perubahan panjang data. Hal ini dimungkinkan dengan adanya segmentasi. Jadi dengan segmentasi membuat penanganan struktur data menjadi fleksibel.

2. Kompilasi ulang independen tanpa mentautkan kembali seluruh program. Teknik ini memungkinkan program-program dikompilasi ulang secara independen tanpa perlu mentautkan kembali seluruh program dan dimuatkan kembali.

   Jika masing-masing prosedur terdapat di segmen terpisah beralamat 0 sebagai alamat awal, maka pentautan prosedur-prosedur yang dikompilasi secara terpisah sangat lebih mudah. Setelah semua prosedur dikompilasi dan ditautkan, panggilan ke prosedur di segmen n akan menggunakan alamat dua bagian yaitu (n,0) mengacu ke word alamat 0 (sebagai titik masuk) segmen ke n.

   Jika prosedur di segmen n dimodifikasi dan dikompilasi ulang, prosedur lain tidak perlu diubah (karena tidak ada modifikasi alamat awal) walau versi baru lebih besar dibanding versi lama.

3. Memudahkan pemakaian memori bersama diantara proses-proses Teknik ini memudahkan pemakaian memori bersama diantara proses-proses. Pemrogram dapat menempatkan program utilitas atau tabel data berguna di segmen yang dapat diacu oleh proses-proses lain. Segmentasi memberi fasilitas pemakaian bersama terhadap prosedur dan data untuk dapat diproses, berupa shared library.

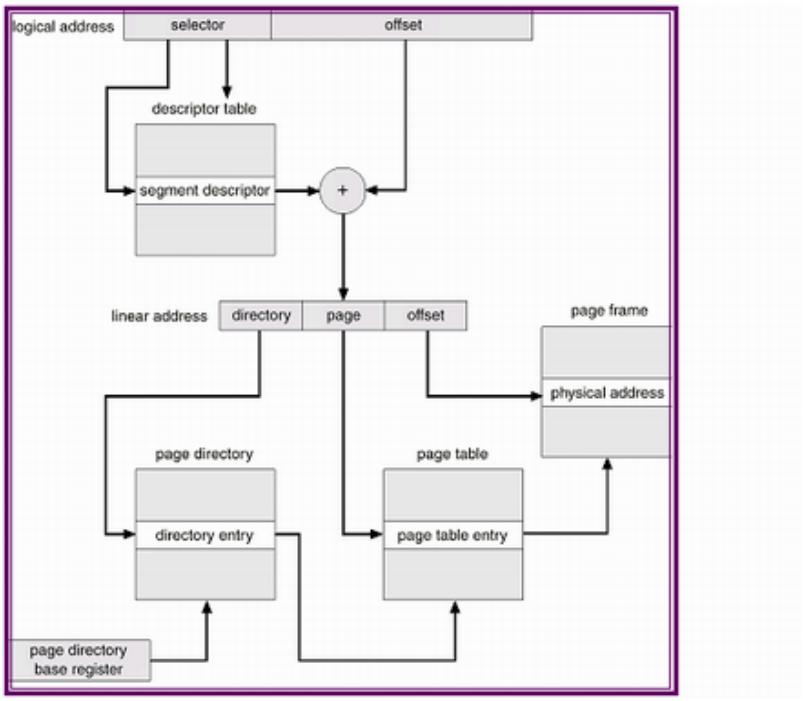
   Pada workstation modern yang menjalankan sistem Windows sering mempunyai pustaka grafis sangat besar. Pustaka ini diacu hampir semua program. Pada sistem bersegmen, pustaka grafis diletakan di satu segmen dan dipakai secara bersama banyak proses sehingga menghilangkan mempunyai pustaka ditiap ruang alamat proses.

Shared libraries di sistem pengalamantan murni lebih rumit, yaitu dengan simulasi segmentasi.

4. Memudahkan proteksi karena segmen dapat dikonstruksi berisi sekumpulan prosedur atau data terdefinisi baik, pemrogram atau administrator sistem dapat memberikan kewenangan pengaksesan secara nyaman.

31.4. Penggunaan Segmentasi Pentium

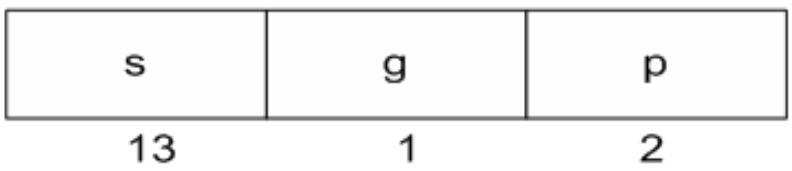
Gambar 31.5. Segmentasi dengan Pemberian Halaman (INTEL 30386)



Arsitektur Pentium memperbolehkan segmen sebanyak 4 GB dan jumlah maksimum segmen per proses adalah 16 KB. Ruang alamat lojik dari proses dibagi menjadi 2 partisi. Partisi pertama terdiri atas segmen-segmen hingga 8 KB (tersendiri dari proses). Partisi kedua terdiri atas segmen-segmen hingga 8 KB yang berbagi dengan semua proses-proses. Informasi partisi pertama terletak di LDT (Local Descriptor Table), informasi partisi kedua terletak di GDT (Global Descriptor Table). Masing-masing entri dari LDT dan GDT terdiri atas 8 byte segmen descriptor dengan informasi yang rinci tentang segmen-segmen tertentu, termasuk lokasi base dan limit dari segmen itu. Alamat logikal adalah sepasang (selector, offset), dimana berjumlah 16 bit.

Gambarnya adalah sebagai berikut:

Gambar 31.6. Selektor



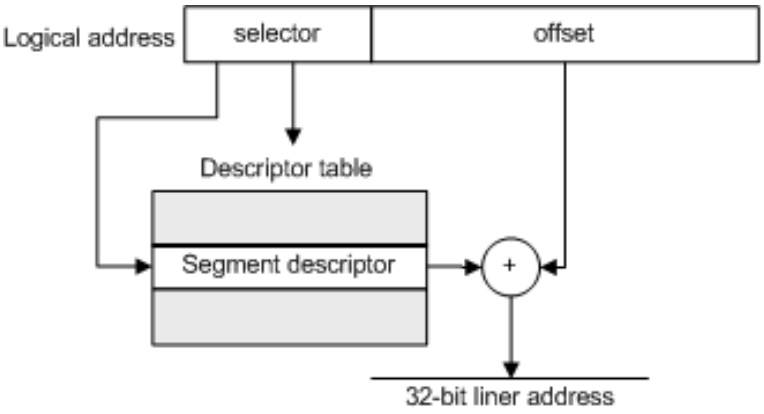
Dimana s menandakan nomor segmen, g mengindikasikan apakah segmen GDT atau LDT, dan p

mengenai proteksi. Offsetnya berjumlah 32 bit yang menspesifikasi lokasi byte (atau word) dalam segmentasi Pentium.

Pada Pentium mempunyai 6 register mikroprogram 8 byte, yang mengijinkan 6 segmen tadi untuk dialamatkan kapan saja. 6 register ini berfungsi untuk menangani deskriptor-deskriptor yang sesuai dengan LDT atau GDT. Cache ini juga mengijinkan Pentium untuk tidak membca deskriptor dari memori.

Alamat linear pada Pentium panjangnya 32 bit dan prosesnya adalah register segmen menunjuk pada entry yang sesuai dalam LDT atau GDT. Informasi base dan limit tentang segmen Pentium digunakan untuk menghasilkan alamat linear. Pertama, limit digunakan untuk memeriksa valid tidaknya suatu alamat. Jika alamat tidak valid, maka kesalahan memori akan terjadi yang menimbulkan trap pada sistem operasi. Jika alamat valid, maka nilai offset dijumlahkan dengan nilai base, yang menghasilkan alamat linear 32 bit. Hal ini ditunjukkan seperti pada gambar berikut:

Gambar 31.7. Segmentasi-Intel-Pentium



31.5. Segmentasi Linux

Pada Pentium, Linux hanya menggunakan 6 segmen:
1. Segmen untuk kode kernel.
2. Segmen untuk data kernel.
3. Segmen untuk kode pengguna.
4. Segmen untuk data pengguna.
5. Segmen Task-state (TSS).
6. Segmen default LDT.

Segmen untuk kode pengguna dan data pengguna berbagi dengan semua proses yang running pada pengguna mode, karena semua proses menggunakan ruang alamat lojik yang sama dan semua descriptor segmen terletak di GDT. TSS (Task-state Segment) digunakan untuk menyimpan context hardware dari tiap proses selama context switch. Tiap proses mempunyai TSS sendiri, dimana deskriptornya terletak di GDT. Segmen default LDT normalnya berbagi dengan semua proses dan biasanya tidak digunakan. Jika suatu proses membutuhkan LDT-nya, maka proses dapat membuatnya dan tidak menggunakan default LDT.

Seperti yang telah dijelaskan, tiap selektor segmen mempunyai 2 bit proteksi. Mak, Pentium mengijinkan proteksi 4 level. Dari 4 level ini, Linux hanya mengenal 2 level, yaitu pengguna mode dan kernel mode.

31.6. Rangkuman

1. Segmentasi merupakan skema manajemen memori yang mendukung cara pandang seorang

programmer terhadap memori. Masing-masing segment mempunyai panjang dan nama yang dapat mewakili sebagai alamat.
2. Segmentasi mempunyai keuntungan yang dapat dilihat dari sisi sudut pandang programmer.
3. Segmentasi juga bisa diterapkam dalam pentium baik dengan OS Windows ataupun Linux.

Rujukan

[Hariyanto1997] Bambang Hariyanto. 1997. *Sistem Operasi*. Buku Teks Ilmu Komputer. Edisi Kedua. Informatika. Bandung.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBAmirSch2000] YairTheo AmirSchlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

Bab 32. Memori Virtual

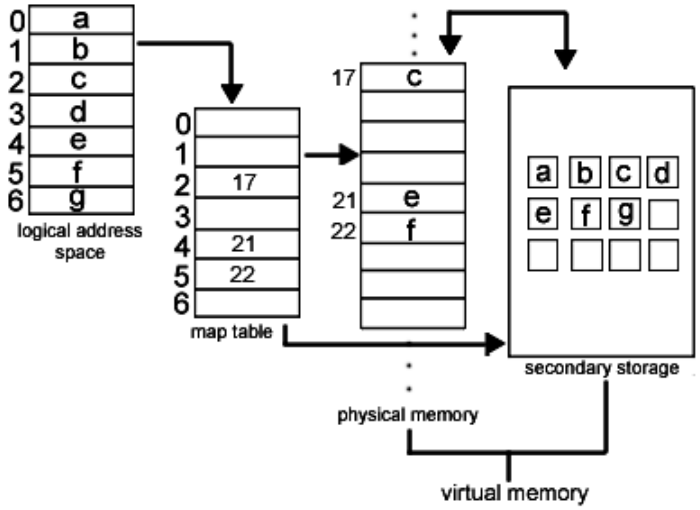
32.1. Pendahuluan

Manajemen memori pada intinya adalah menempatkan semua bagian proses yang akan dijalankan kedalam memori sebelum proses itu dijalankan. Untuk itu, semua bagian proses itu harus memiliki tempat sendiri di dalam memori fisik.

Tetapi tidak semua bagian dari proses itu akan dijalankan, misalnya:
- Pernyataan atau pilihan yang hanya akan dieksekusi pada kondisi tertentu. Contohnya adalah: pesan-pesan *error* yang hanya muncul bila terjadi kesalahan saat program dijalankan.
- Fungsi-fungsi yang jarang digunakan.
- Pengalokasian memori yang lebih besar dari yang dibutuhkan. Contoh: *array*, *list* dan tabel.

Pada memori berkapasitas besar, hal-hal ini tidak akan menjadi masalah. Akan tetapi, pada memori yang sangat terbatas, hal ini akan menurunkan optimalisasi utilitas dari ruang memori fisik. Sebagai solusi dari masalah-masalah ini digunakanlah konsep memori virtual.

Gambar 32.1. Memori Virtual



Memori virtual adalah suatu teknik yang memisahkan antara memori logis dan memori fisiknya. Teknik ini menyembunyikan aspek-aspek fisik memori dari pengguna dengan menjadikan memori sebagai lokasi alamat virtual berupa *byte* yang tidak terbatas dan menaruh beberapa bagian dari memori virtual yang berada di memori logis.

Setiap program yang dijalankan harus berada di memori. Memori merupakan suatu tempat penyimpanan utama (primary storage) yang bersifat sementara (volatile). Ukuran memori yang terbatas menimbulkan masalah bagaimana menempatkan program yang berukuran lebih besar dari ukuran memori fisik dan masalah penerapan multiprogramming yang membutuhkan tempat lebih besar di memori. Dengan pengaturan oleh sistem operasi dan didukung perangkat keras, memori virtual dapat mengatasi masalah kebutuhan memori tersebut.

Konsep memori virtual dikemukakan pertama kali oleh John Fotheringham pada tahun 1961 dengan menggunakan *dynamic storage allocation* pada sistem komputer atlas di Universitas Manchester. Sedangkan istilah memori virtual dipopulerkan oleh Peter J. Denning yang mengambil istilah 'virtual' dari dunia optik.

Memori virtual adalah teknik yang memisahkan memori logis dan memori fisik. Memori logis merupakan kumpulan keseluruhan halaman dari suatu program. Tanpa memori virtual memori logis

ini akan langsung dibawa ke memori fisik. Memori virtual melakukan pemisahan dengan menaruh memori logis ke disk sekunder dan hanya membawa halaman yang diperlukan ke memori utama. Teknik ini menjadikan seolah-olah ukuran memori fisik yang dimiliki lebih besar dari yang sebenarnya dengan menempatkan keseluruhan program di disk sekunder dan membawa halaman-halaman yang diperlukan ke memori fisik. Jadi jika proses yang sedang berjalan membutuhkan instruksi atau data yang terdapat pada suatu halaman tertentu maka halaman tersebut akan dicari di memori utama. Jika halaman yang diinginkan tidak ada maka akan dicari di disk. Ide ini seperti menjadikan memori sebagai cache untuk disk.

Sebagaimana dikatakan di atas bahwa hanya sebagian dari program yang diletakkan di memori fisik. Hal ini memberikan keuntungan:
- Berkurangnya proses M/K yang dibutuhkan (lalu lintas M/K menjadi rendah). Misalnya untuk program butuh membaca dari *disk* dan memasukkan dalam memory setiap kali diakses.
- Ruang menjadi lebih leluasa karena berkurangnya memori fisik yang digunakan. Contoh, untuk program 10 MB tidak seluruh bagian dimasukkan dalam memori fisik. Pesan-pesan error hanya dimasukkan jika terjadi error.
- Meningkatnya respon, karena menurunnya beban M/K dan memori.
- Bertambahnya jumlah pengguna yang dapat dilayani. Ruang memori yang masih tersedia luas memungkinkan komputer untuk menerima lebih banyak permintaan dari pengguna.

Gagasan utama dari memori virtual adalah ukuran gabungan program, data dan stack melampaui jumlah memori fisik yang tersedia. Sistem operasi menyimpan bagian-bagian proses yang sedang digunakan di memori fisik (memori utama) dan sisanya diletakkan di disk. Begitu bagian yang berada di disk diperlukan, maka bagian di memori yang tidak diperlukan akan dikeluarkan dari memori fisik (*swap-out*) dan diganti (*swap-in*) oleh bagian disk yang diperlukan itu.

Memori virtual diimplementasikan dalam sistem *multiprogramming*. Misalnya: 10 program dengan ukuran 2 Mb dapat berjalan di memori berkapasitas 4 Mb. Tiap program dialokasikan 256 KByte dan bagian-bagian proses *swap in*) masuk ke dalam memori fisik begitu diperlukan dan akan keluar (*swap out*) jika sedang tidak diperlukan. Dengan demikian, sistem *multiprogramming* menjadi lebih efisien.

Prinsip dari memori virtual yang perlu diingat adalah bahwa "Kecepatan maksimum eksekusi proses di memori virtual dapat sama, tetapi tidak pernah melampaui kecepatan eksekusi proses yang sama di sistem yang tidak menggunakan memori virtual".

Memori virtual dapat diimplementasikan dengan dua cara:
1. Demand paging. Menerapkan konsep pemberian halaman pada proses.
2. Demand segmentation. Lebih kompleks diterapkan karena ukuran segmen yang bervariasi. Demand segmentation tidak akan dijelaskan pada pembahasan ini.

32.2. *Demand Paging*

Demand paging atau permintaan pemberian halaman adalah salah satu implementasi dari memori virtual yang paling umum digunakan. *Demand paging* pada prinsipnya hampir sama dengan permintaan halaman (*paging*) hanya saja halaman (*page*) tidak akan dibawa ke ke dalam memori fisik sampai ia benar-benar diperlukan. Untuk itu diperlukan bantuan perangkat keras untuk mengetahui lokasi dari halaman saat ia diperlukan.

Karena *demand paging* merupakan implementasi dari memori virtual, maka keuntungannya sama dengan keuntungan memori virtual, yaitu:
- Sedikit M/K yang dibutuhkan.
- Sedikit memori yang dibutuhkan.
- Respon yang lebih cepat.
- Dapat melayani lebih banyak pengguna.

Ada tiga kemungkinan kasus yang dapat terjadi pada saat dilakukan pengecekan pada halaman yang dibutuhkan, yaitu: halaman ada dan sudah berada di memori – statusnya valid ("1"); halaman ada tetapi masih berada di disk atau belum berada di memori (harus menunggu sampai dimasukkan) – statusnya tidak valid ("0"). Halaman tidak ada, baik di memori maupun di *disk* (*invalid reference*).

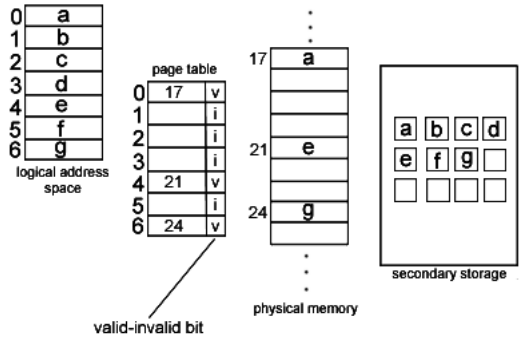
Saat terjadi kasus kedua dan ketiga, maka proses dinyatakan mengalami kesalahan halaman.

Perangkat keras akan menjebaknya ke dalam sistem operasi.

32.3. Skema Bit Valid – Tidak Valid

Dengan meminjam konsep yang sudah pernah dijelaskan dalam Bab 31, *Segmentasi*, maka dapat ditentukan halaman mana yang ada di dalam memori dan mana yang tidak ada di dalam memori. Konsep itu adalah skema bit valid-tidak valid, di mana di sini pengertian valid berarti bahwa halaman legal dan berada dalam memori (kasus 1), sedangkan tidak valid berarti halaman tidak ada (kasus 3) atau halaman ada tapi tidak ditemui di memori (kasus 2).

Gambar 32.2. Tabel halaman untuk skema bit valid-tidak valid



Pengaturan bit dilakukan dengan:
- Bit=1 berarti halaman berada di memori.
- Bit=0 berarti halaman tidak berada di memori.

Apabila ternyata hasil dari translasi, bit halaman bernilai 0, berarti kesalahan halaman terjadi.

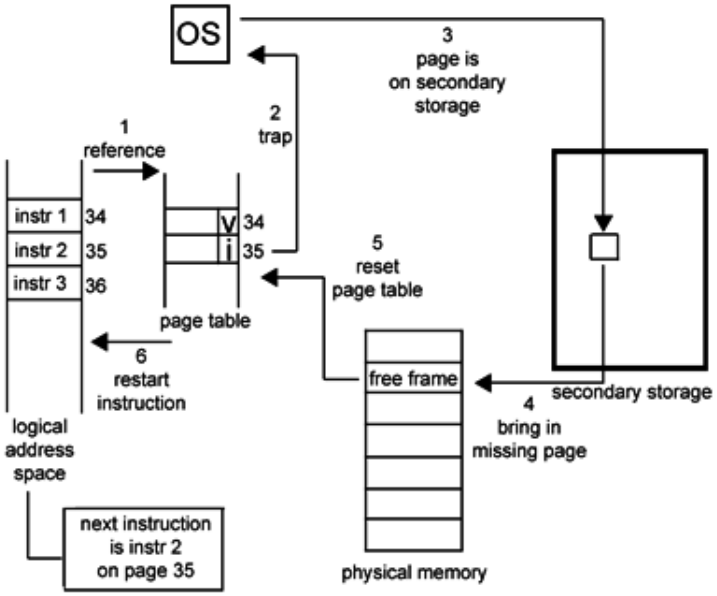
Kesalahan halaman adalah interupsi yang terjadi ketika halaman yang diminta tidak berada di memori utama. Proses yang sedang berjalan akan mengakses tabel halaman untuk mendapatkan referensi halaman yang diinginkan. Kesalahan halaman dapat diketahui dari penggunaan skema bit valid-tidak valid. Bagian inilah yang menandakan terjadinya suatu permintaan halaman (demand paging).

Jika proses mencoba mengakses halaman dengan bit yang diset tidak valid maka akan terjadi kesalahan halaman. Proses akan terhenti sementara halaman yang diminta dicari di disk.

32.4. Penanganan Kesalahan Halaman

Pada gambar dapat dilihat alur jalannya instruksi yang mengalami kesalahan halaman. Penanganan kesalahan halaman dapat dituliskan sebagai berikut:
- CPU mengambil instruksi dari memori untuk dijalankan. Lakukan pengambilan instruksi dari halaman pada memori dengan mengakses tabel halaman. Pada tabel halaman bit terset tidak valid.
- Terjadi interupsi kesalahan halaman, maka interupsi itu menyebabkan *trap* pada sistem operasi.
- Jika referensi alamat yang diberikan ke sistem operasi ilegal atau dengan kata lain halaman yang ingin diakses tidak ada maka proses akan dihentikan. Jika referensi legal maka halaman yang diinginkan diambil dari disk.
- Halaman yang diinginkan dibawa ke memori fisik.
- Mengatur ulang tabel halaman sesuai dengan kondisi yang baru. Jika tidak terdapat ruang di memori fisik untuk menaruh halaman yang baru maka dilakukan penggantian halaman dengan memilih salah satu halaman. Penggantian halaman dilakukan menurut algoritma tertentu yang akan dibahas pada bab selanjutnya. Jika halaman yang digantikan tersebut sudah dimodifikasi oleh proses maka halaman tersebut harus ditulis kembali ke disk.
- Setelah halaman yang diinginkan sudah dibawa ke memori fisik maka proses dapat diulang.

Gambar 32.3. Gambaran pada saat penanganan kesalahan halaman



Kesalahan halaman menyebabkan urutan kejadian berikut:
- Ditangkap oleh Sistem Operasi.
- Menyimpan *register pengguna* dan proses.
- Tetapkan bahwa interupsi merupakan kesalahan halaman.
- Periksa bahwa referensi halaman adalah legal dan tentukan lokasi halaman pada disk.
- Kembangkan pembacaan disk ke *frame* kosong.
- Selama menunggu, alokasikan CPU ke pengguna lain dengan menggunakan penjadwalan CPU.
- Terjadi interupsi dari disk bahwa M/K selesai.
- Simpan register dan status proses untuk pengguna yang lain.
- Tentukan bahwa interupsi berasal dari disk.
- Betulkan tabel halaman dan tabel yang lain bahwa halaman telah berada di memory.
- Tunggu CPU untuk untuk dialokasikan ke proses tersebut
- Kembalikan register pengguna, status proses, tabel halaman, dan meneruskan instruksi interupsi.

Pada berbagai kasus, ada tiga komponen yang kita hadapi pada saat melayani kesalahan halaman:
- Melayani interupsi kesalahan halaman
- Membaca halaman
- Mengulang kembali proses

32.5. Kelebihan/Kekurangan Demand Paging

Manajemen memori dengan permintaan halaman (demand paging) memiliki kelebihan yang sama dengan manajemen memori dengan pemberian halaman, antara lain menghilangkan masalah fragmentasi eksternal sehingga tidak diperlukan pemadatan (compaction). Selain itu permintaan halaman memiliki kelebihan yang lain, yaitu:

1. **Memori virtual yang besar.** Memori logis tidak lagi terbatas pada ukuran memori fisik. Hal ini berarti bahwa besar suatu program tidak akan terbatas hanya pada ukuran memori fisik tersedia.
2. **Penggunaan memori yang lebih efisien.** Bagian program yang dibawa ke memori fisik hanyalah bagian program yang dibutuhkan sementara bagian lain yang jarang digunakan tidak akan dibawa.
3. **Meningkatkan derajat *multiprogamming*.** Derajat multiprogramming menunjukkan banyaknya proses yang berada di memori fisik. Dengan penggunaan permintaan halaman maka ukuran suatu program di memori akan lebih kecil mengingat bahwa hanya bagian program yang diperlukan saja yang akan dibawa ke memori fisik. Penggunaan memori yang lebih kecil oleh sebuah proses

memberi sisa ruang memori fisik yang lebih besar sehingga lebih banyak proses yang bisa berada di memori fisik. Hal ini berpengaruh pada utilisasi CPU dan throughput (banyaknya proses yang dapat diselesaikan dalam satu satuan waktu) yang lebih besar.
4. **Penggunaan M/K yang lebih sedikit.** Hal ini dapat terjadi karena permintaan halaman hanya membawa bagian yang diperlukan dari suatu program. Penggunaan M/K pada permintaan halaman lebih sedikit dibandingkan dengan manajemen memori lain yang membawa seluruh memori logis sebuah program ke memori fisik.

Permintaan halaman juga memiliki beberapa kekurangan, antara lain:
1. ***Processor overhead*.** Interupsi kesalahan halaman memberikan kerja tambahan kepada CPU untuk mengambil halaman yang tidak berada di memori fisik pada saat diperlukan.
2. ***Thrashing*.** Suatu kondisi yang terjadi akibat kesalahan halaman yang melewati batas normal. Akibat dari *thrashing* adalah CPU lebih banyak mengurusi kesalahan halaman daripada menangani proses itu sendiri. Hal ini dapat menurunkan kinerja dari CPU.

32.6. Kinerja *Demand Paging*

Salah satu hal yang menjadi pertimbangan dalam penggunaan permintaan halaman adalah waktu akses memori menjadi lebih lambat akibat perlunya penanganan kesalahan halaman.

Halaman Fault Time

Lamanya waktu untuk mengatasi kesalahan halaman disebut dengan halaman *fault time*. Ada tiga faktor utama yang mempengaruhi halaman *fault time* ini, yaitu:
1. **Melayani interupsi dari kesalahan halaman.** Aktivitas yang dilakukan dalam melayani kesalahan halaman ini, yaitu:
   a. Memberitahu sistem operasi saat terjadinya kesalahan halaman.
   b. Menyimpan status dari proses bersangkutan.
   c. Memeriksa apakah referensi halaman yang diberikan legal atau tidak. Bila referensi yang diberikan legal maka dicari lokasi dari halaman tersebut di disk.
2. **Pembacaan halaman.** Aktivitas yang terjadi dalam pembacaan halaman ini, yaitu:
   a. Menunggu dalam antrian sampai mendapatkan giliran untuk membaca.
   b. Menunggu disk untuk membaca lokasi yang diminta. Disk melakukan kerja mekanis untuk membaca data sehingga lebih lambat dari memori.
   c. Mengirim halaman yang diminta ke memori fisik.
3. **Pengulangan instruksi.** Aktivitas yang terjadi untuk mengulangi instruksi ini, yaitu:
   a. Interupsi proses yang sedang berjalan untuk menandakan bahwa proses yang sebelumnya terhenti akibat kesalahan halaman telah selesai dalam membaca halaman yang diminta.
   b. Menyimpan status dari proses yang sedang berjalan.
   c. Membetulkan tabel halaman untuk menunjukkan bahwa halaman yang ingin dibaca sudah ada di memori fisik.
   d. Mengambil kembali status proses bersangkutan untuk selanjutnya dijalankan di CPU.

Effective Access Time

Untuk mengetahui kinerja permintaan halaman dapat dilakukan dengan menghitung effective access time-nya.

Effective Access Time (EAT) = (1-p) x ma + p x halaman fault time

- p = kemungkinan terjadi halaman fault ($0 < p < 1$)

- ma = memory access time (10 - 200 ns)

Jika p = 0 maka tidak ada kesalahan halaman, sehingga EAT = memory access time.

Jika p = 1 maka semua pengaksesan mengalami kesalahan halaman.

Untuk menghitung EAT, kita harus mengetahui waktu yang dibutuhkan untuk melayani kesalahan halaman.

Contoh penggunaan Effective Access Time

Diketahui waktu pengaksesan memori (ma) = 100 ns, waktu kesalahan halaman (halaman fault time) = 20 ms. Berapakah Effective Access Time-nya?

```
EAT = (1-p) x ma + p x halaman fault time
    = (1-p) x 100 + p x 20.000.000
    = 100 - 100p + 20.000.000p
    = 100 + 19.999.900p nanosecond
```

Pada permintaan halaman diusahakan agar kemungkinan terjadinya halaman fault rendah karena bila EAT-nya meningkat, maka proses akan berjalan lebih lambat.

32.7. Persyaratan Perangkat Keras

Locality of References

Jika terjadi banyak kesalahan halaman maka efisiensi sistem akan menurun. Oleh karena itu, kemungkinan terjadinya kesalahan halaman harus dikurangi atau dibatasi dengan memakai prinsip lokalitas ini.

Prinsip lokalitas ini dibagi menjadi 2 bagian:
- **temporal.** Lokasi yang sekarang ditunjuk kemungkinan akan ditunjuk lagi.
- **spatial.** Kemungkinan lokasi yang dekat dengan lokasi yang sedang ditunjuk akan ditunjuk juga.

Pure Demand Paging

Penjalanan (running) sebuah program dimulai dengan membawa hanya satu halaman awal ke main memori. Setelah itu, setiap kali terjadi permintaan halaman, halaman tersebut baru akan dimasukkan ke dalam memori. Halaman akan dimasukkan ke dalam memori hanya bila diperlukan.

Multiple Page Fault

Kesalahan halaman yang terjadi karena satu instruksi memerlukan pengaksesan beberapa halaman yang tidak ada di memori utama. Kejadian seperti ini dapat mengurangi kinerja dari program.

Pemberian nomor halaman melibatkan dukungan perangkat keras, sehingga ada persyaratan perangkat keras yang harus dipenuhi. Perangkat-perangkat keras tersebut sama dengan yang digunakan untuk *paging* dan *swapping*, yaitu:
- tabel halaman "bit valid-tidak valid"
  - Valid ("1") artinya halaman sudah berada di memori
  - Tidak valid ("0") artinya halaman masih berada di disk.
- Memori sekunder, digunakan untuk menyimpan proses yang belum berada di memori.

Lebih lanjut, sebagai konsekuensi dari persyaratan ini, akan diperlukan pula perangkat lunak yang dapat mendukung terciptanya pemberian nomor halaman.

Restart Instruction

Salah satu penanganan jika terjadi kesalahan halaman adalah kebutuhan akan pengulangan instruksi. Penanganan pengulangam instruksi berbeda-beda tergantung pada kemungkinan terjadinya kesalahan halaman dan kompleksitas instruksi.
1. Jika kesalahan halaman terjadi saat pengambilan instruksi maka pengulangan proses dilakukan dengan mengambil instruksi itu lagi.
2. Jika kesalahan halaman terjadi saat pengambilan operan maka pengulangan proses dilakukan dengan mengambil instruksi itu lagi, men-dekode-kan, baru mengambil operan.

3. Pengulangan proses ketika instruksi memodifikasi beberapa lokasi yang berbeda, misalnya pada instruksi move character dari blok *source* ke blok *destination* dimana kedua blok tersebut tumpang tindih (overlapping) dan kesalahan halaman terjadi saat sebagian instruksi tersebut sudah dijalankan, tidak dapat dilakukan secara langsung karena ada bagian dari blok source yang telah termodifikasi. Ada dua solusi untuk mengatasi hal ini, yaitu:
   a. Komputasi kode mikro dan berusaha untuk mengakses kedua ujung dari blok, agar tidak ada modifikasi halaman yang sempat terjadi.
   b. Menggunakan register sementara (temporary register) untuk menyimpan nilai yang berada pada lokasi yang "overwritten" sehingga bila terjadi kesalahan halaman semua nilai lama dapat ditulis kembali ke memori dan pengulangan dapat dilakukan.
   c. Pengulangan proses ketika instruksi menggunakan mode pengalamatan spesial seperti *autoincrement* dan *autodacrement*, misalnya instruksi MOV (R2)+,-(R3) yang menyalin isi lokasi yang ditunjuk register R2 ke lokasi yang ditunjuk register R3. Misalkan R2 = 10 dan R3 = 20 maka isi alamat memori 10 akan disalin ke alamat 19. Apabila terjadi kesalahan halaman saat penyimpanan ke lokasi yang ditunjuk R3 dan proses langsung terulang maka penyalinan akan dilakukan dari lokasi 11 ke lokasi 18. Solusinya yaitu dengan memakai register status khusus yang akan menyimpan angka pada register dan jumlah perubahan (modifikasi) yang dialami oleh register sehingga register dapat diubah kembali ke nilai lama dan sistem operasi dapat mengulang sebagian proses instruksi yang telah dilakukan.

32.8. Rangkuman

Memori virtual dapat mengatasi masalah keterbatasan memori fisik dalam menyimpan suatu proses yang besar dengan cara menyimpan salinan seluruh program dalam disk dan hanya membawa ke memori utama bagian program yang diperlukan. Teknik permintaan halaman (demand paging) digunakan untuk mengimplementasikan konsep memori virtual. Permintaan halaman ini terjadi akibat kesalahan halaman (halaman fault) yang berarti bahwa halaman yang diminta tidak berada di memori utama dan harus dibawa dari disk.

Waktu untuk menangani kesalahan halaman berpengaruh pada kinerja pengaksesan memori. Untuk mengurangi jumlah kesalahan halaman yang terjadi maka digunakan konsep lokalitas (locality of references). Pengimplentasian permintaan halaman pada memori virtual juga tidak lepas dari masalah-masalah seperti perangkat keras yang dibutuhkan dan masalah pengulangan instruksi.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBAmirSch2000] Yair Amir dan Theo Schlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBCACMF1961] John Fotheringham. " Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store – http://www.eecs.harvard.edu/ cs261/ papers/ frother61.pdf ". Diakses 29 Juni 2006. *Communications of the ACM* . 4. 10. October 1961.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBHP1997] Hewlett-Packard Company. 1997. *HP-UX Memory Management – Overview of Demand Paging – http://docs.hp.com/en/5965-4641/ch01s10.html* . Diakses 29 Juni 2006.

[WEBJupiter2004] Jupitermedia Corporation. 2004. *Virtual Memory – http://www.webopedia.com/ TERM/ v/ virtual_memory.html* . Diakses 29 Juni 2006.

[WEBOCWEmer2005] Joel Emer dan Massachusetts Institute of Technology. 2005. *OCW – Computer System Architecture – Fall 2005 – Virtual Memory Basics – http://ocw.mit.edu/ NR/ rdonlyres/ Electrical -Engineering -and -Computer -Science/ 6 -823Computer -System -ArchitectureSpring2002/ C63EC0D0 -0499 -474F -BCDA -A6868A6827C4/ 0/ lecture09.pdf* . Diakses 29 Juni 2006.

[WEBRegehr2002] John Regehr dan University of Utah. 2002. *CS 5460 Operating Systems – Demand Halamand Virtual Memory – http://www.cs.utah.edu/ classes/ cs5460-regehr/ lecs/ demand_paging.pdf* . Diakses 29 Juni 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

Bab 33. Permintaan Halaman Proses

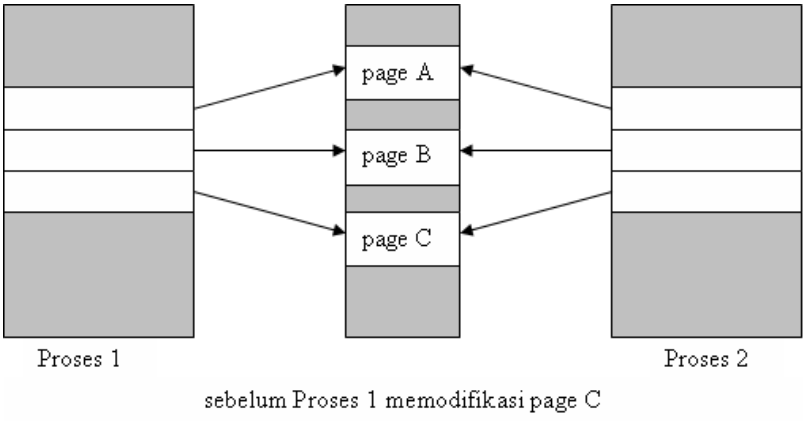
33.1. Pendahuluan

Seperti yang telah dibahas dalam bagian sebelumnya, memori virtual memungkinkan proses untuk berbagi pakai memori. Proses berbagi pakai ini adalah proses berbagi pakai halaman memori virtual. Karena setiap proses membutuhkan halaman tersendiri maka akan dibutuhkan teknik untuk mengelola halaman dan pembuatannya. Teknik untuk mengoptimasi pembuatan dan penggunaan halaman proses adalah dengan Copy-On-Write dan Memory-Mapped-File.

33.2. *Copy-On-Write*

Dengan memanggil sistem pemanggilan **fork()**, sistem operasi akan membuat proses anak sebagai salinan dari proses induk. Sistem pemanggilan *fork()* bekerja dengan membuat salinan alamat proses induk untuk proses anak, lalu membuat salinan halaman milik proses induk tersebut.Tapi, karena setelah pembuatan proses anak selesai, proses anak langsung memanggil sistem pemanggilan *exec()* yang menyalin alamat proses induk yang kemungkinan tidak dibutuhkan.

Gambar 33.1. Sebelum



sebelum Proses 1 memodifikasi page C

Oleh karena itu, lebih baik kita menggunakan teknik lain dalam pembuatan proses yang disebut sistem ***copy-on-write***. Teknik ini bekerja dengan memperbolehkan proses anak untuk menginisialisasi penggunaan halaman yang sama secara bersamaan. halaman yang digunakan bersamaan itu, disebut dengan "halaman *copy-on-write*", yang berarti jika salah satu dari proses anak atau proses induk melakukan penulisan pada halaman tersebut, maka akan dibuat juga sebuah salinan dari halaman itu.

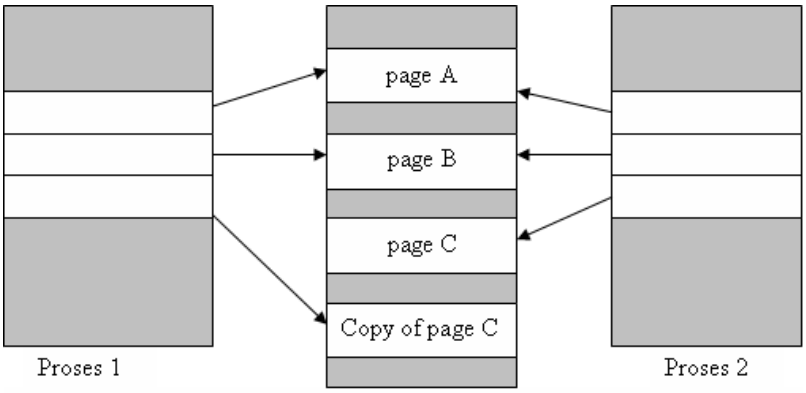
Sebagai contoh, sebuah proses anak hendak memodifikasi sebuah halaman yang berisi sebagian dari *stack*. Sistem operasi akan mengenali hal ini sebagai *copy-on-write*, lalu akan membuat salinan dari halaman ini dan memetakannya ke alamat memori dari proses anak, sehingga proses anak akan memodifikasi halaman salinan tersebut, dan bukan halaman milik proses induk. Dengan tekik *copy-on-write* ini, halaman yang akan disalin adalah halaman yang dimodifikasi oleh proses anak atau proses induk. Halaman-halaman yang tidak dimodifikasi akan bisa dibagi untuk proses anak dan proses induk.

Saat suatu halaman akan disalin menggunakan teknik *copy-on-write*, digunakan teknik ***zero-fill-on-demand*** untuk mengalokasikan halaman kosong sebagai tempat meletakkan hasil duplikat. Halaman kosong tersebut dialokasikan saat *stack* atau *heap* suatu proses akan diperbesar atau untuk mengatur halaman *copy-on-write*. Halaman *Zero-fill-on-demand* akan dibuat kosong sebelum dialokasikan, yaitu dengan menghapus isi awal dari halaman. Karena itu, dengan

copy-on-write, halaman yang sedang disalin akan disalin ke sebuah halaman *zero-fill-on*.

Teknik *copy-on-write* digunakan oleh beberapa sistem operasi seperti Windows 2000, Linux, dan Solaris2.

Gambar 33.2. Sesudah



Dengan COW, beberapa proses dapat berbagi pakai halaman yang sama, namun jika ada salah satu proses akan menulis atau melakukan modifikasi, maka dibuat halaman baru (sebagai salinan dari halaman copy-on-write). Pada halaman salinan tersebut proses melakukan modifikasi. Halaman yang lama tetap. Halaman Copy-On-Write diberi tanda sebagai "halaman Copy-On-Write" dan bersifat "read only", sedangkan halaman salinan tidak diberi tanda dan bisa dimodifikasi.

Misalkan, halaman C digunakan bersama oleh proses 1 dan proses 2. Ketika proses 1 akan melakukan modifikasi terhadap halaman C, maka sistem operasi membuat halaman baru sebagai salinan dari halaman C yang ditunjuk oleh proses 1. Proses 1 melakukan modifikasi pada halaman yang baru tersebut (ditunjuk oleh proses 1), sedangkan halaman C tetap (ditunjuk oleh proses 2).

Untuk menentukan halaman baru sebagai salinan dari halaman Copy-On-Write tersebut Sistem Operasi harus mengetahui letak halaman-halaman yang kosong. Beberapa Sistem Operasi menyediakan pool dari halaman-halaman yang kosong. Selain untuk salinan dari halaman Copy-On-Write, halaman-halaman kosong tersebut disediakan untuk proses yang melakukan penambahan stack atau heap.

Teknik yang biasa digunakan oleh sistem operasi untuk menyediakan halaman tersebut disebut zero-fill-on-demand. Teknik ini dilakukan dengan mengosongkan halaman-halaman sebelum digunakan oleh proses yang baru.

Keuntungan teknik COW
1. Jika tidak ada modifikasi pada halaman maka pembuatan salinan dari halaman tidak akan pernah dilakukan atau jumlah memori fisik yang dialokasikan untuk proses tidak pernah bertambah sampai terjadi penulisan data.
2. Penggunaan memori fisik sangat jarang, memori fisik baru digunakan hanya jika terjadi penyimpanan data.

Kekurangan teknik COW:
1. Bertambahnya kompleksitas pada level kernel, pada saat kernel menulis ke halaman, maka harus dibuat salinan halaman jika halaman tersebut diberi tanda COW.

33.3. *Memory-Mapped Files*

Memory-Mapped Files adalah teknik yang digunakan untuk pengaksesan file, dimana blok berkas dalam disk dipetakan ke halaman memori. File tersebut belum berada dalam memori. Karena

pengaksesan berkas dilakukan melalui demand paging, akses kali pertama pada berkas akan menghasilkan halaman fault. Namun sesudah itu sebagian dari halaman berkas akan dibaca dari sistem berkas ke memori fisik. Selanjutnya pembacaan dan penulisan pada berkas dilakukan dalam memori. Hal ini menyederhanakan pengaksesan dan penggunaan berkas daripada pengaksesan langsung ke disk melalui sistem call read() dan write().

Bila terjadi penulisan/perubahan pada bagian berkas dalam memori, perubahan ini tidak langsung dilakukan pada berkas dalam disk. Beberapa sistem operasi melakukan perubahan secara periodik dengan memeriksa apakah ada perubahan pada berkas atau tidak. Ketika berkas ditutup semua perubahan ditulis ke disk dan berkas dihapus dari memori virtual.

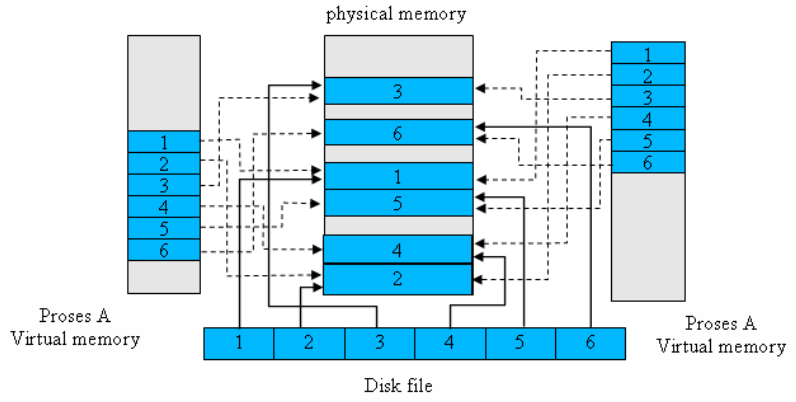
Implementasi Memory-Mapped File (MMF)

Pada beberapa sistem operasi, MMF dilakukan dengan menggunakan sistem call khusus sementara sistem call biasa digunakan untuk berkas M/K yang lainya. Pada beberapa sistem lain, walaupun sistem call khusus untuk MMF ada, semua berkas akan dipetakan meskipun file tersebut tidak ditandai sebagai berkas MMF. Pada Solaris misalnya, sistem ini memiliki sistem call mmap() untuk mengaktifkan MMF. Namun Solaris tetap akan memetakan file yang diakses menggunakan sistem call open(), read() dan write().

MMF juga dapat memetakan banyak proses pada berkas yang sama secara bersamaan sehingga proses-proses tersebut dapat saling berbagi data. Pada multiproses, perubahan yang dibuat oleh salah satu proses dapat dilihat oleh proses lainnya. MMF juga mendukung teknik COW. Jika ditambahkan teknik COW, halaman yang dipetakan berada dalam mode read only dan apabila ada proses yang merubah isi berkas maka proses tersebut akan diberi salinan halaman yang akan dirubahnya sehingga perubahan yang terjadi tidak dapat dilihat oleh proses lain.

Dalam beberapa hal, sharing memori pada MMF sama dengan sharing memori yang telah dijelaskan pada bab sebelumnya. Unix dan Linux menggunakan mekanisme yang berbeda untuk MMF dan sharing memori, untuk MMF digunakan sistem call mmap(), sedangkan untuk sharing memori digunakan sistem call shmget() dan shmat(). Pada Windows NT, 2000, dan XP penerapan sharing memori dilakukan dengan mekanisme MMF.

Gambar 33.3. MMF



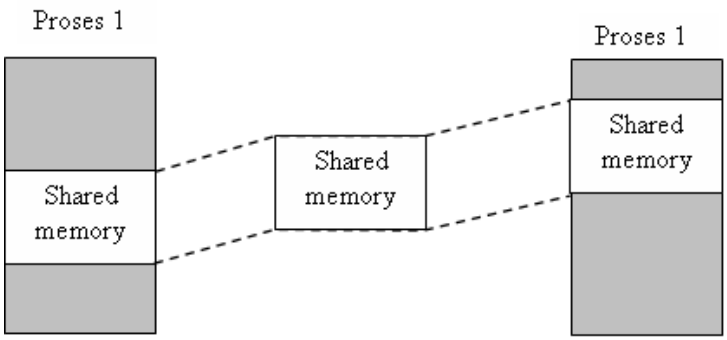
Berbagi Memori Dalam Win32API

Untuk melakukan sharing memori menggunakan MMF, pertama kali harus dibuat (create) pemetaan berkas dari disk ke memori. Setelah itu, dibuat view dari berkas tersebut pada alamat virtual proses. Proses kedua (lainnya) dapat membuka dan membuat view dari berkas tersebut dalam ruang alamat virtualnya. Pemetaan berkas ini merupakan representasi sharing memori, dimana proses dapat berkomunikasi satu sama lain.

Misalnya pada proses produsen dan konsumen. Produsen membuat obyek sharing memori

menggunakan fasilitas memori mapped-file Win32 API. Kemudian produsen menulis pesan ke dalam memori tersebut. Setelah itu, konsumen dapat membuat pemetaan ke obyek sharing memori tersebut, membukanya, dan membaca pesan yang ditulis oleh produsen.

Gambar 33.4. MMF pada Win32 API



Untuk membuat memori mapped file, proses harus membuka (open) berkas yang akan dipetakan dengan fungsi CreateFile(). Kemudian proses memetakan berkas dengan fungsi CreateFileMapping(). Setelah berkas dipetakan maka view dari file dapat dimunculkan pada alamat virtual proses dengan menggunakan fungsi MapViewOfFile().

Fungsi CreateFileMapping() akan membuat obyek sharing memori yang disebut SharedObject. Konsumen dapat berkomunikasi menggunakan sharing memori dengan membuat pemetaan ke obyek dengan nama yang sama. Sementara fungsi MapViewOfFile() akan memunculkan pointer pada obyek shared-memori, seluruh akses pada bagian memori ini merupakan akses pada memory-mapped file.

Kelebihan Memory-Mapped Files

1. Akses pada berkas dilakukan dengan pointer, tidak langsung dengan sistem call read() dan write().
2. Akses dapat dilakukan secara acak.
3. "On demand loading", permintaan M/K baru akan terjadi kalau berkas/bagian berkas yang bersangkutan di akses.
4. Data/bagian berkas yang tak terpakai akan di-swap out.

Kelemahan Memory-Mapped Files

1. Bila terjadi penambahan berkas harus dibuat MMF yang baru.
2. Proses M/K secara asinkron tidak dapat dilakukan, semuanya dilakukan per blok.
3. Kadang-kadang terjadi bug ketika OS menulis ke dalam file.

33.4. Rangkuman

Copy-On-Write merupakan suatu teknik optimasi yang digunakan dalam Memori Virtual. Teknik ini bekerja dengan cara berbagi pakai halaman untuk beberapa proses, jika ada proses yang menulis ke halaman tersebut maka dibuatkan halaman baru, dan di halaman baru tersebut proses melakukan penulisan. Teknik ini sangat menguntungkan karena menghemat ruang memori, walaupun akan menambah kompleksitas operasi pada kernel

Memori Mapped File memetakan blok disk yang berisi berkas pada halaman-halaman memori. Dengan cara ini akses dan penggunakan berkas ditangani langsung oleh rutin akses memori, bukan lagi sistem call yang berhubungan langsung dengan disk. Hal ini sudah barang tentu akan menghemat waktu akses file.

Memori Mapped File sama atau mirip dengan Sharing Memori. Linux dan Unix membedakan mekanisme MMF dengan Sharing Memori menggunakan sistem call yang berbeda. Sedangkan pada keluarga Windows implementasi sharing memori adalah menggunakan teknik Memori-Mapped File.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBAmirSch2000] YairTheo AmirSchlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBEgui2006] Equi4 Software. 2006. *Memory Mapped Files – http://www.equi4.com/mkmmf.html* . Diakses 3 Juli 2006.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

[WEBWiki2006c] From Wikipedia, the free encyclopedia. 2006. *Memory Management Unit – http://en.wikipedia.org/ wiki/ Memory_management_unit* . Diakses 30 Juni 2006.

[WEBWiki2006d] From Wikipedia, the free encyclopedia. 2006. *Page Fault – http://en.wikipedia.org/ wiki/ Page_fault* . Diakses 30 Juni 2006.

[WEBWiki2006e] From Wikipedia, the free encyclopedia. 2006. *Copy on Write – http://en.wikipedia.org/ wiki/ Copy_on_Write* . Diakses 03 Juli 2006.

Bab 34. Algoritma Ganti Halaman

34.1. Pendahuluan

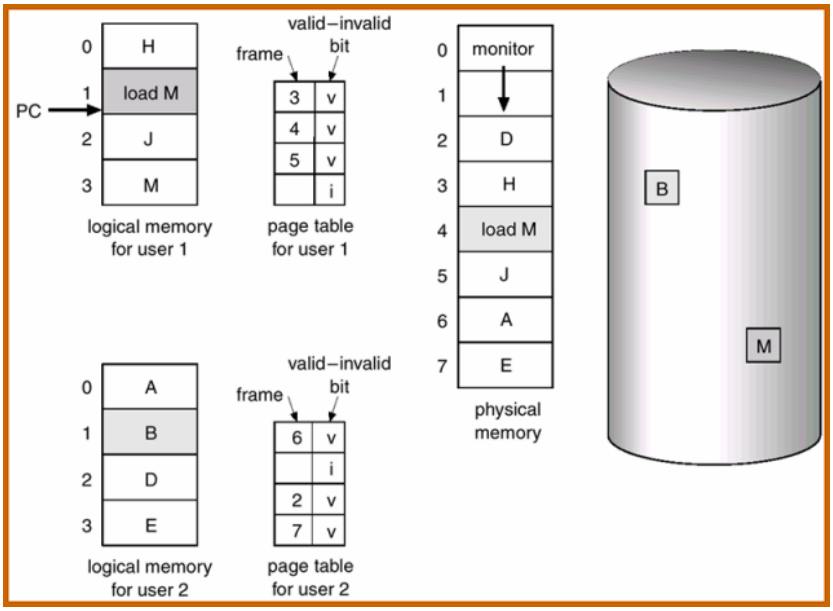
Masalah kesalahan halaman pasti akan dialami oleh setiap halaman minimal satu kali. Akan tetapi, sebenarnya sebuah proses yang memiliki N buah halaman hanya akan menggunakan N/2 diantaranya. Kemudian permintaan halaman akan menyimpan M/K yang dibutuhkan untuk mengisi N/2 halaman sisanya. Dengan demikian utilisasi CPU dan *throughput* dapat ditingkatkan.

Upaya yang dilakukan oleh permintaan halaman dalam mengatasi kesalahan halaman didasari oleh pemindahan halaman. Sebuah konsep yang akan kita bahas lebih lanjut dalam sub bab ini.

Prinsip dalam melakukan pemindahan halaman adalah sebagai berikut: "Jika tidak ada bingkai yang kosong, cari bingkai yang tidak sedang digunakan dalam jangka waktu yang lama, lalu kosongkan dengan memindahkan isinya ke dalam ruang pertukaran dan ubah semua tabel halamannya sebagai indikasi bahwa halaman tersebut tidak akan lama berada di dalam memori."

Pada kasus pemindahan halaman di atas, bingkai kosong yang diperoleh akan digunakan sebagai tempat penyimpanan halaman yang salah.

Gambar 34.1. Ilustrasi Kondisi yang Memerlukan Pemindahan Halaman

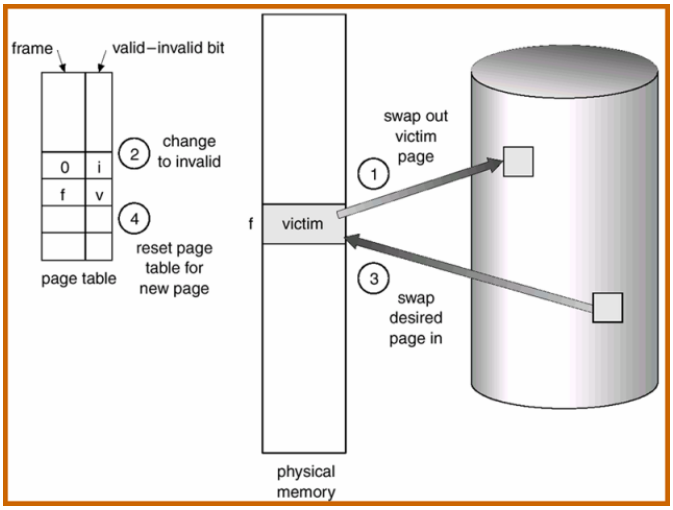


Rutinitas yang dilakukan dalam pemindahan halaman antara lain:

- Mencari lokasi dari halaman yang diinginkan pada *disk*.
- Mencari *frame* yang kosong:
  a. Jika ada, maka gunakan *frame* tersebut.
  b. Jika tidak ada, maka tentukan *frame* yang tidak sedang dipakai atau yang tidak akan digunakan dalam jangka waktu lama, lalu kosongkan *frame* tersebut. Gunakan algoritma pemindahan halaman untuk menentukan *frame* yang akan dikosongkan.

     Usahakan agar tidak menggunakan frame yang akan digunakan dalam waktu dekat. Jika terpaksa, maka sebaiknya segera masukkan kembali frame tersebut agar tidak terjadi *overhead*.
  c. Tulis halaman yang dipilih ke *disk*, ubah tabel halaman dan tabel *frame*.
- Membaca halaman yang diinginkan ke dalam *frame* kosong yang baru.
- Mengulangi proses pengguna dari awal.

Gambar 34.2. Pemindahan Halaman



Rutinitas di atas belum tentu berhasil. Jika kita tidak dapat menemukan *frame* yang kosong atau akan dikosongkan, maka sebagai jalan keluarnya kita dapat melakukan pentransferan dua halaman (satu masuk, satu keluar). Cara ini akan menambah waktu pelayanan kesalahan halaman dan waktu akses efektif. Oleh karena itu, perlu digunakan bit tambahan untuk masing-masing halaman dan *frame* yang diasosiasikan dalam perangkat keras.

Sebagai dasar dari permintaan halaman, pemindahan halaman merupakan "jembatan pemisah" antara memori logis dan memori fisik. Mekanisme yang dimilikinya memungkinkan memori virtual berukuran sangat besar dapat disediakan untuk pemrogram dalam bentuk memori fisik yang berukuran lebih kecil.

Dalam permintaan halaman, jika kita memiliki banyak proses dalam memori, kita harus menentukan jumlah *frame* yang akan dialokasikan ke masing-masing proses. Ketika pemindahan halaman diperlukan, kita harus memilih *frame* yang akan dipindahkan (dikosongkan). Masalah ini dapat diselesaikan dengan menggunakan algoritma pemindahan halaman.

Ada beberapa macam algoritma pemindahan halaman yang dapat digunakan. Algoritma yang terbaik adalah yang memiliki tingkat kesalahan halaman terendah. Selama jumlah *frame* meningkat, jumlah kesalahan halaman akan menurun. Peningkatan jumlah *frame* dapat terjadi jika memori fisik diperbesar.

Evaluasi algoritma pemindahan halaman dapat dilakukan dengan menjalankan sejumlah string acuan di memori dan menghitung jumlah kesalahan halaman yang terjadi. Sebagai contoh, suatu proses memiliki urutan alamat: 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105; per 100 *bytes*-nya dapat kita turunkan menjadi string acuan: 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.

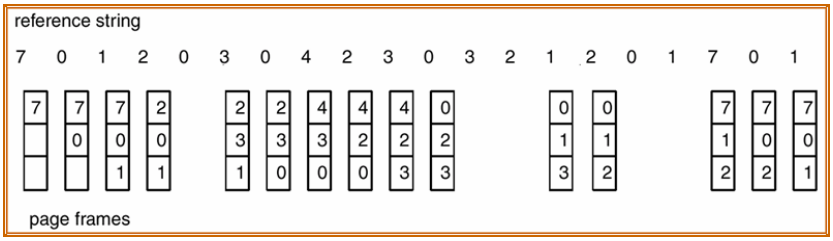
Pemindahan halaman diimplementasikan dalam algoritma yang bertujuan untuk menghasilkan tingkat kesalahan halaman terendah. Ada beberapa algoritma pemindahan halaman yang berbeda. Pemilihan halaman yang akan diganti dalam penggunaan algoritma-algoritma tersebut bisa dilakukan dengan berbagai cara, seperti dengan memilih secara acak, memilih dengan berdasarkan pada penggunaan, umur halaman, dan lain sebagainya. Pemilihan algoritma yang kurang tepat dapat menyebabkan peningkatan tingkat kesalahan halaman sehingga proses akan berjalan lebih lambat.

34.2. Algoritma *First In First Out*

Prinsip yang digunakan dalam algoritma FIFO yaitu menggunakan konsep antrian, halaman yang diganti adalah halaman yang paling lama berada di memori. Algoritma ini adalah algoritma pemindahan halaman yang paling mudah diimplementasikan, akan tetapi paling jarang digunakan

dalam keadaan sebenarnya. Biasanya penggunaan algoritma FIFO ini dikombinasikan dengan algoritma lain.

Gambar 34.3. Contoh Penerapan Algoritma FIFO



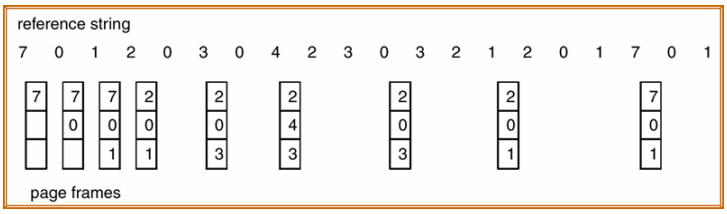
Pengimplementasian algoritma FIFO dilakukan dengan menggunakan antrian untuk menandakan halaman yang sedang berada di dalam memori. Setiap halaman baru yang diakses diletakkan di bagian belakang (ekor) dari antrian. Apabila antrian telah penuh dan ada halaman yang baru diakses maka halaman yang berada di bagian depan (kepala) dari antrian akan diganti.

Kelemahan dari algoritma FIFO adalah kinerjanya yang tidak selalu baik. Hal ini disebabkan karena ada kemungkinan halaman yang baru saja keluar dari memori ternyata dibutuhkan kembali. Di samping itu dalam beberapa kasus, tingkat kesalahan halaman justru bertambah seiring dengan meningkatnya jumlah *frame*, yang dikenal dengan nama anomali Belady.

34.3. Algoritma Optimal

Algoritma optimal pada prinsipnya akan mengganti halaman yang tidak akan digunakan untuk jangka waktu yang paling lama. Kelebihannya antara lain dapat menghindari terjadinya anomali Belady dan juga memiliki tingkat kesalahan halaman yang terendah diantara algoritma-algoritma pemindahan halaman yang lain.

Gambar 34.4. Contoh Algoritma Optimal



Meski pun tampaknya mudah untuk dijelaskan, tetapi algoritma ini sulit atau hampir tidak mungkin untuk diimplementasikan karena sistem operasi harus dapat mengetahui halaman-halaman mana saja yang akan diakses berikutnya, padahal sistem operasi tidak dapat mengetahui halaman yang muncul di waktu yang akan datang.

34.4. Algoritma *Least Recently Used* (LRU)

Cara kerja algoritma LRU adalah menggantikan halaman yang sudah tidak digunakan dalam jangka waktu yang paling lama. Pertimbangan algoritma ini yaitu berdasarkan observasi bahwa halaman yang sering diakses kemungkinan besar akan diakses kembali. Sama halnya dengan algoritma optimal, algoritma LRU juga tidak akan mengalami anomali Belady. Namun, sama halnya juga dengan algoritma optimal, algoritma LRU susah untuk diimplementasikan, walaupun sedikit lebih mudah dari algoritma optimal. Pengimplementasian algoritma LRU dapat dilakukan dengan dua cara.

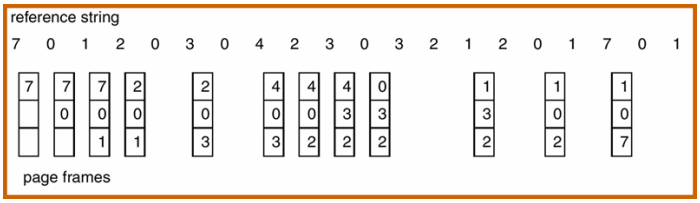
Counter

Cara ini dilakukan dengan menggunakan counter atau logical clock. Setiap halaman memiliki nilai yang pada awalnya diinisialisasi dengan 0. Ketika mengakses ke suatu halaman baru, nilai pada clock di halaman tersebut akan bertambah 1. Untuk melakukan hal itu dibutuhkan extra write ke memori. Halaman yang diganti adalah halaman yang memiliki nilai clock terkecil. Kekurangan dari cara ini adalah hanya sedikit sekali mesin yang mendukung hardware counter.

Stack

Cara ini dilakukan dengan menggunakan stack yang menandakan halaman-halaman yang berada di memori. Setiap kali suatu halaman diakses, akan diletakkan di bagian paling atas stack. Apabila ada halaman yang perlu diganti, maka halaman yang berada di bagian paling bawah stack akan diganti sehingga setiap kali halaman baru diakses tidak perlu mencari kembali halaman yang akan diganti. Dibandingkan pengimplementasian dengan counter, cost untuk mengimplementasikan algoritma LRU dengan menggunakan stack akan lebih mahal karena isi stack harus di-update setiap kali mengakses halaman.

Gambar 34.5. Algoritma LRU



34.5. Algoritma Perkiraan LRU

Pada dasarnya algoritma perkiraan LRU memiliki prinsip yang sama dengan algoritma LRU, yaitu halaman yang diganti adalah halaman yang tidak digunakan dalam jangka waktu terlama, hanya saja dilakukan modifikasi pada algoritma ini untuk mendapatkan hasil yang lebih baik. Perbedaannya dengan algoritma LRU terletak pada penggunaan bit acuan. Setiap halaman yang berbeda memiliki bit acuan. Pada awalnya bit acuan diinisialisasikan oleh perangkat keras dengan nilai 0. Nilainya akan berubah menjadi 1 bila dilakukan akses ke halaman tersebut. Terdapat beberapa cara untuk mengimplementasikan algoritma ini

Algoritma NFU (*Not Frequently Used*)

Pada prinsipnya algoritma LRU dapat diimplementasikan, jika ada mesin yang menyediakan hardware counter atau stack seperti yang digunakan pada LRU. Pendekatan lainnya adalah dengan mengimplementasikannya pada software. Salah satu solusinya disebut algoritma NFU (Not Frequently Used).

Seperti LRU, NFU juga menggunakan counter tapi counter ini diimplementasikan pada software. Setiap halaman memiliki counter, yang diinisiasi dengan nol. Pada setiap clock interval, semua halaman yang pernah diakses pada interval tersebut akan ditambah nilai counter-nya dengan bit R, yang bernilai 0 atau 1. Maka, halaman dengan nilai counter terendah akan diganti.

Kelemahan NFU adalah NFU menyimpan informasi tentang sering- tidaknya sebuah halaman diakses tanpa mempertimbangkan rentang waktu penggunaan. Jadi, halaman yang sering diakses pada pass pertama akan dianggap sebagai halaman yang sangat dibutuhkan pada pass kedua, walaupun halaman tersebut tidak lagi dibutuhkan pada pass kedua karena nilai counter-nya tinggi. Hal ini dapat mengurangi kinerja algoritma ini.

Setiap halaman akan memiliki bit acuan yang terdiri dari 8 *bit* (*byte*) sebagai penanda. Pada awalnya semua bit nilainya 0, contohnya: 00000000. Setiap selang beberapa waktu, pencatat waktu melakukan interupsi kepada sistem operasi, kemudian sistem operasi menggeser 1 bit ke kanan

dengan bit yang paling kiri adalah nilai dari bit acuan, yaitu bila halaman tersebut diakses nilainya 1 dan bila tidak diakses nilainya 0. Jadi halaman yang selalu digunakan pada setiap periode akan memiliki nilai 11111111. Halaman yang diganti adalah halaman yang memiliki nilai terkecil.

Algoritma *Aging*

Algoritma Aging adalah turunan dari NFU yang mempertimbangkan rentang waktu penggunaan suatu halaman. Tidak seperti NFU yang hanya menambahkan bit R pada counter, algoritma ini menggeser bit counter ke kanan (dibagi 2) dan menambahkan bit R di paling kiri bit counter. Halaman yang akan diganti adalah halaman yang memiliki nilai counter terendah.

Contoh, jika suatu halaman diakses pada interval pertama, tidak diakses dalam dua interval selanjutnya, diakses pada dua interval berikutnya, tidak diakses dalam interval berikutnya, dan seterusnya, maka bit R dari halaman itu: 1, 0, 0, 1, 1, 0, ... Dengan demikian counter-nya akan menjadi: 10000000, 01000000, 00100000, 10010000, 11001000, 01100100, ...

Algoritma ini menjamin bahwa halaman yang paling baru diakses, walaupun tidak sering diakses, memiliki prioritas lebih tinggi dibanding halaman yang sering diakses sebelumnya. Yang perlu diketahui, aging dapat menyimpan informasi pengaksesan halaman sampai 16 atau 32 interval sebelumnya. Hal ini sangat membantu dalam memutuskan halaman mana yang akan diganti. Dengan demikian, aging menawarkan kinerja yang mendekati optimal dengan harga yang cukup rendah.

Algoritma *Second-Chance*

Algoritma ini adalah modifikasi dari FIFO yang, seperti namanya, memberikan kesempatan kedua bagi suatu halaman untuk tetap berada di dalam memori karena halaman yang sudah lama berada di memori mungkin saja adalah halaman yang sering digunakan dan akan digunakan lagi. Kesempatan kedua itu direalisasikan dengan adanya bit acuan yang di-set untuk suatu halaman.

Penempatan halaman pada antrian sama seperti pada FIFO, halaman yang lebih dulu diakses berada di depan antrian dan yang baru saja diakses berada di belakang antrian. Ketika terjadi kesalahan halaman, algoritma ini tidak langsung mengganti halaman di depan antrian tapi terlebih dahulu memeriksa bit acuannya. Jika bit acuan = 0, halaman tersebut akan langsung diganti. Jika bit acuan = 1, halaman tersebut akan dipindahkan ke akhir antrian dan bit acuannya diubah menjadi 0, kemudian mengulangi proses ini untuk halaman yang sekarang berada di depan antrian.

Algoritma ini dapat terdegenerasi menjadi FIFO ketika semua bit acuan di-set 1. Ketika halaman pertama kembali berada di awal antrian, bit acuan pada semua halaman sudah diubah menjadi 0, sehingga halaman pertama langsung diganti dengan halaman baru.

Proses pemindahan halaman yang diberi kesempatan kedua akan membuat algoritma menjadi tidak efisien karena harus terus memindahkan halaman dalam antrian. Untuk mengatasi masalah ini, digunakan antrian berbentuk lingkaran yang dijelaskan pada algoritma di bawah ini.

Pengimplementasian algoritma ini dilakukan dengan menyimpan halaman pada sebuah *linked-list* dan halaman-halaman tersebut diurutkan berdasarkan waktu ketika halaman tersebut tiba di memori yang berarti menggunakan juga prinsip algoritma FIFO disamping menggunakan algoritma LRU. Apabila nilai bit acuan-nya 0, halaman dapat diganti. Dan apabila nilai bit acuan-nya 1, halaman tidak diganti tetapi bit acuan diubah menjadi 0 dan dilakukan pencarian kembali.

Algoritma *Clock*

Seperti yang disebutkan di atas, algoritma Clock Halaman (atau Clock saja) menggunakan prinsip Second-Chance tapi dengan antrian yang berbentuk melingkar.

Pada antrian ini terdapat pointer yang menunjuk ke halaman yang paling lama berada di antrian. Ketika terjadi kesalahan halaman, halaman yang ditunjuk oleh pointer akan diperiksa bit acuannya seperti pada Second-Chance. Jika bit acuan = 0, halaman tersebut akan langsung diganti. Jika bit acuan = 1, bit acuannya diubah menjadi 0 dan pointer akan bergerak searah jarum jam ke halaman yang berada di sebelahnya.

Penggunaan antrian berbentuk melingkar menyebabkan halaman tidak perlu dipindahkan setiap saat, yang bergerak cukup pointer saja. Meski pun algoritma *second-chance* sudah cukup baik, namun pada kenyataannya penggunaan algortima tersebut tidak efisien. Algoritma *clock* adalah penyempurnaan dari algoritma tersebut. Pada prinsipnya kedua algoritma tersebut sama, hanya terletak perbedaan pada pengimplementasiannya saja. Algortima ini menggunakan antrian melingkar yang berbentuk seperti jam dengan sebuah penunjuk yang akan berjalan melingkar mencari halaman untuk diganti.

34.6. Algoritma *Counting*

Dilakukan dengan menyimpan pencacah dari nomor acuan yang sudah dibuat untuk masing-masing halaman. Penggunaan algoritma ini memiliki kekurangan yaitu lebih mahal. Algoritma *Counting* dapat dikembangkan menjadi dua skema dibawah ini:

- **Algoritma *Least Frequently Used* (LFU)**

  Halaman yang diganti adalah halaman yang paling sedikit dipakai (nilai pencacah terkecil) dengan alasan bahwa halaman yang digunakan secara aktif akan memiliki nilai acuan yang besar.
- **Algoritma *Most Frequently Used* (MFU)**

  Halaman yang diganti adalah halaman yang paling sering dipakai (nilai pencacah terbesar) dengan alasan bahwa halaman dengan nilai terkecil mungkin baru saja dimasukkan dan baru digunakan

34.7. Algoritma NRU (Not Recently Used)

Dalam algoritma ini terdapat bit acuan dan bit modifikasi yang akan di-update setiap kali mengakses halaman. Ketika terjadi halaman fault, sistem operasi akan memeriksa semua halaman dan membagi halaman-halaman tersebut ke dalam kelas-kelas. Algoritma NRU mudah untuk dimengerti, efisien, dan memiliki kinerja yang cukup baik. Algoritma ini mempertimbangkan dua hal sekaligus, yaitu bit acuan dan bit modifikasi. Ketika terjadi kesalahan halaman, sistem operasi memeriksa semua halaman dan membagi halaman-halaman tersebut ke dalam empat kelas:

- Kelas 1: Tidak digunakan dan tidak dimodifikasi, Bit terbaik untuk dipindahkan.
- Kelas 2: Tidak digunakan tapi dimodifikasi, tidak terlalu baik untuk dipindahkan karena halaman ini perlu ditulis sebelum dipindahkan.
- Kelas 3: Digunakan tapi tidak dimodifikasi, ada kemungkinan halaman ini akan segera digunakan lagi.
- Kelas 4: Digunakan dan dimodifikasi, halaman ini mungkin akan segera digunakan lagi dan halaman ini perlu ditulis ke disk sebelum dipindahkan.

Halaman yang akan diganti adalah halaman dari kelas yang paling rendah. Jika terdapat lebih dari satu kelas terendah yang sama, maka sistem operasi akan memilih secara acak. Kelebihan algoritma ini adalah mudah untuk dimengerti, efisien, dan memiliki kinerja yang cukup baik.

34.8. Algoritma *Page Buffering*

Pada algoritma ini, sistem menyimpan pool dari bingkai yang kosong dan dijadikan sebagai buffer. Prosedur ini memungkinkan suatu proses mengulang dari awal secepat mungkin, tanpa perlu menunggu halaman yang akan dipindahkan untuk ditulis ke disk karena bingkai-nya telah ditambahkan ke dalam pool bingkai kosong. Teknik seperti ini digunakan dalam sistem VAX/VMS. Kelebihan algoritma ini adalah dapat membuat proses lebih cepat dikembalikan ke status ready queue, namun akan terjadi penurunan kinerja karena sedikit sekali halaman yang digunakan sehingga halaman lain yang tidak digunakan menjadi sia-sia (utilitas sistem rendah).

34.9. Rangkuman

Pendekatan untuk pemindahan halaman: "Jika tidak ada bingkai yang kosong, cari bingkai yang tidak sedang digunakan dalam jangka waktu yang lama, lalu kosongkan dengan memindahkan isinya ke dalam ruang pertukaran dan ubah semua tabel halamannya sebagai indikasi bahwa

halaman tersebut tidak akan lama berada di dalam memori."

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBAmirSch2000] Yair Amir dan Theo Schlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

[WEBWiki2006f] From Wikipedia, the free encyclopedia. 2006. *Page replacement algorithms – http://en.wikipedia.org/ wiki/ Page_replacement_algorithms* . Diakses 04 Juli 2006.

Bab 35. Strategi Alokasi Frame

35.1. Pendahuluan

Hal yang mendasari strategi alokasi *frame* yang menyangkut memori virtual adalah bagaimana membagi memori yang bebas untuk beberapa proses yang sedang dikerjakan. Dapat kita ambil satu contoh yang mudah pada sistem satu pemakai. Misalnya sebuah sistem mempunyai seratus *frame* bebas untuk proses *user*. Untuk permintaan halaman murni, keseratus *frame* tersebut akan ditaruh pada daftar *frame* bebas. Ketika sebuah *user* mulai dijalankan, akan terjadi sederetan kesalahan halaman. Sebanyak seratus kesalahan halaman pertama akan mendapatkan *frame* dari daftar *frame* bebas. Saat *frame* bebas sudah habis, sebuah algoritma pergantian halaman akan digunakan untuk memilih salah satu dari seratus halaman di memori yang diganti dengan yang ke seratus satu, dan seterusnya. Ketika proses selesai atau diterminasi, seratus *frame* tersebut akan disimpan lagi pada daftar *frame* bebas.

Ada beberapa variasi untuk strategi sederhana ini antara lain kita bisa meminta sistem operasi untuk mengalokasikan seluruh *buffer* dan ruang tabelnya dari daftar *frame* bebas. Saat ruang ini tidak digunakan oleh sistem operasi, ruang ini bisa digunakan untuk mendukung permintaan halaman dari *user*. Kita juga dapat menyimpan tiga *frame* bebas dari daftar *frame* bebas, sehingga ketika terjadi kesalahan halaman, ada *frame* bebas yang dapat digunakan untuk permintaan halaman. Saat penggantian halaman terjadi, penggantinya dapat dipilih, kemudian ditulis ke *disk*, sementara proses *user* tetap berjalan.

Pada dasarnya yaitu proses pengguna diberikan *frame* bebas yang mana saja. Masalah yang muncul ketika penggantian halaman dikombinasikan dengan *multiprogramming*. Hal ini terjadi karena *multiprogramming* menaruh dua (atau lebih) proses di memori pada waktu yang bersamaan.

Jumlah *Frame* Minimum

Ada suatu batasan dalam mengalokasikan *frame*, kita tidak dapat mengalokasikan *frame* lebih dari jumlah *frame* yang ada. Hal yang utama adalah berapa minimum *frame* yang harus dialokasikan agar jika sebuah instruksi dijalankan, semua informasinya ada dalam memori. Jika terjadi kesalahan halaman sebelum eksekusi selesai, instruksi tersebut harus diulang. Sehingga kita harus mempunyai jumlah *frame* yang cukup untuk menampung semua halaman yang dibutuhkan oleh sebuah instruksi.

Jumlah *frame* mimimum yang bisa dialokasikan ditentukan oleh arsitektur komputer. Sebagai contoh, instruksi *move* pada PDP-11 adalah lebih dari satu kata untuk beberapa modus pengalamatan, sehingga instruksi tersebut bisa membutuhkan dua halaman. Sebagai tambahan, tiap operannya mungkin merujuk tidak langsung, sehingga total ada enam *frame*. Kasus terburuk untuk IBM 370 adalah instruksi MVC. Karena instruksi tersebut adalah instruksi perpindahan dari penyimpanan ke penyimpanan, instruksi ini butuh 6 bit dan dapat memakai dua halaman. Satu blok karakter yang akan dipindahkan dan daerah tujuan perpindahan juga dapat memakai dua halaman, sehingga situasi ini membutuhkan enam *frame*.

Algoritma Alokasi

Jumlah minimum dari bingkai untuk setiap proses di definisikan oleh arsitektur komputer, sedangkan maksimum bingkai didefinisikan oleh banyaknya memori fisik yang terdapat di komputer. Sedangkan sistem operasi diberikan kebebasan untuk mengatur pengalokasikan bingkai untuk suatu proses.

Algoritma pertama yaitu equal allocation. Algoritma ini memberikan bagian yang sama, sebanyak m/n bingkai untuk tiap proses. Sebagai contoh ada 100 bingkai tersisa dan lima proses, maka tiap proses akan mendapatkan 20 bingkai. Jika ada sebuah proses sebesar 10K. Sebuah proses basis data 127K dan hanya kedua proses ini yang berjalan pada sistem, maka ketika ada 62 bingkai bebas, tidak masuk akal jika kita memberikan masing-masing proses 31 bingkai. Bingkai tersisa, sebanyak tiga buah dapat digunakan sebagai bingkai bebas cadangan dan ternyata proses pertama cukup hanya mengunakan 10 bingkai dan 21 tidak dipergunakan dan terbuang percuma.

Algoritma kedua yaitu proportional allocation sebuah alternatif yaitu mengunakan alokasi memori yang tersedia kepada setiap proses dengan melihat pada besarnya ukuran proses. Ukuran memori virtual untuk proses pi = si, dan S = si jumlah proses. Lalu jumlah total dari bingkai yang tersedia adalah m, kita mengalokasikan proses ai ke proses pi, dimana ai mendekati:

si = ukuran besarnya proses pi

S = jumlah total dari si

m = jumlah bingkai

ai = alokasi bingkai pi= si/S x m

misal:

m = 64

s1 = 10

s2 = 127

a1= 10/137 x 64 = 5 bingkai

a2 = 127/137 x 64 =59 bingkai

Dalam kedua strategi ini, tentu saja, alokasi untuk setiap proses bisa bervariasi berdasarkan *multiprogramming* level-nya. Jika *multiprogramming* level-nya meningkat, setiap proses akan kehilangan beberapa *frame* guna menyediakan memori yang dibutuhkan untuk proses yang baru. Di sisi lain, jika *multiprogramming* level-nya menurun, *frame* yang sudah dialokasikan pada bagian proses sekarang bisa disebar ke proses-proses yang masih tersisa.

Mengingat hal itu, dengan *equal allocation* atau pun *proportional allocation*, proses yang berprioritas tinggi diperlakukan sama dengan proses yang berprioritas rendah. Berdasarkan definisi tersebut, bagaimana pun juga, kita ingin memberi memori yang lebih pada proses yang berprioritas tinggi untuk mempercepat eksekusi-nya.

Satu pendekatan adalah menggunakan skema *proportional allocation* dimana perbandingan *frame*-nya tidak tergantung pada ukuran relatif dari proses, melainkan lebih pada prioritas proses, atau tergantung kombinasi dari ukuran dan prioritas. Algoritma ini dinamakan alokasi prioritas.

Alokasi Global dan Lokal

Hal penting lainnya dalam pengalokasian *frame* adalah pergantian halaman. Proses-proses bersaing mendapatkan *frame*, maka dari itu kita dapat mengklasifikasikan algoritma penggantian halaman kedalam dua kategori; Penggantian Global dan Penggantian Lokal. Pergantian global memperbolehkan sebuah proses mencari *frame* pengganti dari semua *frame-frame* yang ada, walaupun *frame* tersebut sedang dialokasikan untuk proses yang lain. Hal ini memang efisien. tetapi ada kemungkinan proses lain tidak mendapatkan *frame* karena framenya terambil oleh proses lain. Penggantian lokal memberi aturan bahwa setiap proses hanya boleh memilih *frame* pengganti dari *frame-frame* yang memang dialokasikan untuk proses itu sendiri.

Sebagai contoh, misalkan ada sebuah skema alokasi yang memperbolehkan proses berprioritas tinggi untuk mencari *frame* pengganti dari proses yang berprioritas rendah. Proses berprioritas tinggi ini dapat mancari *frame* pengganti dari *frame-frame* yang telah dialokasikan untuknya atau dari *frame-frame* yang dialokasikan untuk proses berprioritas lebih rendah.

Penggantian lokal memberi aturan bahwa setiap proses hanya boleh memilih bingkai pengganti dari frame-frame yang memang dialokasikan untuk proses itu sendiri. Misalkan ada sebuah skema alokasi yang memperbolehkan proses berprioritas tinggi untuk mencari bingkai pengganti dari proses yang berprioritas rendah. Proses berprioritas tinggi ini dapat mancari bingkai pengganti dari frame- bingkai yang telah dialokasikan atau frame-frame yang dialokasikan untuk proses berprioritas lebih rendah. Dalam penggantian lokal, jumlah bingkai yang teralokasi tidak berubah.

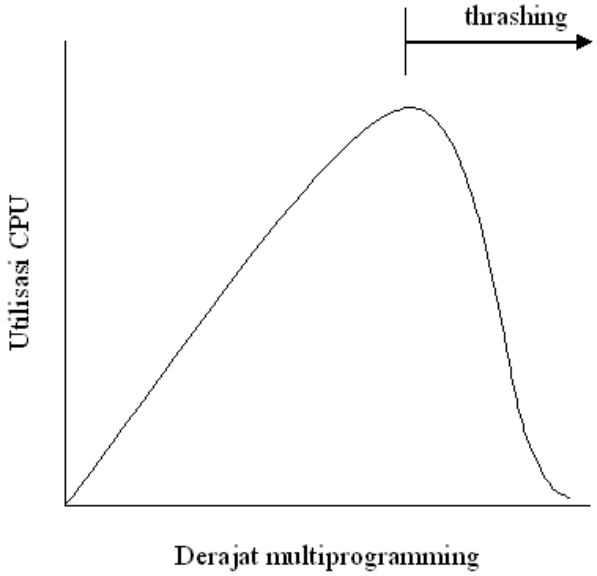
Dengan Penggantian Global ada kemungkinan sebuah proses hanya menyeleksi bingkai-frame yang teralokasi pada proses lain, sehingga meningkatkan jumlah bingkai yang teralokasi pada proses itu sendiri (asumsi bahwa proses lain tidak memilih bingkai proses tersebut untuk penggantian). Masalah pada algoritma Penggantian Global adalah bahwa sebuah proses tidak bisa mengontrol kasalahan halaman-nya sendiri. Halaman-halaman dalam memori untuk sebuah proses tergantung tidak hanya pada prilaku penghalamanan dari proses tersebut, tetapi juga pada prilaku penghalamanan dari proses lain. Karena itu, proses yang sama dapat tampil berbeda (memerlukan 0,5 detik untuk satu eksekusi dan 10,3 detik untuk eksekusi berikutnya).

Dalam Penggantian Lokal, halaman-halaman dalam memori untuk sebuah proses hanya dipengaruhi prilaku penghalamanan proses itu sendiri. Penggantian Lokal dapat menyembunyikan sebuah proses dengan membuatnya tidak tersedia bagi proses lain, menggunakan halaman yang lebih sedikit pada memori. Jadi, secara umum Penggantian Global menghasilkan sistem throughput yang lebih bagus, oleh sebab itu metoda ini yang paling sering digunakan.

35.2. *Thrashing*

Thrashing adalah keadaan dimana terdapat aktifitas yang tinggi dari penghalamanan. Aktifitas penghalamanan yang tinggi ini maksudnya adalah pada saat sistem sibuk melakukan *swap-in* dan *swap-out* dikarenakan banyak kasalahan halaman yang terjadi. Suatu proses dapat mengurangi jumlah *frame* yang digunakan dengan alokasi yang minimum. Tetapi jika sebuah proses tidak memiliki *frame* yang cukup, tetap ada halaman dalam jumlah besar yang memiliki kondisi aktif digunakan. Maka hal ini mengakibatkan kasalahan halaman. Untuk seterusnya sistem harus mengganti beberapa halaman menjadi halaman yang akan dibutuhkan. Karena semua halamannya aktif digunakan, maka halaman yang diganti adalah halaman yang dalam waktu dekat berkemungkinan akan digunakan kembali. Hal ini mengakibatkan kesalahan halaman yang terus-menerus

Gambar 35.1. Derajat dari *Multiprogramming*



Penyebab *Thrashing*

Utilitas dari CPU selalu diharapkan tinggi hingga mendekati 100%. Jika proses yang dikerjakan CPU hanya sedikit, maka kita tidak bisa menjaga agar CPU sibuk. Utilitas dari CPU bisa ditingkatkan dengan meningkatkan jumlah proses. Jika Utilitas CPU rendah, maka sistem akan menambah derajat dari *multiprogramming* yang berarti menambah jumlah proses yang sedang

berjalan. Pada titik tertentu, menambah jumlah proses justru akan menyebabkan utilitas CPU turun drastis dikarenakan proses-proses yang baru tidak mempunya memori yang cukup untuk berjalan secara efisien. Pada titik ini terjadi aktifitas penghalamanan yang tinggi yang akan menyebabkan *thrashing*.

Ketika sistem mendeteksi bahwa utilitas CPU menurun dengan bertambahnya proses, maka sistem meningkatkan lagi derajat dari *multiprogramming*. Proses-proses yang baru berusaha merebut frame-frame yang telah dialokasikan untuk proses yang sedang berjalan. Hal ini mengakibatkan kesalahan halaman meningkat tajam. Utilitas CPU akan menurun dengan sangat drastis diakibatkan oleh sistem yang terus menerus menambah derajat *multiprogramming*.

Pada gambar di bawah ini tergambar grafik dimana utilitas dari CPU akan terus meningkat seiiring dengan meningkatnya derajat dari *multiprogramming* hingga sampai pada suatu titik saat utilitas CPU menurun drastis. Pada titik ini, untuk menghentikan *thrashing*, derajat dari *multiprogramming* harus diturunkan.

35.3. Membatasi Efek *Thrashing*

Efek dari *thrashing* dapat dibatasi dengan algoritma pergantian lokal atau prioritas. Dengan pergantian lokal, jika satu proses mulai *thrashing*, proses tersebut dapat mengambil *frame* dari proses yang lain dan menyebabkan proses itu tidak langsung *thrashing*. Jika proses mulai *thrashing*, proses itu akan berada pada antrian untuk melakukan penghalamanan yang mana hal ini memakan banyak waktu. Rata-rata waktu layanan untuk kesalahan halaman akan bertambah seiring dengan makin panjangnya rata-rata antrian untuk melakukan penghalamanan. Maka, waktu akses efektif akan bertambah walaupun untuk suatu proses yang tidak *thrashing*.

Salah satu cara untuk menghindari *thrashing*, kita harus menyediakan sebanyak mungkin *frame* sesuai dengan kebutuhan suatu proses. Cara untuk mengetahui berapa *frame* yang dibutuhkan salah satunya adalah dengan strategi *Working Set*.

Selama satu proses di eksekusi, model lokalitas berpindah dari satu lokalitas satu ke lokalitas lainnnya. Lokalotas adalah kumpuulan halaman yang aktif digunakan bersama. Suatu program pada umumnya dibuat pada beberapa lokalitas sehingga ada kemungkinan terjadi *overlap*. *Thrashing* dapat muncul bila ukuran lokalitas lebih besar dari ukuran memori total.

Model *Working Set*

Strategi *Working set* dimulai dengan melihat berapa banyak *frame* yang sesungguhnya digunakan oleh suatu proses. *Working set model* mengatakan bahwa sistem hanya akan berjalan secara efisien jika masing-masing proses diberikan jumlah halaman *frame* yang cukup. Jika jumlah *frame* tidak cukup untuk menampung semua proses yang ada, maka akan lebih baik untuk menghentikan satu proses dan memberikan halamannya untuk proses yang lain.

Working set model merupakan model lokalitas dari suatu eksekusi proses. Model ini menggunakan parameter Δ (delta) untuk mendefinisikan *working set window*. Untuk menentukan halaman yang dituju, yang paling sering muncul. Kumpulan dari halaman dengan Δ halaman yang dituju yang paling sering muncul disebut *working set*. *Working set* adalah pendekatan dari program lokalitas.

Keakuratan *working set* tergantung pada pemilihan Δ.

1. Jika Δ terlalu kecil, tidak akan dapat mewakilkan keseluruhan dari lokalitas.
2. Jika Δ terlalu besar, akan menyebabkan *overlap* beberapa lokalitas.
3. Jika Δ tidak terbatas, *working set* adalah kumpulan *page* sepanjang eksekusi program.

Jika kita menghitung ukuran dari *Working Set*, WWSi, untuk setiap proses pada sistem, kita hitung dengan D = WSSi, dimana D merupakan total *demand* untuk *frame*.

Jika total perminataan lebih dari total banyaknya *frame* yang tersedia (D > m), *thrashing* dapat terjadi karena beberapa proses akan tidak memiliki *frame* yang cukup. Jika hal tersebut terjadi, dilakukan satu pengeblokan dari proses-proses yang sedang berjalan.

Strategi *Working Set* menangani *thrashing* dengan tetap mempertahankan derajat dari

multiprogramming setinggi mungkin.

Contoh: △ = 1000 referensi, Penghitung interupsi setiap 5000 referensi.

Ketika kita mendapat interupsi, kita menyalin dan menghapus nilai bit referensi dari setiap halaman. Jika kesalahan halaman muncul, kita dapat menentukan bit referensi sekarang dan 2 pada bit memori untuk memutuskan apakah halaman itu digunakan dengan 10000 ke 15000 referensi terakhir.

Jika digunakan, paling sedikit satu dari bit-bit ini akan aktif. Jika tidak digunakan, bit ini akan menjadi tidak aktif.

Halaman yang memiliki paling sedikit 1 bit aktif, akan berada di *working-set*.

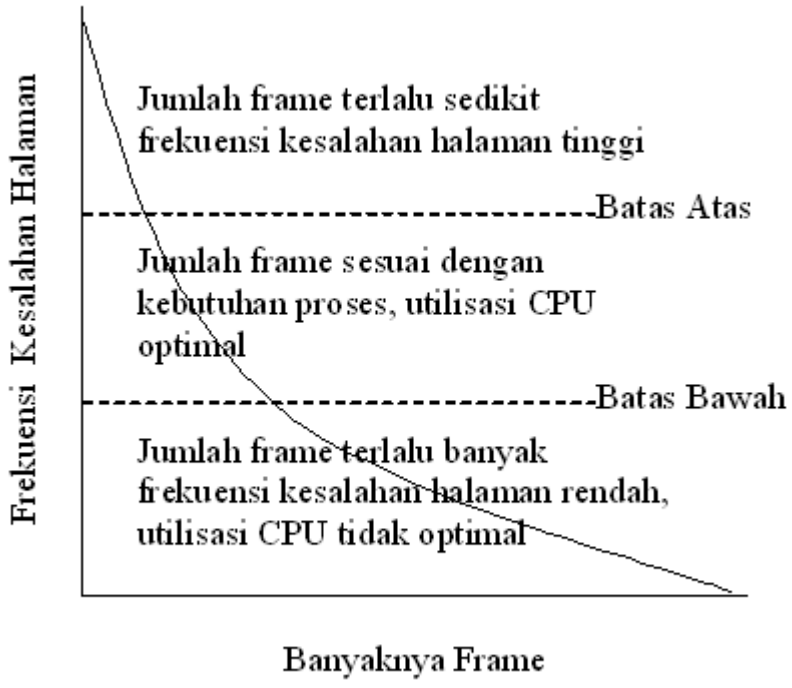
Hal ini tidaklah sepenuhnya akurat karena kita tidak dapat memberitahukan dimana pada interval 5000 tersebut, referensi muncul. Kita dapat mengurangi ketidakpastian dengan menambahkan sejarah bit kita dan frekuensi dari interupsi.

Contoh: 20 bit dan interupsi setiap 1500 referensi.

Frekuensi Kesalahan Halaman

Working-set dapat berguna untuk *prepaging*, tetapi kurang dapat mengontrol *thrashing*. Strategi menggunakan frekuensi kesalahan halaman mengambil pendekatan yang lebih langsung.

Gambar 35.2. Frekuensi-kesalahan-halaman



Thrashing memiliki kecepatan kasalahan halaman yang tinggi. Kita ingin mengontrolnya. Ketika terlalu tinggi, kita mengetahui bahwa proses membutuhkan *frame* lebih. Sama juga, jika terlalu rendah, maka proses mungkin memiliki terlalu banyak *frame*. Kita dapat menentukan batas atas dan bawah pada kecepatan kesalahan halaman seperti terlihat pada gambar berikut ini.

Jika kecepatan kasalahan halaman yang sesungguhnya melampaui batas atas, kita mengalokasikan

frame lain ke proses tersebut, sedangkan jika kecepatan kasalahan halaman di bawah batas bawah, kita pindahkan *frame* dari proses tersebut. Maka kita dapat secara langsung mengukur dan mengontrol kecepatan kasalahan halaman untuk mencegah *thrashing*.

35.4. *Prepaging*

Pemilihan algoritma penggantian dan aturan alokasi adalah keputusan-keputusan utama yang kita buat untuk sistem pemberian halaman. Selain itu, masih banyak pertimbangan lain.

Sebuah ciri dari sistem permintaan pemberian halaman murni adalah banyaknya kesalahan halaman yang terjadi saat proses dimulai. Situasi ini merupakan hasil dari percobaan untuk mendapatkan tempat pada awalnya. Situasi yang sama mungkin muncul di lain waktu. Misalnya saat proses *swapped-out* dimulai kembali, seluruh halaman ada di cakram dan setiap halaman harus dibawa ke dalam memori yang akan mengakibatkan banyaknya kesalahan halaman. *Prepaging* mencoba untuk mencegah pemberian halaman awal tingkat tinggi ini. Strateginya adalah untuk membawa seluruh halaman yang akan dibutuhkan pada satu waktu ke memori.

Sebagai contoh, pada sistem yang menggunakan model *working set*, untuk setiap proses terdapat daftar dari semua halaman yang ada pada *working set*nya. Jika kita harus menunda sebuah proses (karena menunggu M/K atau kekurangan *frame* bebas), daftar *working set* untuk proses tersebut disimpan. Saat proses itu akan dilanjutkan kembali (permintaan M/K telah terpenuhi atau *frame* bebas yang cukup), secara otomatis seluruh *working set*-nya akan dibawa ke dalam memori sebelum memulai kembali proses tersebut.

Prepaging dapat berguna pada beberapa kasus. Yang harus dipertimbangkan adalah apakah biaya menggunakan *prepaging* lebih sedikit dari biaya menangani kesalahan halaman yang terjadi bila tidak memakai *prepaging*. Jika biaya *prepaging* lebih sedikit (karena hampir seluruh halaman yang di *prepage* digunakan) maka *prepaging* akan berguna. Sebaliknya, jika biaya *prepaging* lebih besar (karena hanya sedikit halaman dari yang di-*prepage* digunakan) maka *prepaging* akan merugikan.

35.5. Ukuran Halaman

Para perancang sistem operasi untuk mesin yang sudah ada jarang memiliki pilihan terhadap ukuran halaman. Akan tetapi, saat merancang sebuah mesin baru, harus dipertimbangkan berapa ukuran halaman yang terbaik. Pada dasarnya tidak ada ukuran halaman yang paling baik, karena banyaknya faktor-faktor yang mempengaruhinya.

Salah satu faktor adalah ukuran tabel halaman. Untuk sebuah memori virtual dengan ukuran 4 *megabytes* (2^22), akan ada 4.096 halaman berukuran 1.024 *bytes*, tapi hanya 512 halaman jika ukuran halaman 8.192 *bytes*. Setiap proses yang aktif harus memiliki salinan dari tabel halaman-nya, jadi lebih masuk akal jika dipilih ukuran halaman yang besar.

Di sisi lain, pemanfaatan memori lebih baik dengan halaman yang lebih kecil. Jika sebuah proses dialokasikan di memori, mengambil semua halaman yang dibutuhkannya, mungkin proses tersebut tidak akan berakhir pada batas dari halaman terakhir. Jadi, ada bagian dari halaman terakhir yang tidak digunakan walaupun telah dialokasikan. Asumsikan rata-rata setengah dari halaman terakhir tidak digunakan, maka untuk halaman dengan ukuran 256 *bytes* hanya akan ada 128 *bytes* yang terbuang, bandingkan dengan halaman berukuran 8192 *bytes*, akan ada 4096 *bytes* yang terbuang. Untuk meminimalkan pemborosan ini, kita membutuhkan ukuran halaman yang kecil.

Masalah lain adalah waktu yang dibutuhkan untuk membaca atau menulis halaman. Waktu M/K terdiri dari waktu pencarian, *latency* dan transfer. Waktu transfer sebanding dengan jumlah yang dipindahkan (yaitu, ukuran halaman). Sedangkan waktu pencarian dan *latency* biasanya jauh lebih besar dari waktu transfer. Untuk laju pemindahan 2 MB/s, hanya dihabiskan 0.25 millidetik untuk memindahkan 512 *bytes*. Waktu *latency* mungkin sekitar 8 millidetik dan waktu pencarian 20 millidetik. Total waktu M/K 28.25 milidetik. Waktu transfer sebenarnya tidak sampai 1%. Sebagai perbandingan, untuk mentransfer 1024 *bytes*, dengan ukuran halaman 1024 *bytes* akan dihabiskan waktu 28.5 milidetik (waktu transfer 0.5 milidetik). Namun dengan halaman berukuran 512 *bytes* akan terjadi 2 kali transfer 512 *bytes* dengan masing-masing transfer menghabiskan waktu 28.25 milidetik sehingga total waktu yang dibutuhkan 56.5 milidetik. Kesimpulannya, untuk meminimalisasi waktu M/K dibutuhkan ukuran halaman yang lebih besar.

Masalah terakhir yang akan dibahas disini adalah mengenai kesalahan halaman. Misalkan ukuran halaman adalah 1 *byte*. Sebuah proses sebesar 100 KB, dimana hanya setengahnya yang menggunakan memori, akan menghasilkan kesalahan halaman sebanyak 51200. Sedangkan bila ukuran halaman sebesar 200 KB maka hanya akan terjadi 1 kali kesalahan halaman. Jadi untuk mengurangi jumlah kesalahan halaman dibutuhkan ukuran halaman yang besar.

Masih ada faktor lain yang harus dipertimbangkan (misalnya hubungan antara ukuran halaman dengan ukuran sektor pada peranti pemberian halaman). Tidak ada jawaban yang pasti berapa ukuran halaman yang paling baik. Sebagai acuan, pada 1990, ukuran halaman yang paling banyak dipakai adalah 4096 *bytes*. Sedangkan sistem modern saat ini menggunakan ukuran halaman yang jauh lebih besar dari itu.

35.6. Jangkauan *TLB*

Hit ratio dari *TLB* adalah persentasi alamat virtual yang diselesaikan dalam *TLB* daripada di tabel halaman. *Hit ratio* sendiri berhubungan dengan jumlah masukan dalam *TLB* dan cara untuk meningkatkan *hit ratio* adalah dengan menambah jumlah masukan dari *TLB*. Tetapi ini tidaklah murah karena memori yang dipakai untuk membuat *TLB* mahal dan haus akan tenaga.

Ada suatu ukuran lain yang mirip dengan *hit ratio* yaitu jangkauan *TLB*. Jangkauan *TLB* adalah jumlah memori yang dapat diakses dari *TLB*, jumlah tersebut merupakan perkalian dari jumlah masukan dengan ukuran halaman. Idealnya, *working set* dari sebuah proses disimpan dalam *TLB*. Jika tidak, maka proses akan menghabiskan waktu yang cukup banyak mengatasi referensi memori di dalam tabel halaman daripada di *TLB*. Jika jumlah masukan dari *TLB* dilipatgandakan, maka jangkauan *TLB* juga akan bertambah menjadi dua kali lipat. Tetapi untuk beberapa aplikasi hal ini masih belum cukup untuk menyimpan *working set*.

Cara lain untuk meningkatkan jangkauan *TLB* adalah dengan menambah ukuran halaman. Bila ukuran halaman dijadikan empat kali lipat dari ukuran awalnya (misalnya dari 32 KB menjadi 128 KB), maka jangkauan *TLB* juga akan menjadi empat kali lipatnya. Namun ini akan meningkatkan fragmentasi untuk aplikasi-aplikasi yang tidak membutuhkan ukuran halaman sebesar itu. Sebagai alternatif, OS dapat menyediakan ukuran halaman yang bervariasi. Sebagai contoh, UltraSparc II menyediakan halaman berukuran 8 KB, 64 KB, 512 KB, dan 4 MB. Sedangkan Solaris 2 hanya menggunakan halaman ukuran 8 KB dan 4 MB.

35.7. Tabel Halaman yang Dibalik

Kegunaan dari tabel halaman yang dibalik adalah untuk mengurangi jumlah memori fisik yang dibutuhkan untuk melacak penerjemahan alamat virtual-ke-fisik. Metode penghematan ini dilakukan dengan membuat tabel yang memiliki hanya satu masukan tiap halaman memori fisik, terdaftar oleh pasangan (proses-id, nomor-halaman).

Karena menyimpan informasi tentang halaman memori virtual yang mana yang disimpan di setiap *frame* fisik, tabel halaman yang dibalik mengurangi jumlah fisik memori yang dibutuhkan untuk menyimpan informasi ini. Bagaimana pun, tabel halaman yang dibalik tidak lagi mengandung informasi yang lengkap tentang ruang alamat logis dari sebuah proses, dan informasi itu dibutuhkan jika halaman yang direferensikan tidak sedang berada di memori. *Demand paging* membutuhkan informasi ini untuk memproses kesalahan halaman. Agar informasi ini tersedia, sebuah tabel halaman eksternal (satu tiap proses) harus tetap disimpan. Setiap tabel tampak seperti tabel halaman per proses tradisional, mengandung informasi dimana setiap halaman virtual berada.

Tetapi, apakah tabel halaman eksternal menegasikan kegunaan tabel halaman yang dibalik? Karena tabel-tabel ini direferensikan hanya saat kesalahan halaman terjadi, mereka tidak perlu tersedia secara cepat. Namun, mereka dimasukkan atau dikeluarkan dari memori sesuai kebutuhan. Sayangnya, sekarang kesalahan halaman mungkin terjadi pada manager memori virtual, menyebabkan kesalahan halaman lain karena pada saat mem-*page in* tabel halaman eksternal, ia harus mencari halaman virtual pada *backing store*. Kasus spesial ini membutuhkan penanganan di kernel dan penundaan pada proses *page-lookup*.

35.8. Struktur Program

Pemilihan struktur data dan struktur pemograman secara cermat dapat meningkatkan *locality* dan karenanya menurunkan tingkat kesalahan halaman dan jumlah halaman di *working set*. Sebuah *stack* memiliki *locality* yang baik, karena akses selalu dari atas. Sebuah *hash table*, di sisi lain, didesain untuk menyebar referensi-referensi, menghasilkan *locality* yang buruk. Tentunya, referensi akan *locality* hanyalah satu ukuran dari efisiensi penggunaan struktur data. Faktor-faktor lain yang berbobot berat termasuk kecepatan pencarian, jumlah total dari referensi dan jumlah total dari halaman yang disentuh.

35.9. *Interlock* M/K

Saat *demand paging* digunakan, kita terkadang harus mengizinkan beberapa halaman untuk dikunci di memori. Salah satu situasi muncul saat M/K dilakukan ke atau dari memori pengguna (virtual). M/K sering diimplementasikan oleh prosesor M/K yang terpisah. Sebagai contoh, sebuah pengendali pita magnetik pada umumnya diberikan jumlah *bytes* yang akan dipindahkan dan alamat memori untuk *buffer*. Saat pemindahan selesai, CPU diinterupsi.

Harus diperhatikan agar urutan dari kejadian-kejadian berikut tidak muncul: Sebuah proses mengeluarkan permintaan M/K, dan diletakkan di antrian untuk M/K tersebut. Sementara itu, CPU diberikan ke proses- proses lain. Proses-proses ini menimbulkan kesalahan halaman, dan, menggunakan algoritma penggantian global, salah satu dari mereka menggantikan halaman yang mengandung memori *buffer* untuk proses yang menunggu tadi. Halaman-halaman untuk proses tersebut dikeluarkan. Kadang-kadang kemudian, saat permintaan M/K bergerak maju menuju ujung dari antrian peranti, M/K terjadi ke alamat yang telah ditetapkan. Bagaimana pun, *frame* ini sekarang sedang digunakan untuk halaman berbeda milik proses lain.

Ada dua solusi untuk masalah ini. Salah satunya adalah jangan pernah menjalankan M/K kepada memori pengguna. Sedangkan solusi lainnya adalah dengan mengizinkan halaman untuk dikunci dalam memori.

35.10. Pemrosesan Waktu Nyata

Diskusi-diskusi di bab ini telah dikonsentrasikan dalam menyediakan penggunaan yang terbaik secara menyeluruh dari sistem komputer dengan meningkatkan penggunaan memori. Dengan menggunakan memori untuk data yang aktif, dan memindahkan data yang tidak aktif ke cakram, kita meningkatkan *throughput*. Bagaimana pun, proses individual dapat menderita sebagai hasilnya, sebab mereka sekarang mendapatkan kesalahan halaman tambahan selama eksekusi.

Pertimbangkan sebuah proses atau *thread* waktu-nyata. Proses tersebut berharap untuk memperoleh kendali CPU, dan untuk menjalankan penyelesaian dengan penundaan yang minimum. Memori virtual adalah kebalikan dari komputasi waktu nyata, sebab dapat menyebabkan penundaan jangka panjang yang tidak diharapkan pada eksekusi sebuah proses saat halaman dibawa ke memori. Untuk itulah, sistem-sistem waktu nyata hampir tidak memiliki memori virtual.

35.11. Keluarga Windows NT

Pada bagian berikut akan dibahas bagaimana Windows XP dan Solaris 2 mengimplementasi memori virtual. Windows XP mengimplementasikan memori virtual dengan menggunakan permintaan halaman melalui clustering. Clustering menanganani kesalahan halaman dengan menambahkan tidak hanya halaman yang terkena kesalahan, tetapi juga halaman-halaman yang berada disekitarnya kedalam memori fisik.

Saat proses pertama dibuat, diberikan working set minimum yaitu jumlah minimum halaman yang dijamin akan dimiliki oleh proses tersebut dalam memori. Jika memori yang cukup tersedia, proses dapat diberikan halaman sampai sebanyak working set maximum. Manager memori virtual akan menyimpan daftar dari bingkai yang bebas. Terdapat juga sebuah nilai batasan yang diasosiasikan dengan daftar ini untuk mengindikasikan apakah memori yang tersedia masih mencukupi. Jika proses tersebut sudah sampai pada working set maximum-nya dan terjadi kesalahan halaman, maka

dia harus memilih bingkai pengganti dengan aturan penggantian lokal.

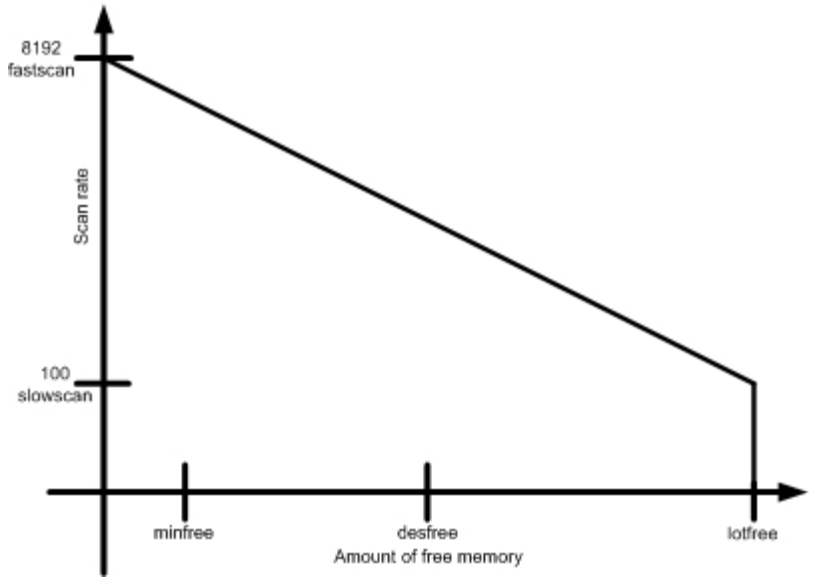
Saat jumlah memori bebas jatuh di bawah nilai batasan, manager memori virtual menggunakan sebuah taktik yang dikenal sebagai automatic working set trimming untuk mengembalikan nilai tersebut di atas nilai batas. Cara ini berguna untuk mengevaluasi jumlah halaman yang dialokasikan kepada proses. Jika proses telah mendapat alokasi halaman lebih besar daripada working set minimum-nya, manager memori virtual akan mengurangi jumlah halamannya sampai working-set minimum. Jika memori bebas sudah tersedia, proses yang bekerja pada working set minimum akan mendapatkan halaman tambahan.

35.12. Solaris 2

Dalam sistem operasi Solaris 2, jika sebuah proses menyebabkan terjadi kesalahan halaman, kernel akan memberikan halaman kepada proses tersebut dari daftar halaman bebas yang disimpan. Akibat dari hal ini adalah, kernel harus menyimpan sejumlah memori bebas. Terhadap daftar ini ada dua parameter yg disimpan yaitu *minfree* dan *lotsfree*, yaitu batasan minimum dan maksimum dari memori bebas yang tersedia.

Empat kali dalam tiap detiknya, kernel memeriksa jumlah memori yang bebas. Jika jumlah tersebut jatuh di bawah *minfree*, maka sebuah proses *pageout* akan dilakukan, dengan pekerjaan sebagai berikut. Pertama *clock* akan memeriksa semua halaman dalam memori dan mengeset bit referensi menjadi 0. Saat berikutnya, *clock* kedua akan memeriksa bit referensi halaman dalam memori, dan mengembalikan bit yang masih di set ke 0 ke daftar memori bebas. Hal ini dilakukan sampai jumlah memori bebas melampaui parameter *lotsfree*. Lebih lanjut, proses ini dinamis, dapat mengatur kecepatan jika memori terlalu sedikit. Jika proses ini tidak bisa membebaskan memori , maka *kernel* memulai pergantian proses untuk membebaskan halaman yang dialokasikan ke proses-proses tersebut.

Gambar 35.3. Solar Halaman Scanner



35.13. Rangkuman

Masalah yang penting dari alokasi *frame* dengan penggunaan memori virtual adalah bagaimana membagi memori dengan bebas untuk beberapa proses yang sedang dikerjakan. Contoh algoritma yang lazim digunakan adalah *equal allocation* dan *proportional allocation*.

Algoritma *page replacement* dapat diklasifikasikan dalam 2 kategori, yaitu penggantian global dan penggantian lokal. Perbedaan antara keduanya terletak pada boleh tidaknya setiap proses memilih

frame pengganti dari semua *frame* yang ada.

Utilitas dari CPU dapat menyebabkan *trashing*, dimana sistem sibuk melakukan *swapping* dikarenakan banyaknya *page-fault* yang terjadi. Efek dari trashing dapat dibatasi dengan algoritma penggantian lokal atau prioritas. Cara untuk mengetahui berapa banyak proses yang dibutuhkan suatu proses salah satunya adalah dengan strategi *working set*.

Pemilihan algoritma penggantian dan aturan alokasi adalah keputusan-keputusan utama yang kita buat untuk sistem *paging*. Selain itu, ada beberapa pertimbangan lain, antara lain *prepaging, TLB reach*, ukuran *page*, struktur program, *I/O interlock*, dan lain sebagainya. Beberapa contoh sistem operasi yang mengimplementasikan virtual memori adalah Windows NT, Solaris 2 dan Linux.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBAmirSch2000] Yair Amir dan Theo Schlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

Bab 36. Memori Linux

36.1. Pendahuluan

Seperti pada Solaris 2, Linux juga menggunakan variasi dari algoritma *clock*. *Thread* dari kernel linux (kswapd) akan dijalankan secara periodik (atau dipanggil ketika penggunaan memori sudah berlebihan). Jika jumlah halaman yang bebas lebih sedikit dari batas atas halaman bebas, maka *thread* tersebut akan berusaha untuk membebaskan tiga halaman. Jika lebih sedikit dari batas bawah halaman bebas, *thread* tersebut akan berusaha untuk membebaskan enam halaman dan tidur untuk beberapa saat sebelum berjalan lagi. Saat dia berjalan, akan memeriksa mem_map, daftar dari semua halaman yang terdapat di memori. Setiap halaman mempunyai *byte* umur yang diinisialisasikan ke tiga. Setiap kali halaman ini diakses, maka umur ini akan ditambahkan (hingga maksimum 20), setiap kali kswapd memeriksa halaman ini, maka umur akan dikurangi. Jika umur dari sebuah halaman sudah mencapai 0 maka dia bisa ditukar. Ketika kswapd berusaha membebaskan halaman, dia pertama akan membebaskan halaman dari *cache*, jika gagal dia akan mengurangi *cache* sistem berkas, dan jika semua cara sudah gagal, maka dia akan menghentikan sebuah proses.

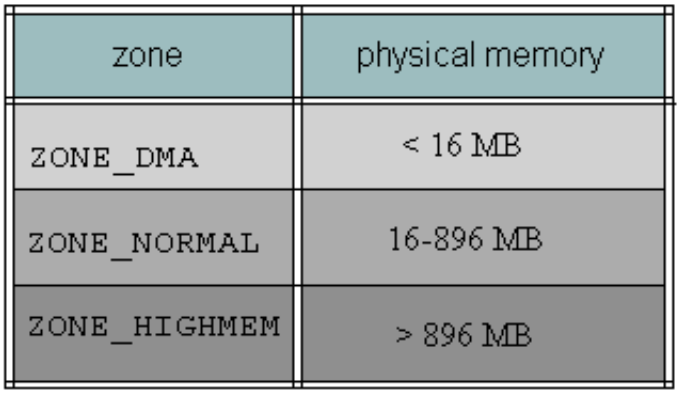
Alokasi memori pada linux menggunakan dua buah alokasi yang utama, yaitu algoritma *buddy* dan *slab*. Untuk algoritma *buddy*, setiap rutin pelaksanaan alokasi ini dipanggil, dia memeriksa blok memori berikutnya, jika ditemukan dia dialokasikan, jika tidak maka daftar tingkat berikutnya akan diperiksa. Jika ada blok bebas, maka akan dibagi jadi dua, yang satu dialokasikan dan yang lain dipindahkan ke daftar yang di bawahnya.

36.2. Manajemen Memori Fisik

Bagian ini menjelaskan bagaimana linux menangani memori dalam sistem. Memori manajemen merupakan salah satu bagian terpenting dalam sistem operasi. Karena adanya keterbatasan memori, diperlukan suatu strategi dalam menangani masalah ini. Jalan keluarnya adalah dengan menggunakan memori virtual. Dengan memori virtual, memori tampak lebih besar daripada ukuran yang sebenarnya. Manajemen memori pada Linux mengandung dua komponen utama (yang merupakan inti dari bab ini):

1. Berkaitan dengan pembebasan dan pengalokasian halaman/blok pada main memori.
2. Berkaitan dengan penanganan memori virtual.

Gambar 36.1. Contoh pembagian zona memori pada arsitektur Intel x86.

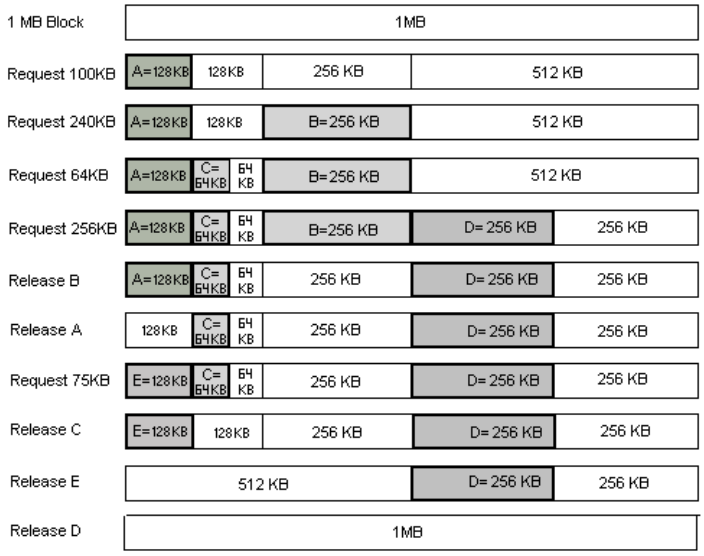


Linux memisahkan memori fisik ke dalam tiga zona berbeda, dimana tiap zona mengindentifikasikan region-region yang berbeda pada memori fisik. Ketiga zona tersebut adalah:

1. **Zona DMA (Direct Memory Access).** Tempat penanganan kegiatan yang berhubungan dengan transfer data antara CPU dengan I/O, dalam hal ini DMA akan menggantikan peran CPU sehingga CPU dapat mengerjakan instruksi lainnya.
2. **Zona NORMAL.** Tempat di memori fisik dimana penanganan permintaan-permintaan yang berhubungan dengan pemanggilan routine untuk alokasi halaman/blok dalam menjalankan proses.
3. **Zona HIGHMEM.** Tempat yang merujuk kepada memori fisik yang tidak dipetakan ke dalam ruang alamat kernel.

Contoh pada arsitektur intel 32-bit, yang berarti menyediakan maksimal 2 pangkat 32 atau 4GB ruang alamat memori fisik. Alokasi zona DMAnya mencapai 16MB, alokasi zona NORMAL antara 16-896MB, dan zona HIGHMEM antara 896-4096MB.

Gambar 36.2. Contoh skema alokasi memori dengan algoritma buddy



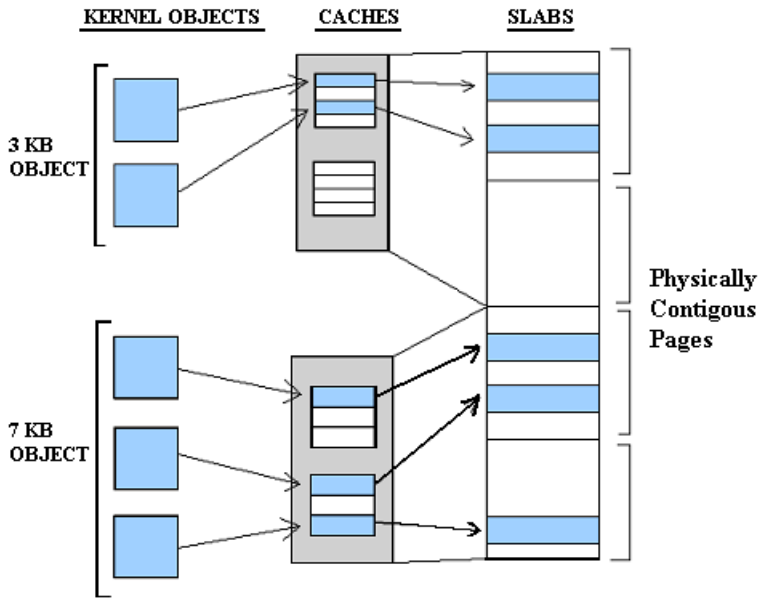
Memori manager di Linux berusaha untuk mengefisienkan ruang alamat pada memori fisik, agar memungkinkan lebih banyak proses yang dapat bekerja di memori dibandingkan dengan yang sudah ditentukan oleh kernel. Oleh karena itu, digunakanlah dua macam tekhnik alokasi, yaitu alokasi halaman yang ditangani oleh halaman allocator dan alokasi slab yang ditangani oleh slab allocator.

Alokasi halaman menggunakan algoritma buddy yang bekerja sebagai berikut: Pada saat kegiatan alokasi data di memori, maka region di memori (yang disediakan oleh kernel kepada suatu proses) akan di-split (dibagi) menjadi dua region yang berukuran sama besar. Kejadian ini akan terus berlanjut hingga didapat region (blok) yang sesuai dengan ukuran data yang diperlukan oleh proses tersebut. Dalam hal ini halaman allocator akan memanggil sistem call kmalloc() yang kemudian akan memerintahkan kernel untuk melakukan kegiatan pembagian region tersebut.

Alokasi slab bertujuan untuk mengalokasikan struktur data (object) kernel yang dibutuhkan di memori fisik untuk menjalankan proses tertentu. Alokasi slab menggunakan algoritma slab. Slab dibentuk dari halaman-halaman memori fisik yang berdekatan serta digunakan terutama untuk kegiatan pengalokasian memori fisik. Sebuah cache (yang terdapat pada disk) terdiri dari satu atau lebih slab, dan diisi oleh beberapa obyek. Obyek merupakan bentuk instansiasi dari struktur data kernel yang direpresentasikan oleh cache yang bersangkutan. Misalkan suatu cache merepresentasikan tempat penyimpanan semaphore, maka obyek yang terdapat di dalamnya merupakan obyek-obyek semaphore yang disediakan oleh kernel.

Ketika sebuah cache dibentuk, maka semua obyek (di dalam cache tersebut) berstatus free, dan ketika terjadi sebuah permintaan (dari suatu proses), maka obyek-obyek yang dibutuhkan untuk memenuhi permintaan tersebut akan diset berstatus used. Kemudian obyek-obyek yang berstatus used tersebut (yang telah dikelompokkan ke dalam slab-slab) akan dipetakan dari cache ke dalam memori fisik.

Gambar 36.3. Contoh skema alokasi slab di GNU/Linux



Di Linux, sebuah slab dapat berstatus:
1. Full, dimana semua obyek di dalam slab tersebut adalah used.
2. Empty, dimana semua obyek di dalam slab tersebut adalah free.
3. Partial, dimana ada obyek yang used dan ada pula yang free.

Keuntungan algoritma slab:
1. Tidak terdapatnya fragmentasi pada memori fisik, karena ukuran obyek-obyek tersebut telah ditetapkan sesuai dengan yang dibutuhkan proses dalam membantu melakukan kerjanya di memori fisik.
2. Request-permintaan oleh memori cepat terpenuhi (dengan mendayagunakan kerja dari cache yang dibentuk pada disk).

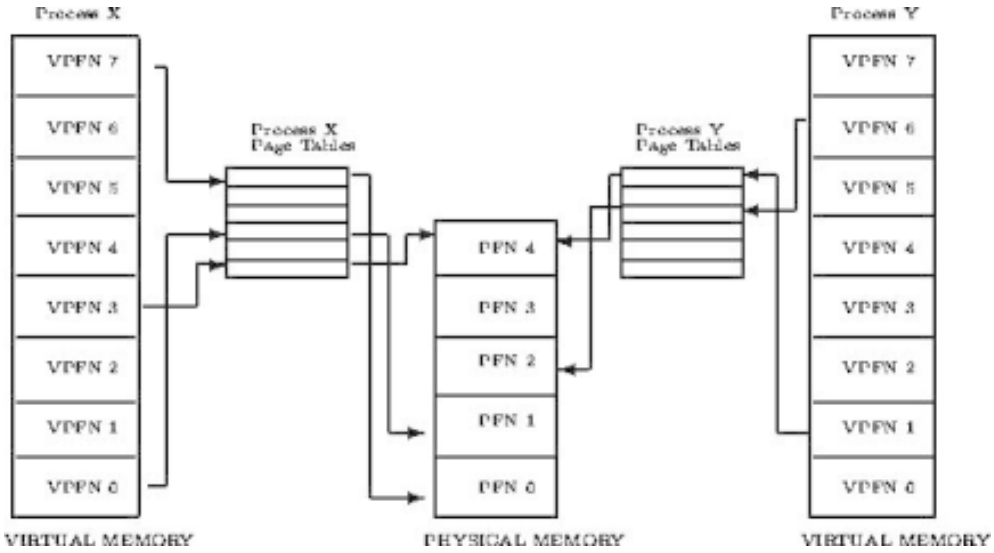
Sistem memori virtual berkaitan erat dengan sistem halaman cache, karena pembacaan halaman data dari memori ke dalam halaman cache (yang dilakukan oleh kernel) membutuhkan proses pemetaan halaman ke dalam halaman cache yang dikerjakan oleh sistem memori virtual.

Keuntungan sistem memori virtual pada Linux:
1. Ruang alamat yang besar, bahkan lebih besar daripada memori fisik.
2. Pembagian memori fisik yang adil, karena adanya manajemen memori.
3. Proteksi, manajemen memori memungkinkan setiap proses terlindungi dari proses-proses lainnya di memori. Sehingga jika terjadi crash yang dialami oleh suatu proses, maka proses-proses lainnya tidak terpengaruh.
4. Penggunaan memori virtual bersama. Contohnya konsep shared library.

36.3. Memori Virtual

Kernel menyediakan suatu ruang alamat virtual untuk tiap proses yang bersifat konstan dan architecture-dependant. Entri-entri dalam tabel halaman yang dipetakan ke dalam ruang alamat ini berstatus protected, dalam arti tidak kelihatan dan tidak dapat dimodifikasi ketika sistem berjalan dalam mode pengguna. Area dalam memori virtual kernel dibagi 2:
1. **Area statis (static area).** Mengandung referensi tabel halaman untuk setiap halaman fisik di memori yang tersedia. Sehingga ketika menjalankan kode kernel, translasi dari alamat fisik ke alamat virtualnya menjadi lebih mudah dan sederhana.
2. **Area sisa (remainder of reserved section).** Area ini kebanyakan tidak terpakai, namun kernel dapat memodifikasi area ini untuk dipakai ditujukan ke area lain di memori.

Gambar 36.4. Pemetaan Memori Virtual ke Alamat Fisik.



Pada saat kita menginstall OS linux misalkan distro Mandarake, maka kita akan diminta untuk menentukan suatu partisi khusus yang berperan sebagai memori virtual, yakni yang disebut: swap. Sebagaimana kita ketahui bahwa memori virtual merupakan jembatan penghubung antara Disk dengan Memori fisik. Di Linux, halaman- halaman yang dibutuhkan supaya suatu proses dapat berjalan, pada awalnya tidak akan langsung dialokasikan ke memori fisik, tetapi dialokasikan ke memori virtual (swap) terlebih dahulu.

Sistem memori virtual linux berperan dalam mengatur beberapa hal:
1. Mengatur ruang alamat supaya dapat dilihat oleh tiap proses.
2. Membentuk halaman-halaman(halamans) yang dibutuhkan.
3. Mengatur alokasi halaman-halaman tersebut dari disk ke memori fisik atau sebaliknya, yang biasa disebut swapping.

Secara mendasar, sistem memori virtual linux mengenal dan mengatur dua ruang alamat proses, yakni sebagai kumpulan halaman (halamans) dan sebagai kumpulan region. Region bisa diartikan sebagai blok-blok atau bagian-bagian atau ruang-ruang dalam memori virtual dengan ukuran yang bervariasi, sedangkan halaman(halaman), ukurannya sama.

Sistem memori virtual linux juga mengatur dua view berkenaan dengan ruang alamat:
1. **Logical View.** Mendeskripsikan instruksi-instruksi yang diterima oleh sistem memori virtual berkenaan dengan susunan ruang alamat.
2. **Physical View.** Berupa entri-entri tabel halaman (halaman tables), dimana entri- entrinya akan menentukan apakah halaman ybs. berada di memori fisik(sedang dipakai untuk proses) atau masih berada di disk (belum dipakai).

Region Memori Virtual

Sehubungan dengan region memori virtual, maka memori virtual dalam Linux memiliki karakteristik:
1. **Backing Store untuk region.** Di dalam region terdapat halaman-halaman(halamans). Backing store mendeskripsikan berasal dari mana halaman-halaman ini. Kebanyakan region dalam memori virtual berasal dari suatu berkas (dari disk) atau kosong(nothing). region dengan backing store yang kosong biasa disebut "demand zero memory", yang merupakan tipe paling sederhana dari memori virtual.
2. **Reaksi region dalam melakukan write.** Pemetaan dari suatu region ke dalam ruang alamat proses dapat bersifat private ataupun shared. Jika ada proses yang akan me-write region yang bersifat private, maka akan dilakukan mekanisme Copy-On-Write (menulis salinannya).

Lifetime dari Ruang Alamat Virtual

Kernel berperan penting dalam manejemen memori virtual, dimana kernel akan membentuk ruang alamat yang baru di memori virtual dalam dua kondisi:

1. Proses menjalankan suatu program dengan sistem call exec() Ketika sistem call exec() dipanggil oleh proses untuk menjalankan suatu program, maka proses akan diberikan ruang alamat virtual yang masih kosong. Kemudian routine-routine akan berkerja me-load program dan mengisi ruang alamat ini.
2. Pembentukan proses baru dengan sistem call fork() Intinya menyalin secara keseluruhan ruang alamat virtual dari proses yang ada. Langkah-langkahnya adalah sebagai berikut:
   - kernel menyalin deskriptor vm_area_struct dari proses induk,
   - kernel membentuk tabel halaman untuk proses anak,
   - kernel menyalin isi tabel halaman proses induk ke proses anak,
   - setelah fork(), maka induk dan anak akan berbagi halaman fisik yang sama.

Disamping itu, ada kasus khusus yang harus diperhatikan, yaitu ketika proses penyalinan dilakukan terhadap region di memori virtual yang bersifat private, dimana region tersebut dipakai lebih dari satu proses -selain proses induk dan anak yang memang berbagi halaman yang sama- dan ada proses yang hendak me-write region tersebut. Jika ini terjadi maka akan dilakukan mekanisme Copy-On-Write, yang berarti mengubah dan memakai salinannya.

36.4. *Demand Paging*

Cara untuk menghemat memori fisik adalah dengan hanya meload page virtual yang sedang digunakan oleh program yang sedang dieksekusi. Tehnik dimana hanya meload page virtual ke memori hanya ketika program dijalankan disebut demand paging.

Ketika proses mencoba mengakses alamat virtual yang tidak ada di dalam memori, CPU tidak dapat menemukan anggota tabel page. Contohnya, dalam gambar, tidak ada anggota tabel page untuk proses x untuk virtual PFN dua dan jika proses x ingin membaca alamat dari virtual PFN 2, CPU tidak dapat menterjemahkan alamat ke alamat fisik. Saat ini CPU bergantung pada sistem operasi untuk menangani masalah ini. CPU menginformasikan kepada sistem operasi bahwa page fault telah terjadi, dan sistem operasi membuat proses menunggu selama sistem operasi menagani masalah ini.

CPU harus membawa page yang benar ke memori dari image di disk. Akses disk membutuhkan waktu yang sangat lama dan proses harus menunggu sampai page selesai diambil. Jika ada proses lain yang dapat dijalankan, maka sistem operai akan memilihnya untuk kemudian dijalankan. Page yang diambil kemudian dituliskan di dalam page fisik yang masih kosong dan anggota dari virtual PFN ditambahkan dalam tabel page proses. Proses kemudian dimulai lagi pada tempat dimana page fault terjadi. Saat ini terjadi pengaksesan memori virtual, CPU membuat penerjemahan dan kemudian proses dijalankan kembali.

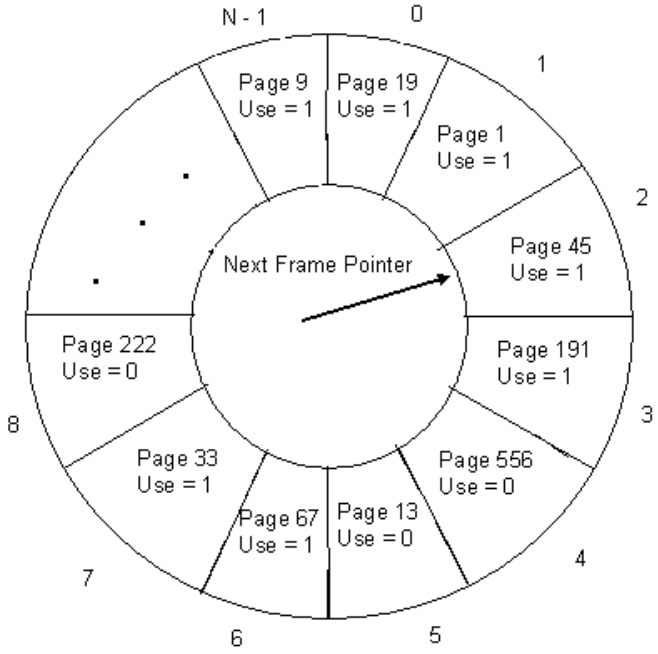
Demand paging terjadi saat sistem sedang sibuk atau saat image pertama kali diload ke memori. Mekanisme ini berarti sebuah proses dapat mengeksekusi image dimana hanya sebagian dari image tersebut terdapat dalam memori fisik.

36.5. *Swaping*

Linux menggunakan tehnik page aging agar adil dalam memilih page yang akan dihapus dari sistem. Ini berarti setiap page memiliki usia sesuai dengan berapa sering page itu diakses. Semakin sering sebuah page diakses, semakin muda page tersebut. Page yang tua adalah kandidat untuk diswap.

Dua mekanisme terpenting dalam sistem memori virtual adalah swapping dan paging, yakni mekanisme dalam mengalokasikan halaman-halaman (halamans) dari memori virtual ke memori fisik (swap-in) maupun sebaliknya (swap-out). Mekanisme ini cukup penting mengingat kemungkinan adanya halaman-halaman yang sudah tidak dibutuhkan lagi, tapi masih 'nongkrong' di memori fisik, padahal ada proses yang memiliki prioritas tinggi membutuhkan halaman di memori fisik namun masih berada di memori virtual. Mekanisme swap-in mengatur pengeluaran (relokasi) halaman yang sudah tidak dibutuhkan lagi dari memori fisik ke memori virtual, sementara mekanisme swap-out mengatur pemasukan(alokasi) halaman yang dibutuhkan dari memori virtual ke memori fisik.

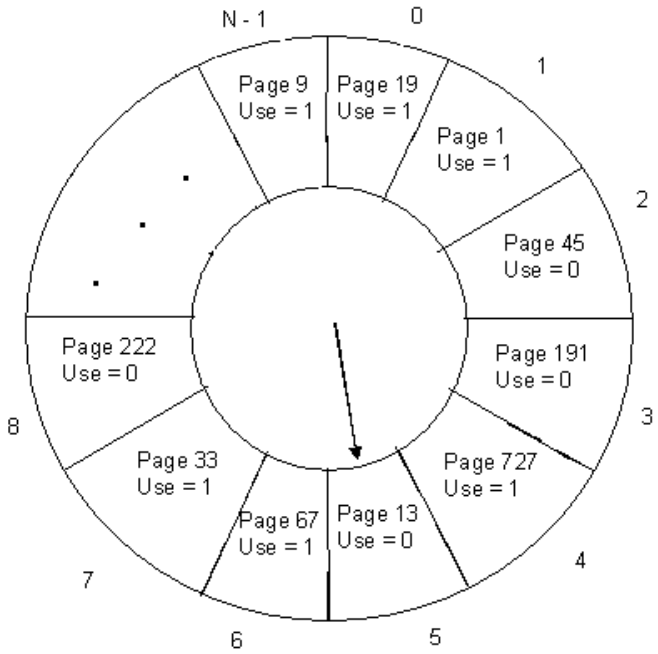
Gambar 36.5. Skema Algoritma Clock



Sistem paging di linux dibagi dua, antara lain:

1. **Halaman-out Policy.** Lebih fokus pada metoda swap-out, yakni menentukan halaman-halaman mana yang akan dikeluarkan dari memori fisik dan kapan. Contoh: algoritma pergantian halaman (halaman replacement),yakni algoritma clock yang mengacu pada Least Frequently Used (LFU) policy.
2. **Paging Mechanism.** Lebih fokus pada metoda swap-in, namun lebih menitikberatkan pada halaman-halaman yang akan dibutuhkan kembali ke memori fisik, dalam artian halaman-halaman tersebut sebelumnya pernah digunakan dan berada di memori fisik.

Gambar 36.6. Clock



36.6. Pengaksesan Memori Virtual Bersama

Memori virtual mempermudah proses untuk berbagi memori saat semua akses ke memori menggunakan tabel page. Proses yang akan berbagi memori virtual yang sama, page fisik yang sama direference oleh banyak proses. Tabel page untuk setiap proses mengandung anggota page table yang mempunyai PFN fisik yang sama.

36.7. Efisiensi

Desainer dari CPU dan sistem operasi berusaha meningkatkan kinerja dari sistem. Disamping membuat prosesor, memori semakin cepat, jalan terbaik adalah manggunakan cache. Berikut ini adalah beberapa cache dalam manajemen memori di linux:

- **Page Cache.** Digunakan untuk meningkatkan akses ke image dan data dalam disk. Saat dibaca dari disk, page dicache di page cache. Jika page ini tidak dibutuhkan lagi pada suatu saat, tetapi dibutuhkan lagi pada saat yang lain, page ini dapat segera diambil dari page cache.
- **Buffer Cache.** Page mungkin mengandung buffer data yang sedang digunakan oleh kernel, device driver dan lain-lain. Buffer cache tampak seperti daftar buffer. Contohnya, device driver membutuhkan buffer 256 bytes, adalah lebih cepat untuk mengambil buffer dari buffer cache daripada mengalokasikan page fisik lalu kemudian memecahnya menjadi 256 bytes buffer-buffer.
- **Swap Cache.** Hanya page yang telah ditulis ditempatkan dalam swap file. Selama page ini tidak mengalami perubahan setelah ditulis ke dalam swap file, maka saat berikutnya page di swap out tidak perlu menuliskan kembali jika page telah ada di swap file. Di sistem yang sering mengalami swap, ini dapat menghemat akses disk yang tidak perlu. Salah satu implementasi yang umum dari hardware cache adalah di CPU, cache dari anggota tabel page. Dalam hal ini, CPU tidak secara langsung membaca tabel page, tetap mencache terjemahan page yang dibutuhkan.

36.8. Load dan Eksekusi Program

Eksekusi dari Kernel Linux dilakukan oleh panggilan terhadap sistem call exec(). System call exec() memerintahkan kernel untuk menjalankan program baru di dalam proses yang sedang berlangsung (current process), dengan cara meng-overwrite current execution dengan initial context dari program baru yang akan dijalankan. Untuk meng-overwrite dan mengeksekusi, akan dilakukan dua kegiatan, yakni:

1. Memeriksa apakah proses baru yang dipanggil memiliki izin untuk melakukan overwrite terhadap berkas yang sedang dieksekusi.

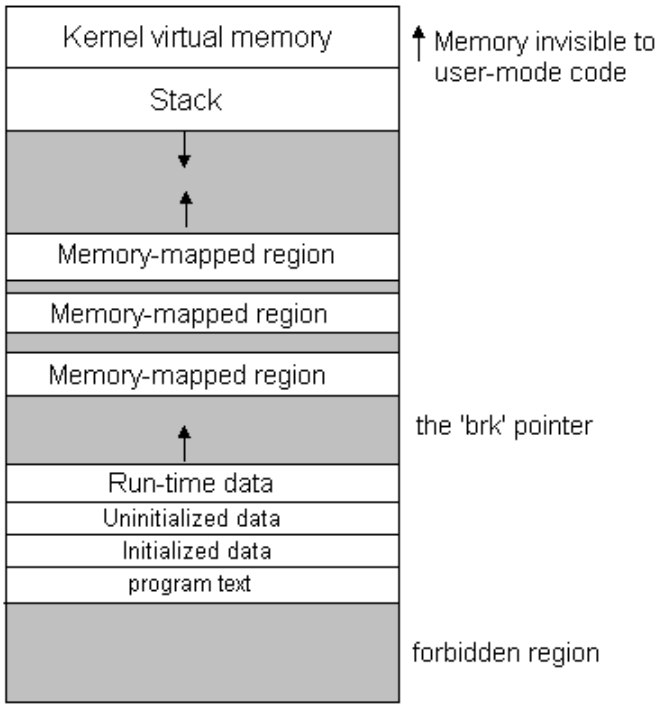
2. Kernel memanggil loader routine untuk memulai menjalankan program. Loader tidak perlu untuk me-load isi dari berkas program ke memori fisik, tetapi paling tidak mengatur pemetaan program ke memori virtual.

Linux menggunakan tabel loader untuk loading program baru. Dengan menggunakan tabel tersebut, Linux memberikan setiap fungsi kesempatan untuk me-load program ketika sistem call exec() dipanggil. Linux menggunakan tabel loader, karena format standar berkas binary Linux telah berubah antara kernel Linux 1.0 dan 1.2. Format Linux versi 1.0 menggunakan format a.out, sementara Linux baru (sejak 1.2) menggunakan format ELF. Format ELF memiliki fleksibilitas dan ekstensibilitas dibanding dengan a.out, karena dapat menambahkan sections baru ke binary ELF (contoh, menambahkan informasi debugging) tanpa menyebabkan loader routine menjadi bingung. Saat ini Linux mendukung pemakaian baik format binary ELF dan a.out di single running sistem, karena menggunakan registrasi dari multiple loader routine. Dua subbab (Pemetaan Program ke Memori dan Static and Dynamic Linking) selanjutnya yang dibahas adalah format binary ELF.

Pemetaan Program ke Memori

Di Linux, binary loader tidak perlu me-load berkas biner ke memori fisik, melainkan dengan cara memetakan halaman dari binary file ke region dari memori virtual. Sehingga hanya ketika program mengakses halaman tertentu akan menyebabkan halaman fault yang mengakibatkan halaman yang dibutuhkan diload ke memori fisik.

Gambar 36.7. ELF



Static dan Dynamic Linking

Ketika program diload dan sudah mulai dieksekusi, semua berkas biner yang dibutuhkan telah diload ke ruang alamat virtual. Meskipun demikian, sebagian besar program juga butuh menjalankan fungsi yang terdapat di sistem librari seperti algoritma sorting, fungsi-fungsi aritmatika, dan lain-lain. Untuk itulah fungsi librari perlu untuk diload juga. Untuk mendapatkan fungsi-fungsi yang terdapat di sistem librari, ada dua cara, yaitu:

1. Static Linking. Fungsi librari yang dibutuhkan diload langsung ke berkas biner yang dapat dijalankan (executable) program. Kerugian static linking adalah setiap program yang dibuat harus meng-copy fungsi-fungsi dari sistem librari, sehingga tidak efisien dalam penggunaan memori fisik dan pemakaian ruang disk.
2. Dynamic Linking. Dynamic linking menggunakan single loading, sehingga lebih efisien dalam penggunaan memori fisik dan pemakaian ruang disk. Link librari dapat menentukan fungsi-fungsi yang dibutuhkan program dengan cara membaca informasi yang terkandung di dalam section dari ELF.

36.9. Rangkuman

Algoritma *page replacement* dapat diklasifikasikan dalam dua kategori, yaitu penggantian global dan penggantian lokal. Perbedaan antara keduanya terletak pada boleh tidaknya setiap proses memilih *frame* pengganti dari semua *frame* yang ada.

Utilitas dari CPU dapat menyebabkan *trashing*, dimana sistem sibuk melakukan *swapping* dikarenakan banyaknya *page-fault* yang terjadi. Efek dari trashing dapat dibatasi dengan algoritma penggantian lokal atau prioritas. Cara untuk mengetahui berapa banyak proses yang dibutuhkan suatu proses salah satunya adalah dengan strategi *working set*.

Pemilihan algoritma penggantian dan aturan alokasi adalah keputusan-keputusan utama yang kita buat untuk sistem *paging*. Selain itu, ada beberapa pertimbangan lain, antara lain *prepaging, TLB reach*, ukuran *page*, struktur program, *I/O interlock*, dan lain sebagainya. Beberapa contoh sistem operasi yang mengimplementasikan virtual memori adalah Windows NT, Solaris 2 dan Linux.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBAmirSch2000] Yair Amir dan Theo Schlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

Bagian VI. Penyimpanan Sekunder

Pada umumnya, penyimpanan sekunder berbentuk disk magnetik. Kecepatan pengaksesan memori sekunder ini jauh lebih lambat dibandingkan memori utama. Pada bagian ini akan diperkenalkan konsep-konsep yang berhubungan dengan memori sekunder seperti sistem berkas, atribut dan operasi sistem berkas, struktur direktori, atribut dan operasi struktur direktori, sistem berkas jaringan, sistem berkas virtual, sistem berkas GNU/Linux, keamanan sistem berkas, FHS (*File Hierarchy System*), serta alokas blok sistem berkas.

Bab 37. Sistem Berkas

37.1. Pendahuluan

Semua aplikasi komputer butuh menyimpan dan mengambil informasi. Ketika sebuah proses sedang berjalan, proses tersebut menyimpan sejumlah informasi yang terbatas, dibatasi oleh ukuran alamat virtual. Untuk beberapa aplikasi, ukuran ini cukup, namun untuk lainnya terlalu kecil.

Masalah berikutnya adalah apabila proses tersebut berhenti maka informasinya hilang. Padahal ada beberapa informasi yang penting dan harus bertahan beberapa waktu bahkan selamanya. Ada pun masalah ketiga yaitu sangatlah perlu terkadang untuk lebih dari satu proses mengakses informasi secara bersamaan. Untuk memecahkan masalah ini, informasi tersebut harus dapat berdiri sendiri tanpa tergantung dengan sebuah proses.

Pada akhirnya, kita memiliki masalah-masalah yang cukup signifikan dan penting untuk dicari solusinya. Pertama kita harus dapat menyimpan informasi dengan ukuran yang besar. Kedua, informasi harus tetap ketika proses berhenti. Ketiga, informasi harus dapat diakses oleh lebih dari satu proses secara bersamaan. Solusi dari ketiga masalah diatas adalah sesuatu yang disebut berkas.

37.2. Konsep Berkas

Berkas adalah sebuah koleksi informasi berkaitan yang diberi nama dan disimpan di dalam *secondary storage*. Biasanya sebuah berkas merepresentasikan data atau program. Beberapa jenis berkas diantaranya:

- ***Text file*.** yaitu urutan dari karakter-karakter yang diatur menjadi barisan dan mungkin halaman.
- ***Source file*.** yaitu urutan dari berbagai subroutine dan fungsi yang masing-masing kemudian diatur sebagai deklarasi-deklarasi diikuti oleh pernyataan-pernyataan yang dapat diexecute.
- ***Object file*.** yaitu urutan dari byte-byte yang diatur menjadi blok-blok yang dapat dipahami oleh penghubung system.
- ***Executable file*.** adalah kumpulan dari bagian-bagian kode yang dapat dibawa ke memori dan dijalankan oleh loader.

37.3. Atribut berkas

Selain nama dan data, sebuah berkas dikaitkan dengan informasi-informasi tertentu yang juga penting untuk dilihat pengguna, seperti kapan berkas itu dibuat, ukuran berkas, dan lain-lain. Kita akan sebut informasi-informasi ekstra ini atribut. Setiap sistem mempunyai sistem atribusi yang berbeda-beda, namun pada dasarnya memiliki atribut-atribut dasar seperti berikut ini:

- **Nama.** nama berkas simbolik ini adalah informasi satu-satunya yang disimpan dalam format yang dapat dibaca oleh pengguna.
- ***Identifier*.** Tanda unik ini yang biasanya merupakan sebuah angka, mengenali berkas didalam sebuah sistem berkas; tidak dapat dibaca oleh pengguna.
- **Jenis.** Informasi ini diperlukan untuk sistem-sistem yang mendukung jenis berkas yang berbeda.
- **Lokasi.** Informasi ini adalah sebuah penunjuk pada sebuah *device* dan pada lokasi berkas pada *device* tersebut.
- **Ukuran.** Ukuran dari sebuah berkas (dalam bytes, words, atau blocks) dan mungkin ukuran maksimum dimasukkan dalam atribut ini juga.
- **Proteksi.** Informasi yang menentukan siapa yang dapat melakukan *read*, *write*, *execute*, dan lainnya.
- **Waktu dan identifikasi pengguna.** Informasi ini dapat disimpan untuk pembuatan berkas, modifikasi terakhir, dan penggunaan terakhir. Data-data ini dapat berguna untuk proteksi, keamanan, dan *monitoring* penggunaan.

37.4. Jenis Berkas

Jenis berkas merupakan salah satu atribut berkas yang cukup penting. Saat kita mendesain sebuah sistem berkas, kita perlu mempertimbangkan bagaimana sistem operasi akan mengenali

berkas-berkas dengan jenis yang berbeda. Apabila sistem operasi dapat mengenali, maka membuka berkas tersebut bukan suatu masalah. Seperti contohnya, apabila kita hendak mencetak bentuk obyek biner sebuah program, yang tercetak biasanya tidak terbaca, namun hal ini dapat dihindari apabila sistem operasi telah diberitahu akan adanya jenis berkas tersebut.

Cara yang paling umum untuk mengimplementasikan jenis berkas tersebut adalah dengan memasukkan jenis berkas tersebut ke dalam nama berkas. Nama berkas dibagi menjadi dua bagian. Bagian pertama adalah nama dari berkas tersebut, dan yang kedua, atau biasa disebut extention adalah jenis dari berkas tersebut. Kedua nama ini biasanya dipisahkan dengan tanda '.', contoh: berkas.txt.

JENIS BERKAS	EXTENSION	FUNGSI
Executable	exe, com, bin, atau tidak ada	Siap menjalankan program bahasa mesin
Object	obj, o	Dikompilasi, bahasa mesin, tidak terhubung (link)
Source code	c, cc, java, pas, asm	Kode-kode program dalam berbagai bahasa pemrograman
Batch	bat, sh	Memerintahkan ke command interpreter
Text	txt, doc	Data text, dokumen
Word processor	wp, tex, rtf, doc	Macam-macam format dari word processor
Library	lib, a, so, dll	Libraries dari rutin untuk programmer
Print/view	ps,pdf, jpg	Berkas ASCII/binary dalam format untuk mencetak atau melihat
Archive	arc, zip, tar	Berkas-berkas yang berhubungan dikelompokkan ke dalam satu berkas, dikompres, untuk pengarsipan
Multimedia	mpeg, mov, rm	Berkas binary yang berisi informasi audio atau A/V

37.5. Operasi Berkas

Fungsi dari berkas adalah untuk menyimpan data dan mengizinkan kita membacanya. Dalam proses ini ada beberapa operasi yang dapat dilakukan berkas. Ada pun operasi-operasi dasar yang dilakukan berkas, yaitu:

- Membuat Berkas (*Create*):

  Kita perlu dua langkah untuk membuat suatu berkas. Pertama, kita harus temukan tempat didalam sistem berkas. Kedua, sebuah entri untuk berkas yang baru harus dibuat dalam direktori. Entri dalam direktori tersebut merekam nama dari berkas dan lokasinya dalam sistem berkas.

- Menulis sebuah berkas (*Write*):

Untuk menulis sebuah berkas, kita membuat sebuah *system call* yang menyebutkan nama berkas dan informasi yang akan di-nulis kedalam berkas.

- Membaca Sebuah berkas (*Read*):

  Untuk membaca sebuah berkas menggunakan sebuah system call yang menyebut nama berkas yang dimana dalam blok memori berikutnya dari sebuah berkas harus diposisikan.

- Memposisikan Sebuah Berkas (*Reposition*):

  Direktori dicari untuk entri yang sesuai dan *current-file-position* diberi sebuah nilai. Operasi ini di dalam berkas tidak perlu melibatkan M/K, selain itu juga diketahui sebagai *file seek.*

- Menghapus Berkas (*Delete*):

  Untuk menghapus sebuah berkas kita mencari dalam direktori untuk nama berkas tersebut. Setelah ditemukan, kita melepaskan semua spasi berkas sehingga dapat digunakan kembali oleh berkas-berkas lainnya dan menghapus entry direktori.

- Menghapus Sebagian Isi Berkas (*Truncate*):

  User mungkin mau menghapus isi dari sebuah berkas, namun menyimpan atributnya. Daripada memaksa pengguna untuk menghapus berkas tersebut dan membuatnya kembali, fungsi ini tidak akan mengganti atribut, kecuali panjang berkas dan mendefinisikan ulang panjang berkas tersebut menjadi nol.

Keenam operasi diatas merupakan operasi-operasi dasar dari sebuah berkas yang nantinya dapat dikombinasikan untuk membentuk operasi-operasi baru lainnya. Contohnya apabila kita ingin menyalin sebuah berkas, maka kita memakai operasi *create* untuk membuat berkas baru, *read* untuk membaca berkas yang lama, dan *write* untuk menulisnya pada berkas yang baru.

37.6. Struktur Berkas

Berkas dapat di struktur dalam beberapa cara. Cara yang pertama adalah sebuah urutan *bytes* yang tidak terstruktur. Akibatnya sistem operasi tidak tahu atau peduli apa yang ada dalam berkas, yang dilihatnya hanya bytes. Ini menyediakan fleksibilitas yang maksimum. User dapat menaruh apa pun yang mereka mau dalam berkas, dan sistem operasi tidak membantu, namun tidak juga menghalangi.

Cara berikutnya, adalah dengan *record sequence*. Dalam model ini, sebuah berkas adalah sebuah urutan dari rekaman-rekaman yang telah ditentukan panjangnya, masing-masing dengan beberapa struktur internal. Artinya adalah bahwa sebuah operasi *read* membalikan sebuah rekaman dan operasi *write* menimpa atau menambahkan suatu rekaman.

Struktur berkas yang ketiga, adalah menggunakan sebuah *tree*. Dalam struktur ini sebuah berkas terdiri dari sebuah *tree* dari rekaman-rekaman tidak perlu dalam panjang yang sama, tetapi masing-masing memiliki sebuah *field key* dalam posisi yang telah ditetapkan dalam rekaman tersebut. Tree ini disort dalam *field key* dan mengizinkan pencarian yang cepat untuk sebuah *key tertentu*.

37.7. Metode Akses

Berkas menyimpan informasi. Apabila sedang digunakan informasi ini harus diakses dan dibaca melalui memori komputer. Informasi dalam berkas dapat diakses dengan beberapa cara. Berikut adalah beberapa caranya:

- **Akses Sekuensial.** Akses ini merupakan yang paling sederhana dan paling umum digunakan. Informasi di dalam berkas diproses secara berurutan. Sebagai contoh, editor dan kompilator biasanya mengakses berkas dengan cara ini.
- **Akses Langsung.** Metode berikutnya adalah akses langsung atau dapat disebut *relative access*.

Sebuah berkas dibuat dari rekaman-rekaman logical yang panjangnya sudah ditentukan, yang mengizinkan program untuk membaca dan menulis rekaman secara cepat tanpa urutan tertentu.

37.8. Rangkuman

Di dalam sebuah sistem operasi, salah satu hal yang paling penting adalah sistem berkas. Sistem berkas ini muncul karena ada tiga masalah utama yang cukup signifikan: kebutuhan untuk menyimpan data dalam jumlah yang besar, kebutuhan agar data tidak mudah hilang (*non-volatile*), dan informasi harus berdiri sendiri tidak bergantung pada proses. Pada sistem berkas ini, diatur segala rupa macam yang berkaitan dengan sebuah berkas mulai dari atribut, tipe, operasi, struktur, sampai metode akses berkas.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBCarter2004] John Carter. 2004. *CS 5460 Operating Systems – Lecture 19: File System Operations and Optimizations – http://www.cs.utah.edu/ classes/ cs5460/ lectures/ lecture19.pdf* . Diakses 29 Mei 2006.

[WEBGooch1999] Richard Gooch. 1999. *Overview of the Virtual File System – http://www.atnf.csiro.au/ people/ rgooch/ linux/ docs/ vfs.txt* . Diakses 29 Mei 2006.

[WEBIBM1997] IBM Coorporation. 1997. *General Programming Concepts: Writing and Debugging Programs – Threads Scheduling http://www.unet.univie.ac.at/ aix/ aixprggd/ genprogc/ threads_sched.htm* . Diakses 1 Juni 2006.

[WEBWiki2006g] From Wikipedia, the free encyclopedia. 2006. *File system – http://en.wikipedia.org/wiki/File_system* . Diakses 04 Juli 2006.

Bab 38. Struktur Direktori

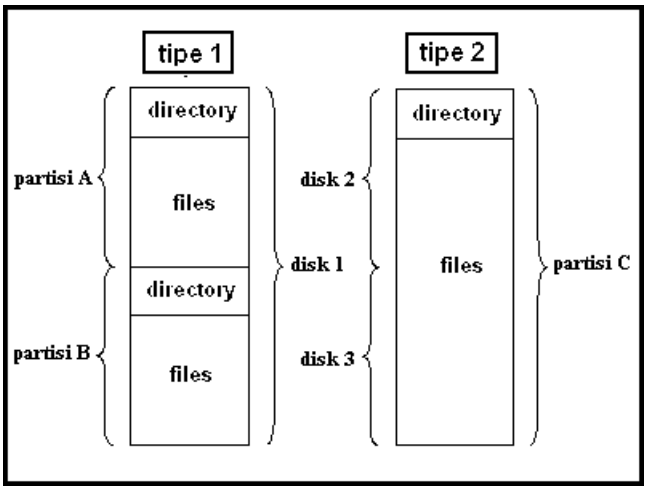
38.1. Pendahuluan

Direktori atau folder merupakan suatu entitas dalam sebuah berkas sistem yang mengandung berkas atau mengandung direktori lain. Sebenarnya, pada hakikatnya berkas atau berkas terdapat dalam disk, direktori hanya menyediakan link atau menunjuk pada berkas yang ada.

Dengan demikian, dapat disimpulkan bahwa direktori digunakan sebagai sarana untuk pengorganisasian berkas pada suatu sistem komputer. Dengan direktori, berkas-berkas dapat dikelompokkan. Berkas tersebut dapat berisi berkas ataupun direktori lain, sehingga direktori dapat juga disebut sebagai berkas istimewa.

Dalam pengorganisasian berkas, sistem operasi dapat mempartisi disk menjadi beberapa volume/direktori, ataupun menjadikan dua disk menjadi sebuah volume/direktori. Istilah volume digunakan sebagai root direktori pada Windows.

Gambar 38.1. File Organization



38.2. Atribut Direktori

Sebagai sebuah berkas, direktori mempunyai atribut, yaitu:

1. **Nama.** Merupakan nama dari direktori itu sendiri.
2. **Alamat.** Merupakan alamat dari direktori. Sebagai contoh, alamat dari direktori lib dalam Linux adalah "/usr/lib", sedangkan alamat direktori sistem dalam Windows adalah "C:/windows/system".
3. **Ukuran.** Merupakan besarnya ukuran direktori, biasanya dalam satuan byte, KiloByte, MegaByte atau GigaByte. Ukuran tersebut memuat ukuran dari berkas-berkas yang ada dalam direktori tersebut.
4. **Tanggal.** Berisi keterangan mengenai tanggal pembuatan dari direktori tersebut
5. **Proteksi.** Merupakan atribut yang berguna sebagai proteksi. Hal ini mencakup siapa saja yang berhak mengakses, penyembunyian file, read only, dan yang lainnya. Dalam Unix, untuk mengubah atribut berkas digunakan perintah "chmod".

Atribut pada direktori dirancang sewaktu pembuatan sistem operasi tersebut, sehingga atribut yang ada bergantung pada para pembuat sistem operasi. Atribut di atas merupakan atribut yang umum dan lazim digunakan.

Perbedaan dari atribut pada direktori dan atribut pada berkas yaitu direktori tidak mempunyai tipe, sedangkan berkas mempunyai banyak tipe. Sedangkan pada Windows, perbedaan lainnya yaitu

direktori hanya mempunyai keterangan tanggal pembuatan, tidak mempunyai keterangan tanggal pemodifikasian ataupun tanggal pengaksesan. Pada Windows, kita dapat melihat atribut direktori dengan cara meng-klik kanan pada direktori, kemudian memilih properties.

38.3. Operasi Direktori

Silberschatz, Galvin dan Gagne mengkategorikan operasi-operasi terhadap direktori sebagai berikut:
- **Mencari Berkas.** Mencari lewat struktur direktori untuk dapat menemukan entri untuk suatu berkas tertentu. berkas-berkas dengan nama yang simbolik dan mirip, mengindikasikan adanya keterkaitan diantara berkas-berkas tersebut. Oleh karena itu, tentunya perlu suatu cara untuk menemukan semua berkas yang benar-benar memenuhi kriteria khusus yang diminta.
- **Membuat Berkas.** Berkas-berkas baru perlu untuk dibuat dan ditambahkan ke dalam direktori.
- **Menghapus Berkas.** Saat suatu berkas tidak diperlukan lagi, berkas tersebut perlu dihapus dari direktori.
- **Menampilkan isi Direktori.** Menampilkan daftar berkas-berkas yang ada di direktori, dan semua isi direktori dari berkas-berkas dalam daftar tersebut.
- **Mengubah nama berkas.** Nama berkas mencerminkan isi berkas terhadap pengguna. Oleh karena itu, nama berkas harus dapat diubah-ubah ketika isi dan kegunaannya sudah berubah atau tidak sesuai lagi. Mengubah nama berkas memungkinkan posisinya berpindah dalam struktur direktori.
- **Akses Sistem berkas.** Mengakses tiap direktori dan tiap berkas dalam struktur direktori. Sangatlah dianjurkan untuk menyimpan isi dan stuktur dari keseluruhan sistem berkas setiap jangka waktu tertentu. Menyimpan juga dapat berarti menyalin seluruh berkas ke pita magnetik. Teknik ini membuat suatu cadangan salinan dari berkas tersebut jika terjadi kegagalan sistem atau jika berkas itu tidak diperlukan lagi.

Sedangkan Tanenbaum juga menambahkan hal-hal berikut sebagai operasi yang dapat dilakukan terhadap direktori tersebut:
- Membuka direktori
- Menutup direktori
- Menambah direktori
- Mengubah nama direktori
- Menghubungkan berkas-berkas di direktori berbeda
- Menghapus hubungan berkas-berkas di direktori berbeda

Seiring berkembangnya sistem operasi, tentunya stuktur direktori juga mengalami perkembangan dan perbaikan. Berikut ini adalah jenis-jenis struktur direktori yang telah ada.

38.4. Direktori Satu Tingkat

Gambar 38.2. Single Level Directory



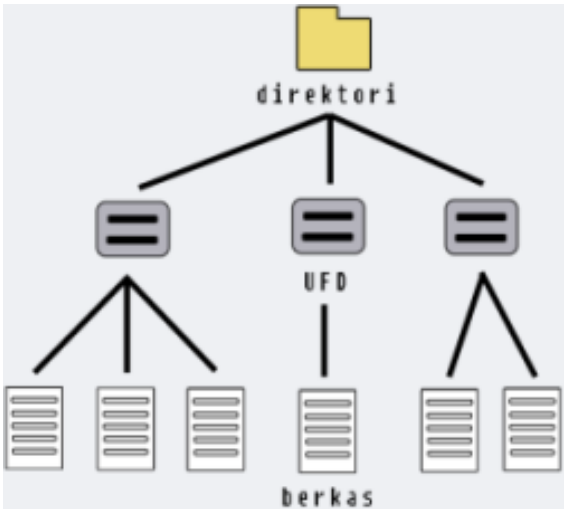
Direktori Satu Tingkat (*Single Level Directory*) ini merupakan struktur direktori yang paling sederhana. Semua berkas disimpan dalam direktori yang sama. Direktori satu tingkat memiliki keterbatasan, yaitu bila berkas bertambah banyak atau bila sistem memiliki lebih dari satu pengguna. Hal ini disebabkan karena tiap berkas harus memiliki nama yang unik.

38.5. Direktori Dua Tingkat

Direktori Dua Tingkat (*Two Level Directory*) membuat direktori yang terpisah untuk tiap pengguna, yang disebut *User File Directory* (UFD). Ketika pengguna login, *master directory* berkas dipanggil. MFD memiliki indeks berdasarkan nama pengguna dan setiap entri menunjuk pada UFD pengguna tersebut. Maka, pengguna boleh memiliki nama berkas yang sama dengan berkas lain.

Meski pun begitu, struktur ini masih memiliki kelemahan, terutama bila beberapa pengguna ingin mengerjakan sesuatu secara kerjasama dan ingin mengakses berkas milik pengguna lain. Beberapa sistem secara sederhana tidak mengizinkan berkas seorang pengguna diakses oleh pengguna lain.

Gambar 38.3. Two Level Directory



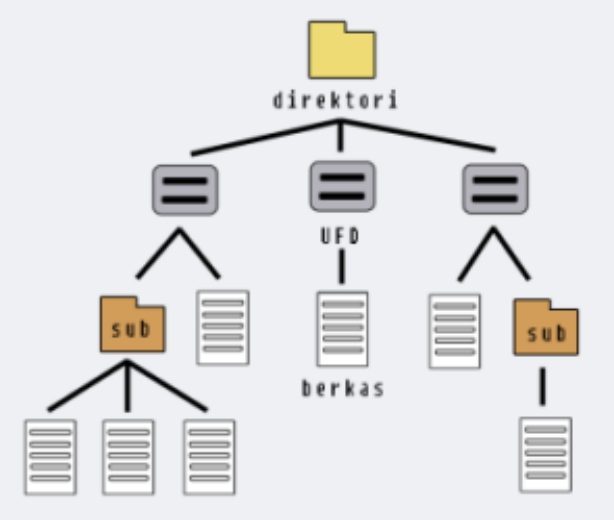
38.6. Direktori dengan Struktur Pohon

Pada direktori dengan Struktur Pohon (*Tree-Structured Directory*), setiap pengguna dapat membuat subdirektori sendiri dan mengorganisasikan berkas-berkasnya. Dalam penggunaan normal, tiap pengguna memiliki apa yang disebut direktori saat ini. Direktori saat ini mengandung berkas-berkas yang baru-baru ini digunakan oleh pengguna.

Terdapat dua istilah, *path* (lintasan) relatif dan lintasan mutlak. Lintasan relatif adalah lintasan yang dimulai dari direktori saat ini, sedangkan lintasan mutlak adalah path yang dimulai dari *root directory*.

Karena keterbatasan dari single level directory maka dibuat struktur two level directory, yaitu dengan membuat direktori yang terpisah untuk tiap pengguna. Dalam suatu sistem terdapat satu direktori utama yang disebut Master File Directory (MFD) dan beberapa direktori milik pengguna yang disebut User File Directory (UFD). Jumlah UFD yang ada sama dengan jumlah pengguna. MFD hanya memiliki informasi tentang UFD dan tidak memiliki informasi tentang berkas-berkas yang ada di dalam UFD. Ketika pengguna login maka MFD akan dipanggil. MFD memiliki indeks berdasarkan pengguna pengguna. Seorang pengguna dapat memiliki nama berkas yang sama dengan berkas milik pengguna lain. Keterbatasan dari struktur ini adalah dalam pengorganisasian berkas, pengguna hanya mempunyai satu direktori sehingga mereka tidak dapat mengelompokkan berkas miliknya.

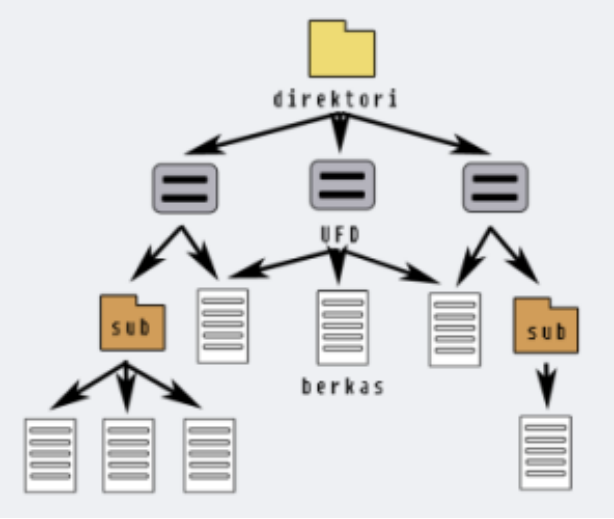
Gambar 38.4. Directori Struktur Pohon



38.7. Direktori dengan Struktur Graf Asiklik

Direktori dengan struktur pohon melarang pembagian berkas/direktori. Oleh karena itu, struktur graf asiklik (*Acyclic-Structured Directory*) memperbolehkan direktori untuk berbagi berkas atau subdirektori. Jika ada berkas yang ingin diakses oleh dua pengguna atau lebih, maka struktur ini menyediakan fasilitas *sharing*. Acyclic graph directory mengatasi permasalahan pada direktori dengan sruktur pohon, karena pada acyclic graph directory diperbolehkan adanya sharing berkas.

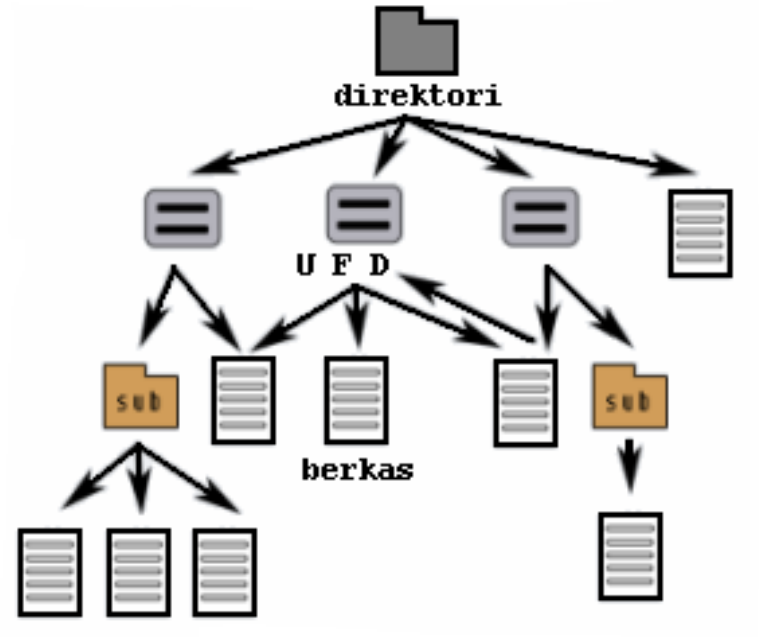
Gambar 38.5. Acyclic-Structured Directory



38.8. Direktori dengan Struktur Graf Umum

Masalah yang timbul dalam penggunaan struktur graf asiklik adalah meyakinkan apakah tidak ada siklus. Bila kita mulai dengan struktur direktori tingkat dua dan memperbolehkan pengguna untuk membuat subdirektori, maka kita akan mendapatkan struktur direktori pohon. Sangatlah mudah untuk mempertahankan sifat pohon, akan tetapi, bila kita tambahkan sambungan pada direktori dengan struktur pohon, maka sifat pohon akan musnah dan menghasilkan struktur graf sederhana.

Gambar 38.6. Graf Umum



Bila siklus diperbolehkan dalam direktori, tentunya kita tidak ingin mencari sebuah berkas dua kali. Algoritma yang tidak baik akan menghasilkan *infinite loop* dan tidak pernah berakhir. Oleh karena itu diperlukan skema pengumpulan sampah (*garbage-collection scheme*).

Skema ini menyangkut memeriksa seluruh sistem berkas dengan menandai tiap berkas yang dapat diakses. Kemudian mengumpulkan apa pun yang tidak ditandai pada tempat yang kosong. Hal ini tentunya dapat menghabiskan banyak waktu.

Pada direktori dengan struktur pohon, setiap pengguna dapat membuat direktori sendiri, sehingga dalam UFD akan terdapat direktori yang dibuat oleh pengguna dan dalam direktori itu juga dapat dibuat direktori lain (sub direktori), begitu seterusnya. Hal ini tentunya memudahkan pengguna dalam pengelompokan dan pengorganisasian file. File-file dapat dikelompokkan berdasarkan tipenya atau yang lainnya.

Masalah yang muncul adalah ketika pengguna ingin menggunakan suatu berkas secara bersama-sama. Karena sistem tidak mengizinkan seorang pengguna mengakses direktori pengguna lain.

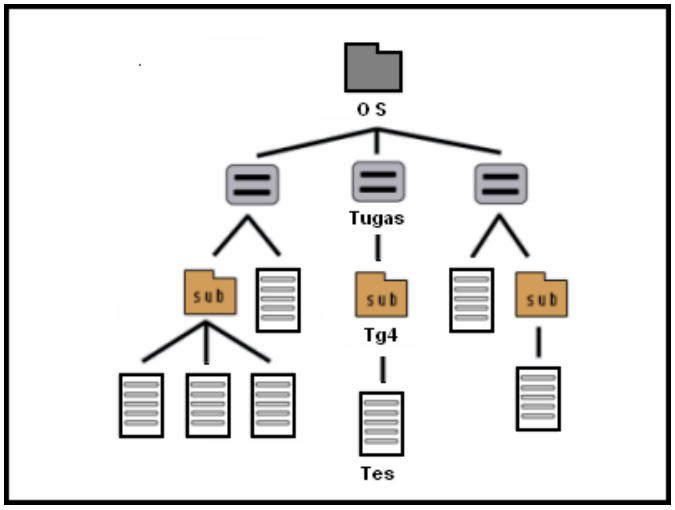
Pada general graph directory, sebuah direktori me-link pada direktori yang me-linknya. Dengan kata lain misalnya direktori A berisi/me-link direktori B. Maka ketika direktori B dibuka akan terdapat direktori A (ada siklus).

Dalam struktur direktori terdapat istilah path (lintasan). Terdapat dua jenis path, yaitu absolut path dan relatif path.
- **Absolut Path.** Berarti menuliskan lintasan sebuah berkas mulai dari root direktori sampai keberadaan sebuah berkas yang dituju.
- **Relatif Path.** Berarti menuliskan lintasan sebuah berkas mulai dari direktori saat ini (direktori yang sedang diakses pengguna) sampai keberadaan sebuah berkas yang dituju.

Misalkan pada gambar di bawah, kita sedang berada pada direktori Tg4, maka penulisan lintasan/path dari berkas Tes:

Relatif path yaitu "../Tg4/Tes".
- Absolut path yaitu "/OS/Tugas/Tg4/Tes".

Gambar 38.7. Path



38.9. *Mounting*

Mounting adalah proses mengkaitkan sebuah sistem berkas yang baru ditemukan pada sebuah piranti ke struktur direktori utama yang sedang dipakai. Piranti-piranti yang akan di-*mount* dapat berupa *cd-rom*, disket atau sebuah *zip-drive*. Tiap-tiap sistem berkas yang akan di-*mount* akan diberikan sebuah *mount point*, atau sebuah direktori dalam pohon direktori sistem Anda, yang sedang diakses.

Mounting bisa dilakukan secara remote maupun secara local. Kalau Anda menggunakan Linux, kemudian ingin membaca berkas yang terdapat dalam USB, maka Anda harus me-mount sistem berkas dalam USB itu secara manual. Sistem berkas yang dideskripsikan di /etc/fstab (fstab adalah singkatan dari *filesystem tables*) biasanya akan di-*mount* saat komputer baru mulai dinyalakan, tapi dapat juga me-*mount* sistem berkas tambahan dengan menggunakan perintah:

```
mount [nama piranti]
```

atau dapat juga dengan menambahkan secara manual *mount point* ke berkas /etc/fstab. Daftar sistem berkas yang di-*mount* dapat dilihat kapan saja dengan menggunakan perintah *mount*. Karena izinnya hanya diatur *read-only* di berkas fstab, maka tidak perlu khawatir pengguna lain akan mencoba mengubah dan menulis *mount point* yang baru.

Seperti biasa saat ingin mengutak-atik berkas konfigurasi seperti mengubah isi berkas fstab, pastikan untuk membuat berkas cadangan untuk mencegah terjadinya kesalahan teknis yang dapat menyebabkan suatu kekacauan. Kita dapat melakukannya dengan cara menyediakan sebuah disket atau *recovery-disk* dan mem-*back-up* berkas fstab tersebut sebelum membukanya di editor teks untuk diutak-atik.

GNU/Linux dan sistem operasi lainnya yang mirip dengan UNIX mengakses berkas dengan cara yang berbeda dari MS-DOS, Windows dan Macintosh. Di linux, segalanya disimpan di dalam sebuah lokasi yang dapat ditentukan dalam sebuah struktur data. Linux bahkan menyimpan perintah-perintah sebagai berkas. Seperti sistem operasi modern lainnya, Linux memiliki struktur pohon, hirarki, dan organisasi direktori yang disebut sistem berkas.

Semua ruang kosong yang tersedia di *disk* diatur dalam sebuah pohon direktori tunggal. Dasar sistem ini adalah direktori *root* yang dinyatakan dengan sebuah garis miring ("/"). Pada linux, isi sebuah sistem berkas dibuat nyata tersedia dengan menggabungkan sistem berkas ke dalam sebuah sistem direktori melalui sebuah proses yang disebut *mounting*.

Sistem berkas dapat di-*mount* maupun di-*umount* yang berarti sistem berkas tersebut dapat tersambung atau tidak dengan struktur pohon direktori. Perbedaannya adalah sistem berkas tersebut akan selalu di-*mount* ke direktori *root* ketika sistem sedang berjalan dan tidak dapat di-*mount*. Sistem berkas yang lain di-*mount* seperlunya, contohnya yang berisi *hard drive* berbeda dengan *floppy disk* atau CD-ROM.

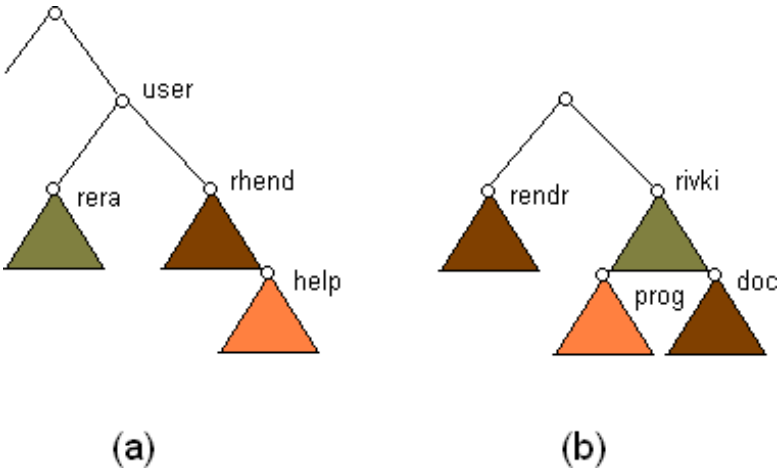
Mounting adalah memasukkan sistem berkas ke struktur direktori utama, baik ke dalam direktori kosong maupun ke dalam direktori yang sudah berisi. Hanya kalau dimasukkan ke direktori yang berisi, isi dari direktori itu tidak bisa diakses selama sistem berkas masih di-mount. Selama sistem berkas masih di-mount, isi yang akan terlihat saat membuka direktori itu adalah isi dari sistem berkas. Ketika sistem berkas telah di-unmount, barulah isi sesungguhnya dari direktori itu dapat terlihat.

Sebenarnya setiap akan memproses suatu sistem berkas (read and write) kita harus me-mount sistem berkas itu terlebih dahulu. Untungnya sistem operasi menyediakan fasilitas mounting secara otomatis pada saat sistem operasi dijalankan. Pada beberapa sistem operasi, ada device-device tertentu yang harus di-mount terlebih dahulu secara manual untuk memproses sistem berkas di dalamnya.

Untuk me-mount suatu sistem berkas, sistem operasi memerlukan data tentang device yang membawakan sistem berkas tersebut dan mountpoint tempat sistem berkas itu hendak diletakkan.

Mountpoint adalah direktori tempat di mana akan meletakkan sistem berkas tersebut. Kalau kita hendak me-mount sistem berkas berupa direktori, maka mountpointnya harus berupa direktori. Sebaliknya, jika yang hendak kita mount adalah file, maka mountpointnya juga harus berupa file.

Gambar 38.8. Mounting



Bisa juga dilakukan dengan cara memasukkan mountpoint ke berkas /etc/fstab. File ini menyimpan daftar sistem berkas yang akan di-mount secara otomatis ketika sistem operasi mulai berjalan beserta direktori lokal di mana mereka bisa diakses. Di berkas ini juga tersimpan data mengenai pengguna mana saja yang bisa me-mount suatu device.

Mounting membuat sistem berkas, direktori, piranti dan berkas lainnya menjadi dapat digunakan di lokasi-lokasi tertentu, sehingga memungkinkan direktori itu menjadi dapat diakses. Perintah *mount* menginstruksikan sistem operasi untuk mengkaitkan sebuah sistem berkas ke sebuah direktori khusus.

Memahami *Mount Point*

Mount point adalah sebuah direktori dimana berkas baru menjadi dapat diakses. Untuk me-*mount* suatu sistem berkas atau direktori, titik *mount*-nya harus berupa direktori, dan untuk me-*mount* sebuah berkas, *mount point*-nya juga harus berupa sebuah berkas.

Biasanya, sebuah sistem berkas, direktori, atau sebuah berkas di-*mount* ke sebuah *mount point* yang kosong, tapi biasanya hal tersebut tidak diperlukan. Jika sebuah berkas atau direktori yang akan menjadi *mount point* berisi data, data tersebut tidak akan dapat diakses selama direktori/berkas tersebut sedang dijadikan *mount point* oleh berkas atau direktori lain. Sebagai akibatnya, berkas yang di-*mount* akan menimpa apa yang sebelumnya ada di direktori/berkas tersebut. Data asli dari direktori itu dapat diakses kembali bila proses *mounting* sudah selesai.

Saat sebuah sistem berkas di-*mount* ke sebuah direktori, izin direktori *root* dari berkas yang di-*mount* akan mengambil alih izin dari *mount point*. Pengecualiannya adalah pada direktori induk akan memiliki atribut .. (*double dot*). Agar sistem operasi dapat mengakses sistem berkas yang baru, direktori induk dari *mount point* harus tersedia.

Untuk segala perintah yang membutuhkan informasi direktori induk, pengguna harus mengubah izin dari direktori *mounted-over*. Kegagalan direktori *mounted-over* untuk mengabulkan izin dapat menyebabkan hasil yang tidak terduga, terutama karena izin dari direktori *mounted-over* tidak dapat terlihat. Kegagalan umum terjadi pada perintah pwd. Tanpa mengubah izin direktori *mounted-over*, akan timbul pesan error seperti ini:

```
pwd: permission denied
```

Masalah ini dapat diatasi dengan mengatur agar izin setidaknya di-set dengan 111.

Mounting Sistem Berkas, Direktori, dan Berkas

Ada dua jenis *mounting*: *remote mounting* dan *mounting* lokal. *Remote mounting* dilakukan dengan sistem *remote* dimana data dikirimkan melalui jalur telekomunikasi. *Remote* sistem berkas seperti Network File Systems (NFS), mengharuskan agar file diekspor dulu sebelum di-*mount*. *mounting* lokal dilakukan di sistem lokal.

Tiap-tiap sistem berkas berhubungan dengan piranti yang berbeda. Sebelum kita menggunakan sebuah sistem berkas, sistem berkas tersebut harus dihubungkan dengan struktur direktori yang ada (dapat *root* atau berkas yang lain yang sudah tersambung).

Sebagai contoh, kita dapat me-*mount* dari /home/server/database ke *mount point* yang dispesifikasikan sebagai /home/user1, /home/user2, and /home/user3:

- /home/server/database /home/user1
- /home/server/database /home/user2
- /home/server/database /home/user3

38.10. Rangkuman

Beberapa sistem komputer menyimpan banyak sekali berkas-berkas dalam *disk*, sehingga diperlukan suatu struktur pengorganisasian data sehingga data lebih mudah diatur. Dalam struktur direktori satu tingkat, semua berkas diletakkan pada direktori yang sama, sehingga memiliki suatu keterbatasan karena nama berkas harus unik. Struktur direktori dua tingkat mencoba mengatasi masalah tersebut dengan membuat direktori yang terpisah untuk tiap pengguna yang disebut dengan *user file directory* (UFD). Sedangkan dalam struktur direktori pohon setiap pengguna dapat membuat subdirektori sendiri dan mengorganisasikan berkas-berkasnya. Direktori dengan struktur pohon melarang berbagi berkas atau direktori. Oleh karena itu, struktur dengan *acyclic-graph* memperbolehkan direktori untuk berbagi berkas atau sub-direktori. Struktur Direktori *general graph* mengatasi masalah yang timbul dalam struktur *acyclic* dengan metode *Garbage Collection*.

Mounting adalah proses mengaitkan sebuah sistem berkas yang baru ditemukan pada sebuah piranti ke struktur direktori utama yang sedang dipakai. *Mount point* adalah sebuah direktori dimana berkas baru menjadi dapat diakses.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBArpaciD2005] Andrea C Arpaci-Dusseau dan Remzi H Arpaci-Dusseau. 2005. *CS 537: Introduction to Operating Systems – File System: User Perspective – http://www.cs.wisc.edu/ ~remzi/ Classes/ 537/ Fall2005/ Lectures/ lecture18.ppt* . Diakses 8 Juli 2006.

[WEBBabicLauria2005] G Babic dan Mario Lauria. 2005. *CSE 660: Introduction to Operating Systems – Files and Directories – http://www.cse.ohio-state.edu/ ~lauria/ cse660/ Cse660.Files.04-08-2005.pdf* . Diakses 8 Juli 2006.

[WEBCarter2004] John Carter. 2004. *CS 5460 Operating Systems – Lecture 19: File System Operations and Optimizations – http://www.cs.utah.edu/ classes/ cs5460/ lectures/ lecture19.pdf* . Diakses 29 Mei 2006.

[WEBChung2005] Jae Chung. 2005. *CS4513 Distributed Computer Systems – File Systems – http://web.cs.wpi.edu/ ~goos/ Teach/ cs4513-d05/ slides/ fs1.ppt* . Diakses 7 Juli 2006.

[WEBCook2006] Tony Cook. 2006. *G53OPS Operating Systems – Directories – http://www.cs.nott.ac.uk/ ~acc/ g53ops/ lecture14.pdf* . Diakses 7 Juli 2006.

[WEBGooch1999] Richard Gooch. 1999. *Overview of the Virtual File System – http://www.atnf.csiro.au/ people/ rgooch/ linux/ docs/ vfs.txt* . Diakses 29 Mei 2006.

[WEBIBM1997] IBM Coorporation. 1997. *General Programming Concepts: Writing and Debugging Programs – Threads Scheduling http://www.unet.univie.ac.at/ aix/ aixprggd/ genprogc/ threads_sched.htm* . Diakses 1 Juni 2006.

[WEBJeffay2005] Kevin Jeffay. 2005. *Secondary Storage Management – http://www.cs.unc.edu/ ~jeffay/ courses/ comp142/ notes/ 15-SecondaryStorage.pdf* . Diakses 7 Juli 2006.

[WEBKaram1999] Vijay Karamcheti. 1999. *Honors Operating Systems – Lecture 15: File Systems – http://cs.nyu.edu/ courses/ spring99/ G22.3250-001/ lectures/ lect15.pdf* . Diakses 5 Juli 2006.

[WEBKessler2005] Christhope Kessler. 2005. *File System Interface – http://www.ida.liu.se/ ~TDDB72/ slides/ 2005/ c10.pdf* . Diakses 7 Juli 2006.

[WEBLee2000] Insup Lee. 2000. *CSE 380: Operating Systems – File Systems – http://www.cis.upenn.edu/ ~lee/ 00cse380/ lectures/ ln11b-fil.ppt* . Diakses 7 Juli 2006.

[WEBRamam2005] B Ramamurthy. 2005. *File Management – http://www.cse.buffalo.edu/ faculty/ bina/ cse421/ spring2005/ FileSystemMar30.ppt* . Diakses 5 Juli 2006.

[WEBWiki2006g] From Wikipedia, the free encyclopedia. 2006. *File system – http://en.wikipedia.org/ wiki/ File_system* . Diakses 04 Juli 2006.

Bab 39. Aspek Jaringan dan Keamanan

39.1. Pendahuluan

Kita dapat berbagi berkas dengan pengguna lainnya yang teregistrasi. Hal pertama yang harus kita lakukan adalah menentukan dengan siapa berkas tersebut akan dibagi dan akses seperti apa yang akan diberikan kepada mereka. Berbagi bekas berguna bagi pengguna yang ingin bergabung dengan pengguna lain dan mengurangi usaha untuk mencapai sebuah hasil akhir.

Saat sebuah sistem operasi dibuat untuk *multiple user*, masalah berbagi berkas, penamaan berkas dan proteksi berkas menjadi sangat penting. Oleh karena itu, sistem operasi harus dapat mengakomodasikan/mengatur pembagian berkas dengan memberikan suatu struktur direktori yang membiarkan pengguna untuk saling berbagi.

Berkaitan dengan permasalahan akses berkas, kita dapat mengizinkan pengguna lain untuk melihat, mengedit atau menghapus suatu berkas. Proses mengedit berkas yang menggunakan *web-file system* berbeda dengan menggunakan aplikasi seperti Windows Explorer. Untuk mengedit sebuah file dengan *web-file system*, kita harus menduplikasi berkas tersebut dahulu dari *web-file system* ke komputer lokal, mengeditnya di/ komputer lokal, dan mengirim file tersebut kembali ke sistem dengan menggunakan nama berkas yang sama.

Sebagai contoh, kita dapat mengizinkan semua pengguna yang terdaftar untuk melihat berkas-berkas yang ada di direktori (tetapi mereka tidak dapat mengedit atau menghapus berkas tersebut). Contoh lainnya, kita dapat mengizinkan satu pengguna saja untuk melakukan apa pun terhadap sebuah direktori dan segala isinya (izin untuk melihat semua berkas, mengeditnya, menambah berkas bahkan menghapus isi berkas). Kita juga dapat memberikan kesempatan bagi pengguna untuk mengubah izin dan kontrol akses dari sebuah isi direktori, namun hal tersebut biasanya di luar kebiasaan, sebab seharusnya satu-satunya pengguna yang berhak mengubah izin adalah kita sendiri.

Sistem berkas web memungkinkan kita untuk menspesifikasikan suatu akses dalam tingkatan berkas. Jadi, kita dapat mengizinkan seluruh orang untuk melihat isi dari sebuah direktori atau mengizinkan sebagian kecil pengguna saja untuk mengakses suatu direktori. Bahkan, dalam kenyataannya, kita dapat menspesifikasikan jenis akses yang berbeda dengan jumlah pengguna yang berbeda pula.

Kebanyakan pada sistem banyak pengguna menerapkan konsep direktor berkas *owner/user* dan group.

- *Owner*: pengguna yang dapat mengubah atribut, memberikan akses, dan memiliki sebagian besar kontrol di dalam sebuah berkas atau direktori.
- *Group*: sebagian pengguna yang sedang berbagi berkas.

Sebagian besar sistem mengimplementasikan atribut owner dengan mengatur daftar pengguna name dan mengasosiasikannya dengan pengguna ID. Saat pengguna log-in ke sistem, akan dicek apakah pengguna ID-nya tepat atau tidak. User ID diasosiasikan dengan seluruh proses dan thread yang berkaitan dengan pengguna.

Fungsionalitas group juga dapat didaftarkan sebagai group name dan group identifier. Setiap pengguna dapat tergabung dalam satu atau lebih group, tergantung dari desain sistem operasi yang digunakan.

Owner dan group ID dari sebuah berkas atau direktori disimpan bersama atribut berkas lainnya. Ketika pengguna melakukan operasi terhadap berkas, user ID dan group ID akan dicocokkan dengan atribut berkas yang telah disimpan. Hasil pengecekan user ID dan group ID menentukan apakah permintaan melakukan operasi terhadap berkas diizinkan atau tidak.

39.2. *Remote File System*

Jaringan menyebabkan berbagi data terjadi di seluruh dunia. Dalam metode implementasi pertama, yang digunakan untuk berbagi data adalah program FTP (*File Transfer Protocol*). User secara manual mentransfer berkas antara mesin melalui program. FTP digunakan untuk akses anonim (mentransfer file tanpa memiliki account di sistem remote) dan akses autentik (membutuhkan izin). WWW biasanya menggunakan akses anonim, dan DFS menggunakan akses autentik.

Yang kedua terbesar adalah DFS (*Disributed File System*) yang memungkinkan remote direktori terlihat dari mesin lokal. Remote direktori dapat dipantau dari local machine. DFS mencakup integrasi yang lebih kompleks antara mesin yang mengakses remote berkas dan mesin yang menyediakan berkas.

Metode yang ketiga adalah WWW (*World Wide Web*) Pada remote berkas sistem, dikenal akses anonim dan akses autentik. Akses anonim di mana pengguna dapat mentransfer berkas tanpa harus memiliki account di remote sistem. Akses autentik adalah akses yang membutuhkan izin.

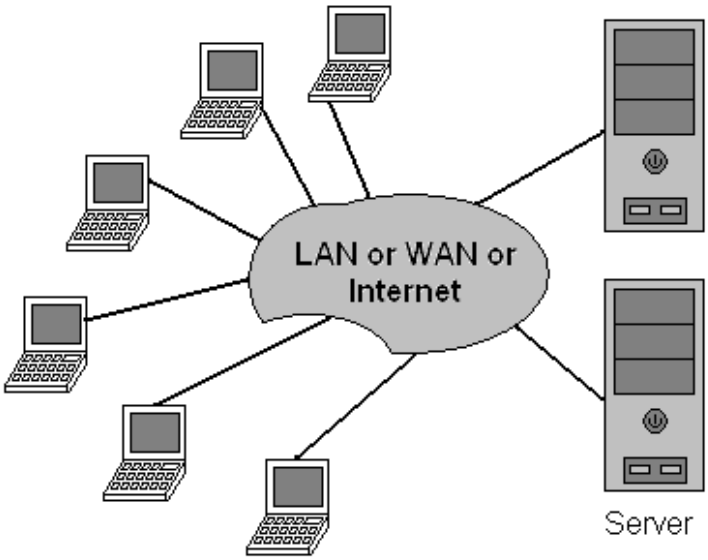
39.3. Model *Client-Server*

Server dapat melayani banyak pengguna dan klien dapat menggunakan banyak server. Proses identifikasi klien biasanya sulit, dan cara yang biasa digunakan adalah melacak alamat IP, namun karena alamat IP dapat dipalsukan, cara ini menjadi kurang efektif. Ada juga yang menggunakan proses kunci terenkripsi, namun hal ini lebih rumit lagi, sebab klien-server harus menggunakan algoritma enkripsi yang sama dan pertukaran kunci yang aman.

Remote berkas sistem memungkinkan komputer untuk melakukan proses mounting lebih dari satu sistem berkas dari satu atau lebih remote machine. Mesin yang berisi berkas disebut server. Mesin yang mengakses berkas disebut client. Satu server dapat melayani lebih dari satu client dan satu client dapat menggunakan lebih dari satu server, tergantung dari fasilitas client server.

Client dapat dispesifikasi melalui network name atau IP address. Namun, IP address dapat dipalsukan sehingga proses identifikasi dengan cara ini tidak efektif. Cara lain dengan menggunakan proses kunci terenkripsi. Sayangnya hal ini agak rumit karena client dan server harus menggunakan algoritma enkripsi dan pertukaran kunci yang aman.

Gambar 39.1. Client Server



39.4. Consistency Semantics

Konsistensi semantik merupakan kriteria penting dalam evaluasi sistem berkas yang menunjang berkas berbagi. Konsistensi semantik menunjukkan karakteristik sistem yang menspesifikasi semantik dari pengguna ganda yang mengakses berkas yang sama secara simultan. Konsistensi semantik berhubungan langsung dengan algoritma pada proses sinkronisasi.

Beberapa contoh penting konsistensi semantik sebagai berikut:
1. **UNIX Semantics.** Apa yang ditulis pengguna pada sebuah open berkas dapat dilihat pengguna lain yang juga sedang membuka berkas yang sama Sharing memungkinkan pengguna untuk berbagi pointer
2. **Session Semantics.** Apa yang ditulis pengguna pada sebuah open berkas tidak dapat dilihat pengguna lain yang juga sedang membuka berkas yang sama. Setelah berkas itu di-close, perubahan yang terjadi karena ada pengguna yang menulis berkas dapat dilihat.
3. **Immutable-Shared Files Semantics.** Sebuah immutable berkas tidak dapat dimodifikasi. Walaupun beberapa pengguna mengakses immutable file, isi berkas tidak dapat diubah.

39.5. Proteksi

Dalam pembahasan mengenai proteksi berkas, kita akan berbicara lebih mengenai sisi keamanan dan mekanisme bagaimana menjaga keutuhan suatu berkas dari gangguan akses luar yang tidak dikehendaki. Sebagai contoh bayangkan saja Anda berada di suatu kelompok kerja dimana masing-masing staf kerja disediakan komputer dan mereka saling terhubung membentuk suatu jaringan; sehingga setiap pekerjaan/dokumen/berkas dapat dibagi-bagikan ke semua pengguna dalam jaringan tersebut. Misalkan lagi Anda harus menyerahkan berkas RAHASIA.txt ke atasan Anda, dalam hal ini Anda harus menjamin bahwa isi berkas tersebut tidak boleh diketahui oleh staf kerja lain apalagi sampai dimodifikasi oleh orang yang tidak berwenang. Suatu mekanisme pengamanan berkas mutlak diperlukan dengan memberikan batasan akses ke setiap pengguna terhadap berkas tertentu.

Ketika suatu informasi disimpan dalam sistem komputer, kita harus menjaganya dari kerusakan fisik (reliability) dan akses yang tidak diinginkan (protection). Reliability biasanya ditanggulangi oleh duplikasi berkas, sedangkan proteksi dapat dilakukan dengan banyak cara. Untuk sistem pengguna tunggal, proteksi dapat dilakukan dengan cara memindahkan berkas ke floppy disk dan menguncinya dalam berkas cabinet. Namun, mekanisme proteksi lebih dibutuhkan dalam sistem pengguna-jamak.

39.6. Tipe Akses

Proteksi berkaitan dengan kemampuan akses langsung ke berkas tertentu. Panjangnya, apabila suatu sistem telah menentukan secara pasti akses berkas tersebut selalu ditutup atau selalu dibebaskan ke setiap pengguna lain maka sistem tersebut tidak memerlukan suatu mekanisme proteksi. Tetapi tampaknya pengimplementasian seperti ini terlalu ekstrim dan bukan pendekatan yang baik. Kita perlu membagi akses langsung ini menjadi beberapa jenis-jenis tertentu yang dapat kita atur dan ditentukan (akses yang terkontrol).

Dalam pendekatan ini, kita mendapatkan suatu mekanisme proteksi yang dilakukan dengan cara membatasi jenis akses ke suatu berkas. Beberapa jenis akses tersebut antara lain:
- **Baca (*Read*).** Membaca berkas.
- **Tulis (*Write*).** Menulis Berkas.
- **Eksekusi (*Execute*).** Memasukkan berkas ke memori dan dieksekusi.
- **Sisip (*Append*).** Menulis informasi baru pada baris akhir berkas.
- **Hapus (*Delete*).** Menghapus berkas.
- **Daftar (*List*).** Mendaftar nama dan atribut berkas.

Operasi lain seperti *rename*, *copying*, atau *editing* yang mungkin terdapat di beberapa sistem merupakan gabungan dari beberapa jenis kontrol akses diatas. Sebagai contoh, menyalin sebuah berkas dikerjakan sebagai runtutan permintaan baca dari pengguna. Sehingga dalam hal ini, seorang pengguna yang memiliki kontrol akses *read* dapat pula meng-*copy*, mencetak dan sebagainya.

39.7. Kontrol Akses

Pendekatan yang paling umum dipakai dalam mengatasi masalah proteksi berkas adalah dengan membiarkan akses ke berkas ditentukan langsung oleh pengguna (dalam hal ini pemilik/pembuat berkas itu). Sang pemilik bebas menentukan jenis akses apa yang diperbolehkan untuk pengguna lain. Hal ini dapat dilakukan dengan menghubungkan setiap berkas atau direktori dengan suatu daftar kontrol-akses (*Access-Control Lists/ACL*) yang berisi nama pengguna dan jenis akses apa yang diberikan kepada pengguna tersebut.

Sebagai contoh dalam suatu sistem VMS, untuk melihat daftar direktori berikut daftar kontrol-akses, ketik perintah "DIR/SECURITY", atau "DIR/SEC". Salah satu keluaran perintah itu adalah daftar seperti berikut ini:

```
WWW-HOME.DIR;1       [HMC2000,WWART]              (RW,RWED,,E)
    (IDENTIFIER=WWW_SERVER_ACCESS,OPTIONS=DEFAULT,ACCESS=READ)
    (IDENTIFIER=WWW_SERVER_ACCESS,ACCESS=READ)
```

Baris pertama menunjukkan nama berkas tersebut WWW-HOME.DIR kemudian disebelahnya nama grup pemilik HMC2000 dan nama pengguna WWART diikuti dengan sekelompok jenis akses RW, RWED,,E (R=Baca, W=Tulis, E=Eksekusi, D=Hapus). Dua baris dibawahnya itulah yang disebut daftar konrol-akses. Satu-satu baris disebut sebagai masukan kontrol-akses (*Access Control Entry/ACE*) dan terdiri dari 3 bagian. Bagian pertama disebut sebagai IDENTIFIER/Identifikasi, menyatakan nama grup atau nama pengguna (seperti [HMC2000, WWART]) atau akses khusus (seperti WWW_SERVER_ACCESS). Bagian kedua merupakan daftar OPTIONS/Plihan-pilihan. Dan terakhir adalah daftar izin ACCESS/akses, seperti *read* atau *execute*, yang diberikan kepada siapa saja yang mengacu pada bagian Identifikasi.

Cara kerjanya: apabila seorang pengguna meminta akses ke suatu berkas/direktori, sistem operasi akan memeriksa ke daftar kontrol-akses apakah nama pengguna itu tercantum dalam daftar tersebut. Apabila benar terdaftar, permintaan akses akan diberikan dan sebaliknya bila tidak, permintaan akses akan ditolak.

Pendekatan ini memiliki keuntungan karena penggunaan metodologi akses yang kompleks sehingga sulit ditembus sembarangan. Masalah utamanya adalah ukuran dari daftar akses tersebut. Bayangkan apabila kita mengizinkan semua orang boleh membaca berkas tersebut, kita harus mendaftar semua nama pengguna disertai izin akses baca mereka. Lebih jauh lagi, tehnik ini memiliki dua konsekuensi yang tidak diinginkan:

- Pembuatan daftar semacam itu merupakan pekerjaan yang melelahkan dan tidak efektif.
- Entri direktori yang sebelumnya memiliki ukuran tetap, menjadi ukuran yang dapat berubah-ubah, mengakibatkan lebih rumitnya manajemen ruang kosong.

Masalah ini dapat diselesaikan dengan penggunaan daftar akses yang telah disederhanakan.

Untuk menyederhanakan ukuran daftar kontrol akses, banyak sistem menggunakan tiga klasifikasi pengguna sebagai berikut:

- *Owner*: pengguna yang telah membuat berkas tersebut.
- *Group*: sekelompok pengguna yang saling berbagi berkas dan membutuhkan akses yang sama.
- *Universe*: keseluruhan pengguna.

Pendekatan yang dipakai belum lama ini adalah dengan mengkombinasikan daftar kontrol-akses dengan konsep kontrol- akses pemilik, grup dan semesta yang telah dijabarkan diatas. Sebagai contoh, Solaris 2.6 dan versi berikutnya menggunakan tiga klasifikasi kontrol-akses sebagai pilihan umum, tetapi juga menambahkan secara khusus daftar kontrol-akses terhadap berkas/direktori tertentu sehingga semakin baik sistem proteksi berkasnya.

Masalah yang paling penting dalam proteksi berkas adalah membuat akses yang yang bergantung pada identitas pengguna yang mengakses berkas. Implementasi yang umum untuk menerapkan akses yang bergantung pada identitas sebuah berkas atau objek adalah Access Control List (ACL). ACL menspesifikasikan nama pengguna dan tipe akses yang mana yang dizinkan untuk setiap pengguna. Akan tetapi, terdapat kelemahan jika mengimplementasikan ACL untuk proteksi berkas:

1. Harus melist satu persatu pengguna terhadap tipe akses yang diizinkan terhadap berkas.
2. Manajemen ruang kosong pada memori akan lebih rumit.

Penulisannya adalah file/objek (owner, group, right). Sebagai contoh, ada empat pengguna (A, B , C, dan D) yang masing-masing termasuk dalam group system, staff, dan student.

```
File0 (A,*, RWX)
File1 (A, system, RWX)
File2 (A, *, RW-) (B, staff, R--) (D , *, RW-)
File3 (*, student, R--)
File4 (C,*,---) (*, student, R--)
```

File0 dapat dibaca, dieksekusi dan ditulis oleh pengguna A pada semua group yang ada. File1 dapat dibaca, dieksekusi dan ditulis oleh pengguna A pada group system. File2 dapat dibaca dan ditulis oleh pengguna A dan D pada semua group, dibaca oleh pengguna B pada group staff. File3 dapat dibaca oleh semua member dari group student. File4 memiliki keistimewaan yaitu ia mengatakan bahwa pengguna C di setiap group tidak memiliki akses apapun, tetapi semua member group student dapat membacanya, dengan mengunakan ACL memungkinkan menjelaskan spesifik pengguna, group yang mengakses sebuah berkas atau objek.

Proteksi berkas juga dapat dilakukan dengan menggunakan password. Mekanisme kendali akses dengan password adalah memberikan password untuk setiap berkas yang akan diproteksi. Proteksi berkas dengan password akan efektif jika kombinasi password yang dipilih acak dan sering diganti secara berkala. Namun, kendali akses dengan password memiliki kelemahan, antara lain:

1. Tidak praktis karena banyak password yang harus diingat pengguna.
2. Jika hanya menggunakan satu password untuk seluruh berkas dan ada orang lain yang mengetahui password itu, maka seluruh berkas dapat diakses tanpa proteksi.

39.8. *File Permission* dalam UNIX

Contoh lain yaitu sistem UNIX dimana konrol-aksesnya dinyatakan dalam tiga bagian. Masing-masing bagian merupakan klasifikasi pengguna (yaitu pemilik, grup dan semesta). Setiap bagian kemudian dibagi lagi menjadi tiga bit jenis akses -*rwx*, dimana *r* mengontrol akses baca, *w* mengontrol akses tulis dan *x* mengontrol eksekusi. Dalam pendekatan ini, 9 bit diperlukan untuk merekam seluruh informasi proteksi berkas.

Berikut adalah keluaran dari perintah "ls -al" di sistem UNIX:

```
      -rwxr-x---  1  david karyawan 12210 Nov 14 20:12 laporan.txt
```

Baris di atas menyatakan bahwa berkas laporan.txt memiliki akses penuh terhadap pemilik berkas (david), grupnya hanya dapat membaca dan mengeksekusi, sedang lainnya tidak memiliki akses sama sekali.

File permission dalam UNIX terdiri dari 10 bit, yang dikelompokkan dalam empat bagian:

i. 1 bit (MSB), informasi tentang jenis berkas.
ii. 3 bit (ke-2,3,4), adalah izin untuk pemilik berkas (pengguna, u).
iii. 3 bit (ke-5,6,7), adalah izin untuk group dari pemilik berkas (group, g).
iv. 3 bit (ke-8,9,10), adalah izin untuk pengguna lain (other, o).

Permission setiap berkas dapat anda lihat dengan menggunakan perintah "ls -l". Contohnya:

```
-rw-w--w---  1  riv  student  200  Des  23   12:50   117-39.tar.gz
drw-w--w---  1  riv  student  200  Des  20   13:00   Operating
drw-w--w---  1  riv  student  200  Des  20   13:00   System
-rw-w--w---  1  riv  student  200  Des  12   14:00   tes.txt
```

Arti digit pada permission pada berkas tersebut adalah:

1. d (direktori), menunjukkan bahwa jenis berkas adalah direktori.

2. r (read), menunjukkan bahwa berkas dapat dibaca.

3. w (write), menunjukkan bahwa berkas dapat ditulis.

4. x (execute), menunjukkan bahwa berkas dapat dieksekusi.

5. s (suid), menunjukkan bahwa berkas dapat dieksekusi sebagai program yang membutuhkan level root, karena program ini dimiliki sistem. Jadi permission ini memungkinkan pengguna yang bukan root dapat menjalankannya.

6. - (tidak ada). Jika pada bit pertama, berarti berkas tersebut bukan sebuah direktori. Jika pada posisi bit lainnya berarti berkas tersebut tidak dapat diakses dengan (r, w, x).

Untuk memodifikasi permission suatu berkas, kita dapat mengunakan peintah chmod.

39.9. Pendekatan Pengamanan Lainnya

Salah satu pendekatan lain terhadap masalah proteksi adalah dengan memberikan sebuah kata kunci (*password*) ke setiap berkas. Jika kata-kata kunci tersebut dipilih secara acak dan sering diganti, pendekatan ini sangatlah efektif sebab membatasi akses ke suatu berkas hanya diperuntukkan bagi pengguina yang mengetahui kata kunci tersebut.

Meski pun demikian, pendekatan ini memiliki beberapa kekurangan, diantaranya:

- Kata kunci yang perlu diingat oleh pengguna akan semakin banyak, sehingga membuatnya menjadi tidak praktis.
- Jika hanya satu kata kunci yang digunakan di semua berkas, maka jika sekali kata kunci itu diketahui oleh orang lain, orang tersebut dapat dengan mudah mengakses semua berkas lainnya. Beberapa sistem (contoh: TOPS-20) memungkinkan seorang pengguna untuk memasukkaan sebuah kata kunci dengan suatu subdirektori untuk menghadapi masalah ini, bukan dengan satu berkas tertentu.
- Umumnya, hanya satu kata kunci yang diasosiasikan dengan semua berkas lain. Sehingga, pengamanan hanya menjadi semua-atau-tidak sama sekali. Untuk mendukung pengamanan pada tingkat yang lebih mendetail, kita harus menggunakan banyak kata kunci.

39.10. Rangkuman

Berbagi berkas bermanfaat untuk mereduksi usaha untuk mencapai tujuan komputasi. Masalah penamaan, proteksi, dan berbagi berkas menjadi sangat penting apabila sebuah sistem operasi dikembangkan untuk pengguna majemuk. Masalah berbagi berkas terkait dengan sistem berkas jarak jauh (*remote*) dan model *client-server*. Konsistensi semantik perlu diperhatikan jika membahas berbagi berkas.

Tujuan proteksi berkas adalah mencegah akses yang tidak diinginkan ketika berbagi berkas. Diterima atau tidaknya operasi pada berkas salah satunya bergantung pada tipe akses. Mekanisme proteksi berkas di antaranya mengimplementasikan *Access Control List* (ACL) dan proteksi dengan password.

Mounting adalah proses mengaitkan sebuah sistem berkas yang baru ditemukan pada sebuah piranti ke struktur direktori utama yang sedang dipakai. Mount point adalah sebuah direktori dimana berkas baru menjadi dapat diakses.

Rujukan

[Sidik2004] Betha Sidik. 2004. *Unix dan Linux*. Informatika. Bandung.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[WEBCarter2004] John Carter. 2004. *CS 5460 Operating Systems – Lecture 19: File System Operations and Optimizations – http://www.cs.utah.edu/ classes/ cs5460/ lectures/ lecture19.pdf* . Diakses 29 Mei 2006.

[WEBGooch1999] Richard Gooch . 1999. *Overview of the Virtual File System – http://www.atnf.csiro.au/ people/ rgooch/ linux/ docs/ vfs.txt* . Diakses 29 Mei 2006.

[WEBIBM2003] IBM Corporation. 2003. *System Management Concepts: Operating System and Devices – http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690 / IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm* . Diakses 29 Mei 2006.

[WEBITCUV2006] IT& University of Virginia. 2006. *Mounting File Systems (Linux) – http://www.itc.virginia.edu/ desktop/ linux/ mount.html* . Diakses 20 Juli 2006.

[WEBJonesSmith2000] David Jones dan Stephen Smith. 2000. *85349 – Operating Systems – Study Guide – http://www.infocom.cqu.edu.au/ Courses/ aut2001/ 85349/ Resources/ Study_Guide/ 85349.pdf* . Diakses 20 Juli 2006.

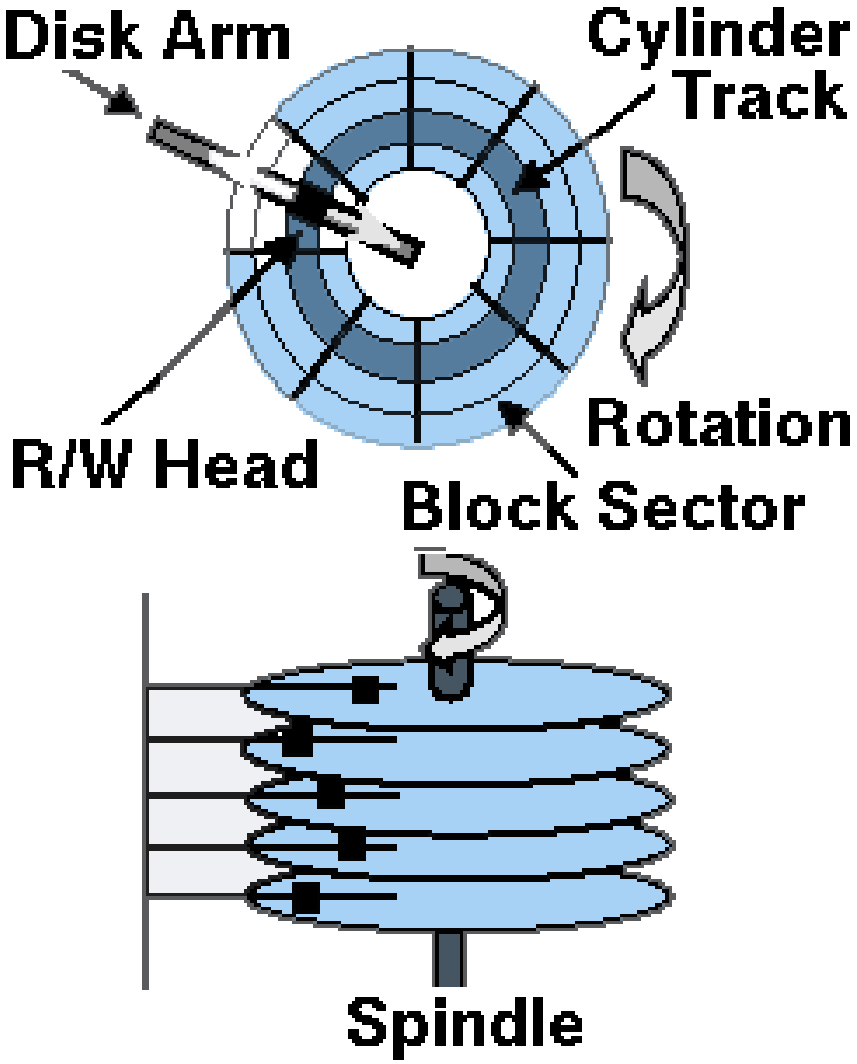
Bab 40. Implementasi Sistem Berkas

40.1. Pendahuluan

Disk – tempat terdapatnya sistem berkas – menyediakan sebagian besar tempat penyimpanan dimana sistem berkas akan dikelola. *Disk* memiliki dua karakteristik penting yang menjadikan *disk* sebagai media yang tepat untuk menyimpan berbagai macam berkas, yaitu:

- Data dapat ditulis ulang di *disk* tersebut, hal ini memungkinkan untuk membaca, memodifikasi, dan menulis di *disk* tersebut.
- Dapat diakses langsung ke setiap blok di *disk*. Hal ini memudahkan untuk mengakses setiap berkas baik secara berurut maupun tidak berurut, dan berpindah dari satu berkas ke berkas lain dengan hanya mengangkat *head disk* dan menunggu *disk* berputar.

Gambar 40.1. Organisasi Disk



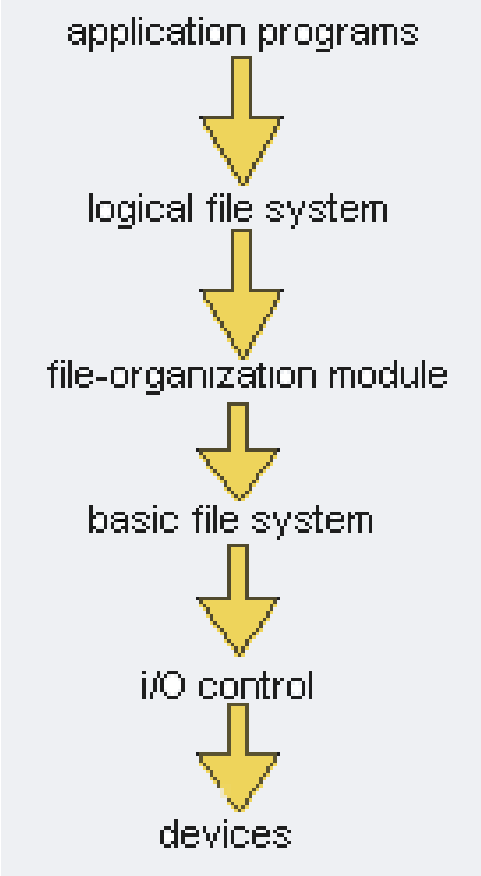
Untuk meningkatkan efisiensi M/K, pengiriman data antara memori dan *disk* dilakukan dalam setiap blok. Setiap *blok* merupakan satu atau lebih sektor. Setiap *disk* memiliki ukuran yang berbeda-beda, biasanya berukuran 512 *bytes*.

Gambar 40.2. File Control Block

Izin Berkas
Data Berkas
Kepemilikan Berkas
Ukuran Berkas
Blok Data Berkas

Sistem operasi menyediakan sistem berkas agar data mudah disimpan, diletakkan dan diambil kembali dengan mudah. Terdapat dua masalah desain dalam membangun suatu sistem berkas. Masalah pertama adalah definisi dari sistem berkas. Hal ini mencakup definisi berkas dan atributnya, operasi ke berkas, dan struktur direktori dalam mengorganisasikan berkas-berkas. Masalah kedua adalah membuat algoritma dan struktur data yang memetakan struktur logikal sistem berkas ke tempat penyimpanan sekunder.

Gambar 40.3. *Layered File System*



Sistem berkas dari sistem operasi modern diimplementasikan dengan menggunakan struktur berlapis. Keuntungan struktur berlapis ini adalah fleksibilitas yang dimilikinya. Penggunaan dari struktur berlapis ini memungkinkan adanya implementasi yang lebih dari satu secara bersamaan, terutama pada kendali M/K dan tingkatan organisasi berkas. Hal ini memungkinkan untuk mendukung lebih dari satu implementasi sistem berkas.

Lapisan struktur sistem berkas menghubungkan antara perangkat keras dengan aplikasi program yang ada, yaitu (dari yang terendah):

- Kendali M/K, terdiri atas *driver device* dan *interrupt handler*. *Driver device* adalah perantara komunikasi antara sistem operasi dengan perangkat keras. *Input* didalamnya berisikan perintah tingkat tinggi seperti "ambil blok 133", sedangkan *output*-nya adalah perintah tingkat rendah, instruksi spesifik perangkat keras yang digunakan oleh *controller* perangkat keras.
- *Basic file system*, diperlukan untuk mengeluarkan perintah *generic* ke *device driver* untuk *read* dan *write* pada suatu blok dalam *disk*.
- *File-organization module*, informasi tentang alamat logika dan alamat fisik dari berkas tersebut. Modul ini juga mengatur sisa *disk* dengan melacak alamat yang belum dialokasikan dan menyediakan alamat tersebut saat *pengguna* ingin menulis berkas ke dalam *disk*. Di dalam *File-organization module* juga terdapat *free- space manager*.
- *Logical file-system*, tingkat ini berisi informasi tentang simbol nama berkas, struktur dari direktori, dan proteksi dan sekuriti dari berkas tersebut. Sebuah *File Control Block* (FCB) menyimpan informasi tentang berkas, termasuk kepemilikan, izin dan lokasi isi berkas.

Di bawah ini merupakan contoh dari kerja struktur berlapis ini ketika suatu program mau membaca informasi dari *disk*. Urutan langkahnya:

1. ***Application program* memanggil sistem berkas dengan *system call*.** Contoh: *read (fd, input, 1024)* akan membaca *section* sebesar 1 Kb dari *disk* dan menempatkannya ke variabel *input*.
2. **Diteruskan ke *system call interface*.** System call merupakan *software interrupt*. Jadi, *interrupt handler* sistem operasi akan memeriksa apakah *system call* yang menginterupsi. *Interrupt handler* ini akan memutuskan bagian dari sistem operasi yang bertanggung-jawab untuk menangani *system call*. *Interrupt handler* akan meneruskan *system call*.
3. **Diteruskan ke *logical file system*.** Memasuki lapisan sistem berkas. Lapisan ini menyediakan *system call*, operasi yang akan dilakukan dan jenis berkas. Yang perlu ditentukan selanjutnya adalah *file organization module* yang akan meneruskan permintaan ini. *File organization module* yang akan digunakan tergantung dari jenis sistem berkas dari berkas yang diminta. Contoh: Misalkan kita menggunakan Linux dan berkas yang diminta ada di Windows 95. Lapisan *logical file system* akan meneruskan permintaan ke *file organization module* dari Windows 95.
4. **Diteruskan ke *file organization module*.** *File organization module* yang mengetahui pengaturan (organisasi) direktori dan berkas pada *disk*. Sistem berkas yang berbeda memiliki organisasi yang berbeda. Windows 95 menggunakan VFAT-32. Windows NT menggunakan format NTFS. Linux menggunakan EXT2. Sistem operasi modern memiliki beberapa *file organization module* sehingga dapat membaca format yang berbeda.Pada contoh di atas, *logical file system* telah meneruskan permintaan ke *file organization module* VFAT32. Modul ini menterjemahkan nama berkas yang ingin dibaca ke lokasi fisik yang biasanya terdiri dari *disk* antarmuka, *disk drive*, *surface*, *cylinder*, *track*, *sector*.
5. **Diteruskan ke *basic file system*.** Dengan adanya lokasi fisik, kita dapat memberikan perintah ke piranti keras yang dibutuhkan. Hal ini merupakan tanggung-jawab *basic file system*. *Basic file system* ini juga memiliki kemampuan tambahan seperti *buffering* dan *caching*. Contoh: Sektor tertentu yang dipakai untuk memenuhi permintaan mungkin saja berada dalam *buffers* atau *caches* yang diatur oleh *basic file system*. Jika terjadi hal seperti ini, maka informasi akan didapatkan secara otomatis tanpa perlu membaca lagi dari *disk*.
6. **Kendali M/K.** Tingkatan yang paling rendah ini yang memiliki cara untuk memerintah/memberitahu piranti keras yang diperlukan.

40.2. Implementasi Sistem Berkas

Untuk mengimplementasikan suatu sistem berkas biasanya digunakan beberapa struktur *on-disk* dan *in-memory*. Struktur ini bervariasi tergantung pada sistem operasi dan sistem berkas, tetapi beberapa prinsip dasar harus tetap diterapkan. Pada struktur *on-disk*, sistem berkas mengandung informasi tentang bagaimana mem-*boot* sistem operasi yang disimpan, jumlah blok, jumlah dan lokasi blok yang masih kosong, struktur direktori, dan berkas individu.

Struktur *on-disk*:

- ***Boot Control Block*.** Informasi yang digunakan untuk menjalankan mesin mulai dari partisi yang diinginkan untuk menjalankan mesin mulai dari partisi yang diinginkan. Dalam UPS disebut *boot block*. Dalam NTFS disebut *partition boot sector*.
- ***Partition Block Control*.** Spesifikasi atau detil-detil dari partisi (jumlah blok dalam partisi, ukuran blok, ukuran blok, dsb). Dalam UPS disebut *superblock*. Dalam NTFS disebut tabel *master file*.
- **Struktur direktori.** Mengatur berkas-berkas.
- **File Control Block (FCB).** Detil-detil berkas yang spesifik. Di UPS disebut *inode*. Di NTFS, informasi ini disimpan di dalam tabel *Master File*.

Struktur in-memory:
- **Tabel Partisi *in-memory*.** Informasi tentang partisi yang di-*mount*.
- **Struktur Direktori *in-memory*.** Menyimpan informasi direktori tentang direktori yang paling sering diakses.
- **Tabel *system-wide open-file*.**
  - menyimpan *open count* (informasi jumlah proses yang membuka berkas tsb)
  - menyimpan atribut berkas (pemilik, proteksi, waktu akses, dsb), dan lokasi *file blocks*.
  - Tabel ini digunakan bersama-sama oleh seluruh proses.
- **Tabel *per-process open-file*.**
  - menyimpan *pointer* ke entri yang benar dalam tabel *open- file*
  - menyimpan posisi pointer pada saat itu dalam berkas.
  - modus akses

Untuk membuat suatu berkas baru, program aplikasi memanggil *logical file system*. *Logical file system* mengetahui format dari struktur direktori. Untuk membuat berkas baru, *logical file system* akan mengalokasikan FCB, membaca direktori yang benar ke memori, memperbaharui dengan nama berkas dan FCB yang baru dan menulisnya kembali ke dalam *disk*.

Beberapa sistem operasi, termasuk Unix, memperlakukan berkas sebagai direktori. Sistem operasi Windows NT mengimplementasi beberapa *system calls* untuk berkas dan direktori. Windows NT memperlakukan direktori sebagai sebuah kesatuan yang berbeda dengan berkas. *Logical file system* dapat memanggil *file-organization module* untuk memetakan direktori M/K ke *disk-block numbers*, yang dikirimkan ke sistem berkas dasar dan sistem kendali M/K *File-organization module* juga mengalokasikan blok untuk penyimpanan data-data berkas.

Setelah berkas selesai dibuat, mula-mula harus dibuka terlebih dahulu. Perintah *open* mengirim nama berkas ke sistem berkas. Ketika sebuah berkas dibuka, struktur direktori mencari nama berkas yang diinginkan. Ketika berkas ditemukan, FCD disalin ke ke tabel *system-wide open-file* pada memori. Tabel ini juga mempunyai entri untuk jumlah proses yang membuka berkas tersebut.

Selanjutnya, entri dibuat di tabel *per-process open-file* dengan penunjuk ke entri di dalam tabel *system-wide open-file*. Seluruh operasi pada berkas akan diarahkan melalui penunjuk ini.

Ketika proses menutup file, entri tabel per-per proses dihapus, dan entri dari jumlah proses yang membuka berkas tersebut (open count) dalam system-wide dikurangi. Ketika semua pengguna yang mengakses file menutup berkas tersebut, informasi yang telah di-update disalinan kembali ke disk dan tabel system-wide open-file dihapus.

Realitanya system call open pertama-tama mencari pada tabel system-wide open apakah berkas yang ingin dibuka telah dibuka pengguna lain. Jika telah dibuka maka tabel entri open-file akan mengacu pada tabel system-wide open-file tersebut.

40.3. Partisi dan *Mounting*

Setiap partisi dapat merupakan raw atau cooked. Raw adalah partisi yang tidak memiliki sistem berkas dan cooked sebaliknya. Raw disk digunakan jika tidak ada sistem berkas yang tepat. Raw disk juga dapat menyimpan informasi yang dibutuhkan oleh sistem disk RAID dan database kecil yang menyimpan informasi konfigurasi RAID.

Informasi boot dapat disimpan di partisi yang berbeda. Semuanya mempunyai formatnya

masing-masing karena pada saat boot, sistem tidak punya sistem berkas dari perangkat keras dan tidak dapat memahami sistem berkas. Sebuah komputer dapat memiliki lebih dari satu Sistem Operasi. Boot-Loader dapat mengenali lebih dari satu sistem operasi dan sistem berkas. Ketika di-load, Boot-Loader dapat mem-boot salah satu sistem operasi yang tersedia di disk. Setiap disk dapat memiliki lebih dari satu partisi, yang masing-masing dapat memiliki sistem operasi dan sistem berkas yang berbeda.

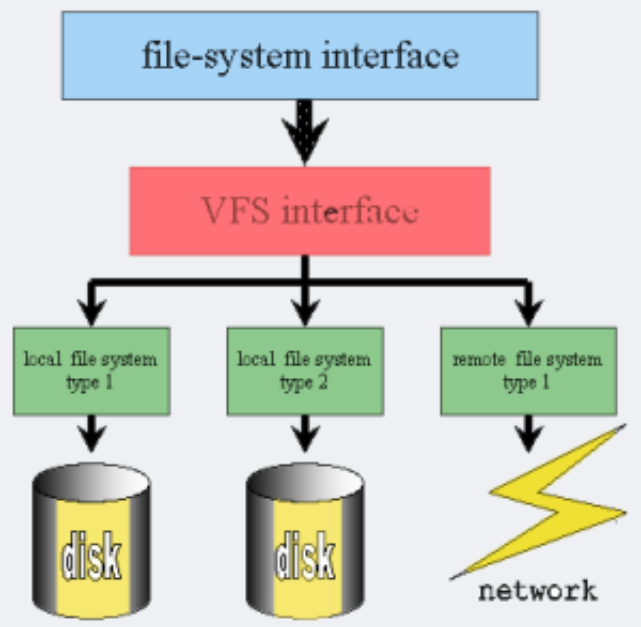
Root partition adalah partisi yang mengandung kernel sistem operasi dan sistem berkas yang lain, di-mount saat boot. Partisi yang lain di-mount secara otomatis atau manual (tergantung sistem operasi). Sistem operasi menyimpan informasi tentang dimana sistem berkas di-mount dan jenis dari sistem berkas di dalam struktur tabel mount di memori.

Pada UNIX, sistem berkas dapat di-mount di direktori mana pun. Ini diimplementasikan dengan mengatur flag di salinan in-memori dari jenis direktori itu. Flag itu mengindikasikan bahwa direktori adalah puncak mount.

40.4. Sistem Berkas Virtual

Suatu direktori biasanya menyimpan beberapa berkas dengan jenis-jenis yang berbeda. Sistem operasi harus dapat menyatukan berkas-berkas berbeda itu di dalam suatu struktur direktori. Untuk menyatukan berkas-berkas tersebut digunakan metoda implementasi beberapa jenis sistem berkas dengan menulis di direktori dan berkas routine untuk setiap jenis.

Gambar 40.4. ***Schematic View of Virtual File System***



Sistem operasi pada umumnya, termasuk UNIX, menggunakan teknik berorientasi objek untuk menyederhakan, mengorganisir dan mengelompokkannya sesuai dengan implementasinya. Penggunaan metoda ini memungkinkan berkas-berkas yang berbeda jenisnya diimplementasikan dalam struktur yang sama.

Implementasi spesifiknya menggunakan struktur data dan prosedur untuk mengisolasi fungsi dasar dari system call. Implementasi sistem berkas terdiri dari tiga lapisan utama:

1. **Interface sistem berkas.** Perintah open, read, write, close dan berkas descriptor.
2. **Virtual File System(VFS).** Suatu lapisan perangkat lunak dalam kernel yang menyediakan antar muka sistem berkas untuk program ruang-pengguna. VFS juga menyediakan suatu abstraksi dalam kernel yang mengijinkan implementasi sistem berkas yang berbeda untuk muncul.
3. **Sistem berkas lokal dan sistem berkas remote.** Untuk jaringan.

VFS ini memiliki 2 fungsi yang penting yaitu:
1. Memisahkan operasi berkas generic dari implementasinya dengan mendefinisikan VFS antar muka yang masih baru.
2. VFS didasarkan pada struktur file-representation yang dinamakan vnode, yang terdiri dari designator numerik untuk berkas unik network-wide.

40.5. Implementasi Direktori

Sebelum sebuah berkas dapat dibaca, berkas tersebut harus dibuka terlebih dahulu. Saat berkas tersebut dibuka, sistem operasi menggunakan *path name* yang dimasukkan oleh pengguna untuk mengalokasikan direktori entri yang menyediakan informasi yang dibutuhkan untuk menemukan *block disk* tempat berkas itu berada. Tergantung dari sistem tersebut, informasi ini dapat berupa alamat *disk* dari berkas yang bersangkutan (*contiguous allocation*), nomor dari blok yang pertama (kedua skema *linked list*), atau nomor dari *inode*. Dalam semua kasus, fungsi utama dari direktori entri adalah untuk memetakan nama ASCII dari berkas yang bersangkutan kepada informasi yang dibutuhkan untuk mengalokasikan data.

Masalah berikutnya yang kemudian muncul adalah dimana atribut yang dimaksud akan disimpan. Kemungkinan paling nyata adalah menyimpan secara langsung di dalam direktori entri, dimana kebanyakan sistem menggunakannya. Untuk sistem yang menggunakan *inodes*, kemungkinan lain adalah menyimpan atribut ke dalam *inode*, selain dari direktori entri. Cara yang terakhir ini mempunyai keuntungan lebih dibandingkan menyimpan dalam direktori entri.

Cara pengalokasian direktori dan pengaturan direktori dapat meningkatkan efisiensi, performa dan kehandalan. Ada beberapa macam algoritma yang dapat digunakan.

40.6. Algoritma *Linear List*

Metode paling sederhana. Menggunakan nama berkas dengan penunjuk ke data blok.

Proses:
- Mencari (tidak ada nama berkas yang sama).
- Menambah berkas baru pada akhir direktori.
- Menghapus (mencari berkas dalam direktori dan melepaskan tempat yang dialokasikan).

Penggunaan suatu berkas:

Memberi tanda atau menambahkan pada daftar direktori bebas.

Kelemahan:

Pencarian secara *linier (linier search)* untuk mencari sebuah berkas, sehingga implementasi sangat lambat saat mengakses dan mengeksekusi berkas.

Solusi:

Linked list dan *Tree Structure Data Software Cache*

40.7. Algoritma *Hash Table*

Linear List menyimpan direktori entri, tetapi sruktur data *hash* juga digunakan.

Proses:

Hash table mengambil nilai yang dihitung dari nama berkas dan mengembalikan sebuah penunjuk ke nama berkas yang ada di *linier list*.

Kelemahan:
- Ukuran tetap:
- Adanya ketergantungan fungsi *hash* dengan ukuran *hash table*

Alternatif:

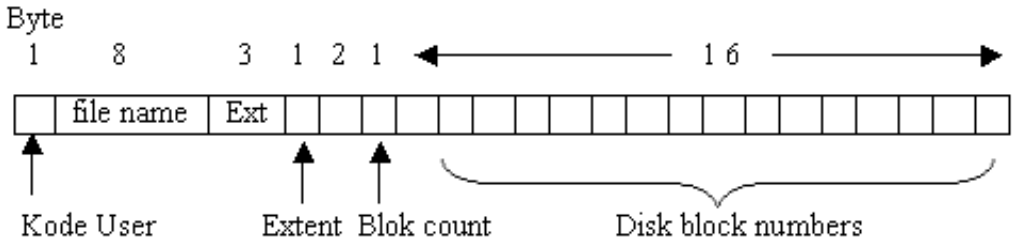
Chained-overflow hash table yaitu setiap *hash table* mempunyai *linked list* dari nilai individual dan *crash* dapat diatasi dengan menambah tempat pada *linked list* tersebut. Namun penambahan ini dapat memperlambat.

40.8. Direktori pada CP/M

Direktori pada CP/M merupakan direktori entri yang mencakup nomor *block disk* untuk setiap berkas. Contoh direktori ini (Golden dan Pechura, 1986), berupa satu direktori saja. Jadi, Semua sistem berkas harus melihat nama berkas dan mencari dalam direktori satu-satunya ini. Direktori ini terdiri dari 3 bagian yaitu:

- ***User Code*.** Merupakan bagian yang menetapkan *track* dari *user* mana yang mempunyai berkas yang bersangkutan, saat melakukan pencarian, hanya entri tersebut yang menuju kepada *logged-in user* yang bersangkutan. Dua bagian berikutnya terdiri dari nama berkas dan ekstensi dari berkas.
- **Nama Berkas.**
- **Ekstensi.**
- ***Extent*.** Bagian ini diperlukan oleh berkas karena berkas yang berukuran lebih dari 16 blok menempati direktori entri yang banyak. Bagian ini digunakan untuk memberitahukan entri mana yang datang pertama, kedua, dan seterusnya.
- ***Block Count*.** Bagian ini memberitahukan seberapa banyak dari ke-enambelas *block disk* potensial, sedang digunakan. Enambelas bagian akhir berisi nomor *block disk* yang bersangkutan. Bagian blok yang terakhir dapat saja penuh, jadi sistem tidak dapat menentukan kapasitas pasti dari berkas sampai ke *byte* yang terakhir. Saat CP/M menemukan entri, CP/M juga memakai nomor *block disk*, saat berkas disimpan dalam direktori entri, dan juga semua atributnya. Jika berkas menggunakan *block disk* lebih dari satu entri, berkas dialokasikan dalam direktori yang ditambahkan.
- **Alamat Blok Berkas.**

Gambar 40.5. Direktori CPM



Saat CP/M menemukan entri, CP/M juga memakai nomor block disk, saat berkas disimpan dalam direktori entri, dan juga semua atributnya. Jika berkas menggunakan block disk lebih dari satu entri, berkas dialokasikan dalam direktori yang ditambahkan.

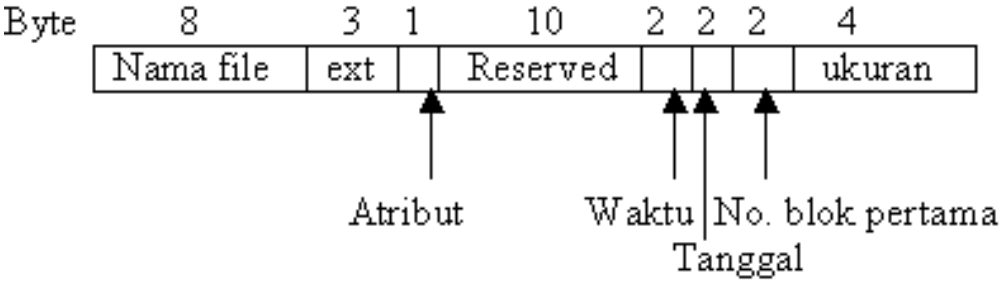
40.9. Direktori pada MS-DOS

Merupakan sistem dengan *tree hierarchy directory*. Mempunyai panjang 32 *bytes*, yang mencakup nama berkas, atribut, dan nomor dari *block disk* yang pertama. Nomor dari *block disk* yang pertama digunakan sebagai indeks dari tabel MS-DOS direktori entri. Dengan sistem rantai, semua blok dapat ditemukan.

1. Nama berkas.
2. Ekstensi (jenis berkas).
3. Atribut.
4. Cadangan untuk penggunaan di masa datang.

5. Waktu, mencatat pertama kali berkas diciptakan atau terakhir kali dimodifikasi.
6. Tanggal, mencatat pertama kali diciptakan atau terakhir kali dimodifikasi.
7. Nomor blok pertama, menujuk ke alamat pertama dari blok disk berkas dan FAT (File Allocation Table) menunjukan blok-blok berkas berikutnya dengan sistem rantai.
8. Ukuran berkas.

Gambar 40.6. Direktori MS-DOS



Dalam MS-DOS, direktori dapat berisi direktori lain, tergantung dari hirarki sistem berkas. Dalam MS-DOS, program aplikasi yang berbeda dapat dimulai oleh setiap program dengan membuat direktori dalam direktori *root*, dan menempatkan semua berkas yang bersangkutan di dalam sana. Jadi antar aplikasi yang berbeda tidak dapat terjadi konflik.

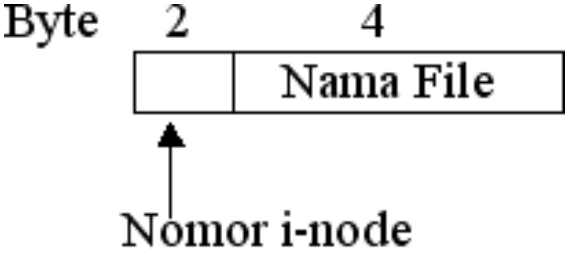
Apabila ingin menghapus berkas, maka entri direktori dari berkas tersebut akan diberi flag. Dalam MS-DOS, direktori dapat berisi direktori lain, tergantung dari hirarki sistem berkas. Dalam MS-DOS, program aplikasi yang berbeda dapat dimulai oleh setiap program dengan membuat direktori dalam direktori root, dan menempatkan semua berkas yang bersangkutan di dalam sana. Jadi antar aplikasi yang berbeda tidak dapat terjadi konflik.

40.10. Direktori pada Unix

Struktur direktori yang digunakan dalam UNIX adalah struktur direktori tradisional. Seperti yang terdapat dalam gambar direktori entri dalam UNIX, setiap entri berisi nama berkas dan nomor inode yang bersangkutan. Semua informasi dari jenis, kapasitas, waktu dan kepemilikan, serta block disk yang berisi inode. Sistem UNIX terkadang mempunyai penampakan yang berbeda,tetapi pada beberapa kasus, direktori entri biasanya hanya string ASCII dan nomor inode. Dari nomor inode ini kita memperoleh inode yang merupakan suatu struktur data yang menyimpan informasi berkas. Penghapusan berkas dilakukan dengan cara melepas inode. I-node berisi informasi tentang:

- jenis
- ukuran
- waktu
- pemilik
- blok disk

Gambar 40.7. Direktori Unix



Saat berkas dibuka, sistem berkas harus mengambil nama berkas dan mengalokasikan block disk

yang bersangkutan, sebagai contoh, nama path /usr/ast/mbox dicari, dan kita menggunakan UNIX sebagai contoh, tetapi algoritma yang digunakan secara dasar sama dengan semua hirarki sistem direktori sistem.

Pertama, sistem berkas mengalokasikan direktori root. Dalam UNIX inode yang bersangkutan ditempatkan dalam tempat yang sudah tertentu dalam disk. Kemudian, UNIX melihat komponen pertama dari path, usr dalam direktori root menemukan nomor inode dari direktori /usr. Mengalokasikan sebuah nomor inode adalah secara straight-forward, sejak setiap inode mempunyai lokasi yang tetap dalam disk. Dari inode ini, sistem mengalokasikan direktori untuk /usr dan melihat komponen berikutnya, dst. Saat dia menemukan entri untuk ast, dia sudah mempunyai inode untuk direktori /ust/ast. Dari inode ini, dia dapat menemukan direktorinya dan melihat mbox. Inode untuk berkas ini kemudian dibaca ke dalam memori dan disimpan disana sampai berkas tersebut ditutup.

Nama path dilihat dengan cara yang relatif sama dengan yang absolut. Dimulai dari direktori yang bekerja sebagai pengganti root directory. Setiap direktori mempunyai entri untuk. dan ".." yang dimasukkan ke dalam saat direktori dibuat. Entri "." mempunyai nomor inode yang menunjuk ke direktori di atasnya/orangtua (parent), "." kemudian melihat ../dick/prog.c hanya melihat tanda ".." dalam direktori yang bekerja, dengan menemukan nomor inode dalam direktori di atasnya/parent dan mencari direktori disk. Tidak ada mekanisme spesial yang dibutukan untuk mengatasi masalah nama ini. Sejauh masih di dalam sistem direktori, mereka hanya merupakan ASCII string yang biasa.

40.11. Rangkuman

Sebagai implementasi direktori yang merupakan implementasi dari Implementasi Sistem Berkas, implementasi direktori memiliki algoritma seperti *Linear List dan Hashtable*. Direktori pada MS/Dos merupakan sistem dengan direktori hirarki tree. Direktori pada Unix merupakan struktur direktori tradisional.

Sebagai implementasi direktori yang merupakan implementasi dari Implementasi Sistem Berkas, implementasi direktori memiliki algoritma seperti *Linear List dan Hashtable*. Direktori pada MS/Dos merupakan sistem dengan direktori hirarki tree. Direktori pada Unix merupakan struktur direktori tradisional.

Implementasi sistem berkas dapat bervariasi, pada umumnya digunakan struktur sistem berkas Layered file system yaitu: Application program -> Logical File System -> File Organization Module -> Basic File System -> M/K control -> Devices

Sistem berkas virtual adalah suatu lapisan perangkat lunak dalam kernel yang menyediakan antar muka sistem berkas untuk program ruang pengguna. VFS juga menyediakan suatu abstraksi dalam kernel yang mengijinkan implementasi sistem berkas yang berbeda untuk muncul. VFS ini memiliki dua fungsi yang penting yaitu:

1. Memisahkan operasi berkas generic dari implementasinya dengan mendefinisikan VFS antar muka yang masih baru.
2. VFS didasarkan pada struktur file-representation yang dinamakan vnode, yang terdiri dari designator numerik untuk berkas unik network-wide.

Sebagai implementasi direktori yang merupakan implementasi dari Implementasi Sistem Berkas, implementasi direktori memiliki algoritma seperti Linear List dan Hashtable. Direktori pada CP/M merupakan sistem dengan direktori tunggal. Direktori pada MS/Dos merupakan sistem dengan direktori hirarki pohon. Direktori pada UNIX merupakan struktur direktori tradisional.

Rujukan

[Hariyanto1997] Bambang Hariyanto. 1997. *Sistem Operasi*. Buku Teks Ilmu Komputer. Edisi Kedua. Informatika. Bandung.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

[WEBBraam1998] Peter J Braam. 1998. *Linux Virtual File System – http://www.coda.cs.cmu.edu/ doc/ talks/ linuxvfs/* . Diakses 25 Juli 2006.

[WEBCarter2004] John Carter. 2004. *CS 5460 Operating Systems – Lecture 19: File System Operations and Optimizations – http://www.cs.utah.edu/ classes/ cs5460/ lectures/ lecture19.pdf* . Diakses 29 Mei 2006.

[WEBGooch1999] Richard Gooch. 1999. *Overview of the Virtual File System – http://www.atnf.csiro.au/ people/ rgooch/ linux/ docs/ vfs.txt* . Diakses 29 Mei 2006.

[WEBIBM2003] IBM Corporation. 2003. *System Management Concepts: Operating System and Devices – http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690 / IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm* . Diakses 29 Mei 2006.

[WEBJonesSmith2000] David Jones dan Stephen Smith. 2000. *85349 – Operating Systems – Study Guide – http://www.infocom.cqu.edu.au/ Courses/ aut2001/ 85349/ Resources/ Study_Guide/ 85349.pdf* . Diakses 20 Juli 2006.

[WEBKozierok2005] Charles M Kozierok. 2005. *Reference Guide – Hard Disk Drives http://www.storagereview.com/ guide/* . Diakses 9 Agustus 2006.

Bab 41. FHS

41.1. Pendahuluan

Filesystem Hierarchy Standard (FHS) merupakan bakuan (*standard*) yang digunakan oleh perangkat lunak dan pengguna untuk mengetahui lokasi dari berkas atau direktori yang berada pada komputer. Hal ini dilakukan dengan cara menetapkan prinsip-prinsip dasar pada setiap daerah pada sistem berkas, menetapkan berkas dan direktori minimum yang dibutuhkan, mengatur banyaknya pengecualian dan mengatur kasus yang sebelumnya pernah mengalami konflik secara spesifik.

Proses pengembangan sebuah standar untuk filesystem hierarchy dimulai pada Agustus 1993 dengan usaha untuk merestrukturisasi struktur berkas dan direktori Linux. FSSTND,sebuah FHS spesifik untuk OS Linux dirilis pada 9 Oktober 1994.Di awal 1995, tujuan untuk mengembangkan sebuah versi FSSTND yang lebih komprehensif ditujukan tidak hanya kepada Linux, tetapi juga sistem lain yang UNIX-like. Jadilah saat ini namanya menjadi FHS karena scope-nya sudah meluas.

FHS melakukan standardisasi dengan cara: menetapkan prinsip-prinsip untuk setiap area dari sistem berkas, menetapkan berkas dan direktori minimum yang dibutuhkan dari sistem berkas, memperhitungkan exception untuk prinsip tersebut, dan memperhitungkan kasus spesifik di mana terjadi historical conflict.

Pengguna dokumen FHS adalah: penyedia perangkat lunak untuk menciptakan aplikasi yang sesuai FHS, pembuat OS untuk menyediakan sistem yang sesuai FHS, pengguna untuk mengerti dan memelihara keadaan FHS dari sebuah sistem.

Anda masih bingung tentang apa itu FHS? Jika anda adalah pengguna Microsoft Windows, maka anda akan menemukan beberapa direktori default, seperti direktori Program Files, My Documents, System, dan lain sebagainya. Direktori-direktori tersebut sudah ditentukan fungsinya masing-masing. Penerapan FHS adalah tidak jauh berbeda dengan penerapan di atas.

Komponen dari nama berkas yang bervariasi diapit oleh karakter "<" dan ">". Komponen optional dari filename diapitdengan karakter "[" dan "]" dan dapat dikombinasikan dengan "<" dan ">". Contoh: jika sebuah berkas diperbolehkan untuk dituliskan dengan atau tanpa extension, maka dapat direpresentasikan dengan <filename> [.<extension>].

41.2. Sistem Berkas

Standar ini mengasumsikan bahwa OS yang memakai mode FHS mendukung fitur sekuriti yang sama dengan yang digunakan di sebagian besar sistem berkas UNIX. Kita dapat menggolongkan berkas menurut karakteristiknya:

Berkas shareable adalah berkas-berkas yang dapat diletakkan pada satu host dan digunakan oleh yang host yang lain. Jadi,file tersebut bisa di-share antar host, bukan hanya terbatas antar pengguna. Contohnya adalah direktori "/usr". Bukan suatu halyang mustahil jika dari sepuluh komputer yang terhubung, hanya satu komputer yang memiliki direktori "/usr" karena direktoritersebut dapat diakses oleh semua komputer. Berkas unshareable adalah berkas-berkas yang tidak dapat di-share.

Berkas statik tidak dapat diubah tanpa intervensi administrator sistem. Contohnya adalah binary file, libraries,berkas dokumentasi. Berkas variable adalah berkas yang tidak statik. Berkas statik dan variable sebaiknya diletakkan di direktori yang berbeda. Hal ini bertujuan untuk memudahkan berkas dengan karakteristik yang berbeda pada filesystem yang berbeda.

Berkas shareable dan statik adalah pada "/usr" dan "/opt".Berkas yang statik dan unshareable adalah pada "/etc" dan "/boot".Berkas yang shareable dan variable adalah pada "/var/mail" dan "/var/spool/news". Berkas yang unshareable dan variable "/var/run" dan "/var/lock".

41.3. Sistem Berkas *Root*

Tujuan dan Prasyarat

Isi dari sistem berkas *root* harus memadai untuk melakukan operasi *boot*, *restore*, *recover*, dan atau perbaikan pada sistem. Untuk melakukan operasi *boot* pada sistem, perlu dilakukan hal-hal untuk *mounting* sistem berkas lain. Hal ini meliputi konfigurasi data, informasi *boot loader* dan keperluan-keperluan lain yang mengatur *start-up* data. Untuk melakukan *recovery* dan atau perbaikan dari sistem, hal-hal yang dibutuhkan untuk mendiagnosa dan memulihkan sistem yang rusak harus diletakkan dalam sistem berkas *root*.

Untuk *restore* suatu sistem, hal-hal yang dibutuhkan untuk *back-up* sistem, seperti *floppy disk*, *tape*, dsb, harus berada dalam sistem berkas *root*. Aplikasi pada komputer tidak diperbolehkan untuk membuat berkas atau subdirektori di dalam direktori *root*, karena untuk meningkatkan *performance* dan keamanan, partisi *root* sebaiknya dibuat seminimum mungkin. Selain itu, lokasi-lokasi lain dalam FHS menyediakan fleksibilitas yang lebih dari cukup untuk *package* mana pun.

Terdapat beberapa direktori yang merupakan persyaratan dari sistem berkas *root*. Setiap direktori akan dibahas dalam sub-bagian di bawah. /usr dan /var akan dibahas lebih mendetail karena direktori tersebut sangat kompleks.

Isi dari filesystem root haruslah mencukupi untuk boot, restore, recover, dan atau mereparasi sistem. Untuk boot sistem, beberapa hal harus terdapat di partisi root untuk mountfilesystem lain. Ini termasuk utilisasi, konfigurasi, informasiboot loader, dan start-up data esensial yang lain. "/usr","/opt", "/var" di-design sehingga dapat diletakkan di partisi lain dari sistem. Untuk memungkinkan recovery dan atau reparasi dari sistem, utilitas yang dibutuhkan pemelihara untuk mendiagnosa dan merekonstruksi sistem yang rusak haruslah terdapat di root filesytem. Untuk restore sistem, utilitas yang dibutuhkan untuk restore dari sistem back-up haruslah terdapat di sistem berkas root .

Untuk beberapa hal, adalah beralasan untuk menginginkan agar root berukuran kecil:
1. Terkadang di-mount dari media yang sangat kecil. Hal ini membuat kita tidak dapat memperbesar root lagi karena keterbatasan kapasitas media.
2. Sistim berkas root mengandung banyak berkas konfigurasi yang system-specific.
3. Menjaga kompatibilitas.
4. Mencegah disk error yang membuat data corrupt. Disk error di root filesystem membuat masalah yang jauh lebih besar jika dibandingkan dengan error di partisi lainnya.

Oleh karena itu, membuat sub-direktori baru pada root adalah sangat tidak dianjurkan.

Tabel 41.1. Direktori/link yang wajib dalam "/."

Direktori	Keterangan
bin	Instruksi dasar biner
boot	Berkas statik untuk me-*load boot*
dev	Berkas peranti
etc	Konfigurasi sistem *host-specific*
lib	Pustaka dasar bersama dan modul *kernel*
media	*Mount point* untuk media-media *removable*
mnt	*Mount point* untuk *mounting* sistem berkas secara temporer
opt	Penambahan aplikasi *package* perangkat lunak
sbin	Sistem biner dasar
srv	Data untuk servis yang disediakan oleh sistem
tmp	Berkas temporer
usr	Hirarki sekunder
var	Data variabel

Tabel 41.2. Direktori/link yang optional dalam ''/.''

Direktori	Keterangan
home	Direktori *home* pengguna
lib<qual>	Format alternatif dari pustaka dasar bersama
root	Direktori *home* untuk *root* pengguna

"/bin": Perintah Biner Dasar Umum

"/bin" berisi perintah-perintah yang dapat digunakan oleh administrator sistem dan pengguna, namun dibutuhkan apabila tidak ada sistem berkas lain yang di-*mount*. "/bin" juga berisi perintah-perintah yang digunakan secara tidak langsung oleh *script*.

"/boot": Berkas statik untuk me-*load boot*

Dalam direktori ini, terdapat segala sesuatu yang dibutuhkan untuk melakukan *boot* proses. "/boot" menyimpan data yang digunakan sebelum *kernel* mulai menjalankan program mode pengguna. Hal ini dapat meliputi sektor *master boot* dan sektor berkas map.

"/dev": Berkas peranti

Direktori "/dev" adalah lokasi dari berkas-berkas peranti. Direktori ini harus memiliki perintah bernama "MAKEDEV" yang dapat digunakan untuk menciptakan peranti secara manual. Jika dibutuhkan, "MAKEDEV" harus memiliki segala ketentuan untuk menciptakan peranti-peranti yang ditemukan dalam sistem, bukan hanya implementasi partikular yang di-*install*.

"/etc": Konfigurasi sistem *host-specific*

Direktori "/etc" mernyimpan berkas-berkas konfigurasi. Yang dimaksud berkas konfigurasi adalah berkas lokal yang digunakan untuk mengatur operasi dari sebuah program. Berkas ini harus statik dan bukan merupakan biner *executable*.

"/home": Direktori home pengguna

"/home" adalah konsep standar sistem berkas yang *site-specific*, artinya *setup* dalam *host* yang satu dan yang lainnya akan berbeda-beda. Maka, program sebaiknya tidak diletakkan dalam direktori ini.

"/lib": Pustaka dasar bersama dan modul *kernel*

Direktori "/lib" meliputi gambar-gambar pustaka bersama yang dibutuhkan untuk *boot* sistem tersebut dan menjalankan perintah dalam sistem berkas *root*, contohnya berkas biner di "/bin" dan "/sbin".

"/lib<qual>": Alternatif dari pustaka dasar bersama

Pada sistem yang mendukung lebih dari satu format biner, mungkin terdapat satu atau lebih perbedaan dari direktori "/lib". Jika direktori ini terdapat lebih dari satu, maka persyaratan dari isi tiap direktori adalah sama dengan direktori "/lib" normalnya, namun "/lib<qual>/cpp" tidak dibutuhkan.

"/media": Mount point media *removable*

Direktori ini berisi subdirektori yang digunakan sebagai *mount point* untuk media-media *removable* seperti *floppy disk*, dll. *cdrom*, dll.

"/mnt": *Mount point* temporer

Direktori ini disediakan agar administrator sistem dapat *mount* suatu sistem berkas yang dibutuhkan secara temporer. Isi dari direktori ini adalah *issue* lokal, dan tidak mempengaruhi sifat-sifat dari program yang sedang dijalankan.

/opt: Aplikasi tambahan untuk paket peringkat lunak

"/opt" disediakan untuk aplikasi tambahan paket peringkat lunak. Paket yang di-*install* di "/opt" harus menemukan berkas statiknya di direktori "/opt/<package>" atau "/opt/<provider>", dengan <package> adalah nama yang mendeskripsikan paket perangkat lunak tersebut, dan <provider> adalah nama dari *provider* yang bersangkutan.

"/root": Direktori *home* untuk *root* pengguna

Direktori *home root* dapat ditentukan oleh pengembang atau pilihan-pilihan lokal, namun direktori ini adalah lokasi *default* yang direkomendasikan.

"/sbin": Sistem Biner

Kebutuhan yang digunakan oleh administrator sistem disimpan di "/sbin", "/usr/sbin", dan "/usr/local/sbin". "/sbin" berisi biner dasar untuk *boot* sistem, mengembalikan sistem, memperbaiki sistem sebagai tambahan untuk biner-biner di "/bin". Program yang dijalankan setelah /usr diketahui harus di-*mount*, diletakkan dalam "/usr/bin". Sedangkan, program-program milik administrator sistem yang di-*install* secara lokal sebaiknya diletakkan dalam "/usr/local/sbin".

"/srv": Data untuk servis yang disediakan oleh sistem

"/srv" berisi data-data *site-specific* yang disediakan oleh sistem.

"/tmp": Berkas-berkas temporer

Direktori "/tmp" harus tersedia untuk program-program yang membutuhkan berkas temporer.

41.4. Hirarki "/usr"

Tujuan

"/usr" adalah bagian utama yang kedua dari sistem berkas. "/usr" bersifat *shareable* dan *read-only*. Hal ini berarti "/usr" bersifat *shareable* diantara bermacam-macam *host* FHS-*compliant*, dan tidak boleh di-*write*. *Package* perangkat lunak yang besar tidak boleh membuat subdirektori langsung di bawah hirarki "/usr" ini.

Persyaratan

Tabel 41.3. Direktori/link yang dibutuhkan dalam "/usr".

Direktori	Keterangan
bin	Sebagian besar perintah pengguna
include	Berkas *header* yang termasuk dalam program-program C
lib	Pustaka
local	Hirarki lokal (kosong sesudah instalasi main)
sbin	Sistem biner non-vital
share	Data arsitektur yang independen

Pilihan spesifik

Tabel 41.4. Direktori/link yang merupakan pilihan dalam ''/usr''.

Direktori	Keterangan
X11R6	Sistem X Window, Versi 11 *Release* 6
games	*Games*dan *educational* biner
lib<qual>	Format pustaka alternatif
src	Kode *source*

Link-link simbolik seperti di bawah ini dapat terjadi, apabila terdapat kebutuhan untuk menjaga keharmonisan dengan sistem yang lama, sampai semua implementasi dapat diasumsikan untuk menggunakan hirarki "/var":
- /usr/spool --> /var/spool
- /usr/temp --> /var/tmp
- /usr/spool/locks --> /var/lock

Saat sistem tidak lagi membutuhkan *link-link* di atas, *link* tersebut dapat dihapus.

"/usr/X11R6": Sistem X Window, Versi 11 Release 6

Hirarki ini disediakan untuk Sistem X Window, Versi 11 *Release* 6 dan berkas-berkas yang berhubungan. Untuk menyederhanakan persoalan dan membuat XFree86 lebih kompatibel dengan Sistem X Window, link simbolik di bawah ini harus ada jika terdapat direktori "/usr/X11R6":
- /usr/bin/X11 --> /usr/X11R6/bin
- /usr/lib/X11 --> /usr/X11R6/lib/X11
- /usr/include/X11 --> /usr/X11R6/include/X11

Link-link di atas dikhususkan untuk kebutuhan dari pengguna saja, dan perangkat lunak tidak boleh di-*install* atau diatur melalui *link-link* tersebut.

"/usr/bin": Sebagian perintah pengguna

Direktori ini adalah direktori primer untuk perintah- perintah *executable* dalam sistem.

"/usr/include": Direktori untuk *include-files* standar

Direktori ini berisi penggunaan umum berkas *include* oleh sistem, yang digunakan untuk bahasa pemrograman C.

"/usr/lib": Pustaka untuk pemrograman dan *package*

"/usr/lib" meliputi berkas obyek, pustaka dan biner internal yang tidak dibuat untuk dieksekusi secara langsung melalui pengguna atau *shell script*. Aplikasi-aplikasi dapat menggunakan subdirektori tunggal di bawah "/usr/lib". Jika aplikasi tersebut menggunakan subdirektori, semua data yang arsitektur-*dependent* yang digunakan oleh aplikasi tersebut, harus diletakkan dalam subdirektori tersebut juga.

Untuk alasan historis, "/usr/lib/sendmail" harus merupakan *link* simbolik ke "/usr/sbin/sendmail". Demikian juga, jika "/lib/X11" ada, maka "/usr/lib/X11" harus merupakan *link* simbolik ke "/lib/X11", atau ke mana pun yang dituju oleh *link* simbolik "/lib/X11".

"/usr/lib<qual>": Format pustaka alternatif

"/usr/lib<qual>" melakukan peranan yang sama seperti "/usr/lib" untuk format biner alternatif, namun tidak lagi membutuhkan *link* simbolik seperti "/usr/lib<qual>/sendmail" dan

"/usr/lib<qual>/X11".

"/usr/local/share"

Direktori ini sama dengan "/usr/share". Satu-satunya pembatas tambahan adalah bahwa direktori '/usr/local/share/man' dan '/usr/local/man' harus *synonomous* (biasanya ini berarti salah satunya harus merupakan *link* simbolik).

"/usr/sbin": Sistem biner standar yang non-vital

Direktori ini berisi biner non-vital mana pun yang digunakan secara eksklusif oleh administrator sistem. Program administrator sistem yang diperlukan untuk perbaikan sistem, *mounting* "/usr" atau kegunaan penting lainnya harus diletakkan di "/sbin".

"/usr/share": Data arsitektur independen

Hirarki "/usr/share" hanya untuk data-data arsitektur independen yang *read-only*. Hirarki ini ditujukan untuk dapat di-*share* diantara semua arsitektur *platform* dari sistem operasi; sebagai contoh: sebuah *site* dengan *platform* i386, Alpha dan PPC dapat me-*maintain* sebuah direktori /usr/share yang di-*mount* secara sentral.

Program atau paket mana pun yang berisi dan memerlukan data yang tidak perlu dimodifikasi harus menyimpan data tersebut di "/usr/share" (atau "/usr/local/share", apabila di- *install* secara lokal). Sangat direkomendasikan bahwa sebuah subdirektori digunakan dalam /usr/share untuk tujuan ini.

"/usr/src": Kode *source*

Dalam direktori ini, dapat diletakkan kode-kode *source*, yang digunakan untuk tujuan referensi.

41.5. Hirarki "/var"

Tujuan

"/var" berisi berkas data variable. Termasuk berkas dan direktori spool, data administratif dan logging, serta berkas transient dan temporer. Beberapa bagian dari "/var" tidak dapat di-share diantara sistem yang berbeda, antara lain: "/var/log", "/var/lock" dan "/var/run". Sedangkan, "/var/mail", "/var/cache/man", "/var/cache/fonts" dan "/var/spool/news" dapat di-share antar sistem yang berbeda.

"/var" dispesifikan di ini untuk memungkinkan operasi mount "/usr" read- only. Segala sesuatu yang melewati "/usr", yang telah ditulis selama operasi sistem, harus berada di "/var". Jika "/var" tidak dapat dibuatkan partisi yang terpisah, biasanya "/var" dipindahkan ke luar dari partisi root dan dimasukkan ke dalam partisi "/usr". (Hal ini kadang dilakukan untuk mengurangi ukuran partisi root atau saat ruangan di partisi root mulai berkurang.)

Walaupun demikian, "/var" tidak boleh di-link ke "/usr", karena hal ini membuat pemisahan antara "/usr" dan "/var" semakin sulit dan biasa menciptakan konflik dalam penamaan. Sebaliknya, buat link "/var" ke "/usr/var".

Aplikasi harus secara menyeluruh tidak menambahnkan direktori di level teratas /var. Direktori seperti itu hanya bisa ditambahkan jika mereka memiliki implikasi system-wide, dan berkonsultasi dengan FHS mailing list.

Tabel 41.5. Direktori/link yang dibutuhkan dalam ''/var''

Direktori	Keterangan
cache	Data *cache* aplikasi
lib	Informasi status variabel

Direktori	Keterangan
local	Data variabel untuk "/usr/local"
lock	*Lock*berkas
log	Berkas dan direktori *log*
opt	Data variabel untuk "/opt"
run	Relevansi data untuk menjalankan proses
spool	Aplikasi data *spool*
tmp	Berkas temporer lintas *reboot*

Pilihan Spesifik

Direktori atau link simbol yang menuju ke direktori di bawah ini, dibutuhkan dalam "/var", jika subsistem yang berhubungan dengan direktori tersebut di-*install*:

Tabel 41.6. Direktori/link yang dibutuhkan di dalam ''/var''

Direktori	Keterangan
account	*Log accounting* proses
crash	*System crash dumps*
games	Data variabel *game*
mail	Berkas *mailbox* pengguna
yp	Network Information Service (NIS) berkas *database*

"/var/account": *Log accounting*proses

Direktori ini memegang *log accounting* dari proses yang sedang aktif dan gabungan dari penggunaan data.

"/var/cache": Aplikasi data *cache*

"/var/cache" ditujukan untuk data *cache* dari aplikasi. Data tersebut diciptakan secara lokal sebagai *time-consuming* M/K atau kalkulasi. Aplikasi ini harus dapat menciptakan atau mengembalikan data. Tidak seperti "/var/spool", berkas *cache* dapat dihapus tanpa kehilangan data.

Berkas yang ditempatkan di bawah "/var/cache" dapat *expired* oleh karena suatu sifat spesifik dalam aplikasi, oleh administrator sistem, atau keduanya, maka aplikasi ini harus dapat *recover* dari penghapusan berkas secara manual.

Beberapa contoh dari sistem Ubuntu ialah: ``/var/cache/apt", ``/var/cache/cups", ``/var/cache/debconf", ``/var/cache/debtags", ``/var/cache/dictionaties-common", ``/var/cache/fonts", ``/var/cache/locate", ``/var/cache/man", ``/var/cache/pppconfig".

"/var/crash": *System crash dumps*

Direktori ini mengatur *system crash dumps*. Saat ini, *system crash dumps* belum dapat di-*support* oleh Linux, namun dapat di-*support* oleh sistem lain yang dapat memenuhi FHS.

"/var/games": Data variabel *game*

Data variabel mana pun yang berhubungan dengan *games* di "/usr" harus diletakkan di direktori ini. "/var/games" harus meliputi data variabel yang ditemukan di /usr; data statik, seperti *help text*, deskripsi level, dll, harus ditempatkan di lain direktori, seperti "/usr/share/games".

"/var/lib": Informasi status variabel

Hirarki ini berisi informasi status suatu aplikasi dari sistem. Yang dimaksud dengan informasi status adalah data yang dimodifikasi program saat program sedang berjalan. Pengguna tidak diperbolehkan untuk memodifikasi berkas di "/var/lib" untuk mengkonfigurasi operasi *package*. Informasi status ini digunakan untuk memantau kondisi dari aplikasi, dan harus tetap valid setelah *reboot*, tidak berupa *output logging* atau pun data *spool*.

Sebuah aplikasi harus menggunakan subdirektory "/var/lib" untuk data-datanya. Terdapat satu subdirektori yang dibutuhkan lagi, yaitu "/var/lib/misc", yang digunakan untuk berkas-berkas status yang tidak membutuhkan subdirektori.

Beberapa contoh dari sistem Ubuntu ialah: ``/var/lib/acpi-support", ``/var/lib/alsa", ``/var/lib/apt", ``/var/lib/aptitude", ``/var/lib/aspell", ``/var/lib/belocs", ``/var/lib/debtags", ``/var/lib/defome", ``/var/lib/dhcp3", ``/var/lib/dictionaties-common", ``/var/lib/discover", ``/var/lib/doc-base", ``/var/lib/dpkg", ``/var/lib/fontconfig", ``/var/lib/gcj4.1", ``/var/lib/gconf", ``/var/lib/gstreamer", ``/var/lib/guidance", ``/var/lib/initramfs-tools", ``/var/lib/kdm", dst.

"/var/lock": *Lock* berkas

Lock berkas harus disimpan dalam struktur direktori /var/lock. *Lock* berkas untuk peranti dan sumber lain yang di-*share* oleh banyak aplikasi, seperti *lock* berkas pada serial peranti yang ditemukan dalam "/usr/spool/locks" atau "/usr/spool/uucp", sekarang disimpan di dalam "/var/lock".

Format yang digunakan untuk isi dari *lock* berkas ini harus berupa format *lock* berkas HDB UUCP. Format HDB ini adalah untuk menyimpan pengidentifikasi proses (Process Identifier - PID) sebagai 10 byte angka desimal ASCII, ditutup dengan baris baru. Sebagai contoh, apabila proses 1230 memegang *lock* berkas, maka HDO formatnya akan berisi 11 karakter: spasi, spasi, spasi, spasi, spasi, spasi, satu, dua, tiga, nol dan baris baru.

"/var/log": Berkas dan direktori *log*

Direktori ini berisi bermacam-macam berkas *log*. Sebagian besar *log* harus ditulis ke dalam direktori ini atau subdirektori yang tepat.

Beberapa contoh dari sistem Ubuntu ialah: ``/var/log/aptitude", ``/var/log/auth.log", ``/var/log/cups", ``/var/log/daemon.log", ``/var/log/debug", ``/var/log/dmesg", ``/var/log/dpkg.log", ``/var/log/faillog", ``/var/log/installer", ``/var/log/kdm.log", ``/var/log/lpr.log", ``/var/log/mail.log", ``/var/log/messages", ``/var/log/syslog", ``/var/log/udev", ``/var/log/user.log", ``/var/log/wtmp", dst.

"/var/mail": Berkas *mailbox*pengguna

Mail spool harus dapat diakses melalui "/var/mail" dan berkas mail spool harus menggunakan form <nama_pengguna>. Berkas *mailbox* pengguna dalam lokasi ini harus disimpan dengan format standar *mailbox* UNIX.

"/var/opt": Data variabel untuk "/opt"

Data variabel untuk paket di dalam "/opt" harus di-*install* dalam "/var/opt/<subdir>", di mana <subdir> adalah nama dari *subtree* dalam "/opt" tempat penyimpanan data statik dari *package* tambahan perangkat lunak.

"/var/run": Data variabel *run-time*

Direktori ini berisi data informasi sistem yang mendeskripsikan sistem sejak di *boot*. Berkas di dalam direktori ini harus dihapus dulu saat pertama memulai proses *boot*. Berkas pengidentifikasi proses (PID), yang sebelumnya diletakkan di "/etc", sekarang diletakkan di "/var/run".

Program yang membaca berkas-berkas PID harus fleksibel terhadap berkas yang diterima, sebagai contoh: program tersebut harus dapat mengabaikan ekstra spasi, baris-baris tambahan, angka nol

yang diletakkan di depan, dll.

"/var/spool": Aplikasi data *spool*

"/var/spool" berisi data yang sedang menunggu suatu proses. Data di dalam "/var/spool" merepresentasikan pekerjaan yang harus diselesaikan dalam waktu depan (oleh program, pengguna atau administrator); biasanya data dihapus sesudah selesai diproses.

"/var/tmp": Berkas temporer lintas *reboot*

Direktori "/var/tmp" tersedia untuk program yang membutuhkan berkas temporer atau direktori yang diletakkan dalam *reboot* sistem. Karena itu, data yang disimpan di "/var/tmp" lebih bertahan daripada data di dalam "/tmp". Berkas dan direktori yang berada dalam "/var/tmp" tidak boleh dihapus saat sistem di-*boot*. Walaupun data-data ini secara khusus dihapus dalam *site-specific manner*, tetap direkomendasikan bahwa penghapusan dilakukan tidak sesering penghapusan di "/tmp".

"/var/yp": Berkas database NIS

Data variabel dalam Network Information Service (NIS) atau yang biasanya dikenal dengan Sun Yellow Pages (YP) harus diletakkan dalam direktori ini.

41.6. Tambahan untuk Linux

Bagian ini berisi syarat-syarat tambahan dan anjuran yang hanya dapat digunakan untuk Sistem Operasi Linux. Materi dari bagian ini tidak bertentangan dengan standard yang telahdijelaskan sebelumnya.

1. **"/": Direktori Root.** Pada sistem Linux, jika kernel terletak di "/", direkomendasikan untuk menggunakan nama vmlinux atau vmlinuz, yang telah digunakan di paket kernel source Linux saat ini.
2. **"/bin".** Pada "/bin", Sistem linux membutuhkan berkas tambahan setserial.
3. **"/dev: berkas piranti".** berkas-berkas yang harus terdapa di direktori ini antara lain: "/dev/null", "/dev/zero", "/dev/tty".
4. **"/etc": Sistem Konfigurasi host-specific.** Pada "/etc", sistem linux membutuhkan berkas tambahan, yaitu lilo.cnof
5. **"/proc".** Proc filesystem adalah method standard de-facto Linux untuk mengendalikan proses dan sistem informasi, dibandingkan dengan "/dev/kmem" dan method lain yang mirip. Sangat dianjurkan untuk menggunakan direktori ini untuk penerimaan dan storage informasi proses, serta informasi tentang kernel dan informasi memori.

41.7. Rangkuman

Standar Hirarki Sistem Berkas (*File Hierarchy Standard*) adalah rekomendasi penulisan direktori dan berkas-berkas yang diletakkan di dalamnya. FHS ini digunakan oleh perangkat lunak dan pengguna untuk menentukan lokasi dari berkas dan direktori.

Rujukan

[WEBRusQuYo2004] Rusty Russell, Daniel Quinlan, dan Christopher Yeoh. 2004. *Filesystem Hierarchy Standard – http://www.pathname.com/fhs/* . Diakses 27 Juli 2006.

Bab 42. Alokasi Blok Sistem Berkas

42.1. Pendahuluan

Untuk kebutuhan efisiensi dalam pertukaran data antara memori dengan disk, disk membagi ruang tempat penyimpanannya menjadi blok-blok. Hal ini berarti jumlah data yang mengalir antara keduanya dapat dihitung dengan satuan blok. Satu blok terdiri dari satu atau beberapa sektor. Suatu berkas yang disimpan di dalam disk mengisi blok-blok ini sehingga berkas tersebut terpotong-potong menjadi bagian-bagian yang sesuai dengan ukuran satu blok.

Dengan adanya pembagian ruang di disk menjadi blok-blok seperti ini maka timbul permasalahan tentang pengalokasian blok-blok tersebut. Tujuan dari penyelesaian masalah ini adalah agar kapasitas disk dapat diutilisasikan dengan sebaik-baiknya dan berkas-berkas dapat diakses dengan cepat. Dalam bab ini akan dibahas beberapa metoda-metoda pengalokasian blok sistem berkas yang sering digunakan oleh beberapa sistem operasi. Setelah itu kita akan membahas hal-hal lain yang juga berkenaan dengan konsep blok ini.

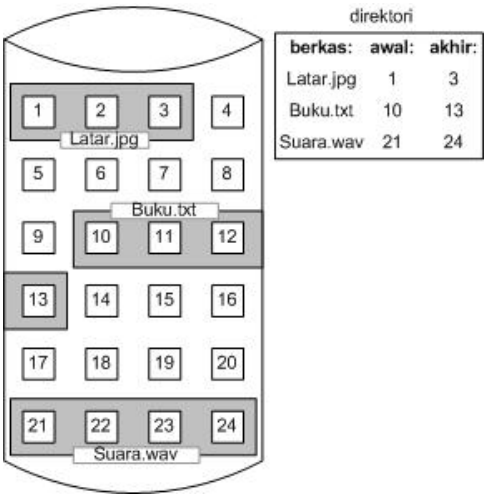
42.2. Metode Alokasi

Kegunaan penyimpanan sekunder yang utama adalah menyimpan berkas-berkas yang kita buat, karena sifat disk akan mempertahankan berkas walaupun tidak ada arus listrik. Oleh karena itu, agar kita dapat mengakses berkas-berkas dengan cepat dan memaksimalisasikan ruang yang ada di disk tersebut, maka lahirlah metode-metode untuk mengalokasikan berkas ke disk. Metode-metode yang akan dibahas lebih lanjut dalam buku ini adalah *contiguous allocation*, *linked allocation*, dan *indexed allocation*. Metode-metode tersebut memiliki beberapa kelebihan dan juga kekurangan. Biasanya sistem operasi memilih satu dari metode diatas untuk mengatur keseluruhan berkas.

Contiguous Allocation

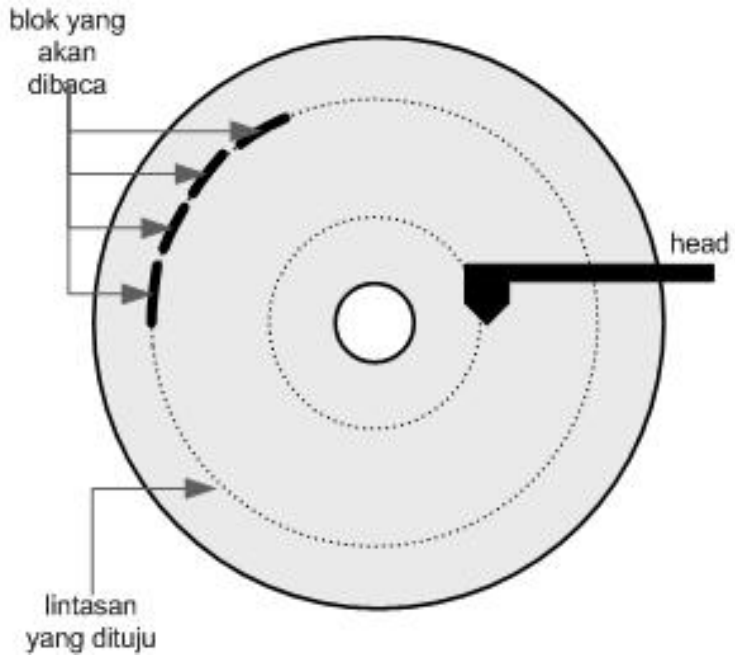
Metoda pengalokasian berurutan akan menempatkan satu berkas ke dalam blok-blok yang bersebelahan (Gambar 42.1, “Alokasi Berkesinambungan”). Penempatan dalam blok-blok yang bersebelahan berarti sektor-sektor yang digunakan juga pada posisi yang bersebelahan. Adapun kasus di mana sektor-sektor dalam satu lintasan sudah habis terpakai sehingga sektor selanjutnya yang dapat digunakan berada di lintasan yang berbeda. Namun dalam kasus ini, lintasan baru yang digunakan tersebut terletak di sebelah lintasan sebelumnya sehingga perpindahan head yang diperlukan hanyalah sedikit (Gambar 42.2, “Kondisi Disk”).

Gambar 42.1. Alokasi Berkesinambungan



Keuntungan lain dari alokasi berurutan adalah ia menunjang pengaksesan bagian tengah berkas secara langsung (direct access). Apabila suatu berkas dimulai dengan blok posisi b dan panjang berkas tersebut adalah p maka blok di posisi b + i dengan i < p pasti juga bagian dari berkas tersebut. Selain pengaksesan langsung, metoda alokasi ini juga menunjang pengaksesan berkas secara sekuensial yaitu berkas dibaca mulai dari bagian awal sampai akhir.

Gambar 42.2. Kondisi Disk



Setiap metoda pengalokasian mendefinisikan suatu berkas dengan cara yang berbeda. Suatu berkas dalam metoda ini didefinisikan atas alamat di mana blok awal berkas tersebut berada dan blok akhir berkas. Sebagai contoh, berkas dengan alamat 2 dan panjang 5 berarti berkas tersebut menempati blok mulai posisi ke-2 hingga blok di posisi ke-6.

Pengalokasian blok sistem berkas seperti ini akan menimbulkan kesulitan dalam mencari ruang kosong dari beberapa ruang yang tersedia untuk kebutuhan berkas baru. Untuk mengatasinya, ada dua cara yang biasa dilakukan, yaitu best fit dan first fit seperti yang telah dijelaskan dalam bab Alokasi Memori Berkesinambungan. Walaupun diantara keduanya tidak ada yang dapat dikatakan solusi terbaik tetapi algoritma first fit bekerja paling cepat.

Kedua algoritma di atas memiliki kelemahan yang disebut dengan fragmentasi eksternal. Setelah berkas-berkas ditulis kemudian dihapus dari disk, maka ruang yang tersedia di disk menjadi bagian-bagian kecil yang tersebar. Apabila potongan-potongan ruang ini tidak ada yang bisa memuat suatu berkas baru maka terjadilah masalah fragmentasi eksternal ini. Untuk mengatasi hal ini, ada beberapa metoda seperti yang pernah digunakan oleh komputer mikro pada masa lalu. Pada masa itu, tempat penyimpanan yang populer adalah floppy disk. Untuk mengatasi berkurangnya utilisasi ruang disk akibat fragmentasi eksternal, maka secara berkala dilakukan pemadatan isi disk. Operasi ini terdiri dari tahap pemindahan isi suatu floppy disk ke tempat penyimpanan sementara (floppy disk lain atau tape) sehingga floppy disk yang pertama menjadi kosong. Selanjutnya, proses diteruskan dengan tahap penulisan kembali floppy disk tersebut dengan isi dari tempat penyimpanan sementara tadi. Hasil yang didapatkan adalah satu floppy disk yang blok-bloknya terisi secara berkesinambungan.

Walaupun cara ini berhasil dan pernah dilakukan pada zaman dahulu, tetapi pemraktekannya pada hard disk dihindari. Hard disk biasanya memuat data yang jauh lebih banyak dari floppy disk. Operasi pemadatan di atas apabila dilakukan pada hard disk maka akan membutuhkan waktu berjam-jam atau mungkin sampai berminggu-minggu. Selain itu tempat penyimpanan sementara yang dibutuhkan pasti sangat besar dan mahal.

Permasalahan lain dari metoda alokasi ini adalah menentukan ruang yang dibutuhkan untuk berkas baru. Ketika suatu berkas akan dibuat maka ruang yang dapat menampung berkas tersebut harus dapat diperkirakan sebelumnya. Jika tidak, maka berkas tersebut mungkin tidak bisa diperbesar. Hal ini karena blok-blok setelah blok data berkas yang terakhir dapat sudah teralokasikan untuk berkas lain. Dalam memperkirakan ukuran berkas dibutuhkan perhitungan yang matang. Suatu berkas bisa saja berkembang menjadi sangat besar sehingga pada awalnya perlu dialokasikan ruang yang besar pula.

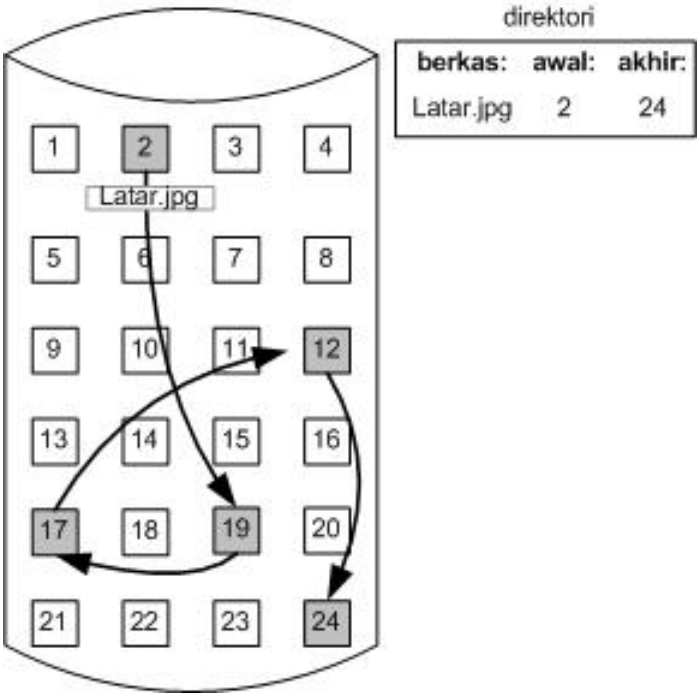
Untuk mengatasi permasalahan di atas, maka pengalokasian ruang untuk suatu berkas bisa langsung dalam jumlah yang besar. Dengan demikian, bisa jadi ruang yang ternyata terpakai hanya sebagian kecil saja dan sisanya dibiarkan terlantar. Kasus seperti ini di mana suatu ruang besar yang telah disediakan ternyata hanya terpakai sebagian disebut dengan fragmentasi internal.

Untuk menghindari kekurangan dari metoda pengalokasian ini, sebagian sistem operasi menggunakan metoda pengalokasian berkesinambungan yang dimodifikasi. Perbedaannya dengan metoda yang biasa adalah metoda yang dimodifikasi ini memungkinkan ruang tambahan - yang juga berupa blok-blok yang berkesinambungan - ditambahkan ke ruang yang pertama kali disediakan apabila ruang ini ternyata tidak cukup untuk menampung suatu berkas. Antara ruang yang satu dengan yang lain dihubungkan dengan suatu penghubung sehingga bisa saja ruang tambahan tersebut tidak berada di sebelah ruang yang pertama kali disediakan. Fragmentasi internal masih bisa terjadi apabila blok-blok tambahan ini terlalu besar dan fragmentasi eksternal dapat terjadi karena penambahan dan penghapusan ruang tambahan ini.

Linked Allocation

Metoda alokasi dengan penghubung blok memungkinkan blok-blok yang akan dialokasikan untuk suatu berkas tidak berada dalam posisi yang bersebelahan. Bisa saja suatu berkas menempati blok-blok yang berada di posisi ke-10, lalu ke-2, lalu ke-67 dst. Hal ini dimungkinkan karena setiap blok yang dialokasikan untuk suatu berkas memuat informasi tentang blok selanjutnya sehingga membentuk satu kesatuan (Gambar 42.3, “Alokasi dengan penghubung blok”). Penghubung ini juga memerlukan ruang dalam blok, sehingga pada metoda alokasi ini, ruang yang dapat diisi dengan data berkas dalam satu blok menjadi lebih sedikit dibandingkan metoda alokasi berkesinambungan.

Gambar 42.3. Alokasi dengan penghubung blok



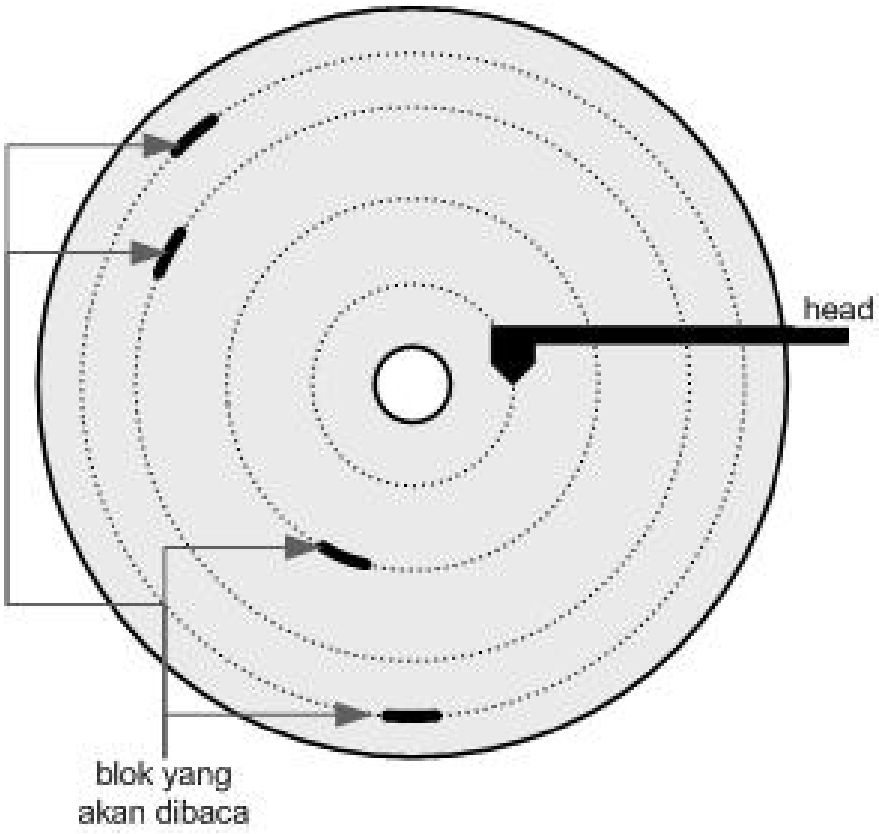
Untuk mendefinisikan suatu berkas, pada direktori tempat berkas tersebut berada, terdapat penghubung ke awal dan akhir blok dari berkas yang dimaksud. Sebagai contoh, misalkan blok awal berkas A berada di posisi ke-10 dan akhirnya di posisi ke-2. Pada direktori tempat berkas A berada akan terdapat informasi tentang berkas A yang bagian awalnya menempati blok ke-10 dan bagian akhirnya berada di blok ke-2.

Untuk membentuk berkas baru, proses awal yang diperlukan adalah menambah daftar berkas ke dalam direktori. Apabila penghubung di blok awal berkas ini diberi nilai nihil maka berkas yang terbentuk adalah berkas kosong. Untuk menambahkan isi berkas maka pencarian blok yang tidak terpakai akan dilakukan dengan menggunakan manajemen ruang kosong (dibahas Bagian 42.3, “Manajemen Ruang Kosong”. Setelah ditemukan, maka penghubung di blok awal berkas akan menunjuk ke blok ini. Selanjutnya blok yang baru ini akan diisi dengan data berkas. Dalam proses pembacaan berkas, operasi yang dilakukan adalah mengakses blok-blok dengan mengikuti penghubung dalam blok-blok tersebut.

Dengan menggunakan metoda alokasi ini maka permasalahan fragmentasi eksternal tidak akan ditemukan. Untuk membentuk berkas pun tidak perlu untuk menyebutkan jumlah blok yang diperlukan. Hal ini berarti berkas-berkas bisa bertambah besar dengan mudah.

Metoda pengalokasian blok seperti ini tidak memungkinkan berkas dibaca tanpa menggunakan cara yang sekuensial. Tidak seperti metoda alokasi sebelumnya, apabila bagian awal suatu berkas berada di blok b dan panjang berkas adalah p, maka blok di posisi b + i dengan i < p belum tentu bagian dari berkas tersebut. Perhatikan juga hubungan antara blok dengan sektor dan lintasan disk. Karena suatu berkas dapat menempati blok-blok di posisi sembarang, berarti berkas tersebut dapat menempati sektor-sektor di posisi yang sembarang pula yang bahkan bisa berada di lintasan yang berbeda. Hal ini menyebabkan dibutuhkannya beberapa waktu tambahan dalam proses pembacaan suatu berkas. Waktu yang dimaksud adalah waktu yang dibutuhkan agar sektor yang ingin dibaca melewati head disk dan waktu pergerakan pembaca disk ke lintasan tempat sektor yang akan dibaca pada (Gambar 42.4, “Alokasi Selain Berkesinambungan”).

Gambar 42.4. Alokasi Selain Berkesinambungan



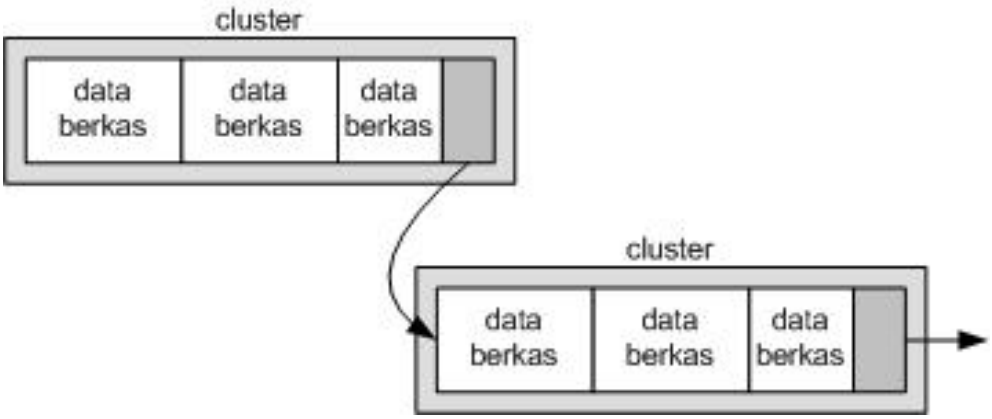
Seperti yang telah dijelaskan sebelumnya, penghubung blok memerlukan ruang tersendiri di dalam blok sehingga data-data berkas yang bisa termuat dalam satu blok akan berukuran lebih kecil dari ukuran blok itu sendiri (Gambar 42.5, “Ruang Blok Dialokasikan Penghubung”). Untuk lebih meningkatkan utilitas disk biasanya berkas-berkas dimasukan ke dalam sekumpulan blok yang disebut cluster dan sistem berkas bekerja berdasarkan cluster, bukan blok. Satu cluster terdiri dari dua atau lebih blok yang berkesinambungan. Antara satu cluster dengan cluster berikutnya yang teralokasikan untuk suatu berkas terdapat penghubung.

Gambar 42.5. Ruang Blok Dialokasikan Penghubung



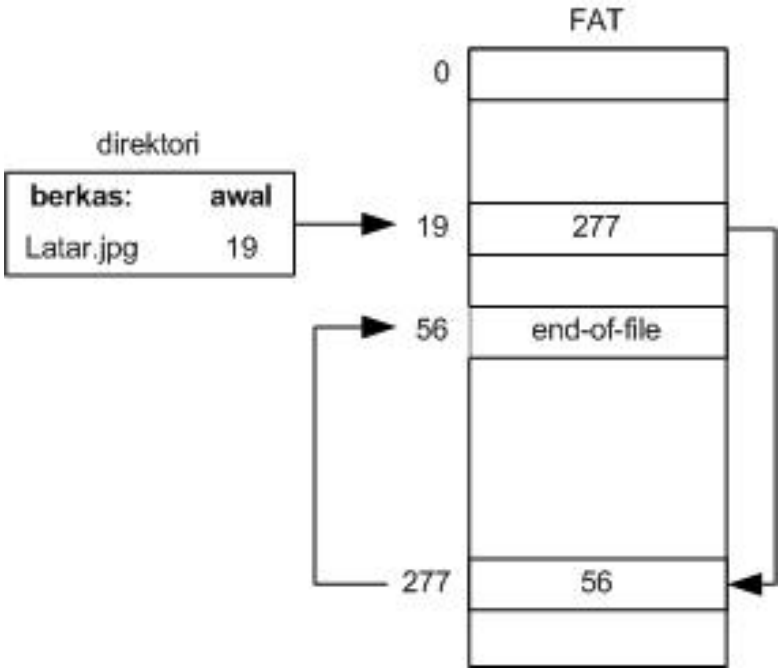
Namun, berbeda dengan metoda alokasi yang biasa, penghubung ini cukup berada di blok terakhir dari setiap cluster (Gambar 42.6, “Cluster”). Dengan menggunakan cluster maka persentase antara ruang yang dibutuhkan untuk penghubung dengan ruang yang benar-benar terpakai untuk data berkas menjadi berkurang. Selain itu, karena jumlah sektor yang terkumpul dalam satu cluster lebih besar dari satu blok saja maka waktu pembacaan suatu berkas mulai dari awal sampai akhir menjadi lebih cepat. Hal ini dikarenakan banyak potongan-potongan berkas berada di sektor yang berkesinambungan sehingga kemungkinan sektor yang akan dibaca berada di lintasan lain menjadi berkurang.

Gambar 42.6. Cluster



Salah satu varian dari metoda alokasi dengan penggunaan penghubung adalah metoda dengan penggunaan tabel alokasi berkas (FAT). Dengan metoda ini maka di setiap bagian awal partisi disk terdapat satu tabel yang berisi diantaranya daftar indeks blok yang berada dalam partisi tersebut, informasi terpakai atau tidaknya blok-blok ini, dan keterhubungan satu blok dengan blok lainnya. Daftar nama berkas-berkas dan blok awal dari masing-masing berkas masih dimiliki oleh setiap direktori dalam partisi tersebut. FAT hanya menggantikan penghubung yang dalam metoda alokasi dengan penghubung biasa dimiliki oleh masing-masing blok. Dengan FAT, hubungan satu blok dengan blok lain sehingga membentuk suatu berkas secara utuh termuat sebagai isi tabel tersebut (Gambar 42.7, “Penggunaan FAT”). Sebagai contoh, apabila blok dengan indeks 10 terhubung dengan blok indeks 6 maka di dalam tabel FAT terdapat informasi bahwa blok 10 berkorespondensi dengan blok 6. Pemetaan ini terus berlangsung hingga blok akhir berkas sehingga akan terbentuk suatu berkas secara utuh. Perbedaan lain dari alokasi dengan penghubung biasa adalah dalam mendefinisikan akhir dari berkas. Dalam metoda ini, untuk menandakan suatu blok merupakan akhir dari berkas maka dalam FAT blok yang dimaksud diberikan informasi end-of-file, sedangkan pada metoda yang biasa informasi ini termuat dalam direktori.

Gambar 42.7. Penggunaan FAT



Pada metoda dengan penggunaan FAT, blok-blok yang belum terpakai ditandai dengan nilai 0. Hal ini menguntungkan apabila suatu berkas hendak diperbesar karena pencarian blok yang dapat dipakai untuk menampung data tambahan berkas menjadi mudah. Pencarian cukup dengan memeriksa FAT lalu menggunakan blok bertanda nilai 0 yang pertama kali ditemukan.

Dengan menggunakan FAT maka direct access ke suatu berkas dapat didukung. Hal ini karena semua bagian dari berkas dapat diketahui posisinya termuat dalam dua pusat informasi. Pusat informasi yang dimaksud adalah dalam setiap direktori untuk informasi blok awal berkas dan FAT untuk informasi blok-blok yang dialokasikan untuk berkas tersebut.

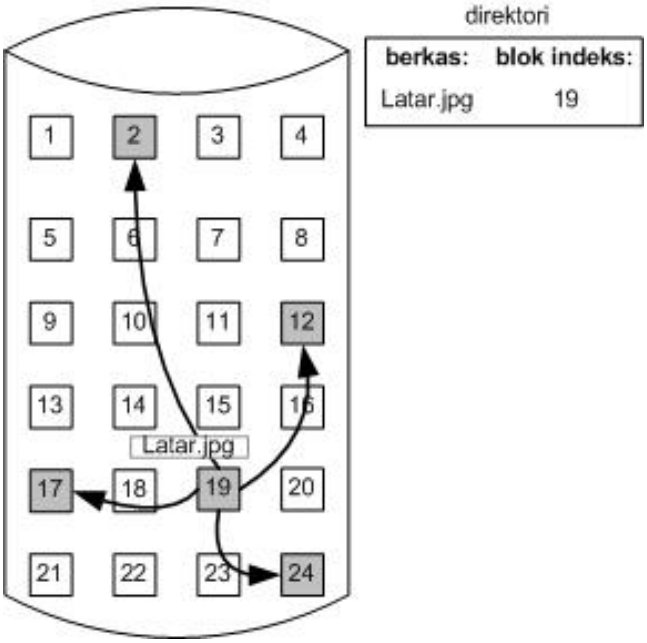
Berdasarkan posisi fisik FAT berada di dalam partisi disk, dapat diketahui bahwa dalam pembacaan suatu berkas mungkin dibutuhkan pergerakan head disk yang banyak. Head harus bergerak ke posisi awal partisi disk untuk membaca FAT lalu bergerak kembali ke posisi fisik blok yang hendak dibaca. Hal ini berarti pengaksesan berkas dengan menggunaan FAT dapat menjadi lebih lambat dibandingkan dengan menggunaan penghubung blok biasa. Masalah ini dapat dihindari apabila ketika proses pembacaan berkas, isi dari FAT disimpan dahulu dalam cache memori. Dengan cara ini head disk tidak perlu bergerak ke awal partisi setiap kali ia telah selesai membaca satu blok berkas karena semua alamat blok yang hendak dibaca sudah diingat.

Indexed Allocation

Di bagian sebelumnya telah dijelaskan keuntungan dari penggunaan alokasi dengan penghubung blok. Namun metoda ini mempunyai kelemahan karena tidak mendukung pengaksesan secara langsung (direct access). Telah dijelasakan juga bahwa permasalahan ini dapat diatasi dengan adanya FAT. Metoda alokasi lain yang juga mempunyai semua kelebihan metoda dengan penggunaan FAT adalah alokasi dengan pengindeksan. Ide dasar dari metoda alokasi ini mengumpulkan semua penghubung blok ke dalam satu lokasi yaitu blok indeks.

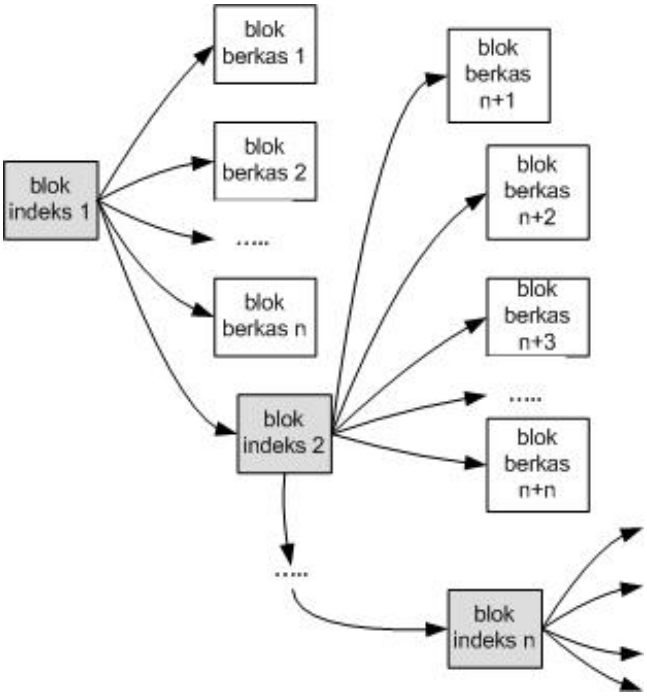
Dalam metoda ini setiap berkas mempunyai blok indeks sendiri yaitu berupa array yang berisi alamat blok-blok yang dialokasikan untuk berkas ini. Isi array dengan indeks ke-i adalah alamat blok ke-i dari berkas yang memiliki array tersebut. Alamat blok indeks itu sendiri terdapat di direktori tempat ia berada (Gambar 42.8, “Pengalokasikan dengan Indeks”). Cara seperti ini mendukung direct access dari berkas karena blok ke-n bisa langsung diakses karena alamatnya termuat di blok indeks.

Gambar 42.8. Pengalokasikan dengan Indeks



Alokasi dengan Pengindeksan tidak akan menimbulkan fragmentasi eksternal karena setiap blok yang belum digunakan dapat dialokasikan untuk berkas baru atau untuk perbesaran berkas yang sudah ada. Walaupun begitu, metoda ini mempunyai kekurangan yang disebabkan karena pasti dibutuhkannya minimal satu blok indeks. Perhatikan bahwa blok indeks merupakan blok tersendiri sehingga apabila suatu berkas dapat ditempatkan ke dalam dua blok maka ia membutuhkan satu blok tambahan yaitu blok indeks. Dalam blok indeks ini dapat dipastikan banyak ruang yang tidak tepakai.

Gambar 42.9. Skema Terhubung

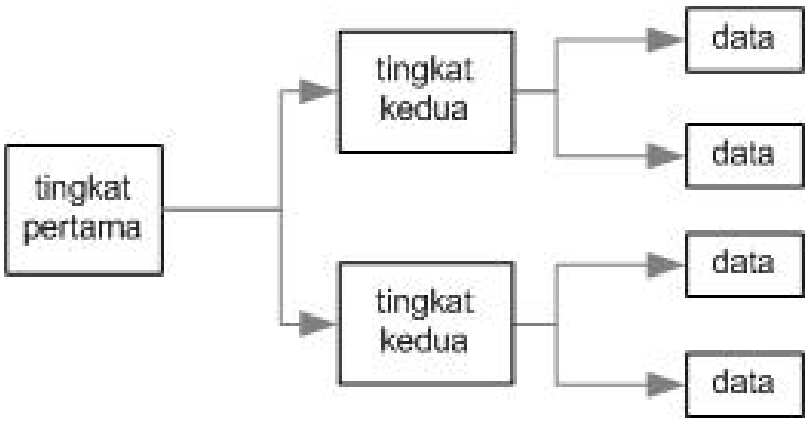


Topik di atas menimbulkan pertanyaan tentang seberapa besar seharusnya ukuran blok indeks. Karena jumlah berkas dalam disk biasanya banyak dan masing-masing berkas membutuhkan blok indeks maka ukuran blok indeks harus sekecil mungkin agar penggunaan ruang disk menjadi efisien. Walaupun begitu, apabila ukurannya terlalu kecil maka blok indeks tidak bisa memuat informasi indeks blok dari berkas yang berukuran besar. Selain itu, mekanisme indeks blok juga harus mendukung konsep-konsep berikut:

Skema terhubung: satu blok indeks bisa terhubung ke blok indeks blok lainnya. Skema seperti ini bisa mendukung berkas yang berukuran besar karena semua informasi alamat blok yang dialokasikan untuknya bisa termuat dalam sejumlah blok indeks. Cara yang dapat digunakan adalah apabila satu blok indeks tidak mencukupi maka salah satu penghubung di blok indeks tersebut akan menunjuk ke blok indeks tambahan (Gambar 42.9, “Skema Terhubung”). Blok indeks juga digunakan untuk menginformasikan blok-blok mana saja yang dialokasikan untuk berkas tersebut.

Pengindeksan bertingkat: blok indeks di tingkat pertama berisi penghubung ke blok-blok indeks di tingkat kedua. Baru di tingkat kedua inilah indeks blok-blok berkas termuat (Gambar 42.10, “Pengindeksan bertingkat”). Jumlah tingkatan bisa lebih besar dari dua. Untuk mengakses suatu blok berkas, sistem operasi menggunakan blok indeks tingkat pertama lalu kedua dst sampai tingkat terbawah yang akan menunjuk blok berkas yang diharapkan.

Gambar 42.10. Pengindeksan bertingkat

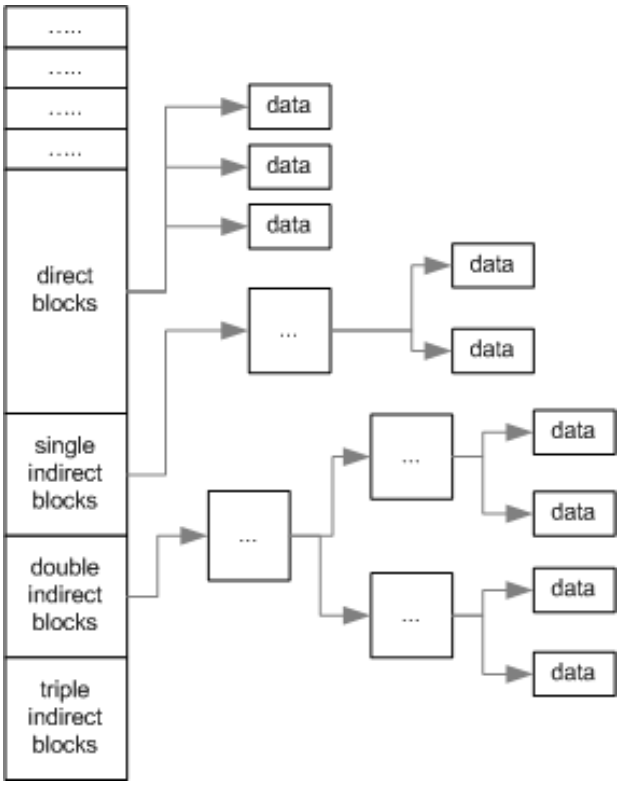


Skema gabungan: blok indeks terbagi-bagi menjadi bagian-bagian yang menunjuk ke blok-blok yang berbeda jenis. Blok seperti ini dalam BSD UNIX disebut inode. Ruang dalam inode terbagi menjadi delapan bagian (Gambar 42.11, “Skema Gabungan”). Empat bagian pertama berisi informasi tentang berkas yang memiliki inode tersebut dan tentang inode itu sendiri. Bagian inode yang kelima langsung menunjuk ke blok data berkas. Bagian dari inode yang menunjuk ke ke blok data berkas ini disebut direct blocks. Perhatikan bahwa direct blocks adalah merupakan bagian dari ruang inode, bukan blok data berkas yang ditunjuknya. Bagian keenam dari inode menunjuk ke blok indeks tingkat kedua, lalu blok indeks tingkat kedua tersebut menunjuk ke blok data berkas. Bagian dari inode yang disebutkan di atas disebut single indirect blocks. Bagian ketujuh menunjuk ke blok indeks tingkat kedua, lalu blok indeks ini menunjuk ke blok indeks tingkat ketiga, lalu blok indeks tingkat ketiga ini menunjuk ke blok data berkas. Bagian ketujuh ini disebut juga dengan double indirect blocks. Bagian terakhir hampir sama dengan bagian sebelumnya hanya saja masing-masing blok indeks akan menunjuk blok-blok indeks dengan tingkat di bawahnya sampai tingkatan keempat. Blok indeks tingkat keempat inilah yang akan menunjuk ke blok data berkas. Bagian inode yang terakhir ini disebut juga dengan triple indirect blocks.

Perhatikan bahwa metoda alokasi dengan pengindeksan dapat mengakibatkan banyaknya pergerakan head disk ketika membaca berkas. Hal ini dikarenakan posisi blok indeks dengan blok-blok berkas dapat saling berjauhan. Dengan banyaknya pergerakan head disk maka waktu yang dibutuhkan ketika proses pengaksesan berkas menjadi lama. Hal ini dapat dikurangi apabila blok indeks ditaruh di dalam memori cache sehingga pergerakan head disk hanya ditujukan untuk membaca blok-blok berkas saja.

Metode yang satu ini memecahkan masalah fragmentasi eksternal dari metode *contiguous allocation* dan ruang yang cuma-cuma untuk petunjuk pada metode *linked allocation*, dengan cara menyatukan semua petunjuk kedalam blok indeks yang dimiliki oleh setiap berkas. Jadi, direktori hanya menyimpan alamat dari blok indeks tersebut, dan blok indeks tersebut yang menyimpan alamat dimana blok-blok berkas berada. Untuk berkas yang baru dibuat, maka blok indeksnya di set dengan *null*.

Gambar 42.11. Skema Gabungan



Metode ini mendukung pengaksesan secara langsung, bila kita ingin mengakses blok ke-i, maka kita hanya mencari isi dari blok indeks tersebut yang ke-i untuk dapatkan alamat blok tersebut. Metode *indexed allocation* tidak menyia-nyiakan ruang disk untuk petunjuk, karena dibandingkan dengan metode *linked allocation*, maka metode ini lebih efektif, kecuali bila satu berkas tersebut hanya memerlukan satu atau dua blok saja.

Metode ini juga memiliki masalah. Masalah itu timbul disaat berkas berkembang menjadi besar dan blok indeks tidak dapat menampung petunjuk-petunjuknya itu dalam satu blok. Salah satu mekanisme dibawah ini dapat dipakai untuk memecahkan masalah yang tersebut. Mekanisme-mekanisme itu adalah:

- ***Linked scheme.*** Untuk mengatasi petunjuk untuk berkas yang berukuran besar mekanisme ini menggunakan tempat terakhir dari blok indeks untuk alamat ke blok indeks selanjutnya. Jadi, bila berkas kita masih berukuran kecil, maka isi dari tempat yang terakhir dari blok indeks berkas tersebut adalah *null*. Namun, bila berkas tersebut berkas besar, maka tempat terakhir itu berisikan alamat untuk ke blok indeks selanjutnya, dan begitu seterusnya.
- **Indeks bertingkat.** Pada mekanisme ini blok indeks itu bertingkat-tingkat, blok indeks pada tingkat pertama akan menunjukkan blok-blok indeks pada tingkat kedua, dan blok indeks pada tingkat kedua menunjukkan alamat-alamat dari blok berkas, tapi bila dibutuhkan dapat dilanjutkan kelevel ketiga dan keempat tergantung dengan ukuran berkas tersebut. Untuk blok indeks 2 level dengan ukuran blok 4.096 byte dan petunjuk yang berukuran 4 byte, dapat mengalokasikan berkas hingga 4 GB, yaitu 1.048.576 blok berkas.
- ***Combined scheme*.** Mekanisme ini menggabungkan *direct block* dan *indirect block*. *Direct block* akan langsung menunjukkan alamat dari blok berkas, tetapi pada *indirect block* akan menunjukkan blok indeks terlebih dahulu seperti dalam mekanisme indeks bertingkat. *Single*

indirect block akan menunjukkan ke blok indeks yang akan menunjukkan alamat dari blok berkas, *double indirect block* akan menunjukkan suatu blok yang bersifat sama dengan blok indeks 2 level, dan *triple indirect block* akan menunjukkan blok indeks 3 level. Dimisalkan ada 15 petunjuk dari mekanisme ini, 12 pertama dari petunjuk tersebut adalah *direct block*, jadi bila ukuran blok 4 byte berarti berkas yang dapat diakses secara langsung didukung sampai ukurannya 48 KB. 3 petunjuk berikutnya adalah *indirect block* yang berurutan dari *single indirect block* sampai *triple indirect block*. Yang hanya mendukung 32 bit petunjuk berkas berarti akan hanya mencapai 4 GB, namun yang mendukung 64 bit petunjuk berkas dapat mengalokasikan berkas berukuran sampai satuan terabyte.

Kinerja Sistem Berkas

Keefisiensian penyimpanan dan waktu akses blok data adalah kriteria yang penting dalam memilih metode yang cocok untuk sistem operasi untuk mengimplementasikan sesuatu. Sebelum memilih sebuah metode alokasi, kita butuh untuk menentukan bagaimana sistem ini akan digunakan.

Untuk beberapa tipe akses, *contiguous allocation* membutuhkan hanya satu akses untuk mendapatkan sebuah blok disk. Sejak kita dapat dengan mudah menyimpan alamat inisial dari sebuah berkas di memori, kita dapat menghitung alamat disk dari blok ke-i (atau blok selanjutnya) dengan cepat dan membacanya dengan langsung.

Untuk *linked allocation*, kita juga dapat menyimpan alamat dari blok selanjutnya di memori dan membacanya dengan langsung. Metode ini bagus untuk akses secara berurutan; untuk akses langsung, bagaimana pun, sebuah akses menuju blok ke-i harus membutuhkan pembacaan disk ke-i. Masalah ini menunjukkan mengapa alokasi yang berurutan tidak digunakan untuk aplikasi yang membutuhkan akses langsung.

Sebagai hasilnya, beberapa sistem mendukung berkas-barkas yang diakses langsung dengan menggunakan *contiguous allocation* dan yang diakses berurutan dengan *linked allocation*. Di dalam kasus ini, sistem operasi harus mempunyai struktur data yang tepat dan algoritma untuk mendukung kedua metode alokasi.

Indexed allocation lebih komplek. Jika blok indeks sudah ada dimemori, akses dapat dibuat secara langsung. Bagaimana pun, menyimpan blok indeks tersebut di memori membutuhkan tempat yang dapat ditolerir. Dengan begitu, kinerja dari *indexed allocation* tergantung dari struktur indeks, ukuran file, dan posisi dari blok yang diinginkan.

Beberapa sistem menggabungkan *contiguous allocation* dengan *indexed allocation* dengan menggunakan *contiguous allocation* untuk berkas-berkas yang kecil (diatas tiga atau empat berkas), dan secara otomatis mengganti ke *indexed allocation* jika berkas bertambah besar.

42.3. Manajemen Ruang Kosong

Setelah kita mempelajari metoda-metoda pengalokasian blok sistem berkas maka kita perlu juga untuk mengetahui manajemen ruang disk yang kosong. Manajemen ruang kosong sangat diperlukan sebelum suatu blok dapat dialokasikan untuk menampung data berkas: blok tersebut harus ditentukan terlebih dahulu dan pastinya blok yang dimaksud adalah ruang kosong. Dalam sub-bab ini kita akan mempelajari cara-cara untuk dapat mengetahui bagian dari disk yang merupakan ruang kosong.

Ruang-ruang kosong di dalam disk akan dapat tercantum dalam daftar ruang kosong. Apabila ada berkas baru, ruang yang dibutuhkan untuk berkas tersebut akan dicari dari daftar ini. Setelah data berkas ditulis, ruang ini dihapus dari daftar ruang kosong. Hal sebaliknya terjadi apabila berkasnya dihapus. Pengimplementasian daftar ruang kosong bisa jadi tidak sebagai berkas, seperti yang akan dijelaskan di bawah ini.

Vektor Bit

Dengan penggunaan vektor bit, setiap blok akan ditandai 1 apabila blok tersebut kosong dan 0 apabila sebaliknya. Sebagai contoh, apabila dalam suatu disk blok-blok yang kosong hanyalah blok-blok dengan indeks 2, 5, dan 6 maka vektor bit yang sesuai adalah 00100110000... Dapat

dilihat bahwa vektor bit sangatlah sederhana. Kelebihan lain dari penggunaan cara ini adalah mudahnya dalam mencari blok pertama yang kosong. Selain itu berdasarkan pola bit dalam vektor, kita juga dapat mengetahui adanya blok-blok kosong yang berkesinambungan.

Kelemahan dari penggunaan vektor bit adalah dibutuhkannya ruang di memori untuk menyimpan vektor ini. Hal ini mungkin dilakukan apabila ukuran disk kecil (seperti pada komputer mikro). Namun apabila ukuran disk besar maka ruang di memori bisa habis hanya untuk menampung vektor bit ini.

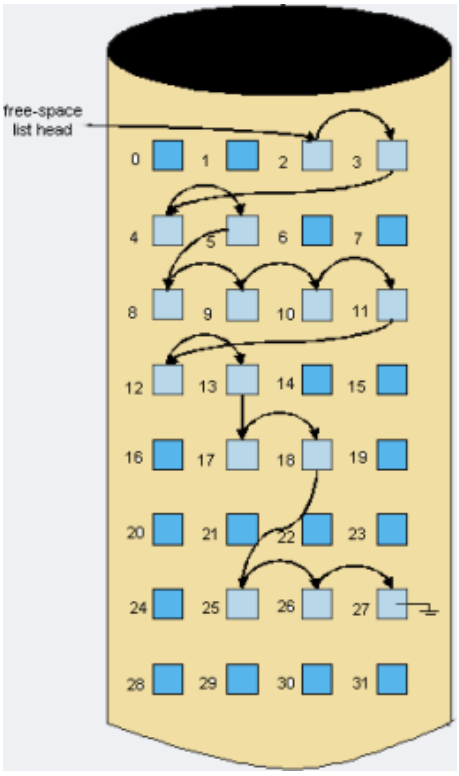
Linked List

Cara kedua untuk menandai blok-blok yang kosong adalah dengan menggunakan penghubung (link) di antara blok-blok kosong tersebut. Blok kosong pertama akan diberikan penghubung khusus. Lalu dari blok ini akan terdapat penghubung ke blok kosong kedua. Selanjutnya, dari blok kosong kedua ini ada penghubung ke blok kosong ketiga dst. Mekanisme semacam ini akan berakibat dibutuhkannya waktu yang lama dalam proses penelusuran blok-blok kosong dalam disk. Untungnya proses semacam ini jarang dilakukan.

Pengelompokan

Cara lain untuk mendaftarkan blok-blok kosong adalah dengan menggunakan satu blok untuk menyimpan alamat blok-blok kosong didekatnya. Di dalam blok tempat alamat tersebut, terdapat pula satu penghubung atau data alamat dari blok semacam ini yang selanjutnya. Dengan cara ini berarti antar blok berisi alamat blok-blok kosong terdapat penghubung (Gambar 42.12, “Ruang kosong *linked list*”). Blok yang berisi data berkas digambarkan dengan kotak yang berwarna abu-abu, blok yang menunjuk ke blok kosong digambarkan dengan kotak dengan sisi dan angka yang ditebalkan, dan blok kosong digambarkan dengan kotak berwarna putih. Keuntungan dari mekanisme dengan pengelompokan adalah sekumpulan besar dari blok-blok kosong dapat dengan cepat diketahui.

Gambar 42.12. Ruang kosong *linked list*



Penghitungan

Umumnya ruang kosong dalam disk berupa urutan dari blok-blok kosong. Berdasarkan hal ini, kita bisa mendaftarkan rangkaian blok-blok kosong tersebut dengan cara memasukan data alamat blok kosong pertama dari rangkaian tersebut lalu disertakan juga jumlah blok kosong yang bersebelahan dengan blok yang pertama. Cara seperi inilah yang merupakan konsep dari pengimplementasian daftar ruang kosong dengan penghitungan.

42.4. Pengimplementasian Direktori

Dalam bagian ini kita akan mempelajari algoritma dalam pengimplementasian direktori serta manajemen di dalamnya. Topik ini sangat berpengaruh terhadap efisiensi, kinerja, dan kehandalan sistem berkas. Dua algoritma yang digunakan dalam berbagai sistem operasi adalah dengan daftar linear dan penggunaan struktur data berupa tabel hash.

Daftar Linear

Cara ini merupakan cara yang paling sederhana yaitu dengan mendaftarkan nama-nama berkas yang termuat dalam direktori tersebut. Selanjutnya, blok-blok berkas yang termuat dalam direktori dihubungkan dengan penghubung. Untuk mencari suatu data berkas maka blok-blok berkas diakses satu persatu. Untuk proses penambahan berkas baru, tahapan yang dilaksanakan adalah memeriksa apakah nama berkas baru tersebut sudah ada dalam direktori, lalu, apabila tidak terjadi konflik, data berkas baru ditambahkan di akhir direktori. Dalam penghapusan berkas diperlukan pencarian nama berkas di direktori lalu membuang isi blok dari berkas tersebut. Setelah itu kita bisa menandai blok-blok tersebut sebagai blok tidak terpakai atau mendaftarkannya di daftar ruang direktori yang kosong. Cara ini dimaksudkan agar ruang-ruang dalam direktori bisa terpakai kembali. Cara yang lebih rumit adalah menggeser isi blok-blok di posisi setelah ruang yang dihapus ke posisi blok-blok yang baru saja dihapus. Selanjutnya, panjang direktori akan dikurangi.

Kelemahan dari cara ini adalah besarnya waktu yang dibutuhkan dalam proses pencarian data berkas. Hal ini merupakan masalah yang biasa ditemukan dalam pengimplementasian dengan penggunaan penghubung antar blok yang disusun secara berurutan. Karena informasi direktori sering digunakan, maka kelemahan ini akan sangat mempengaruhi kinerja sistem. Bahkan banyak sistem operasi yang menyimpan informasi ini dalam cache sehingga setiap kali informasi direktori dibutuhkan, proses yang berkenaan dengan kerja disk tidak lagi dibutuhkan.

Tabel Hash

Untuk mengurangi waktu pencarian data berkas, pengimplementasian direktori bisa dibantu dengan struktur data berupa tabel hash. Dalam cara ini, tabel hash digunakan untuk mempercepat pencarian alamat blok dari data berkas yang dimaksud. Tabel hash akan memroses nilai yang diambil dari nama berkas lalu mengembalikan nilai yang akan menunjuk ke berkas tersebut dalam daftar linear. Dengan cara ini, waktu yang dibutuhkan dalam proses pencarian data berkas di direktori akan berkurang. Perhatikan bahwa daftar linear masih digunakan di sini.

Kesulitan dari penggunaan tabel hash adalah ketidak-fleksibelan struktur data ini untuk berkembang sesuai dengan banyaknya berkas. Sebagai contoh, misalkan ada tabel hash yang memuat 64 masukan. Fungsi dari tabel hash akan mengubah nama berkas menjadi suatu bilangan bulat non-negatif antara 0 sampai dengan 63. Hal di atas berarti jumlah berkas yang dapat didukung oleh tabel ini hanyalah 64. Apabila muncul berkas yang ke-65 maka tabel hash ini harus diperbesar, misalkan menjadi dua kali lipatnya. Dengan ukuran yang baru berarti fungsi pemetaan baru juga harus dibuat sehingga, berdasarkan ukuran di atas, fungsi ini bisa menjangkau nilai 0 sampai 127. Dengan adanya fungsi baru ini berarti seluruh isi direktori juga harus diadaptasikan.

42.5. Efisiensi dan Kinerja

Kita sekarang dapat mempertimbangkan mengenai efek dari alokasi blok dan manajeman direktori dalam kinerja dan penggunanan disk yang efisien. Di bagian ini, kita mendiskusikan tentang bermacam-macam teknik yang digunakan untuk mengembangkan efisiensi dan kinerja dari

penyimpanan kedua.

Efisiensi

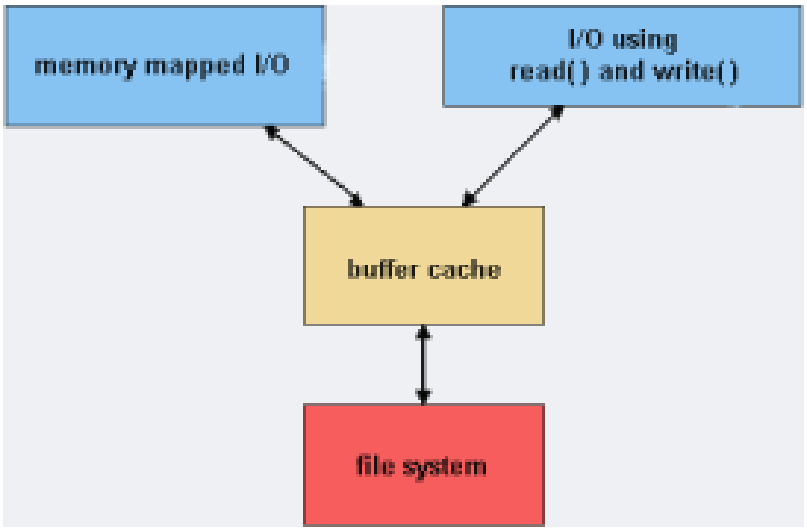
Penggunaan yang efisien dari ruang disk sangat tergantung pada alokasi disk dan algoritma direktori yang digunakan. Sebagai contoh, UNIX mengembangakan kinerjanya dengan mencoba untuk menyimpan sebuah blok data berkas dekat dengan blok inode berkas untuk mengurangi waktu pencarian.

Tipe dari data normalnya disimpan di masukan direktori berkas (atau inode) juga membutuhkan pertimbangan. Biasanya, tanggal terakhir penulisan direkam untuk memberikan informasi kepada pengguna dan untuk menentukan jika berkas ingin di *back up*. Beberapa sistem juga menyimpan sebiuah "last access date", supaya seorang pengguna dapat menentukan kapan berkas terakhir dibaca. Hasil dari menyimpan informasi ini adalah ketika berkas sedang dibaca, sebuah field di struktur direktori harus ditulisi. Prasyarat ini dapat tidak efisien untuk pengaksesan berkas yang berkala. Umumnya setiap persatuan data yang berhubungan dengan berkas membutuhkan untuk dipertimbangkan efeknya pada efisiensi dan kinerja.

Sebagai contoh, mempertimbangkan bagaimana efisiensi dipengaruhi oleh ukuran penunjuk-penunjuk yang digunakan untuk mengakses data. Bagaimana pun, penunjuk-penunjuk membutuhkan ruang lebih untuk disimpan, dan membuat metode alokasi dan manajemen ruang-kosong menggunakan ruang disk yang lebih. Satu dari kesulitan memilih ukuran penunjuk, atau juga ukuran alokasi yang tetap diantara sistem operasi, adalah rencana untuk efek dari teknologi yang berubah.

Kinerja

Gambar 42.13. Menggunakan *unified buffer cache*

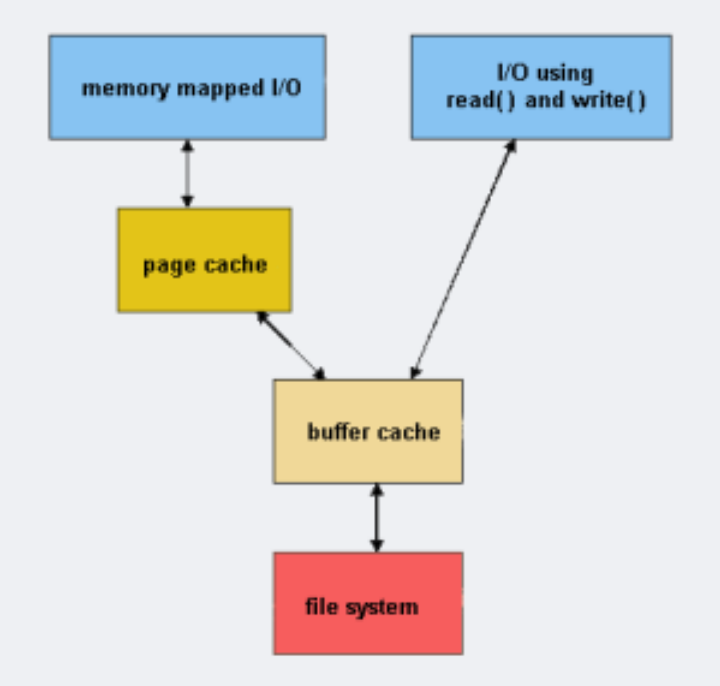


Sekali algoritma sistem berkas dipilih, kita tetap dapat mengembangkan kinerja dengan beberapa cara. Kebanyakan dari *disk controller* mempunyai memori lokal untuk membuat *on-board cache* yang cukup besar untuk menyimpan seluruh *tracks* dengan sekejap.

Beberapa sistem membuat seksi yang terpisah dari memori utama untuk digunakan sebagai *disk cache*, dimana blok-blok disimpan dengan asumsi mereka akan digunakan lagi dengan secepatnya. Sistem lainnya menyimpan data berkas menggunakan sebuah *page cache*. *Page cache* tersebut menggunakan teknik memori virtual untuk menyimpan data berkas sebagai halaman-halaman daripada sebagai blok-blok *file-system-oriented*. Menyimpan data berkas menggunakan alamat-alamat virtual jauh lebih efisien daripada menyimpannya melalui blok disk fisik. Ini dikenal sebagai *unified virtual memory*.

Sebagian sistem operasi menyediakan sebuah *unified buffer cache*. Tanpa sebuah *unified buffer cache*, kita mempunyai situasi panggilan *mapping* memori butuh menggunakan dua cache, *page cache* dan *buffer cache*. Karena sistem memori virtual tidak dapat menggunakan dengan *buffer cache*, isi dari berkas di dalam *buffer cache* harus diduplikat ke *page cache*. Situasi ini dikenal dengan *double caching* dan membutuhkan menyimpan data sistem-berkas dua kali. Tidak hanya membuang-buang memori, tetapi ini membuang CPU dan perputaran M/K dikerenakan perubahan data ekstra diantara memori sistem. Juga dapat menyebabkan korupsi berkas. Sebuah *unified buffer cache* mempunyai keuntungan menghindari *double caching* dan menunjuk sistem memori virtual untuk mengatur data sistem berkas.

Gambar 42.14. Tanpa *unified buffer cache*



42.6. *Recovery*

Sejak berkas-berkas dan direktori-direktori dua-duanya disimpan di memori utama dan pada disk, perawatan harus dilakukan untuk memastikan kegagalan sistem tidak terjadi di kehilangan data atau di tidakkonsistennya data.

Pemeriksaan Konsistensi Berkas

Informasi di direktori di memori utama biasanya lebih baru daripada informasi yang ada di disk, karena penulisan dari informasi direktori yang disimpan ke disk tidak terlalu dibutuhkan secepat terjadinya pembaharuan. Mempertimbangkan efek yang memungkinkan terjadinya *crash* pada komputer. Secara berkala, program khusus akan dijalankan pada saat waktu *reboot* untuk mengecek dan mengoreksi disk yang tidak konsisten. Pemerikasaan rutin membandingkan data yang ada di struktur direktori dengan blok data pada disk, dan mencoba untuk memperbaiki ketidakkonsistenan yang ditemukan.

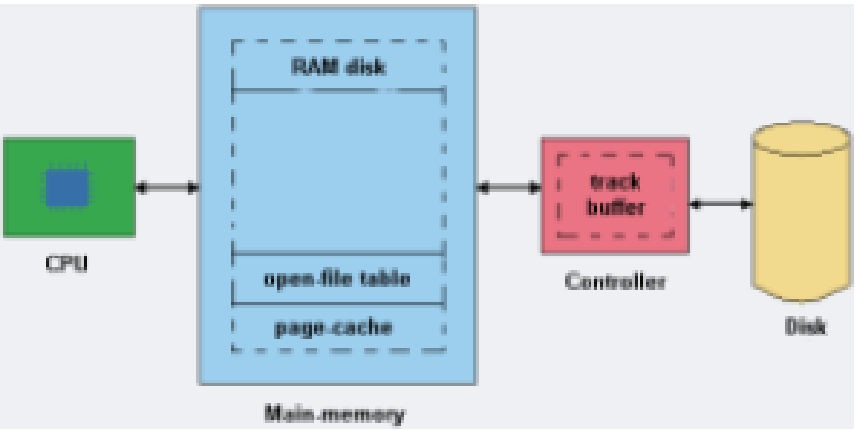
Apabila terjadi kegagalan kerja pada sistem, berkas-berkas yang tersimpan dalam disk dan memori dapat terancam untuk terhapus. Untuk menghindarinya maka diperlukan prosedur yang akan dikerjakan ketika terjadi kesalahan pada sistem. Salah satu prosedur tersebut yang berhubungan dengan konsep alokasi blok sistem berkas adalah pemeriksaan konsistensi.

Seperti yang telah dijelaskan sebelumnya, sebagian informasi direktori disimpan dalam cache untuk

mempercepat pengaksesan. Informasi dalam cache ini biasanya lebih aktual daripada yang terdapat dalam disk. Hal ini dikarenakan apabila terjadi perubahan dalam cache maka perubahan yang sama pada disk belum tentu langsung dilaksanakan.

Sekarang bayangkan hal yang dapat terjadi ketika terjadi crash pada sistem. Tabel berisi berkas-berkas yang dibuka hilang dan bersamanya terdapat perubahan yang dilakukan terhadap direktori tempat berkas tersebut berada. Hal ini bisa menyebabkan ketidak-konsistenan: status terbaru tentang berkas-berkas tidak tercatat pada direktori yang memuat berkas-berkas tersebut. Untuk memeriksa ketidak konsistenan ini ada sebuah program yang akan dijalankan ketika pelaksanaan boot ulang. Program tersebut adalah pemeriksa konsistensi (consistency checker). Program ini akan membandingkan data yang terdapat dalam struktur direktori dengan data blok disk, lalu apabila ditemukan kesalahan maka program tersebut akan mencoba memperbaikinya. Algoritma dalam pengalokasian blok dan manajemen ruang kosong menentukan jenis masalah yang bisa diselesaikan oleh progam ini. Sebagai contoh, misalkan metoda alokasi yang digunakan adalah alokasi dengan penghubung blok dan terdapat penghubung dari setiap blok ke blok selanjutnya maka keseluruhan struktur direktori bisa dibentuk kembali.

Gambar 42.15. Macam-macam lokasi *disk-caching*



Backup dan Restore

Dikarenakan disk magnetik kadang-kadang gagal, perawatan harus dijalankan untuk memastikan data tidak hilang selamanya. Oleh karena itu, program sistem dapat digunakan untuk *back up* data dari disk menuju ke media penyimpanan yang lainnya, seperti sebuah *floppy disk*, tape magnetik, atau disk optikal. *Recovery* dari kehilangan sebuah berkas individu, atau seluruh disk, mungkin menjadi masalah dari *restoring* data dari *backup*.

Untuk meminimalis kebutuhan untuk menduplikat, kita dapat menggunakan inforamsi dari, masing-masing masukan direktori. Sebagai contoh, jika program *backup* mengetahui kapan *backup* terakhir dari berkas telah selesai, dan tanggal terakhir berkas di direktori menunjukkan bahwa berkas tersebut tidak dirubah sejak tanggal tersebut, lalu berkas tersebut tidak perlu diduplikat lagi.

Sebuah tipe jadwal *backup* yaitu sebagai berikut:
- **Hari ke 1.** Menduplikat ke sebuah medium *back up* semua berkas ke disk. Ini disebut sebuah *full backup*.
- **Hari ke 2.** Menduplikat ke medium lainnya semua berkas yang dirubah sejak hari pertama. Ini adalah *incremental backup*.
- **Hari ke 3.** Menduplikat ke medium lainnya semua berkas yang dirubah sejak hari ke-2.
- **Hari ke N.** Menduplikat ke medium lainnya semua berkas yang dirubah sejak hari ke N-1.

Perputaran baru dapat mempunyai *backupnya* ditulis ke semua set sebelumnya, atau ke set yang baru dari media *backup*. N yang terbesar, tentu saja memerlukan tape atau disk yang lebih untuk dibaca untuk penyimpanan yang lengkap. Keuntungan tambahan dari perputaran *backup* ini adalah kita dapat menyimpan berkas apa saja yang tidak sengaja terhapus selama perputaran dengan

mengakses berkas yang terhapus dari *backup* hari sebelumnya.

42.7. *Log-Structured File System*

Algoritma *logging* sudah dilakukan dengan sukses untuk manangani masalah dari pemeriksaan rutin. Hasil dari implementasinya dikenal dengan *log-based transaction-oriented* (atau *journaling* sistem berkas.

Pemanggilan kembali yang mengenai struktur data sistem berkas pada disk--seperti struktur-struktur direktori, penunjuk-penunjuk blok-kosong, penunjuk-penunjuk FCB kosong--dapat menjadi tidak konsisten dikarenakan adanya *system crash*. Sebelum penggunaan dari teknik *log-based* di sisitem operasi, perubahan biasanya dipakaikan pada struktur ini. Perubahan-perubahan tersebut dapat diinterupsi oleh *crash*, dengan hasil strukturnya tidak konsisten.

Ada beberapa masalah dengan adanya pendekatan dari menunjuk struktur untuk memechkan dan memperbaikinya pada *recovery*. Salah satunya adalah ketidakkonsistenan tidak dapat diperbaiki. Pemeriksaan rutin mungkin tidak dapat untuk *recover* struktur tersebut, yang hasilnya kehilangan berkas dan mungkin seluruh direktori.

Solusinya adalah memakai teknik *log-based-recovery* pada sistem berkas metadata yang terbaru. Pada dasarnya, semua perubahan metadata ditulis secara berurutan di sebuah *log*. Masing-masing set dari operasi-operasi yang manampilakan tugas yang spesifik adalah sebuah *transaction*. Jika sistemnya *crashes*, tidak akan ada atau ada kelebihan *transactions* di berkas *log*. *Transactions* tersebut tidak akan pernah lengkap ke sistem berkas walaupun dimasukkan oleh sistem operasi, jadi harus dilengkapi. Keuntungan yang lain adalah proses-proses pembaharuan akan lebih cepat daripada saat dipakai langsung ke struktur data pada disk.

42.8. Sistem Berkas Linux Virtual

Obyek dasar dalam layer-layer virtual file system

1. **File.** File adalah sesuatu yang dapat dibaca dan ditulis. File ditempatkan pada memori. Penempatan pada memori tersebut sesuai dengan konsep file deskriptor yang dimiliki unix.
2. **Inode.** Inode merepresentasikan obyek dasar dalam file sistem. Inode bisa saja file biasa, direktori, simbolik link dan lain sebagainya. Virtual file sistem tidak memiliki perbedaan yang jelas di antara obyek, tetapi mengacu kepada implementasi file sistem yang menyediakan perilaku yang sesuai. Kernel tingkat tinggi menangani obyek yang berbeda secara tidak sama. File dan inode hampir mirip diantara keduanya. Tetapi terdapat perbedaan yang penting diantara keduanya. Ada sesuatu yang memiliki inode tetapi tidak memiliki file, contohnya adalah simbolik link. Ada juga file yang tidak memiliki inode
3. **File sistem.** File system adalah kumpulan dari inode-inode dengan satu inode pembeda yaitu root. Inode lainnya diakses mulai dari root inode dan pencarian nama file untuk menuju ke inode lainnya. File sistem mempunyai beberapa karakteristik yang mencakup seluruh inode dalam file sistem. Salah satu yang terpenting adalah blocksize.
4. **Nama inode.** Semua inode dalam file sistem diakses melalui namanya. Walaupun pencarian nama inode bisa menjadi terlalu berat untuk beberapa sistem, virtual file sistem pada linux tetap memantau cache dan nama inode yang baru saja terpakai agar kinerja meningkat. Cache terdapat di memori sebagai tree, ini berarti jika sembarang inode dari file terdapat di dalam cache, maka parent dari inode tersebut juga terdapat di dalam cache. Virtual file system layer menangani semua pengaturan nama path dari file dan mengubahnya menjadi masukan di dalam cache sebelum mengizinkan file sistem untuk mengaksesnya. Ada pengecualian pada target dari simbolik link, akan diakses file sistem secara langsung. File sistem diharapkan untuk menginterpretasikannya.

42.9. Operasi-operasi Dalam Inode

Linux menyimpan cache dari inode aktif maupun dari inode yang telah terakses sebelumnya. Ada dua path dimana inode ini dapat diakses. Yang pertama telah disebutkan sebelumnya, setiap entri dalam cache menunjuk pada suatu inode dan menjaga inode tetap dalam cache. Yang kedua melalui inode hash table. Setiap inode mempunyai alamat 8 bit sesuai dengan alamat dari file sistem superblok dan nomor inode. Inode dengan nilai hash yang sama kemudian dirangkai di doubly

linked list. Perubahan pada cache melibatkan penambahan dan penghapusan entri-entri dari cache itu sendiri. Entri-entri yang tidak dibutuhkan lagi akan di unhash sehingga tidak akan tampak dalam pencarian berikutnya. Operasi diperkirakan akan mengubah struktur cache harus dikunci selama melakukan perubahan. Unhash tidak memerlukan semaphore karena ini bisa dilakukan secara atomik dalam kernel lock. Banyak operasi file memerlukan dua langkah proses. Yang pertama adalah melakukan pencarian nama di dalam direktori. Langkah kedua adalah melakukan operasi pada file yang telah ditemukan. Untuk menjamin tidak terdapatnya proses yang tidak kompatibel diantara kedua proses itu, setelah proses kedua, virtual file sistem protokol harus memeriksa bahwa parent entry tetap menjadi parent dari entri child-nya. Yang menarik dari cache locking adalah proses rename, karena mengubah dua entri dalam sekali operasi.

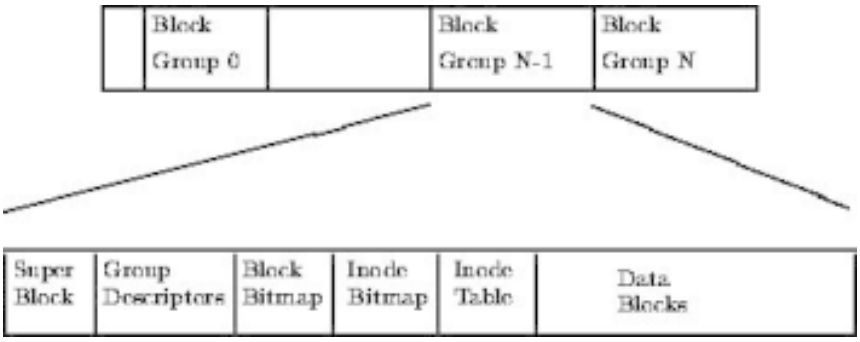
42.10. Sistem Berkas Linux

Sistem Berkas EXT2

EXT2 adalah file sistem yang ampuh di linux. EXT2 juga merupakan salah satu file sistem yang paling ampuh dan menjadi dasar dari segala distribusi linux. Pada EXT2 file sistem, file data disimpan sebagai data blok. Data blok ini mempunyai panjang yang sama dan meski pun panjangnya bervariasi diantara EXT2 file sistem, besar blok tersebut ditentukan pada saat file sistem dibuat dengan perintah mk2fs. Jika besar blok adalah 1024 bytes, maka file dengan besar 1025 bytes akan memakai 2 blok. Ini berarti kita membuang setengah blok per file. EXT2 mendefinisikan topologi file sistem dengan memberikan arti bahwa setiap file pada sistem diasosiasiakan dengan struktur data inode. Sebuah inode menunjukkan blok mana dalam suatu file tentang hak akses setiap file, waktu modifikasi file, dan tipe file. Setiap file dalam EXT2 file sistem terdiri dari inode tunggal dan setiap inode mempunyai nomor identifikasi yang unik. Inode-inode file sistem disimpan dalam tabel inode. Direktori dalam EXT2 file sistem adalah file khusus yang mengandung pointer ke inode masing-masing isi direktori tersebut.

Inode adalah kerangka dasar yang membangun EXT2. Inode dari setiap kumpulan blok disimpan dalam tabel inode bersama dengan peta bit yang menyebabkan sistem dapat mengetahui inode mana yang telah teralokasi dana inode mana yang belum. MODE: mengandung dia informasi, inode apa dan izin akses yang dimiliki user. OWNER INFO: user atau grop yang memiliki file atau direktori SIZE: besar file dalam bytes TIMESTAMPS: kapan waktu pembuatan inode dan waktu terakhir dimodifikasi. DATABLOKS: pointer ke blok yang mengandung data. EXT2 inode juga dapat menunjuk pada device khusus, yang mana device khusus ini bukan merupakan file, tatapi dapat menangani program sehingga program dapat mengakses ke device. Semua file device di dalam drektori /dev dapat membantu program mengakses device.

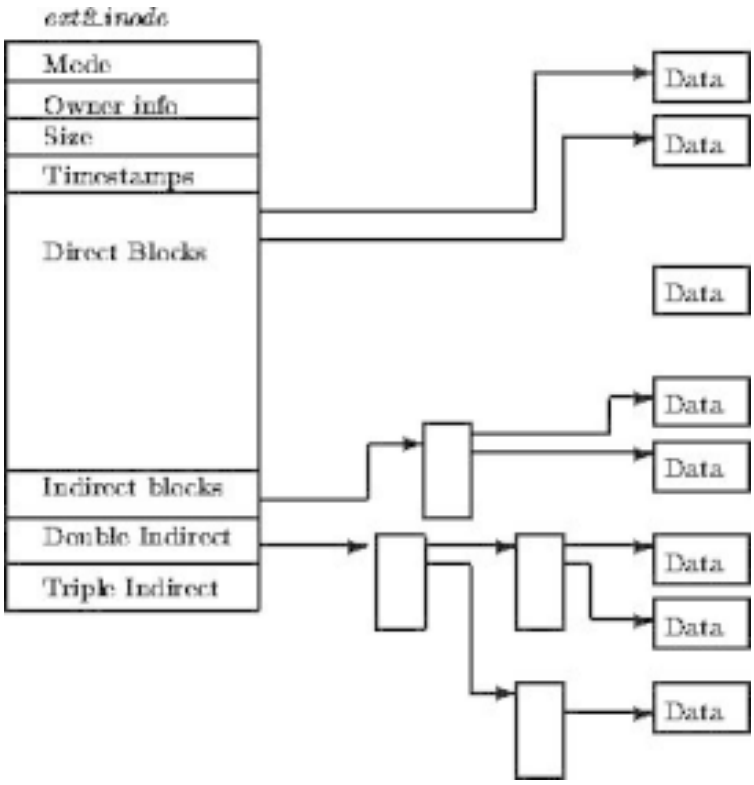
Gambar 42.16. Struktur Sistem Berkas EXT2.



Sistem Berkas EXT3

EXT3 adalah peningkatan dari EXT2 file sistem. Peningkatan ini memiliki beberapa keuntungan, diantaranya:

Gambar 42.17. Inode Sistem Berkas EXT2.



Setelah kegagalan sumber daya, "unclean shutdown", atau kerusakan sistem, EXT2 file sistem harus melalui proses pengecekan dengan program e2fsck. Proses ini dapat membuang waktu sehingga proses booting menjadi sangat lama, khususnya untuk disk besar yang mengandung banyak sekali data. Dalam proses ini, semua data tidak dapat diakses. Jurnal yang disediakan oleh EXT3 menyebabkan tidak perlu lagi dilakukan pengecekan data setelah kegagalan sistem. EXT3 hanya dicek bila ada kerusakan hardware seperti kerusakan hard disk, tetapi kejadian ini sangat jarang. Waktu yang diperlukan EXT3 file sistem setelah terjadi "unclean shutdown" tidak tergantung dari ukuran file sistem atau banyaknya file, tetapi tergantung dari besarnya jurnal yang digunakan untuk menjaga konsistensi. Besar jurnal default memerlukan waktu kira-kira sedetik untuk pulih, tergantung kecepatan hardware.

Integritas data. EXT3 menjamin adanya integritas data setelah terjadi kerusakan atau "unclean shutdown". EXT3 memungkinkan kita memilih jenis dan tipe proteksi dari data.

Kecepatan. Daripada menulis data lebih dari sekali, EXT3 mempunyai throughput yang lebih besar daripada EXT2 karena EXT3 memaksimalkan pergerakan head hard disk. Kita bisa memilih tiga jurnal mode untuk memaksimalkan kecepatan, tetapi integritas data tidak terjamin.

Mudah dilakukan migrasi. Kita dapat berpindah dari EXT2 ke sistem EXT3 tanpa melakukan format ulang.

Sistem Berkas Reiser

Reiser file sistem memiliki jurnal yang cepat. Ciri-cirinya mirip EXT3 file sistem. Reiser file sistem dibuat berdasarkan balance tree yang cepat. Balance tree unggul dalam hal kinerja, dengan algoritma yang lebih rumit tentunya. Reiser file sistem lebih efisien dalam pemenfaatan ruang disk. Jika kita menulis file 100 bytes, hanya ditempatkan dalam satu blok. File sistem lain menempatkannya dalam 100 blok. Reiser file sistem tidak memiliki pengalokasian yang tetap untuk inode. Resier file sistem dapat menghemat disk sampai dengan 6 persen.

Sistem Berkas X

X file sistem juga merupakan jurnaling file sistem. X file sistem dibuat oleh SGI dan digunakan di sistem operasi SGI IRIX. X file sistem juga tersedia untuk linux dibawah lisensi GPL. X file sistem mengunakan B-tree untuk menangani file yang sangat banyak. X file sistem digunakan pada server-server besar.

Sistem Berkas Proc

Sistem Berkas Proc (Proc File Sistem) menunjukkan bagaimana hebatnya virtual file sistem yang ada pada linux. Proc file sistem sebenarnya tidak ada secara fisik, baik subdirektorinya, maupun file-file yang ada di dalamnya. Proc file sistem diregister oleh linux virtual file sistem, jika virtual file sistem memanggilnya dan meminta inode-inode dan file-file, proc file sistem membuat file tersebut dengan informasi yang ada di dalam kernel. Contohnya, /proc/devices milik kernel dibuat dari data struktur kernel yang menjelaskan device tersebut.

Sistem Berkas Web

Sistem Berkas Web (WFS) adalah sistem berkas Linux baru yang menyediakan sebuah sistem berkas seperti antarmuka untuk World Wide Web. Sistem ini dibangun sebagai modul kernel untuk Linux Kernel 2.2.1, dan meng-utilisasi proses level user (web daemon) untuk melayani permintaan pengambilan dokumen HTTP. Sistem berkas ini menyediakan fasilitas caching untuk dokumen remote, dan dapat memproses permintaan-permintaan terkenal multiple secara konkuren. Ketika suatu dokumen remote diambil, isi yang berada dalam hyperlink dalam dokumen tersebut diekstrak dan dipetakan kedalam sistem berkas local. Informasi ini direktori remote ini dibuat untuk setiap direktori yang diatur oleh WFS dalam sebuah file khusus. Utility lsw digunakan untuk daftar dan mengatur ini direktori remote. Partisi yang diatur WFS bersifat read-only bagi client WFS. Namun client dapat menyegarkan entri dari partisi WFS dengan menggunakan utility khusus rwm. Suatu studi unjuk kerja memperlihatkan bahwa WFS lebih kurang 30% lebih lambat daripada AFS untuk penelusuran akses berkas yang berisi 100% cache miss. Unjuk kerja yang lebih rendah ini kemungkinan karena antara lain jumlah proses yang lebih besar dilakukan dalam proses user dalam WFS, dan karena penggunaan general HTTP library untuk pengambilan dokumen.

Sistem Berkas Transparent Cryptographic (TCFS)

TCFS adalah sebuah sistem berkas terdistribusi. Sistem ini mengizinkan akses berkas sensitif yang disimpan dalam sebuah server remote dengan cara yang aman. Sistem ini mengatasi eavesdropping dan data tampering baik pada server maupun pada jaringan dengan menggunakan enkripsi dan message digest. Aplikasi mengakses data pada sistem berkas TCFS ini menggunakan system call regular untuk mendapatkan transparency yang lengkap bagi pengguna.

Sistem Berkas Steganographic

Sistem Berkas Cryptographic menyediakan sedikit perlindungan terhadap instrumen-instrumen legal atau pun ilegal yang memaksa pemilik data untuk melepaskan kunci deskripsinya demi data yang disimpan saat hadirnya data terenkripsi dalam sebuah komputer yang terinfeksi. Sistem Berkas Cryptographic dapat diperluas untuk menyediakan perlindungan tambahan untuk scenario di atas dan telah diperluas sistem berkas Linux (ext2fs) dengan sebuah fungsi enkripsi yang plausible-deniability. Walaupun nyata bahwa komputer kita mempunyai software enkripsi hardisk yang sudah terinstal dan mungkin berisi beberapa data terenkripsi, sebiah inspector tetap akan dapat untuk menentukan apakah kita memberikan kunci akses kepada semua level keamanan atau terbatas. Implementasi ini disebut sistem berkas Steganographic.

42.11. Pembagian Sistem Berkas Ortogonal

Shareable dan Unshareable

1. **Shareable.** Isinya dapat dishare (digunakan bersama) dengan sistem lain, gunanya untuk

menghemat tempat.
2. **Unshareable.** Isinya tidak dapat dishare (digunakan bersama) dengan sistem lain, biasanya untuk alasan keamanan.

Variabel dan Statik

1. **Variabel.** Isinya sering berubah-ubah.
2. **Statik.** Sekali dibuat, kecil kemungkinan isinya akan berubah. Bisa berubah jika ada campur tangan sistem admin.

42.12. Rangkuman

Informasi yang disimpan di dalam suatu berkas harus disimpan ke dalam disk. Artinya, sistem operasi harus memutuskan tempat informasi itu akan disimpan. Ada 3 method untuk menentukan bagaimana sistem operasi menyimpan informasi ke disk yakni manajemen ruang kosong (mengetahui seberapa luang kapasitas disk yang tersedia), efisiensi dan kinerja, dan *recovery*.

Salah satu tujuan OS adalah menyembunyikan kerumitan device hardware dari sistem penggunanya. Contohnya, Sistem Berkas Virtual menyamakan tampilan sistem berkas yang dimount tanpa memperdulikan devices fisik yang berada di bawahnya. Bab ini akan menjelaskan bagaimana kernel Linux mengatur device fisik di sistem.

Salah satu fitur yang mendasar adalah kernel mengabstraksi penanganan device. Semua device hardware terlihat seperti berkas pada umumnya: mereka dapat dibuka, ditutup, dibaca, dan ditulis menggunakan calls sistem yang sama dan standar untuk memanipulasi berkas. Setiap device di sistem direpresentasikan oleh sebuah file khusus device, contohnya disk IDE yang pertama di sistem direpresentasikan dengan /dev/hda. Devices blok (disk) dan karakter dibuat dengan perintah mknod dan untuk menjelaskan device tersebut digunakan nomor devices besar dan kecil. Devices jaringan juga direpresentasikan dengan berkas khusus device, tapi berkas ini dibuat oleh Linux setelah Linux menemukan dan menginisialisasi pengontrol-pengontrol jaringan di sistem. Semua device yang dikontrol oleh driver device yang sama memiliki nomor device besar yang umum. Nomor devices kecil digunakan untuk membedakan antara device-device yang berbeda dan pengontrol-pengontrol mereka, contohnya setiap partisi di disk IDE utama punya sebuah nomor device kecil yang berbeda. Jadi, /dev/hda2, yang merupakan partisi kedua dari disk IDE utama, punya nomor besar 3 dan nomor kecil yaitu 2. Linux memetakan berkas khusus device yang diteruskan ke system call (katakanlah melakukan mount ke sistem berkas device blok) pada driver si device dengan menggunakan nomor device besar dan sejumlah tabel sistem, contohnya tabel device karakter, chrdevs.

Rujukan

[HenPat2002] John L Hennessy dan David A Patterson. 2002. *Computer Architecture*. A Quantitative Approach. Third Edition. Morgan Kaufman. San Francisco.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

[WEBCarter2004] John Carter. 2004. *CS 5460 Operating Systems – Lecture 19: File System Operations and Optimizations – http://www.cs.utah.edu/ classes/ cs5460/ lectures/ lecture19.pdf* . Diakses 29 Mei 2006.

[WEBGooch1999] Richard Gooch. 1999. *Overview of the Virtual File System – http://www.atnf.csiro.au/ people/ rgooch/ linux/ docs/ vfs.txt* . Diakses 29 Mei 2006.

[WEBIBM2003] IBM Corporation. 2003. *System Management Concepts: Operating System and Devices – http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690 / IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm* . Diakses 29 Mei 2006.

[WEBJonesSmith2000] David Jones dan Stephen Smith. 2000. *85349 – Operating Systems – Study Guide – http://www.infocom.cqu.edu.au/ Courses/ aut2001/ 85349/ Resources/ Study_Guide/ 85349.pdf* . Diakses 20 Juli 2006.

[WEBRustling1997] David A Rusling. 1997. *The Linux Kernel – The EXT2 Inode – http://www.science.unitn.it/ ~fiorella/ guidelinux/ tlk/ node96.html* . Diakses 1 Agustus 2006.

Bagian VII. Masukan/Keluaran (M/K)

Sistem Masukan/Keluaran (M/K) merupakan bagian penting yang menentukan kinerja sistem secara keseluruhan. Pada bagian ini akan diperkenalkan perangkat-perangkat M/K, konsep *polling*, interupsi, *Direct Memory Access* (DMA), subsistem kernel, aplikasi antarmuka, *streams*, penjadwalan, RAID, kinerja, M/K sistem Linux, serta sistem penyimpanan tersier.

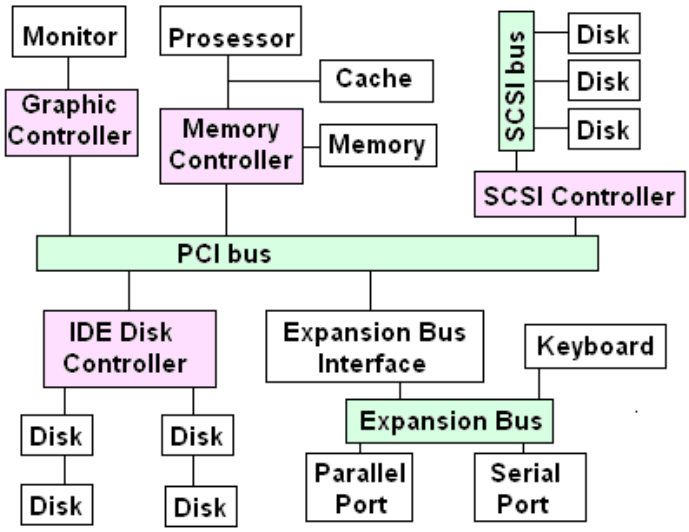
Bab 43. Perangkat Keras M/K

43.1. Pendahuluan

Salah satu fungsi utama sistem operasi adalah mengatur operasi Masukan/Keluaran (M/K) beserta perangkatnya. Sistem operasi harus dapat memberikan perintah ke perangkat-perangkat tersebut, menangkap interupsi, dan menangani *error*/kesalahan yang terjadi. Selain itu, sistem operasi juga menyediakan fasilitas antarmuka (*interface*) antara perangkat-perangkat tersebut dengan keseluruhan sistem yang ada.

Kita akan membahas bagaimana sistem operasi menangani hal-hal yang berkenaan dengan M/K. Pada bab ini, pembahasan yang ada dilihat dari sisi perangkat keras kemudian pada bab selanjutnya dilihat dari sisi perangkat lunak secara umum. Pembahasan hal-hal yang berkenaan dengan M/K pada bab ini kita batasi pada bagaimana perangkat keras tersebut diprogram, bukan cara kerja hal-hal yang ada di dalamnya. Dengan kata lain, kita tidak akan menyinggung komponen-komponen elektronik seperti: resistor, kapasitor, dan lain-lain.

Gambar 43.1. Abstraksi



Saat ini, terdapat berbagai macam perangkat M/K, seperti perangkat penyimpanan (disk, tape), perangkat transmisi (network card, modem), dan perangkat antarmuka dengan pengguna (screen, mouse, keyboard). Secara umum, perangkat M/K dapat dibagi menjadi dua kategori, yaitu:

1. **Perangkat blok.** Perangkat blok adalah perangkat yang menyimpan informasi dalam bentuk blok-blok berukuran tertentu dan setiap blok memiliki alamat masing-masing. Umumnya ukuran blok adalah 512 byte sampai 32.768 byte. Hal penting dari perangkat blok adalah memungkinkan membaca atau menulis setiap blok secara independen. Contoh perangkat blok yaitu disk. Bila kita ingin membuka suatu berkas lagu dalam sebuah direktori di disk, berkas bila langsung kita akses. Berbeda dengan kaset yang harus kita putar terlebih dahulu.
2. **Perangkat karakter.** Perangkat karakter adalah perangkat yang mengirim atau menerima sebarisan karakter, tanpa menghiraukan struktur blok. Printer, *network interface*, dan perangkat yang bukan disk termasuk di dalamnya. Namun, pembagian ini tidak sepenuhnya benar. Pada kenyataannya, terdapat perangkat yang tidak memenuhi salah satu kriteria, yaitu clock. Clock merupakan perangkat yang tidak memiliki blok beralamat, tidak mengirim dan menerima barisan karakter, melainkan perangkat yang hanya menimbulkan interupsi dalam jangka waktu tertentu.

Berbicara tentang M/K, terdapat banyak istilah di dalamnya. Beberapa istilah yang cukup dikenal yaitu port, dan bus. Sebuah perangkat berkomunikasi dengan sistem di komputer dengan cara pengiriman sinyal melalui kabel atau udara. Perangkat tersebut berhubungan dengan komputer

melalui suatu titik yang dinamakan port. Jika satu atau lebih perangkat menggunakan serangkaian kabel atau penghubung yang sama, penghubung itu disebut bus. Jadi, bus adalah serangkaian penghubung yang berfungsi sebagai perantara saat pengiriman pesan/data berlangsung. Selain hal di atas, ada pula bentuk hubungan dimana sebuah perangkat (sebut saja perangkat A) mempunyai kabel yang terhubung ke perangkat B, lalu kabel di B terhubung ke perangkat C, dan perangkat C terhubung ke sebuah port di komputer, pengaturan ini disebut daisy chain. Daisy chain juga berfungsi sebagai sebuah bus.

Port M/K terdiri dari empat register, yaitu:
1. **Status.** Register ini berisi bit-bit yang menandakan status apakah perintah M/K sempurna dilaksanakan perangkat, ada bit di register data-in yang tersedia untuk dibaca, ataupun apakah ada perangkat M/K yang *error*. Semua bit tersebut dibaca oleh CPU.
2. **Control.** Merupakan register yang ditulis oleh CPU untuk memulai perintah atau untuk mengganti modus perangkat. Sebagai gambaran modus perangkat, terdapat bit dalam register control di sebuah serial port yang berfungsi memilih kecepatan transfer yang didukung serial port tersebut.
3. **Data-in.** Register ini merupakan register yang dibaca CPU sebagai input data.
4. **Data-out.** Bit di dalamnya merupakan bit yang ditulis CPU sebagai output data.

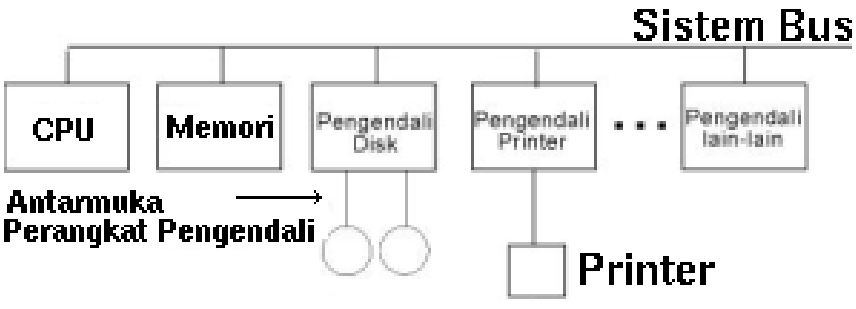
43.2. Komponen M/K

Unit M/K terdiri dari dua komponen, yaitu:
1. **Komponen Mekanis.** Komponen mekanis yakni perangkat M/K itu sendiri, seperti mouse, screen, keyboard, dan lain-lain.
2. **Komponen elektronis.** Komponen elektronis disebut pengendali perangkat M/K (device controller). Device controller hampir selalu berhubungan dengan sistem operasi dalam hal yang berkenaan dengan M/K. Dengan kata lain, dalam menangani operasi M/K, sistem operasi tidak berhubungan langsung dengan perangkat melainkan dengan pengendalinya. Beberapa pengendali perangkat dapat menangani dua, atau lebih perangkat M/K yang sejenis. Pada komputer desktop, komponen ini biasannya berupa kartu sirkuit yang dapat dimasukkan ke dalam slot pada motherboard.

Terdapat berbagai macam antarmuka antara perangkat dengan pengendalinya, antara lain ANSI, IEEE, atau ISO. Selain itu, ada pula IDE (*Integrated Drive Electronics*), dan SCSI (*Small Computer System Interface*). Kedua antarmuka terakhir merupakan antarmuka yang menjadi standar pabrik-pabrik pembuat perangkat M/K ataupun pembuat pengendalinya.

43.3. Penanganan M/K

Gambar 43.2. Model Bus Tunggal



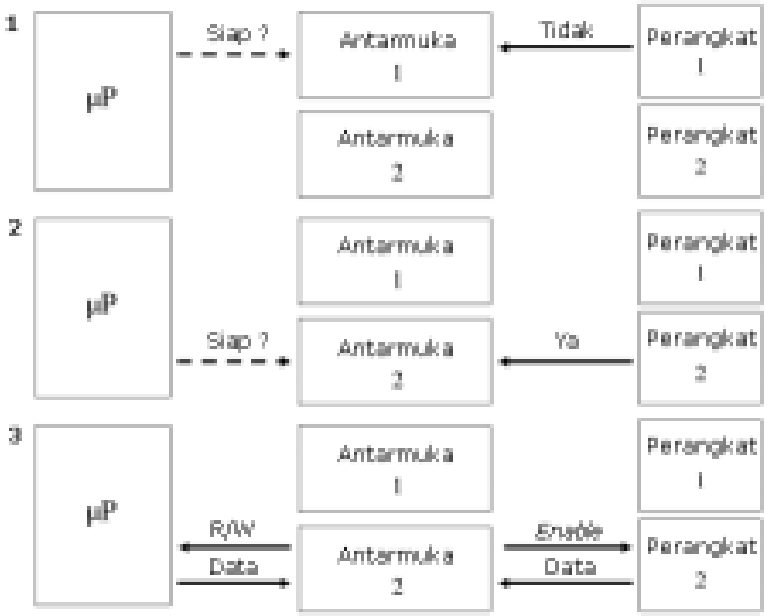
Dalam berkomunikasi dengan controller, terdapat dua cara sistem operasi memberikan perintah dan data, yaitu:
1. **Instruksi M/K.** Merupakan instruksi CPU yang khusus menangani transfer byte atau word ke sebuah port M/K. Cara kerjanya, instruksi tersebut memicu line bus untuk memilih perangkat yang dituju kemudian mentransfer bit-bit dari atau ke register perangkat.

2. **M/K Memory-mapped.** Register-register pengendali perangkat dipetakan ke ruang alamat prosesor. Operasi membaca ataupun menulis di alamat tersebut diinterpretasikan sebagai perintah untuk perangkat M/K. Sebagai contoh, sebuah operasi write digunakan untuk mengirim data ke perangkat M/K dimana data tersebut diartikan sebagai sebuah perintah. Saat CPU menempatkan alamat dan data tersebut di memori bus, memori sistem mengacuhkan operasi tersebut karena alamatnya mengindikasikan jatah ruang memori untuk M/K. Namun, pengendali perangkat melihat operasi tersebut, mengambil data, kemudian mentransmisi ke perangkat sebagai sebuah perintah. Memory-mapped I/O juga bisa digunakan untuk mentransmisikan data dengan cara CPU menulis atau membaca pada alamat tertentu. Perangkat M/K menggunakan alamat tersebut untuk membedakan tipe perintah dan data yang diberikan. Selain itu, terkadang alamat tersebut mempunyai arti sebagai indentitas perangkat dan tipe transmisi antara prosesor dan perangkat.

43.4. Polling

Busy-waiting/polling ialah ketika *host* mengalami *looping* yaitu membaca status *register* secara terus-menerus sampai status *busy* di-*clear*. Pada dasarnya *polling* dapat dikatakan efisien. Akan tetapi *polling* menjadi tidak efisien ketika setelah berulang-ulang melakukan *looping*, hanya menemukan sedikit perangkat yang siap untuk menservis, karena *CPU processing* yang tersisa belum selesai.

Gambar 43.3. Proses *Polling*



Sebagai gambaran polling, kita akan membahas satu contoh interaksi sederhana antara pengendali perangkat dengan CPU. Pada contoh berikut, CPU memberikan perintah kepada perangkat M/K melalui sebuah port (ingat kembali register-register dalam port). Sebagai keterangan tambahan, men-set bit artinya menulis angka 1, dan menghapus bit sebaliknya.

1. CPU terus-menerus membaca bit kerja sampai bit tersebut menjadi hapus. Bit kerja adalah bit dalam register status yang mengindikasikan keadaan pengendali perangkat.
2. CPU mengaktifkan bit write di register perintah sebagai awal pertanda CPU memberikan perintah dan menulis sebuah byte di data-out
3. CPU mengaktifkan command-ready bit, artinya perintah tersedia untuk dijalankan pengendali.
4. Controller melihat command-ready bit di-set sehingga bit kerja di-set.
5. Controller membaca register perintah dan melihat perintah write maka data-out dibaca dan menyuruh perangkat M/K melakukan apa yang diperintah CPU.
6. Controller meng-hapus command-ready bit, bit *error* di status, dan bit kerja.

Langkah pada nomor satu disebut polling atau busy-waiting. CPU terus-menerus memeriksa

keadaan bit untuk mengetahui status pengendali. Polling cocok digunakan bila kinerja perangkat dan pengendalinya cepat. Namun, saat ini kinerja CPU lebih cepat dibandingkan kinerja perangkat M/K dan pengendalinya. Akibatnya, terdapat rentang waktu antara CPU memberikan perintah pertama dengan perintah selanjutnya. Hal ini disebabkan CPU harus menunggu status pengendali selesai menjalankan perintah.

43.5. Interupsi

Mekanisme polling yang dijelaskan sebelumnya menjadi tidak efisien ketika prosesor terus-menerus memeriksa status perangkat M/K, padahal perangkat yang bersangkutan belum siap. Terdapat banyak situasi dimana tugas lain dapat dilakukan oleh prosesor saat menunggu perangkat M/K siap. Kita dapat mengatur agar perangkat M/K memperingatkan prosesor saat perangkat tersebut telah siap, hal ini dapat dilakukan dengan mengirim sinyal yang disebut interupsi ke prosesor.

Mekanisme Dasar Interupsi

Ketika CPU mendeteksi bahwa sebuah pengendali telah mengirimkan sebuah sinyal ke *interrupt request line* (membangkitkan sebuah interupsi), CPU kemudian menjawab interupsi tersebut (juga disebut menangkap interupsi) dengan menyimpan beberapa informasi mengenai keadaan terkini CPU – contohnya nilai instruksi pointer, dan memanggil *interrupt handler* agar *handler* tersebut dapat melayani pengendali atau alat yang mengirim interupsi tersebut.

Langkah-langkah mekanisme interupsi yang disebabkan perangkat M/K, yaitu:
1. Perangkat M/K mengirim sinyal interupsi.
2. Prosesor menerima sinyal interupsi.
3. Penyimpanan informasi proses yang sedang dieksekusi.
4. Prosesor mengidentifikasi penyebab interupsi.
5. Prosesor mengeksekusi interupsi routine sampai return.
6. Prosesor melanjutkan proses yang sebelumnya ditunda.

Di bawah ini adalah penjelasan langkah-langkah mekanisme interupsi yang disebutkan di atas.
1. ada tiga kemungkinan penyebab perangkat M/K mengirimkan interupsi, yaitu:
   a. **Input Ready.** Umpamanya ketika buffer keyboard sudah terisi (terjadi pengetikan), kendali keyboard akan mengirim interupsi untuk memberitahu prosesor bahwa input dari keyboard sudah tersedia.
   b. **Output Complete.** Umpama ketika mencetak pada printer. Misalnya printer hanya menerima satu baris teks pada satu waktu. Karenanya prosesor harus mengirim satu baris teks, menunggu baris tersebut dicetak, kemudian mengirim baris teks berikutnya. Saat menunggu satu baris dicetak, prosesor dapat mengerjakan proses lain. Setelah printer selesai mencetak satu baris, kendali printer mengirim interupsi untuk memberi tahu prosesor bahwa output (satu baris teks) sudah complete (selesai dicetak) sehingga printer siap menerima satu baris teks berikutnya.
   c. ***Error*.** Jika terjadi *error* pada perangkat M/K saat perangkat tersebut sedang melakukan operasi M/K, kontroler perangkat tersebut akan mengirim interupsi untuk memberitahu prosesor bahwa terjadi *error* sehingga operasi M/K tidak bisa dilanjutkan.

   Perangkat M/K mengirim interupsi melalui bus kontrol prosesor yang disebut interupsi-request line.
2. Setiap kali selesai mengeksekusi sebuah instruksi, prosesor akan memeriksa interupsi-request line untuk mendeteksi terjadinya interupsi. Pemeriksaan ini tidak sama dengan mekanisme polling yang dijelaskan sebelumnya (meskipun secara konsep mirip). Pada mekanisme polling, prosesor akan mengeksekusi beberapa baris instruksi dalam loop, dimana instruksi-instruksi dalam loop tersebut berfungsi untuk memeriksa status perangkat M/K. Dalam pemeriksaan terjadinya interupsi, prosesor tidak mengeksekusi instruksi-instruksi seperti pada polling, melainkan hanya semacam validasi bit pada interupsi-request line.
3. Tujuan dari penyimpanan informasi proses adalah agar data dari proses yang akan ditunda ini tidak terganggu oleh eksekusi interupsi routine. Contoh informasi mengenai proses diantaranya adalah: status proses, program counter, isi register prosesor, informasi penjadwalan prosesor, informasi manajemen memori, dll. Proses yang akan ditunda ini akan diubah statusnya dari running ke ready.
4. Untuk mengidentifikasi penyebab interupsi, memori me-load rutin penanganan interupsi (jika belum ada di memori). Kemudian prosesor mengeksekusi routine tersebut. Interupsi handler akan

memeriksa satu-persatu perangkat M/K untuk mengetahui perangkat mana yang mengirim interupsi.
5. Setelah perangkat M/K yang mengirim interupsi ditemukan, prosesor mengeksekusi interupsi routine yang sesuai dengan perangkat itu. interupsi routine akan dieksekusi sampai return.
6. Setelah interupsi routine selesai dieksekusi, prosesor akan kembali mengeksekusi proses yang sebelumnya ditunda.

Dalam mekanisme interupsi terdapat beberapa masalah yang perlu ditangani, yaitu:
1. Interupsi terjadi saat prosesor sedang mengeksekusi critical section dari proses. Untuk mengatasi hal ini prosesor menyediakan dua interupsi-request line, yaitu maskable dan non-maskable. Interupsi yang terjadi melalui interupsi-request line yang maskable akan diabaikan sementara oleh prosesor. Jika eksekusi critical section sudah selesai, prosesor akan melayani interupsi tersebut. Untuk interupsi yang terjadi melalui interupsi-request line yang non-maskable, prosesor akan langsung menghentikan proses yang sedang dieksekusinya kemudian menangani interupsi tersebut.
2. Waktu yang dibutuhkan untuk mencari penyebab interupsi merupakan hal yang tidak efisien, karena prosesor akan memeriksa satu-persatu perangkat M/K yang terhubung ke komputer. Untuk mengatasi hal ini, kita dapat mengatur agar perangkat yang mengirim interupsi juga mengirimkan alamat. Alamat tersebut adalah nomor yang digunakan untuk mencari interupsi routine di sebuah tabel dalam memori yang disebut interupsi vektor. Interupt vektor berisi pasangan-pasangan alamat yang dikirim perangkat M/K dan alamat interupsi routine di memori. Pada kenyataannya komputer saat ini memiliki banyak perangkat M/K sehingga akan membutuhkan tabel yang cukup besar untuk menyimpan alamat-alamat tersebut. Untuk mengatasi hal tersebut, terdapat sebuah teknik yang disebut interupsi chaining. Teknik ini mengatur penyusunan interupsi vektor. Cara penyusunannya yaitu, alamat yang dikirim oleh perangkat M/K merupakan satu elemen yang menunjuk ke head dari list interupsi routine. Ketika terjadi interupsi, interupsi handler yang berada dalam satu list (list dipilih berdasarkan alamat yang dikirim perangkat M/K) dipanggil satu-persatu sampai ditemukan interupsi routine yang sesuai. Interupsi chaining merupakan solusi yang mempertemukan antara kebutuhan penanganan interupsi secepatnya dengan penggunaan memori yang minimal untuk penyimpanan interupsi vektor.
3. Beberapa perangkat M/K membutuhkan pelayanan yang secepatnya dan dan harus didahulukan dari perangkat M/K yang lain(jika pada saat yang sama terdapat lebih dari satu perangkat M/K yang membutuhkan pelayanan oleh prosesor). Dalam situasi tersebut dibutuhkan prioritas untuk memilih perangkat yang akan dilayani lebih dulu. Setiap perangkat M/K memiliki prioritas sendiri. Pengaturan prioritas dan penanganan perangkat berdasarkan prioritasnya diatur oleh prosesor dan interupsi controler.
4. Jika Interupsi yang terjadi merupakan permintaan untuk melakukan transfer data yang besar (contohnya antara disk dan main memori), dalam hal ini penggunaan mekanisme interupsi adalah cara yang tidak efisien. Untuk mengatasi hal ini digunakan DMA yang akan dijelaskan setelah pembahasan interupsi.

Mekanisme interupsi tidak hanya digunakan untuk menangani operasi yang berhubungan dengan perangkat M/K. Sistem operasi menggunakan mekanisme interupsi untuk beberapa hal, diantaranya yaitu:

1. Menangani berbagai macam *exception*. *Exception* bisa disebabkan oleh hal-hal seperti pembagian dengan nol, pengaksesan memori yang tidak eksis, pengaksesan memori oleh pihak yang tidak berhak, dan pengeksekusian instruksi yang privileged melalui pengguna mode.

2. Mengatur virtual memori paging.

3. Menangani software interupsi.

4. Mengatur alur kontrol kernel.

Fitur Tambahan pada Komputer Modern

Pada arsitektur komputer modern, tiga fitur disediakan oleh CPU dan pengendali interupsi (pada perangkat keras) untuk dapat menangani interrupsi dengan lebih bagus. Fitur-fitur ini antara lain ialah kemampuan menghambat sebuah proses penanganan interupsi selama proses berada dalam

critical state, efisiensi penanganan interupsi sehingga tidak perlu dilakukan *polling* untuk mencari perangkat yang mengirimkan interupsi, dan fitur yang ketiga ialah adanya sebuah konsep interupsi multilevel sedemikian rupa sehingga terdapat prioritas dalam penanganan interupsi (diimplementasikan dengan *interrupt priority level system*).

Interrupt Request Line

Pada peranti keras CPU terdapat kabel yang disebut *interrupt request line*, kebanyakan CPU memiliki dua macam *interrupt request line*, yaitu *nonmaskable interrupt* dan *maskable interrupt*. *Maskable interrupt* dapat dimatikan/dihentikan oleh CPU sebelum pengeksekusian deretan *critical instruction* (*critical instruction sequence*) yang tidak boleh diinterupsi. Biasanya, interupsi jenis ini digunakan oleh pengendali perangkat untuk meminta pelayanan CPU.

Interrupt Vector dan *Interrupt Chaining*

Sebuah mekanisme interupsi akan menerima alamat *interrupt handling routine* yang spesifik dari sebuah set, pada kebanyakan arsitektur komputer yang ada sekarang ini, alamat ini biasanya berupa sekumpulan bilangan yang menyatakan offset pada sebuah tabel (biasa disebut vektor interupsi). Tabel ini menyimpan alamat-alamat *interrupt handler* spesifik di dalam memori. Keuntungan dari pemakaian vektor ialah untuk mengurangi kebutuhan akan sebuah interupsi handler yang harus mencari semua kemungkinan sumber interupsi untuk menemukan pengirim interupsi. Akan tetapi, vektor interupsi memiliki hambatan karena pada kenyataannya, komputer yang ada memiliki perangkat (dan *interrupt handler*) yang lebih banyak dibandingkan dengan jumlah alamat pada vektor interupsi. Karena itulah, digunakan teknik *interrupt chaining* setiap elemen dari vektor interupsi menunjuk pada elemen pertama dari sebuah daftar *interrupt handler*. Dengan teknik ini, *overhead* yang dihasilkan oleh besarnya ukuran tabel dan inefisiensi dari penggunaan sebuah *interrupt handler* (fitur pada CPU yang telah disebutkan sebelumnya) dapat dikurangi, sehingga keduanya menjadi kurang lebih seimbang.

Penyebab Interupsi

Interupsi dapat disebabkan berbagai hal, antara lain *exception*, *page fault*, interupsi yang dikirimkan oleh pengendali perangkat, dan *system call*. *Exception* ialah suatu kondisi dimana terjadi sesuatu, atau dari sebuah operasi didapat hasil tertentu yang dianggap khusus sehingga harus mendapat perhatian lebih, contohnya pembagian dengan 0 (nol), pengaksesan alamat memori yang *restricted* atau bahkan tidak valid, dan lain-lain. *System call* ialah sebuah fungsi pada aplikasi (perangkat lunak) yang dapat mengeksekusikan instruksi khusus berupa interupsi perangkat lunak atau *trap*.

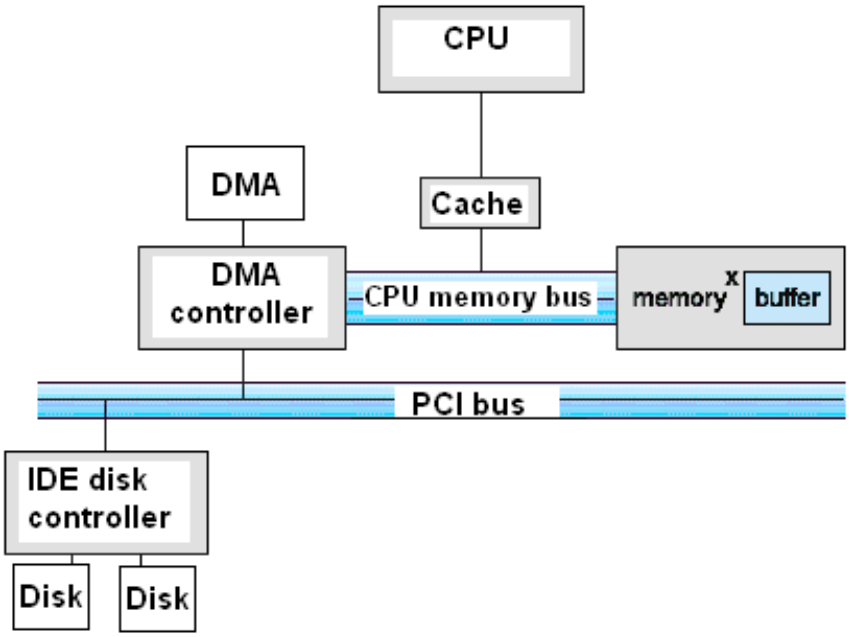
43.6. *Direct Memory Access* (DMA)

DMA ialah sebuah prosesor khusus (*special purpose processor*) yang berguna untuk menghindari pembebanan CPU utama oleh program M/K (PIO). Untuk memulai sebuah transfer DMA, *host* akan menuliskan sebuah DMA *command block* yang berisi *pointer* yang menunjuk ke sumber transfer, *pointer* yang menunjuk ke tujuan transfer, dan jumlah byte yang ditransfer, ke memori. CPU kemudian menuliskan alamat *command block* ini ke pengendali DMA, sehingga pengendali DMA dapat kemudian mengoperasikan bus memori secara langsung dengan menempatkan alamat-alamat pada bus tersebut untuk melakukan transfer tanpa bantuan CPU.

Seperti yang dijelaskan sebelumnya bahwa mekanisme interupsi tidak efisien untuk melakukan transfer data yang besar. Transfer data dilakukan per word. Pada mekanisme interupsi, untuk tiap word data yang ditransfer, prosesor tidak akan menunggu data tersedia pada perangkat yang mengirim data maupun data selesai ditulis oleh perangkat yang menerima data. Dalam situasi tersebut prosesor akan mengganti proses yang sedang dieksekusinya (yang melakukan transfer data) dengan proses lain (context switch). Jika ukuran data yang ditransfer cukup besar, prosesor akan berulang kali melakukan contex switch, padahal context switch akan menimbulkan overhead. Oleh karena itu dapat disimpulkan bahwa kelemahan mekanisme interupsi untuk menangani transfer data yang besar disebabkan oleh context switch. Untuk menangani kelemahan tersebut digunakan pendekatan alternatif. Suatu unit kontrol khusus disediakan untuk memungkinkan transfer data langsung antar perangkat eksternal dan memori utama, tanpa intervensi terus menerus dari prosesor. Pendekatan ini disebut direct memori access, atau DMA. DMA ialah sebuah prosesor khusus

(special purpose processor) yang berguna untuk menghindari pembebanan CPU utama oleh program M/K (PIO).

Gambar 43.4. DMA



Mekanisme DMA

1. Perangkat M/K mengirim interupsi ke CPU untuk memberitahu bahwa perangkat tersebut akan melakukan transfer data.
2. Untuk memulai sebuah transfer DMA, host akan menuliskan sebuah DMA command block yang berisi pointer yang menunjuk ke sumber transfer, pointer yang menunjuk ke tujuan transfer, dan jumlah byte yang ditransfer, ke memori. CPU kemudian menuliskan alamat command block ini ke pengendali DMA, sehingga pengendali DMA dapat kemudian mengoperasikan bus memori secara langsung dengan menempatkan alamat-alamat pada bus tersebut untuk melakukan transfer tanpa bantuan CPU. Setelah transfer data dapat dialihkan, prosesor berhenti mengeksekusi proses tersebut dan meload proses lain. Proses yang meminta transfer data tersebut diubah statusnya dari running ke blocked M/K.
3. DMA melakukan transfer data. Pada dasarnya, DMA mempunyai dua metoda yang berbeda dalam mentransfer data. Metoda yang pertama ialah metode yang sangat baku dan sederhana disebut HALT, atau Burst Mode DMA, karena pengendali DMA memegang kontrol dari sistem bus dan mentransfer semua blok data ke atau dari memori pada single burst. Selagi transfer masih dalam proses, sistem mikroprosessor di-set idle, tidak melakukan instruksi operasi untuk menjaga internal register. Tipe operasi DMA seperti ini ada pada kebanyakan komputer.

   Metoda yang kedua, mengikut-sertakan pengendali DMA untuk memegang kontrol dari sistem bus untuk jangka waktu yang lebih pendek pada periode dimana mikroprosessor sibuk dengan operasi internal dan tidak membutuhkan akses ke sistem bus. Metoda DMA ini disebut cycle stealing mode. Cycle stealing DMA lebih kompleks untuk diimplementasikan dibandingkan HALT DMA, karena pengendali DMA harus mempunyai kepintaran untuk merasakan waktu pada saat sistem bus terbuka.
4. Setelah transfer data selesai, DMA mengirim interupsi ke CPU. Proses yang meminta transfer data tadi, diubah lagi statusnya dari blocked ke ready, sehingga proses itu dapat kembali dipilih oleh penjadwal.

Tiga langkah dalam transfer DMA:
1. Prosesor menyiapkan DMA transfer dengan menyediakan data-data dari perangkat, operasi yang

akan ditampilkan, alamat memori yang menjadi sumber dan tujuan data, dan banyaknya byte yang ditransfer.
2. Pengendali DMA memulai operasi (menyiapkan bus, menyediakan alamat, menulis dan membaca data), sampai seluruh blok sudah di transfer.
3. Pengendali DMA meng-interupsi prosesor, dimana selanjutnya akan ditentukan tindakan berikutnya.

Pada dasarnya, DMA mempunyai dua metode yang berbeda dalam mentransfer data. Metode yang pertama ialah metode yang sangat baku dan sederhana disebut *HALT*, atau *Burst Mode DMA*, karena pengendali DMA memegang kontrol dari sistem bus dan mentransfer semua blok data ke atau dari memori pada *single burst*. Selagi transfer masih dalam prosres, sistem mikroprosessor di-set *idle*, tidak melakukan instruksi operasi untuk menjaga internal *register*. Tipe operasi DMA seperti ini ada pada kebanyakan komputer.

Metode yang kedua, mengikut-sertakan pengendali DMA untuk memegang kontrol dari sistem bus untuk jangka waktu yang lebih pendek pada periode dimana mikroprosessor sibuk dengan operasi internal dan tidak membutuhkan akses ke sistem bus. Metode DMA ini disebut *cycle stealing mode*. *Cycle stealing DMA* lebih kompleks untuk diimplementasikan dibandingkan HALT DMA, karena pengendali DMA harus mempunyai kepintaran untuk merasakan waktu pada saat sistem bus terbuka.

Handshaking

Proses *handshaking* antara pengendali DMA dan pengendali perangkat dilakukan melalui sepasang kabel yang disebut *DMA-request* dan *DMA-acknowledge*. Pengendali perangkat mengirimkan sinyal melalui *DMA-request* ketika akan mentransfer data sebanyak satu word. Hal ini kemudian akan mengakibatkan pengendali DMA memasukkan alamat-alamat yang dinginkan ke kabel alamat memori, dan mengirimkan sinyal melalui kabel *DMA-acknowledge*. Setelah sinyal melalui kabel *DMA-acknowledge* diterima, pengendali perangkat mengirimkan data yang dimaksud dan mematikan sinyal pada *DMA-request*.

Hal ini berlangsung berulang-ulang sehingga disebut *handshaking*. Pada saat pengendali DMA mengambil alih memori, CPU sementara tidak dapat mengakses memori (dihalangi), walaupun masih dapat mengaksees data pada cache primer dan sekunder. Hal ini disebut *cycle stealing*, yang walaupun memperlambat komputasi CPU, tidak menurunkan kinerja karena memindahkan pekerjaan data transfer ke pengendali DMA meningkatkan performa sistem secara keseluruhan.

Cara-cara Implementasi DMA

Dalam pelaksanaannya, beberapa komputer menggunakan memori fisik untuk proses DMA, sedangkan jenis komputer lain menggunakan alamat virtual dengan melalui tahap "penerjemahan" dari alamat memori virtual menjadi alamat memori fisik, hal ini disebut *Direct Virtual-Memory Address* atau DVMA. Keuntungan dari DVMA ialah dapat mendukung transfer antara dua memori *mapped device* tanpa intervensi CPU.

43.7. Rangkuman

Dalam penanganan M/K, sistem operasi hampir selalu berhubungan dengan pengendali perangkat, bukan dengan perangkatnya. Untuk berinteraksi dengan pengendali, terdapat tiga cara yang dilakukan CPU, yaitu: polling, interupsi, dan DMA. Polling baik digunakan bila kinerja pengendali dan perangkat M/K cepat. Namun, saat ini kineja CPU lebih cepat dibandingkan keduanya. Akibatnya, polling menyebabkan pemborosan CPU clock cycle. Hal tersebut dapat diatasi interupsi, tetapi interupsi tidak efisien untuk transfer data yang besar. Kelemahan interupsi tersebut dapat diatasi DMA. DMA dapat mengatasi transfer data yang besar dengan cara yang lebih efisien.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating*

Systems. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

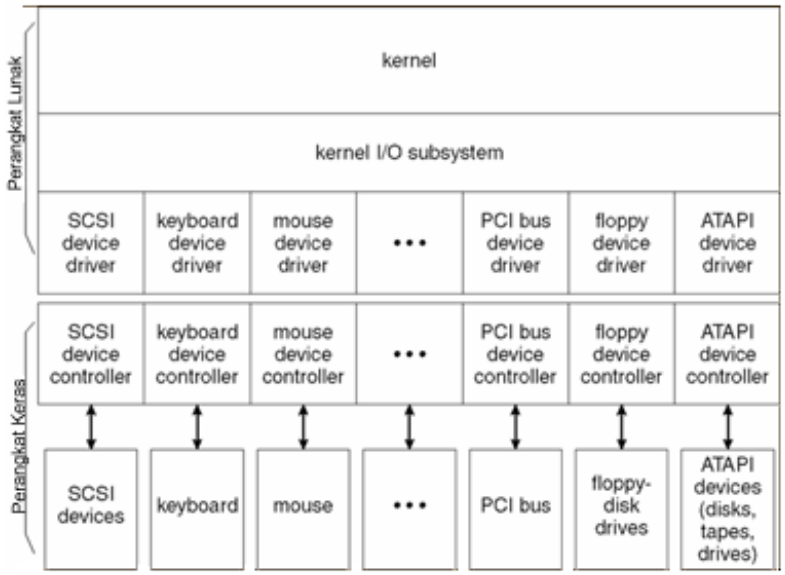
[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

Bab 44. Subsistem M/K Kernel

44.1. Pendahuluan

Pada sistem operasi modern 32-bit seperti Windows dan Linux, aplikasi tidak dapat mengakses M/K ke perangkat keras secara langsung, karena Windows dan Linux menggunakan operasi protected mode yang dimiliki oleh prosesor. Jika begitu bagaimana sebuah program dapat berkomunikasi dengan perangkat keras? Jawabannya adalah sistem operasi mengkoordinasikan semua operasi M/K ke perangkat keras melalui penggunaan *device driver*. *Device Driver* adalah sejenis program khusus yang berfungsi sebagai jembatan antara perangkat keras M/K dengan sistem operasi. Ini berarti aplikasi dapat berinteraksi dengan berbagai perangkat tanpa mengharuskan pemrogram aplikasi tahu banyak tentang mekanisme akses M/K ke perangkat, semuanya sudah dikerjakan oleh sistem operasi melalui penggunaan *device driver*.

Gambar 44.1. Struktur Kernel



Device driver harus mematuhi beberapa protokol khusus dan harus memanggil beberapa *system call* ke sistem operasi yang tidak bisa diakses oleh aplikasi standar. Biasanya *device driver* yang baru di-install pada sistem operasi ketika ada sejenis perangkat keras baru yang akan dipasang pada komputer. Namun biasanya untuk perangkat yang standar seperti mouse atau keyboard, *device driver*-nya sudah tersedia pada sistem operasi modern. Salah satu kelebihan dari mekanisme device driver ini adalah sistem operasi atau vendor dari perangkat keras harus menyediakan sekumpulan *device driver* untuk tiap jenis perangkat M/K atau sistem itu tidak akan pernah populer, hal ini yang mengakibatkan OS/2 dari IBM tidak pernah sukses karena kurangnya keberadaan dari *device driver* yang terdapat pada sistem operasi tersebut.

44.2. Aplikasi Antarmuka M/K

Bagian ini akan membahas bagaimana teknik dan struktur antarmuka yang memungkinkan M/K diperlakukan secara seragam. Salah satu contohnya adalah ketika suatu aplikasi ingin membuka data yang ada dalam suatu disk tanpa mengetahui jenis disk apa yang akan diaksesnya. Untuk mempermudah pengaksesan, sistem operasi melakukan standarisasi pengaksesan pada perangkat M/K. Pendekatan inilah yang dinamakan **aplikasi antarmuka M/K**.

Seperti layaknya permasalahan dari *software-engineering* yang rumit lainnya, aplikasi antarmuka M/K melibatkan abstraksi, enkapsulasi, dan *software layering*. Abstraksi dilakukan dengan

membagi-bagi detail perangkat-perangkat M/K ke dalam kelas-kelas yang lebih umum. Dengan adanya kelas-kelas yang umum ini, maka akan lebih mudah bagi fungsi-fungsi standar (antarmuka) untuk mengaksesnya. Selanjutnya, keberadaan *device driver* pada masing-masing peralatan M/K akan berfungsi meng-enkapsulasi perbedaan-perbedaan yang ada dari setiap anggota kelas-kelas yang umum tadi.

Tujuan dari adanya lapisan *device driver* ini adalah untuk menyembunyikan perbedaan-perbedaan yang ada pada pengendali perangkat dari subsistem M/K yang terdapat dalam kernel. Dengan demikian, subsistem M/K dapat bersifat mandiri dari perangkat keras. Hal ini sangat menguntungkan dari segi pengembangan perangkat keras, karena tidak perlu menunggu vendor sistem operasi untuk mengeluarkan *support code* untuk perangkat-perangkat keras baru yang akan dikeluarkan oleh para vendor perangkat keras tersebut.

Sayangnya untuk manufaktur perangkat keras, masing-masing sistem operasi memiliki standarnya sendiri untuk *device driver* antarmukanya. Karakteristik dari perangkat-perangkat tersebut sangat bervariasi, beberapa yang dapat membedakannya adalah dari segi:

1. ***Character-stream* atau *block*.** Sebuah *stream* karakter memindahkan per satu *bytes*, sedangkan blok memindahkan sekumpulan *bytes* dalam 1 unit.
2. ***Sequential* atau *Random-access*.** Sebuah perangkat yang sekuensial memindahkan data dalam susunan yang sudah pasti seperti yang ditentukan oleh perangkat, sedangkan pengguna akses *random* dapat meminta perangkat untuk mencari ke seluruh lokasi penyimpanan data yang tersedia.
3. ***Synchronous* atau *asynchronous*.** Perangkat yang *synchronous* menampilkan data-data transfer dengan waktu reaksi yang dapat diduga, sedangkan perangkat yang *asynchronous* menampilkan waktu reaksi yang tidak dapat diduga.
4. ***Sharable* atau *dedicated*.** perangkat yang dapat dibagi dapat digunakan secara bersamaan oleh beberapa prosesor atau *thread*, sedangkan perangkat yang *dedicated*tidak dapat.
5. ***Speed of operation*.** Rentangan kecepatan perangkat dari beberapa bytes per detik sampai beberapa gigabytes per detik.
6. ***Read-write, read only*, atau *write only*.** Beberapa perangkat memungkinkan baik input-output dua arah, tapi beberapa lainnya hanya menunjang data satu arah.

Pada umumnya sistem operasi juga memiliki sebuah "*escape*" atau "pintu belakang" yang secara terbuka mengirim perintah yang *arbitrary* dari sebuah aplikasi ke *device driver*. Dalam UNIX, ada **ioctl()** yang memungkinkan aplikasi mengakses seluruh fungsi yang tersedia di *device driver* tanpa perlu membuat sebuah sistem *call* yang baru.

Perintah **ioctl()** ini mempunyai tiga argumen, yang pertama adalah sebuah pendeskripsi berkas yang menghubungkan aplikasi ke *driver* dengan menunjuk perangkat keras yang diatur oleh *driver* tersebut. Kedua, adalah sebuah integer yang memilih satu perintah yang terimplementasi di dalam *driver*. Ketiga, sebuah pointer ke struktur data *arbitrary* di memori, yang memungkinkan aplikasi dan *driver* berkomunikasi dengan data dan mengendalikan informasi data.

Peralatan Blok dan Karakter

Peralatan blok diharapkan dapat memenuhi kebutuhan akses pada berbagai macam *disk drive* dan juga peralatan blok lainnya, memenuhi/mengerti perintah baca, tulis dan juga perintah pencarian data pada peralatan yang memiliki sifat *random-access*.

Keyboard adalah salah satu contoh alat yang dapat mengakses *stream*-karakter. *System call* dasar dari antarmuka ini dapat membuat sebuah aplikasi mengerti tentang bagaimana cara untuk mengambil dan menuliskan sebuah karakter. Kemudian pada pengembangan lanjutannya, kita dapat membuat *library* yang dapat mengakses data/pesan baris demi baris.

Peralatan Jaringan

Karena adanya perbedaan dalam kinerja dan pengalamatan dari jaringan M/K, maka biasanya sistem operasi memiliki antarmuka M/K yang berbeda dari baca, tulis dan pencarian pada disk. Salah satu yang banyak digunakan pada sistem operasi adalah *socket interface*.

Socket berfungsi untuk menghubungkan komputer ke jaringan. *System call* pada *socket interface* dapat memudahkan suatu aplikasi untuk membuat *local socket*, dan menghubungkannya ke *remote*

socket. Dengan menghubungkan komputer ke *socket*, maka komunikasi antar komputer dapat dilakukan.

Jam dan *Timer*

Adanya jam dan *timer* pada perangkat keras komputer, setidaknya memiliki tiga fungsi, memberi informasi waktu saat ini, memberi informasi lamanya waktu sebuah proses, sebagai *trigger* untuk suatu operasi pada suatu waktu. Fungsi-fungsi ini sering digunakan oleh sistem operasi. Sayangnya, *system call* untuk pemanggilan fungsi ini tidak distandarisasi antar sistem operasi.

Perangkat keras yang mengukur waktu dan melakukan operasi *trigger* dinamakan *programmable interval timer*. Dia dapat diatur untuk menunggu waktu tertentu dan kemudian melakukan interupsi. Contoh penerapannya ada pada *scheduler*, dimana dia akan melakukan interupsi yang akan memberhentikan suatu proses pada akhir dari bagian waktunya.

Sistem operasi dapat mendukung lebih dari banyak *timer request* daripada banyaknya jumlah *timer hardware*. Dengan kondisi seperti ini, maka kernel atau *device driver* mengatur daftar dari interupsi dengan urutan yang pertama kali datang akan dilayani terlebih dahulu.

M/K Blok dan Nonblok

Ketika suatu aplikasi menggunakan sebuah *blocking system call*, eksekusi aplikasi itu akan dihentikan sementara lalu dipindahkan ke *wait queue*. Setelah *system call* tersebut selesai, aplikasi tersebut dikembalikan ke *run queue*, sehingga pengeksekusiannya akan dilanjutkan. *Physical action* dari peralatan M/K biasanya bersifat *asynchronous*. Akan tetapi, banyak sistem operasi yang bersifat *blocking*, hal ini terjadi karena *blocking application* lebih mudah dimengerti dari pada *nonblocking application*.

44.3. Penjadwalan M/K

Kernel menyediakan banyak layanan yang berhubungan dengan M/K. Pada bagian ini, kita akan mendeskripsikan beberapa layanan yang disediakan oleh subsistem kernel M/K, dan kita akan membahas bagaimana caranya membuat infrastruktur perangkat keras dan *device driver*. Layanan-layanan yang akan kita bahas adalah penjadwalan M/K, *buffering*, *caching*, *spooling*, reservasi perangkat, *error handling*.

Menjadwal sekumpulan permintaan M/K sama dengan menentukan urutan yang sesuai untuk mengeksekusi permintaan tersebut. Penjadwalan dapat meningkatkan performa sistem secara keseluruhan, dapat membagi perangkat secara adil di antara proses-proses, dan dapat mengurangi waktu tunggu rata-rata untuk menyelesaikan operasi M/K.

Berikut adalah contoh sederhana untuk menggambarkan definisi di atas. Jika sebuah *arm disk* terletak di dekat permulaan disk, dan ada tiga aplikasi yang memblokir panggilan untuk membaca disk tersebut. Aplikasi pertama meminta sebuah blok dekat akhir disk, aplikasi kedua meminta blok yang dekat dengan awal, dan aplikasi tiga meminta bagian tengah dari disk. Sistem operasi dapat mengurangi jarak yang harus ditempuh oleh *arm disk* dengan melayani aplikasi tersebut dengan urutan 2, 3, 1. Pengaturan urutan pekerjaan kembali seperti ini merupakan inti dari penjadwalan M/K.

Pengembang sistem operasi mengimplementasikan penjadwalan dengan mengatur antrian permintaan untuk tiap perangkat. Ketika sebuah aplikasi meminta sebuah *blocking* sistem M/K, permintaan tersebut dimasukkan ke dalam antrian untuk perangkat tersebut. *Scheduler M/K* mengurutkan kembali antrian untuk meningkatkan efisiensi dari sistem dan waktu respon rata-rata yang harus dialami oleh aplikasi. Sistem operasi juga mencoba untuk bertindak secara adil agar tidak ada aplikasi yang menerima layanan yang lebih sedikit, atau dapat memberikan prioritas layanan untuk permintaan penting yang ditunda. Contohnya, pemintaan dari sub sistem mungkin akan mendapatkan prioritas lebih tinggi daripada permintaan dari aplikasi. Beberapa algoritma penjadwalan untuk M/K disk akan dijelaskan pada bagian Penjadwalan Disk.

Salah satu cara untuk meningkatkan efisiensi M/K sub sistem dari sebuah komputer adalah dengan mengatur operasi M/K tersebut. Cara lain adalah dengan menggunakan tempat penyimpanan pada

memori utama atau pada disk, melalui teknik yang disebut *buffering*, *caching*, dan *spooling*.

44.4. *Buffering*

Buffer adalah area memori yang menyimpan data ketika mereka sedang dipindahkan antara dua perangkat atau antara perangkat dan aplikasi. Tiga alasan melakukan buffering:

1. **Mengatasi perbedaan kecepatan antara produsen dengan konsumen dari sebuah *stream* data.** Contoh, sebuah berkas sedang diterima melalui *modem* dan akan disimpan di *hard disk*. Kecepatan *modem* tersebut ribuan kali lebih lambat daripada *hard disk*, sehingga *buffer* dibuat di dalam memori utama untuk mengumpulkan jumlah *byte* yang diterima dari *modem*. Ketika keseluruhan data di *buffer* sudah sampai, *buffer* tersebut dapat ditulis ke disk dengan operasi tunggal. Karena penulisan disk tidak terjadi dengan seketika dan *modem* masih memerlukan tempat untuk menyimpan data yang berdatangan, maka dua buah *buffer* digunakan untuk melakukan operasi ini. Setelah *modem* memenuhi *buffer* pertama, akan terjadi permintaan untuk menulis di disk. *Modem* kemudian mulai memenuhi *buffer* kedua sementara *buffer* pertama dipakai untuk penulisan ke disk. Seiring *modem* sudah memenuhi *buffer* kedua, penulisan ke disk dari *buffer* pertama seharusnya sudah selesai, jadi *modem* akan berganti kembali memenuhi *buffer* pertama sedangkan *buffer* kedua dipakai untuk menulis. Metode *double buffering* ini membuat pasangan ganda antara produsen dan konsumen sekaligus mengurangi kebutuhan waktu diantara mereka.
2. **Untuk menyesuaikan perangkat-perangkat yang mempunyai perbedaan dalam ukuran transfer data.** Hal ini sangat umum terjadi pada jaringan komputer, dimana *buffer* dipakai secara luas untuk fragmentasi dan pengaturan kembali pesan-pesan yang diterima. Pada bagian pengirim, sebuah pesan yang besar akan dipecah ke dalam paket-paket kecil. Paket-paket tersebut dikirim melalui jaringan, dan penerima akan meletakkan mereka di dalam *buffer* untuk disusun kembali.
3. **Untuk mendukung *copy semantics* untuk aplikasi M/K.** Sebuah contoh akan menjelaskan apa arti dari *copy semantics*. Jika ada sebuah aplikasi yang mempunyai *buffer* data yang ingin dituliskan ke disk, aplikasi tersebut akan memanggil sistem penulisan, menyediakan *pointer* ke *buffer*, dan sebuah *integer* untuk menunjukkan ukuran *bytes* yang ingin ditulis. Setelah pemanggilan tersebut, apakah yang akan terjadi jika aplikasi tersebut merubah isi dari buffer? Dengan *copy semantics*, versi data yang ingin ditulis sama dengan versi data waktu aplikasi ini memanggil sistem untuk menulis, tidak tergantung dengan perubahan yang terjadi pada *buffer*. Sebuah cara sederhana untuk sistem operasi untuk menjamin *copy semantics* adalah membiarkan sistem penulisan untuk menyalin data aplikasi ke dalam *buffer kernel* sebelum mengembalikan kontrol kepada aplikasi. Jadi penulisan ke disk dilakukan pada *buffer kernel*, sehingga perubahan yang terjadi pada *buffer* aplikasi tidak akan membawa dampak apa-apa. Menyalin data antara *buffer kernel* data aplikasi merupakan sesuatu yang umum pada sistem operasi, kecuali *overhead* yang terjadi karena operasi *clean semantics*. Kita dapat memperoleh efek yang sama yang lebih efisien dengan memanfaatkan pemetaan virtual-memori dan proteksi *copy-on-wire* dengan lebih pintar.

44.5. *Caching*

Sebuah *cache* adalah daerah memori yang cepat yang berisikan data kopian. Akses ke sebuah kopian yang di-*cached* lebih efisien daripada akses ke data asli. Sebagai contoh, instruksi-instruksi dari proses yang sedang dijalankan disimpan ke dalam disk, dan ter-*cached* di dalam memori fisik, dan kemudian dikopi lagi ke dalam *cache secondary and primary* dari CPU. Perbedaan antara sebuah *buffer* dan *cache* adalah *buffer* dapat menyimpan satu-satunya informasi data sedangkan sebuah *cache* secara definisi hanya menyimpan sebuah data dari sebuah tempat untuk dapat diakses lebih cepat.

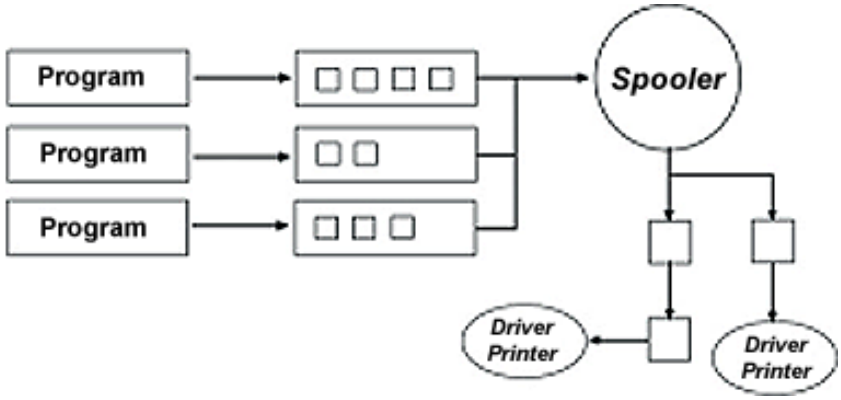
Caching dan *buffering* adalah dua fungsi yang berbeda, tetapi terkadang sebuah daerah memori dapat digunakan untuk keduanya. sebagai contoh, untuk menghemat *copy semantics* dan membuat penjadwalan M/K menjadi efisien, sistem operasi menggunakan *buffer* pada memori utama untuk menyimpan data.

Buffer ini juga digunakan sebagai *cache*, untuk meningkatkan efisiensi IO untuk berkas yang digunakan secara bersama-sama oleh beberapa aplikasi, atau yang sedang dibaca dan ditulis secara berulang-ulang.

Ketika kernel menerima sebuah permintaan berkas M/K, kernel tersebut mengakses *buffer cache* untuk melihat apakah daerah memori tersebut sudah tersedia dalam memori utama. Jika sudah tersedia, sebuah M/K disk fisik dapat dihindari atau bahkan tidak dipakai. Penulisan disk juga terakumulasi ke dalam *buffer cache* selama beberapa detik, jadi transfer yang besar akan dikumpulkan untuk mengefisiensikan jadwal penulisan. Cara ini akan menunda penulisan untuk meningkatkan efisiensi M/K akan dibahas pada bagian *Remote File Access*.

44.6. *Spooling* dan Reservasi Perangkat

Gambar 44.2. Spooling



Spooling adalah proses yang sangat berguna saat berurusan dengan perangkat M/K dalam sistem multiprogram. Sebuah *spool* adalah sebuah buffer yang menyimpan keluaran untuk sebuah perangkat yang tidak dapat menerima *interleaved data* streams. Salah satu perangkat *spool* yang paling umum adalah printer.

Printer hanya dapat melayani satu pekerjaan pada waktu tertentu, namun beberapa aplikasi dapat meminta printer untuk mencetak. *Spooling* memungkinkan keluaran mereka tercetak satu per satu, tidak tercampur. Untuk mencetak sebuah berkas, pertama-tama sebuah proses mengeneralisasi berkas secara keseluruhan untuk di cetak dan ditempatkan pada *spooling directory*. Sistem operasi akan menyelesaikan masalah ini dengan meng-*intercept* semua keluaran kepada printer. Tiap keluaran aplikasi sudah di-*spooled* ke disk berkas yang berbeda. Ketika sebuah aplikasi selesai mencetak, sistem *spooling* akan melanjutkan ke antrian berikutnya.

Di dalam beberapa sistem operasi, *spooling* ditangani oleh sebuah sistem proses *daemon*. Pada sistem operasi yang lain, sistem ini ditangani oleh *in-kernel thread*. Pada kedua penanganan tersebut, sistem operasi menyediakan antarmuka kontrol yang membuat *users* and sistem administrator dapat menampilkan antrian tersebut, untuk mengenyahkan antrian-antrian yang tidak diinginkan sebelum mulai dicetak.

Contoh lain adalah penggunaan *spooling* pada transfer berkas melalui jaringan yang biasanya menggunakan *daemon* jaringan. Untuk mengirim berkas ke suatu tempat, *user* menempatkan berkas tersebut dalam *spooling directory* jaringan. Selanjutnya, *daemon* jaringan akan mengambilnya dan mentransmisikannya. Salah satu bentuk nyata penggunaan *spooling* jaringan adalah sistim *email* via Internet. Keseluruhan sistem untuk mail ini berlangsung di luar sistem operasi.

Beberapa perangkat, seperti drive tape dan printer, tidak dapat me-*multiplex* permintaan M/K dari beberapa aplikasi. Selain dengan *spooling*, dapat juga diatasi dengan cara lain, yaitu dengan membagi koordinasi untuk *multiple concurrent* ini. Beberapa sistem operasi menyediakan dukungan untuk akses perangkat secara eksklusif, dengan mengalokasikan proses ke *device idle* dan membuang perangkat yang sudah tidak diperlukan lagi. Sistem operasi lainnya memaksakan limit suatu berkas untuk menangani perangkat ini. Banyak sistem operasi menyediakan fungsi yang membuat proses untuk menangani koordinat *exclusive akses* diantara mereka sendiri.

44.7. Penanganan Kesalahan (*Error Handling*)

Sebuah sistem operasi yang menggunakan *protected memory* dapat menjaga banyak kemungkinan *error* akibat perangkat keras maupun aplikasi. Perangkat dan transfer M/K dapat gagal dalam banyak cara, dapat karena alasan transient, seperti *overloaded* pada jaringan, maupun alasan permanen yang seperti kerusakan yang terjadi pada *disk controller*. Sistem operasi seringkali dapat mengkompensasikan untuk kesalahan transient. Seperti, sebuah kesalahan baca pada disk akan mengakibatkan pembacaan ulang kembali dan sebuah kesalahan pengiriman pada jaringan akan mengakibatkan pengiriman ulang apabila protokolnya diketahui. Akan tetapi untuk kesalahan permanen, sistem operasi pada umumnya tidak akan dapat mengembalikan situasi seperti semula.

Sebuah ketentuan umum, yaitu sebuah sistem M/K akan mengembalikan satu bit informasi tentang status panggilan tersebut, yang akan menandakan apakah proses tersebut berhasil atau gagal. Sistem operasi pada UNIX menggunakan *integer* tambahan yang dinamakan ***ERRNO*** untuk mengembalikan kode kesalahan sekitar 1 dari 100 nilai yang mengindikasikan sebab dari kesalahan tersebut. Sebaliknya, beberapa perangkat keras dapat menyediakan informasi kesalahan yang detail, walaupun banyak sistem operasi yang tidak mendukung fasilitas ini.

Sebagai contoh, kesalahan pada perangkat SCSI dilaporkan oleh protokol SCSI dalam bentuk ***sense key*** yang mengindentifikasi kesalahan yang umum seperti *error* pada perangkat keras atau permintaan yang ilegal; sebuah ***additional sense code*** yang mengkategorikan kesalahan yang muncul, seperti kesalahan parameter atau kesalahan *self-test*; dan sebuah ***additional sense code qualifier*** yang memberitahukan kesalahan secara lebih mendalam dan mendetil, seperti parameter yang error.

Pada sistem operasi Linux, error dapat dikenali dengan metoda yang disebut oops. Oops adalah cara yang biasanya dilakukan kernel untuk berkomunikasi kepada pengguna ketika terjadi suatu kesalahan. Cara ini melibatkan pencetakan pesan error pada console, mengeluarkan isi dari register dan menyediakan back trace. Oops bisa terjadi karena berbagai alasan, bisa karena pelanggaran akses memori atau juga karena instruksi yang ilegal.

Berikut adalah contoh dari oops

```
Oops: Exception in kernel mode, sig: 4
Unable to handle kernel NULL pointer dereference at virtual
address 00000001

NIP: C013A7F0 LR: C013A7F0 SP: C0685E00 REGS: c0905d10 TRAP: 0700
Not tainted
MSR: 00089037 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last syscall: 120
GPR00: C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08: 000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00000005 00000000 00001032 C3F7C000 00000032 FFFFFFFF C3F7C1C0
Call trace:
[c013ab30] tulip_timer+0x128/0x1c4
[c0020744] run_timer_softirq+0x10c/0x164
[c001b864] do_softirq+0x88/0x104
[c0007e80] timer_interrupt+0x284/0x298
[c00033c4] ret_from_except+0x0/0x34
[c0007b84] default_idle+0x20/0x60
[c0007bf8] cpu_idle+0x34/0x38
[c0003ae8] rest_init+0x24/0x34
```

44.8. Struktur Data Kernel

Kernel membutuhkan informasi keadaan tentang penggunakan komponen M/K. *Kernel* menggunakan banyak struktur yang mirip untuk melacak koneksi jaringan, komunikasi perangkat

karakter, dan aktivitas M/K lainnya.

UNIX menyediakan akses sistem berkas untuk beberapa entiti, seperti berkas pengguna, *raw devices*, dan alamat tempat proses. Walaupun tiap entiti ini didukung sebuah operasi baca, semantiknya berbeda untuk tiap entiti. Seperti untuk membaca berkas pengguna, kernel perlu memeriksa *buffer cache* sebelum memutuskan apakah akan melaksanakan M/K disk. Untuk membaca sebuah *raw disk*, kernel perlu untuk memastikan bahwa ukuran permintaan adalah kelipatan dari ukuran sektor disk, dan masih terdapat di dalam batas sektor. Untuk memproses citra, cukup perlu untuk mengkopi data ke dalam memori. UNIX mengkapsulasikan perbedaan-perbedaan ini di dalam struktur yang seragam dengan menggunakan teknik *object oriented*.

Beberapa sistem operasi bahkan menggunakan metode *object oriented* secara lebih ekstensif. Sebagai contoh, Windows NT menggunakan implementasi *message-passing* untuk M/K. Sebuah permintaan M/K akan dikonversikan ke sebuah pesan yang dikirim melalui kernel kepada M/K manager dan kemudian ke *device driver*, yang masing-masing dapat mengubah isi pesan. Untuk output, isi message adalah data yang akan ditulis. Untuk input, message berisikan *buffer* untuk menerima data. Pendekatan *message-passing* ini dapat menambah *overhead*, dengan perbandingan dengan teknik prosedural yang membagi struktur data, tetapi akan mennyederhanakan struktur dan design dari sistem M/K tersebut dan menambah fleksibilitas.

Kesimpulannya, subsistem M/K mengkoordinasi kumpulan-kumpulan service yang banyak sekali, yang tersedia dari aplikasi maupun bagian lain dari kernel. Subsistem M/K mengawasi:
1. Manajemen nama untuk berkas dan perangkat.
2. Kontrol akses untuk berkas dan perangkat.
3. Kontrol operasi, contoh: model yang tidak dapat dikenali.
4. Alokasi tempat sistem berkas.
5. Alokasi perangkat.
6. *Buffering*, *caching*, *spooling*.
7. Penjadwalan M/K
8. Mengawasi status perangkat, *error handling*, dan kesalahan dalam *recovery*.
9. Konfigurasi dan utilisasi *driver device*.

44.9. Penanganan Permintaan M/K

Di bagian sebelumnya, kita mendeskripsikan *handshaking* antara *device driver* dan pengendali perangkat, tapi kita tidak menjelaskan bagaimana Sistem Operasi menyambungkan permintaan aplikasi untuk menyiapkan jaringan menuju sektor disk yang spesifik.

Sistem Operasi yang modern mendapatkan fleksibilitas yang signifikan dari tahapan-tahapan tabel lookup di jalur diantara permintaan dan pengendali perangkat *physical*. Kita dapat mengenalkan perangkat dan *driver* baru ke komputer tanpa harus mengkompilasi ulang kernelnya. Sebagai fakta, ada beberapa sistem operasi yang mampu untuk me-*load device drivers* yang diinginkan. Pada waktu *boot*, sistem mula-mula meminta *bus* perangkat keras untuk menentukan perangkat apa yang ada, kemudian sistem me-*load* ke dalam *driver* yang sesuai; baik sesegera mungkin, maupun ketika diperlukan oleh sebuah permintaan M/K.

Sistem V UNIX mempunyai mekanisme yang menarik, yang disebut *streams*, yang membolehkan aplikasi untuk men-*assemble pipeline* dari kode *driver* secara dinamis. Sebuah *stream* adalah sebuah koneksi *full duplex* antara sebuah *device driver* dan sebuah proses user-level. Stream terdiri atas sebuah *stream head* yang merupakan antarmuka dengan user process, sebuah *driver end* yang mengontrol perangkat, dan nol atau lebih *stream modules* di antara mereka. *Modules* dapat didorong ke *stream* untuk menambah fungsionalitas di sebuah *layered fashion*. Sebagai gambaran sederhana, sebuah proses dapat membuka sebuah alat *port serial* melalui sebuah *stream*, dan dapat mendorong ke sebuah modul untuk memegang *edit* input. *Stream* dapat digunakan untuk interproses dan komunikasi jaringan. Faktanya, di Sistem V, mekanisme soket diimplementasikan dengan *stream*.

Berikut dideskripsikan sebuah *lifecycle* yang tipikal dari sebuah permintaan pembacaan blok:
1. Sebuah proses mengeluarkan sebuah *blocking read system call* ke sebuah berkas deskriptor dari berkas yang telah dibuka sebelumnya.
2. Kode *system-call* di kernel mengecek parameter untuk kebenaran. Dalam kasus input, jika data telah siap di *buffer cache*, data akan dikembalikan ke proses dan permintaan M/K diselesaikan.
3. Jika data tidak berada dalam *buffer cache*, sebuah physical M/K akan bekerja, sehingga proses

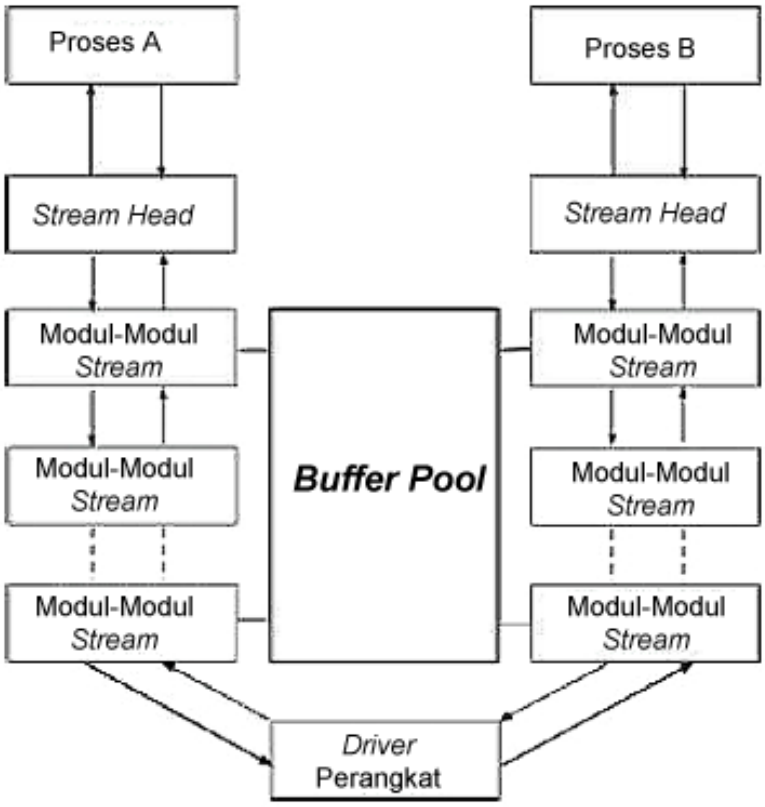
akan dikeluarkan dari antrian jalan (*run queue*) dan diletakkan di antrian tunggu (*wait queue*) untuk alat, dan permintaan M/K pun dijadwalkan. Pada akhirnya, subsistem M/K mengirimkan permintaan ke *device driver*. Bergantung pada sistem operasi, permintaan dikirimkan melalui *call* subrutin atau melalui pesan *in-kernel*.

4. *Device driver* mengalokasikan ruang *buffer* pada kernel untuk menerima data, dan menjadwalkan M/K. Pada akhirnya, driver mengirim perintah ke pengendali perangkat dengan menulis ke *register device control*.
5. Pengendali perangkat mengoperasikan perangkat keras perangkat untuk melakukan transfer data.
6. *Driver* dapat menerima status dan data, atau dapat menyiapkan transfer DMA ke memori kernel. Kita mengasumsikan bahwa transfer diatur oleh sebuah *DMA controller*, yang menggunakan interupsi ketika transfer selesai.
7. *Interrupt handler* yang sesuai menerima interupsi melalui tabel vektor-interupsi, menyimpan sejumlah data yang dibutuhkan, menandai *device driver*, dan kembali dari interupsi.
8. *Device driver* menerima tanda, menganalisa permintaan M/K mana yang telah diselesaikan, menganalisa status permintaan, dan menandai subsistem M/K kernel yang permintaannya telah terselesaikan.
9. Kernel mentransfer data atau mengembalikan kode ke ruang alamat dari proses permintaan, dan memindahkan proses dari antrian tunggu kembali ke antrian siap.
10. Proses tidak diblok ketika dipindahkan ke antrian siap. Ketika penjadwal (*scheduler*) mengembalikan proses ke CPU, proses meneruskan eksekusi pada penyelesaian dari *system call*.

44.10. Arus M/K

Arus M/K (*I/O stream*) merupakan mekanisme pengiriman data secara bertahap dan terus menerus melalui suatu aliran data dari proses ke peranti (begitu pula sebaliknya).

Gambar 44.3. Struktur Stream



Arus M/K terdiri dari:

1. *stream head* yang berhubungan langsung dengan proses.
2. *driver ends* yang mengatur peranti-peranti

3. *stream modules* yang berada di antara *stream head* dan *driver end*, yang bertugas menyampaikan data ke *driver end* melalui *write queue*, maupun menyampaikan data ke proses melalui *read queue* dengan cara *message passing*.

Untuk memasukkan ke dalam stream digunakan *ioctl() system call*, sedangkan untuk menuliskan data ke peranti digunakan *write()*/*putmsg() system calls*, dan untuk membaca data dari peranti digunakan *read()*/*getmsg() system calls*.

44.11. Kinerja M/K

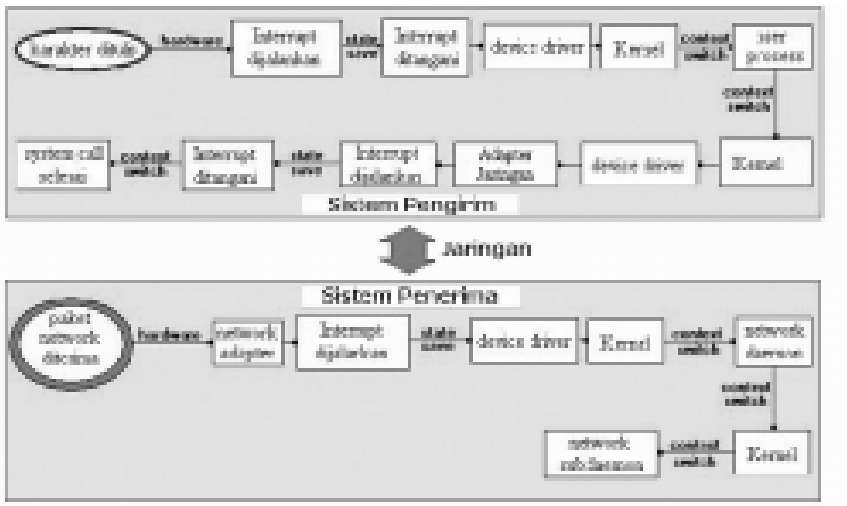
M/K adalah faktor penting dalam kinerja sistem. M/K sering meminta CPU untuk mengeksekusi *device-driver code* dan menjadwal proses secara efisien sewaktu *memblock* dan *unblock*. Hasil *context switch* men-*stress* ke CPU dan *hardware cache*-nya. M/K juga memberitahukan ketidakefisienan mekanisme penanganan interupsi dalam kernel, dan M/K me-*load down memory bus* saat *data copy* antara pengendali dan memori fisik, dan juga saat *copy* antara *kernel buffers* dan *application data space*. Mengkopi dengan semua permintaan ini adalah salah satu kekhawatiran dalam arsitektur komputer.

Walaupun komputer modern dapat menangani beribu-ribu interupsi per detik, namun penanganan interupsi adalah pekerjaan yang sulit. Setiap interupsi mengakibatkan sistem melakukan perubahan status, mengeksekusi *interrupt handler* lalu mengembalikan statusnya kembali. M/K yang terprogram dapat lebih efisien dibanding *interrupt-driven I/O*, jika waktu *cycle* yang dibutuhkan untuk *busy-waiting* tidak berlebihan. M/K yang sudah selesai biasanya meng-*unblock* sebuah proses lalu membawanya ke *full overhead of context switch*.

Network traffic juga dapat menyebabkan *high context-switch rate*. Coba diperhatikan, misalnya sebuah *remote login* dari sebuah mesin ke mesin lainnya. Setiap karakter yang diketikkan pada *local machine* harus dikirim ke *remote machine*. Pada *local machine* karakter akan diketikkan, lalu *keyboard interrupt* dibuat, dan karakter melewati *interrupt handler* menuju *device-driver* lalu ke kernel, setelah itu ke proses. Proses memanggil *network I/O system call* untuk mengirim karakter ke *remote machine*. Karakter lalu melewati local kernel, menuju ke lapisan-lapisan network yang membuat paket network, lalu ke *network device driver*. *Network device driver* mengirim paket itu ke *network controller*, yang mengirim karakter dan membuat interupsi. Interupsi kemudian dikembalikan ke kernel supaya *I/O system call* dapat selesai.

Sekarang *remote system's network hardware* sudah menerima paket, dan interupsi dibuat. Karakter di-*unpack* dari *network protocol* dan dikirim ke *network daemon* yang sesuai. *Network daemon* mengidentifikasi *remote login session* mana yang terlibat, dan mengirim paket ke *subdaemon* yang sesuai untuk *session* itu. Melalui alur ini, ada *context switch* dan *state switch* (lihat Gambar 44.4, “Antar Komputer”). Biasanya, penerima mengirim kembali karakter ke pengirim.

Gambar 44.4. Antar Komputer



Developer Solaris mengimplementasikan kembali telnet daemon menggunakan kernel-thread untuk menghilangkan *context switch* yang terlibat dalam pemindahan karakter dari daemon ke kernel. Sun memperkirakan bahwa perkembangan ini akan menambah jumlah maksimum *network logins* dari beberapa ratus hingga beberapa ribu (pada server besar).

Sistem lain menggunakan *front-end processor* yang terpisah untuk terminal M/K, supaya mengurangi beban interupsi pada *main CPU*. Misalnya, sebuah *terminal concentrator* dapat mengirim sinyal secara bersamaan dari beratus-ratus terminal ke satu port di *large computer*. Sebuah *I/O channel* adalah sebuah CPU yang memiliki tujuan khusus yang ditemukan pada mainframe dan pada *sistem high-end* lainnya. Kegunaan dari *I/O channel* adalah untuk meng-*offload I/O work* dari *main CPU*. Prinsipnya adalah *channel* tersebut menjaga supaya lalu lintas data lancar, sehingga main CPU dapat bebas memproses data. Seperti *device controller* dan *DMA controller* yang ada pada *smaller computer*, sebuah *channel* dapat memproses program-program yang umum dan kompleks, jadi *channel* dapat digunakan untuk *workload* tertentu.

Kita dapat menggunakan beberapa prinsip untuk menambah efisiensi M/K:
1. Mengurangi *context switch*.
2. Mengurangi jumlah pengkopian data dalam memori sewaktu pengiriman antara peranti dan aplikasi.
3. Mengurangi jumlah interupsi dengan menggunakan transfer besar-besaran, *smart controller*, dan *polling* (jika *busy-waiting* dapat diminimalisir).
4. Menambah konkurensi dengan menggunakan pengendali atau *channel* DMA yang sudah diketahui untuk meng-*offload* kopi data sederhana dari CPU.
5. Memindahkan *processing primitives* ke perangkat keras, supaya operasi pada *device controller* konkuren dengan CPU dan operasi *bus*.
6. Keseimbangan antara CPU, *memory subsystem*, *bus* dan kinerja M/K, karena sebuah *overload* pada salah satu area akan menyebabkan keterlambatan pada yang lain.

Kompleksitas peranti berbeda-beda, misalnya *mouse*. *Mouse* adalah peranti yang sederhana. Pergerakan *mouse* dan *button click* diubah menjadi nilai numerik yang dikirim dari perangkat keras (melalui *mouse device driver*) menuju aplikasinya. Kebalikan dari *mouse*, fungsionalitas yang disediakan *NT disk device driver* sangatlah kompleks. *NT disk device driver* tidak hanya mengatur *individual disk*, tapi juga mengimplementasikan *RAID array*. Untuk dapat melakukannya, *NT disk device driver* mengubah *read* atau pun *write request* dari aplikasi menjadi *coordinated set of disk I/O operations*. Terlebih lagi, *NT disk device driver* mengimplementasikan penanganan error dan algoritma *data-recovery*, lalu mengambil langkah-langkah untuk mengoptimalkan kinerja disk, karena kinerja penyimpanan sekunder adalah hal penting untuk keseluruhan kinerja sistem.

Kapan fungsionalitas M/K dapat diimplementasikan? Pada *device hardware*, *device driver*, atau pada aplikasi perangkat lunak?

Mula-mula kita implementasikan eksperimen algoritma M/K pada *application level*, karena *application code* lebih fleksibel, dan *application bug* tidak membuat sistem *crash*. Terlebih lagi dengan mengembangkan kode pada *application level*, kita dapat menghindari *reboot* atau pun *reload device driver* setiap mengganti kode. Bagaimana pun juga sebuah implementasi pada *application level* dapat tidak efisien, karena *overhead of context switch*, dan karena aplikasi tidak dapat menerima kemudahan dari *internal kernel data structure* dan *fungsionalitas kernel* (seperti *internal kernel messaging*, *threading*, dan *locking* yang efisien).

Ketika algoritma *application level* memperlihatkan kegunaannya, kita dapat mengimplementasikan kembali kernel, sehingga dapat menambah kinerja. Akan tetapi, usaha pengembangan sulit dilakukan karena sistem operasi kernel adalah sistem perangkat lunak yang besar dan kompleks.

Terlebih lagi, dalam pengimplementasian internal kernel harus di-*debug* secara hati-hati untuk menghindari *data corrupt* dan sistem *crash*.

Kinerja tertinggi dapat didapatkan dengan cara implementasi spesial dalam perangkat keras, baik dalam peranti atau pun pengendali. Kerugian dari implementasi perangkat keras termasuk kesulitan dan pengorbanan dari membuat kemajuan atau dari pembetulan *bug*, dan bertambahnya *development time* (dalam satuan bulan, bukan hari), dan berkurangnya fleksibilitas.

Misalnya, sebuah *hardware RAID controller* mungkin saja tidak memberikan izin kepada kernel untuk mempengaruhi urutan atau pun lokasi dari *individual block reads and writes*, walaupun kernel

memiliki informasi tertentu tentang *workload* yang mampu membuat kernel meningkatkan kinerja M/K.

44.12. Rangkuman

Subsistem kernel M/K menyediakan layanan yang berhubungan langsung dengan perangkat keras. Layanan Penjadwalan M/K mengurutkan antrian permintaan pada tiap perangkat dengan tujuan untuk meningkatkan efisiensi dari sistem dan waktu respon rata-rata yang harus dialami oleh aplikasi.

Ada tiga alasan melakukan layanan Buffering, yaitu menyangkut perbedaan kecepatan produsen-konsumen, perbedaan ukuran transfer data dan dukungan copy semantics untuk aplikasi M/K. Fungsi buffering dan caching memiliki perbedaan dalam hal tujuan. Caching menyimpan salinan data asli pada area memori dengan tujuan agar bisa diakses lebih cepat, sedangkan buffering menyalin data asli agar dapat menyimpan satu-satunya informasi data.

Subsistem M/K mengkoordinasi kumpulan-kumpulan sevice yang banyak sekali, yang tersedia dari aplikasi atau bagian lain dari kernel. Penanganan permintaan M/K dilakukan dengan suatu mekanisme yang dideskripsikan sebagai sebuah life cycle.

Layanan Arus M/K (I/O Streams) menggunakan suatu mekanisme pengiriman data secara bertahap dan terus menerus melalui suatu aliran data dari piranti ke proses.

Rujukan

[Hyde2003] Randall Hyde. 2003. *The Art of Assembly Language*. First Edition. No Strach Press.

[Love2005] Robert Love. 2005. *Linux Kernel Development* . Second Edition. Novell Press.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

Bab 45. Manajemen Disk I

45.1. Pendahuluan

Struktur *disk* merupakan suatu hal yang penting bagi penyimpanan informasi. Sistem komputer modern menggunakan *disk* sebagai media penyimpanan sekunder. Dulu pita magnetik digunakan sebelum penggunaan *disk* sebagai media penyimpanan, sekunder yang memiliki waktu akses yang lebih lambat dari *disk*. Sejak digunakan *disk*, tape digunakan untuk backup, untuk menyimpan informasi yang tidak sering digunakan, sebagai media untuk memindahkan informasi dari satu sistem ke sistem lain, dan untuk menyimpan data yang cukup besar bagi sistem *disk*.

Bentuk penulisan *Disk* drive modern adalah *array* blok logika satu dimensi yang besar. Blok logika merupakan satuan unit terkecil dari transfer. Ukuran blok logika umumnya sebesar 512 bytes walaupun *disk* dapat diformat di level rendah (*low level formatted*) sehingga ukuran blok logika dapat ditentukan, misalnya 1024 bytes.

Array adalah blok logika satu dimensi yang dipetakan ke sektor dari *disk* secara sekuensial. Sektor 0 merupakan sektor pertama dari *track* pertama yang terletak di silinder paling luar (*outermost cylinder*). Proses pemetaan dilakukan secara berurut dari Sektor 0, lalu ke seluruh *track* dari silinder tersebut, lalu ke seluruh silinder mulai dari silinder terluar sampai silinder terdalam.

Karena jumlah sector tiap track yang tidak konstan, sedangkan sebuah disk harus bisa mentransfer data yang sama jumlahnya dalam waktu tertentu. Dari sinilah dikenal istilah CAV (Constan Angular Velocity) dan CLV (Constan Linier Velocity).

Pada CAV jumlah sector tiap track sama, sehingga disk berotasi sama banyaknya ketika membaca/transfer data pada bagian inner dan outer silinder. Keuntungan menggunakan metoda CAV ini adalah sebuah data bisa langsung dipetakan sesuai pada track dan nomor silinder yang diminta. Tapi kelemahannya juga ada, jumlah data yang bisa di store pada track terluar dan terdalam sama, apdahal kita tahu bahwa panjang track bagian luar lebih panjang daripada bagian dalam.

Kalau pada metoda CLV sesuai dengan definisinya metode ini menggunakan kecepatan linier yang sama tetapi kecepatan berotasi disk ketika head membaca data bagian luar dan bagian tidak sama. Semakin ke luar, perputaran disk semakin lambat karena jumlah data yang dibaca semakin banyak dengan kata lain, jumlah sector pada outer track lebih banyak daripada inner track. Contohnya pada CD-ROM dan DVD-ROM.

45.2. Penjadwalan *Disk*

Penjadwalan disk merupakan salah satu hal yang sangat penting dalam mencapai efisiensi perangkat keras. Bagi *disk* drives, efisiensi dipengaruhi oleh kecepatan waktu akses dan besarnya *disk bandwidth*. Waktu akses memiliki dua komponen utama yaitu waktu pencarian dan waktu rotasi *disk* (*rotational latency*). Waktu pencarian adalah waktu yang dibutuhkan *disk arm* untuk menggerakkan *head* ke bagian silinder *disk* yang mengandung sektor yang diinginkan. Waktu rotasi *disk* adalah waktu tambahan yang dibutuhkan untuk menunggu perputaran *disk* agar *head* dapat berada di atas sektor yang diinginkan. *Disk bandwidth* adalah total jumlah *bytes* yang ditransfer dibagi dengan total waktu dari awal permintaan transfer sampai transfer selesai. Kita dapat meningkatkan waktu akses dan *bandwidth* dengan menjadwalkan permintaan dari M/K dalam urutan tertentu.

Kenapa penjadwalan disk diperlukan? Hal ini perlu untuk meningkatkan efisiensi perangkat keras. Pada disk drive, efisiensi ini dipengaruhi oleh waktu akses dan besarnya disk bandwidth. Waktu akses adalah waktu yang dibutuhkan untuk membaca dan menulis pada sebuah disk block yang dipengaruhi oleh dua faktor:

1. Seek Time yaitu waktu yang dibutuh disk arm untuk mencapai track yang dicari.
2. Rotational delay adalah waktu dari saat disk arm memperoleh track dicari sampai sampai saat disk head memperoleh sector yang sesuai dengan yang dicari.

```
Rumus:

Waktu-akses = seek-time + rotational-delay

T = b/(r.N)
T = transfer time
b = jumlah bite yang ditransfer
r = kecepatan berotasi (rotasi per detik)
N = jumlah bite dalam track
```

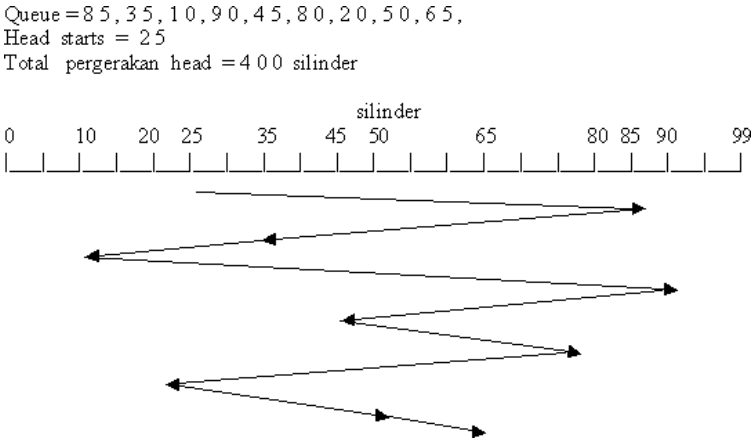
Disk Bandwith (transfer time) adalah banyaknya jumlah bytes yang ditransfer dibandingkan dengan total waktu dari awal permintaan transfer sampai transfer selesai.

Suatu proses yang membutuhkan pelayanan M/K, akan melakukan system call ke sistem operasi. Permintaan layanan ini akan membawa beberapa informasi, yaitu:
1. Jenis operasi (input atau output)
2. Alamat disk untuk proses tersebut
3. Alamat memori proses
4. Jumlah byte yang akan ditransfer

Suatu proses akan dilayani jika disk drive dan penegndalinya tidak sedang sibuk. Jika disk drive dan pengendalinya sedang sibuk melayani proses lain, maka proses tersebut akan diletakkan pada sebuah antrian berupa queue. Dengan demikian, jika disk drive telah melayani suatu proses ia akan melayani proses berikutnya. Mengenai proses mana yang akan diambil berikutnya akan dibahas pada berbagai macam algoritma penjadwalan berikut.

45.3. Penjadwalan FCFS

Gambar 45.1. FCFS



Bentuk algoritma penjadwalan disk yang paling sederhana adalah First Come First Serve (FCFS). Sistem kerja dari algoritma ini adalah melayani permintaan yang lebih dulu datang di queue. Algoritma ini pada hakekatnya adil bagi permintaan M/K yang mengantri di queue karena penjadwalan ini melayani permintaan sesuai waktu tunggunya di queue. Tetapi yang menjadi kelemahan algoritma ini adalah dia tidak menjadi algoritma dengan layanan yang tercepat. Sebagai contoh, misalnya di queue disk terdapat antrian permintaan blok M/K di silinder

85, 35, 10, 90, 45, 80, 20, 50, 65,

secara berurutan. Jika posisi head awal berada di silinder 25, maka pertama kali head akan bergerak dari silinder 25 ke 85, lalu secara berurutan bergerak melayani permintaan di silinder 35, 10, 90, 45, 80, 20, 50 dan akhirnya ke silinder 65. Sehingga total pergerakan headnya adalah 400 silinder.

Untuk lebih jelasnya perhatikan contoh berikut:

```
FCFS
(head awal di silinder 25)

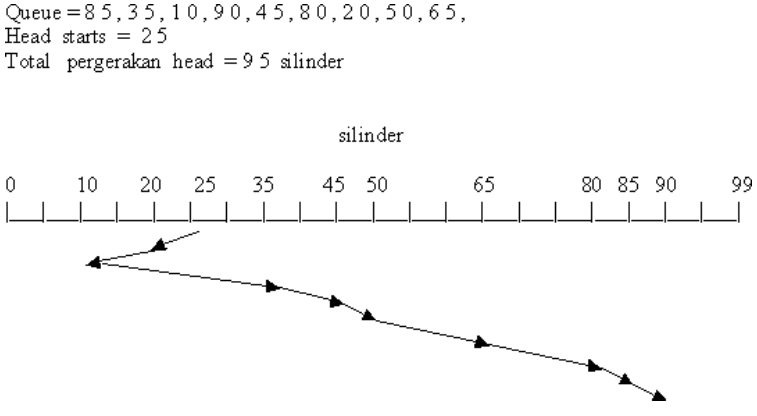
Next  cylinder                Number of cylinder
accessed                      traversed
   85                               60
   35                               45
   10                               25
   90                               80
   45                               45
   80                               35
   20                               60
   50                               30
   65                               15
Total pergerakan head                400 silinder
```

Pergerakan head yang bolak balik dari 25 ke 85 kemudian ke 35 melukiskan masalah dari algoritma penjadwalan ini. Jika permintaan di silinder 20 dan 35 dilayani setelah atau sebelum permintaan di silinder 85 dan 90, total pergerakan head akan berkurang jumlahnya sehingga dapat meningkatkan performa.

45.4. Penjadwalan SSTF

Shortest-Seek-Time-First (SSTF) merupakan algoritma yang melayani permintaan berdasarkan waktu pencarian atau waktu pencarian paling kecil dari posisi *head* terakhir. Sangat beralasan untuk melayani semua permintaan yang berada dekat dengan posisi head yang sebelumnya, sebelum menggerakan head lebih jauh untuk melayani permintaan yang lain. Penjadwalan SSTF merupakan salah satu bentuk dari penjadwalan *shortest-job-first* (SJF), dan karena itu maka penjadwalan SSTF juga dapat mengakibatkan *starvation* pada suatu saat tertentu. Hal ini dapat terjadi bila ada permintaan untuk mengakses bagian yang berada di silinder terdalam. Jika kemudian berdatangan lagi permintaan-permintaan yang letaknya lebih dekat dengan permintaan terakhir yang dilayani maka permintaan dari silinder terluar akan menunggu lama dan sebaliknya. Walaupun algoritma SSTF jauh lebih cepat dibandingkan dengan FCFS, namun untuk keadilan layanan SSTF lebih buruk dari penjadwalan FCFS.

Gambar 45.2. SSTF



Untuk contoh permintaan di queue kita, permintaan yang terdekat dari posisi head awal (25) adalah permintaan silinder 20. Maka silinder itu akan dilayani terlebih dahulu. Setelah head berada di silinder 20, maka permintaan yang terdekat adalah silinder 10. Secara berurutan permintaan yang dilayani selanjutnya adalah silinder 35, lalu 45, 50, 65, 80, 85 dan akhirnya silinder 90. Dengan menggunakan algoritma ini maka total pergerakan headnya menjadi 95 silinder. Hasil yang didapat ternyata kurang dari seperempat jarak yang dihasilkan oleh penjadwalan FCFS.

Untuk lebih jelasnya perhatikan contoh berikut:

SSTF (head awal di silinder 25)

Next cylinder accessed	Number of cylinder traversed
20	5
10	10
35	25
45	10
50	5
65	15
80	15
85	5
90	5
Total pergerakan head	95 silinder

Sistem kerja SSTF yang sama dengan penjadwalan shortest job first (SJF) mengakibatkan kelemahan yang sama dengan kelemahan yang ada di SJF. Yaitu untuk kondisi tertentu dapat mengakibatkan terjadinya starvation. Hal tersebut bisa digambarkan apabila di queue berdatangan permintaan-permintaan baru yang letaknya lebih dekat dengan permintaan terakhir yang dilayani, maka permintaan lama yang letaknya jauh dari permintaan yang dilayani harus menunggu lama sampai permintaan yang lebih dekat itu terlayani semuanya. Walaupun SSTF memberikan waktu pelayanan yang lebih cepat namun bila dilihat dari sudut pandang keadilan bagi permintaan yang menunggu di queue, jelas algoritma ini lebih buruk dibandingkan FCFS Scheduling.

45.5. Penjadwalan SCAN

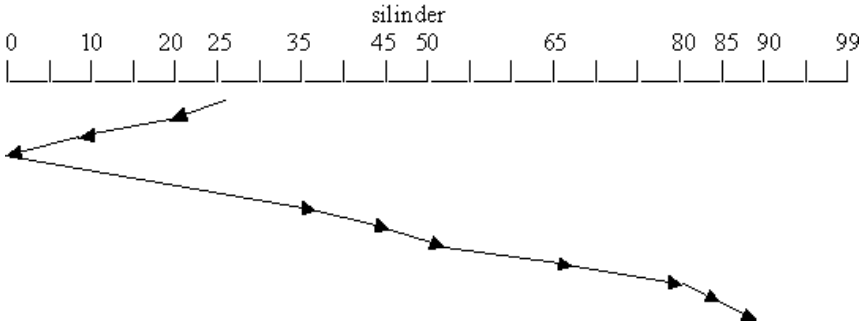
Pada algoritma SCAN, head bergerak ke silinder paling ujung dari disk. Setelah sampai di sana maka head akan berbalik arah menuju silinder di ujung yang lainnya. Head akan melayani permintaan yang dilaluinya selama pergerakannya ini. Algoritma ini disebut juga sebagai Elevator Algorithm karena sistem kerjanya yang sama seperti yang digunakan elevator di sebuah gedung tinggi dalam melayani penggunanya. Elevator akan melayani pengguna yang akan menuju ke atas dahulu sampai lantai tertinggi, baru setelah itu dia berbalik arah menuju lantai terbawah sambil melayani penmggunanya yang akan turun atau sebaliknya. Jika melihat analogi yang seperti itu maka dapat dikatakan head hanya melayani permintaan yang berada di depan arah pergerakannya. Jika ada permintaan yang berada di belakang arah geraknya. Maka permintaan tersebut harus menunggu sampai head menuju silinder di salah satu ujung disk, lalu berbalik arah untuk melayani permintaan tersebut.

Jika head sedang melayani permintaan silinder 25, dan arah pergerakan disk arm-nya sedang menuju ke silinder yang terkecil, maka permintaan berikutnya yang dilayani secara berurutan adalah 20 dan 10 lalu menuju ke silinder 0. Setelah sampai di sini, head akan berbalik arah menuju silinder yang terbesar yaitu silinder 99. Dan dalam pergerakannya itu secara berurutan head akan melayani permintaan 35, 45, 50, 65, 80, 85, dan 90. Sehingga total pergerakan headnya adalah 115 silinder.

Salah satu behavior yang dimiliki oleh algoritma ini adalah, dia memiliki batas atas untuk total pergerakan headnya, yaitu 2 kali jumlah silinder yang dimiliki oleh disk. Jika dilihat dari cara kerjanya yang selalu menuju ke silinder terujung, maka dapat dilihat salah satu dari algoritma ini, yaitu ketidakefisiennya. Mengapa head harus bergerak ke silinder 0, padahal sudah tidak ada lagi permintaan yang lebih kecil dari silinder 10. Bukankah akan lebih efisien jika head langsung berbalik arah setelah melayani permintaan silinder 10 (mengurangi total pergerakan head)?

Kelemahan inilah yang akan dijawab oleh algoritma LOOK yang akan dibahas di sub-bab berikutnya.

Gambar 45.3. SCAN



Untuk lebih jelasnya perhatikan contoh berikut:

```
SCAN
(head awal di silinder 25)
Pergerakan disk arm ke silinder terkecil

Next cylinder                  Number of cylinder
accessed                       traversed
   20                             5
   10                            10
    0                            10
   35                            35
   45                            10
   50                             5
   65                            15
   80                            15
   85                             5
   90                             5
Total pergerakan head            115 silinder
```

Kelemahan lain dari algoritma SCAN adalah dapat menyebabkan permintaan lama menunggu pada kondisi-kondisi tertentu. Misalkan penyebaran banyaknya permintaan yang ada di queue tidak sama. Permintaan yang berada di depan arah pergerakan head sedikit sedangkan yang berada di ujung satunya lebih banyak. Maka head akan melayani permintaan yang lebih sedikit (sesuai arah pergerakannya)dan berbalik arah jika sudah sampai di ujung disk. Jika kemudian muncul permintaan baru, di dekat head yang terakhir, maka permintaan tersebut akan segera dilayani, sehingga permintaan yang lebih banyak yang berada di ujung silinder yang satunya akan semakin kelaparan. Jadi, mengapa head tidak melayani permintaan permintaan yang lebih banyak itu terlebih dahulu. Karena adanya kelemahan inilah maka tercipta satu modifikasi dari algoritma SCAN yaitu C-SCAN yang akan dibahas berikutnya.

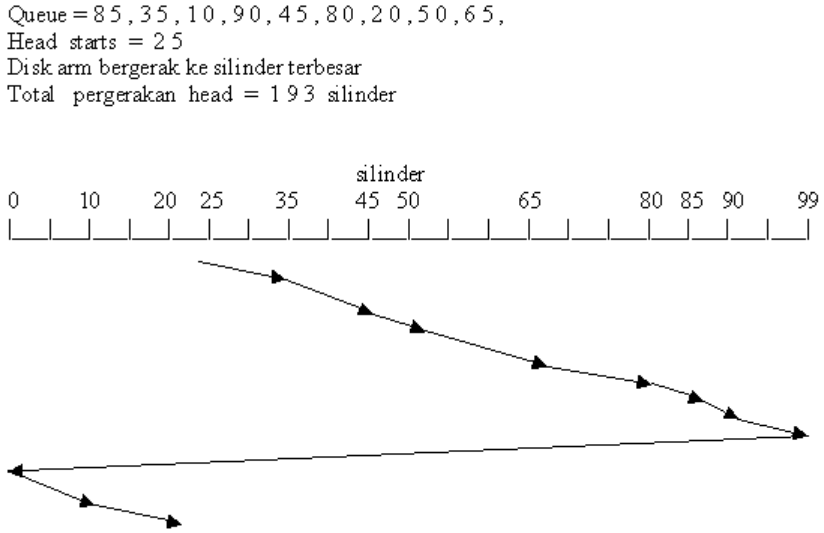
45.6. Penjadwalan C-SCAN

Algoritma C-SCAN atau Circular SCAN merupakan hasil modifikasi dari SCAN untuk mengurangi kemungkinan banyak permintaan yang lama menunggu untuk dilayani. Perbedaan yang paling

mendasar dari kedua algoritma ini adalah pada behavior saat pergerakan head yang berbalik arah setelah sampai di ujung disk. Pada C-SCAN, saat head sudah berada di silinder terujung pada disk, head akan berbalik arah dan bergerak secepatnya menuju silinder di ujung disk yang satu lagi, tanpa melayani permintaan yang dilalui dalam pergerakannya. Sedangkan pada SCAN dia tetap melayani permintaan saat bergerak berbalik arah menuju ujung yang lain.

Untuk contoh permintaan yang sama seperti SCAN, setelah head sampai di silinder 99 (permintaan silinder 35, 45, 50, 65, 80, 85 dan 90 telah dilayani secara berurutan), maka head akan secepatnya menuju silinder 0 tanpa melayani permintaan silinder 20 dan 10. Permintaan tersebut baru dilayani ketika head sudah berbalik arah lagi setelah mencapai silinder 0.

Gambar 45.4. CSCAN



Untuk lebih jelasnya perhatikan contoh berikut:

```
C-SCAN
(head awal di silinder 25)
Pergerakan disk arm ke silinder terbesar

Next cylinder                  Number of cylinder
accessed                       traversed
   35                                10
   45                                10
   50                                 5
   65                                15
   80                                15
   85                                 5
   90                                 5
   99                                 9
    0                                99
   10                                10
   20                                20
Total pergerakan head                193 silinder
```

Dengan sistem kerja yang seperti itu, terlihat bahwa head melayani permintaan hanya dalam satu arah pergerakan saja, saat head bergerak ke silinder terbesar atau saat bergerak ke silinder terkecil. Dan dengan sifatnya yang harus sampai silinder terujung terlebih dahulu sebelum bergerak berbalik arah, C-SCAN seperti halnya SCAN mempunyai ketidakefisienan untuk total pergerakan head.

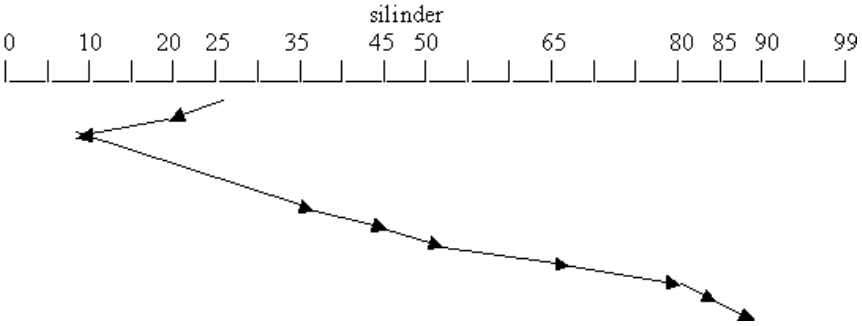
Untuk itulah dibuat algoritma LOOK yang akan dibahas berikut ini.

45.7. Penjadwalan LOOK

Algoritma LOOK adalah algoritma penjadwalan disk yang secara konsep hampir sama dengan algoritma SCAN. Sesuai dengan namanya, algoritma ini seolah-olah seperti dapat "melihat". Algoritma ini memperbaiki kelemahan SCAN dan C-SCAN dengan cara melihat apakah di depan arah pergerakannya masih ada permintaan lagi atau tidak. Bedanya pada algoritma LOOK, disk arm tidak berjalan sampai ujung disk, tetapi disk arm hanya berjalan sampai pada permintaan yang paling dekat dengan ujung disk, setelah melayani permintaan tersebut, disk arm akan berbalik arah dari arah pergerakannya yang pertama dan berjalan sambil melayani permintaan-permintaan yang ada didepannya sesuai dengan arah pergerakannya.

Gambar 45.5. LOOK

Queue = 85, 35, 10, 90, 45, 80, 20, 50, 65,
Head starts = 25
Disk arm bergerak ke siinder dengan nomor kecil
Total pergerakan head = 95 silinder



Untuk lebih jelasnya perhatikan contoh berikut:

```
LOOK
(head awal di silinder 25)
Pergerakan disk arm menuju ke silinder dengan nomor yang
lebih kecil (yaitu ke sebelah kiri)

Next  cylinder                Number of cylinder
accessed                      traversed
   20                               5
   10                              10
   35                              25
   45                              10
   50                               5
   65                              15
   80                              15
   85                               5
   90                               5
Total pergerakan head              95 silinder
```

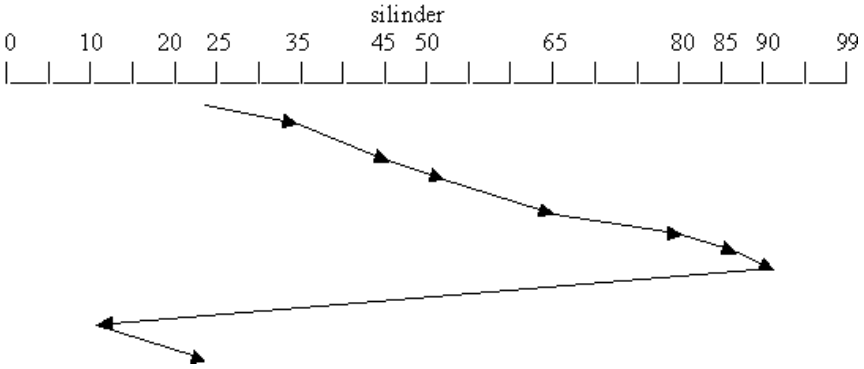
45.8. Penjadwalan C-LOOK

Algoritma ini berhasil memperbaiki kelemahan-kelemahan algoritma SCAN, C-SCAN, dan LOOK.

Algoritma C-LOOK memperbaiki kelemahan LOOK sama seperti algoritma C-SCAN memperbaiki kelemahan SCAN, yaitu pada cara pergerakan *disk arm* setelah mencapai silinder yang paling ujung. Algoritma C-LOOK adalah algoritma penjadwalan disk yang secara konsep hampir sama dengan algoritma C-SCAN. Bedanya pada algoritma C-LOOK, disk arm tidak berjalan sampai ujung disk, tetapi disk arm hanya berjalan sampai pada permintaan yang paling dekat dengan ujung disk, setelah melayani permintaan tersebut, disk arm akan berbalik arah dari arah pergerakannya yang pertama dan langsung berjalan ke permintaan yang paling dekat dengan ujung disk yang lain kemudian melayani permintaan tersebut. Setelah selesai melayani permintaan tersebut, disk arm akan berbalik arah kembali dan melayani permintaan-permintaan lain yang ada didepannya sesuai dengan arah pergerakannya.

Gambar 45.6. CLOOK

Queue = 85, 35, 10, 90, 45, 80, 20, 50, 65,
Head starts = 25
Disk arm bergerak ke silinder dengan nomor besar
Total pergerakan head = 155 silinder



Untuk lebih jelasnya perhatikan gambar berikut:

C-LOOK
(head awal di silinder 25)
Pergerakan disk arm ke silinder terkecil

Next cylinder accessed	Number of cylinder traversed
35	10
45	10
50	5
65	15
80	15
85	5
90	5
10	80
20	10
Total pergerakan head	155 silinder

Dari gambar tersebut, bisa dilihat bahwa disk arm bergerak dari permintaan 25 ke kanan sambil melayani permintaan-permintaan yang ada didepannya yang sesuai dengan arah pergerakannya yaitu permintaan 35, 45, 50, 65, 80, 85 sampai pada permintaan 90. Setelah melayani permintaan 90, disk arm berbalik arah dan langsung menuju ke permintaan 10 (karena permintaan 10 adalah permintaan yang letaknya paling dekat dengan ujung disk 0), setelah melayani permintaan 10, disk arm berbalik arah kembali dan melayani permintaan-permintaan yang ada didepannya sesuai dengan arah pergerakannya yaitu ke permintaan 20.

Catatan:

Arah pergerakan disk arm yang bisa menuju dua arah pada algoritma SCAN, C-SCAN, LOOK dan C-LOOK, menuju silinder terbesar atau terkecil diatur oleh hardware controller, hal ini membuat pengguna tidak bisa menentukan kemana disk arm bergerak.

45.9. Pemilihan Algoritma Penjadwalan *Disk*

Performa dari suatu sistem sebenarnya tidak terlalu bergantung pada algoritma penjadwalan yang kita pakai, karena yang paling mempengaruhi kinerja dari suatu sistem adalah jumlah dan tipe dari permintaan. Dan tipe permintaan juga sangat dipengaruhi oleh metoda pengalokasian file, lokasi direktori dan indeks blok. Karena kompleksitas ini, sebaiknya algoritma penjadwalan disk diimplementasikan sebagai modul yang terpisah dari OS, sehingga algoritma tersebut bisa diganti atau di-replace dengan algoritma lain sesuai dengan kondisi jumlah dan tipe permintaan yang ada. Namun biasanya, OS memiliki algoritma default yang sering dipakai, yaitu algoritma SSTF dan LOOK.

Penerapan algoritma penjadwalan diatas didasarkan hanya pada jarak pencarian saja. Tapi, untuk disk modern, selain jarak pencarian, rotational latency juga sangat berpengaruh, tetapi algoritma untuk mengurangi rotational latency tidak dapat diterapkan oleh OS, karena pada disk modern tidak dapat diketahui lokasi fisik dari blok-blok logikanya. Tapi masalah rotational latency ini dapat dihandle dengan mengimplementasikan algoritma penjadwalan disk pada hardware controller yang terdapat dalam disk drive.

Dari seluruh algoritma yang sudah kita bahas di atas, tidak ada algoritma yang terbaik untuk semua keadaan yang terjadi. SSTF lebih umum dan memiliki prilaku yang lazim kita temui. SCAN dan C-SCAN memperlihatkan kemampuan yang lebih baik bagi sistem yang menempatkan beban pekerjaan yang berat kepada *disk*, karena algoritma tersebut memiliki masalah *starvation* yang paling sedikit. SSTF dan LOOK sering dipakai sebagai algoritma dasar pada sistem operasi.

Dengan algoritma penjadwalan yang mana pun, kinerja sistem sangat tergantung pada jumlah dan tipe permintaan. Sebagai contoh, misalnya kita hanya memiliki satu permintaan, maka semua algoritma penjadwalan akan dipaksa bertindak sama. Sedangkan permintaan sangat dipengaruhi oleh metode penempatan berkas. Karena kerumitan inilah, maka algoritma penjadwalan *disk* harus ditulis dalam modul terpisah dari sistem operasi, jadi dapat saling mengganti dengan algoritma lain jika diperlukan.

Namun perlu diingat bahwa algoritma-algoritma di atas hanya mempertimbangkan jarak pencarian, sedangkan untuk *disk* modern, *rotational latency* dari *disk* sangat menentukan. Tetapi sangatlah sulit jika sistem operasi harus memperhitungkan algoritma untuk mengurangi *rotational latency* karena *disk* modern tidak memperlihatkan lokasi fisik dari blok-blok logikanya. Oleh karena itu para produsen *disk* telah mengurangi masalah ini dengan mengimplementasikan algoritma penjadwalan *disk* di dalam pengendali perangkat keras, sehingga kalau hanya kinerja M/K yang diperhatikan, maka sistem operasi dapat menyerahkan algoritma penjadwalan *disk* pada perangkat keras itu sendiri.

45.10. Rangkuman

Bentuk penulisan disk drive modern adalah array blok logika satu dimensi yang besar. Ukuran blok logika dapat bermacam-macam. Array adalah blok logika satu dimensi yang dipetakan dari disk ke sektor secara bertahap dan berurut. Terdapat dua aturan pemetaan, yaitu:

1. **CLV: *Constant Linear Velocity*.** Kepadatan bit setiap track sama, semakin jauh sebuah track dari tengah disk, maka semakin besar panjangnya, dan juga semakin banyak sektor yang dimilikinya. Digunakan pada CD-ROM dan DVD-ROM.
2. **CAV: Constant Angular Velocity.** Kepadatan bit dari zona terdalam ke zona terluar semakin berkurang, kecepatan rotasi konstan sehingga aliran data pun konstan.

Penjadwalan disk sangat penting dalam hal meningkatkan efisiensi penggunaan perangkat keras. Efisiensi penggunaan disk terkait dengan kecepatan waktu akses dan besarnya disk bandwidth. Untuk meningkatkan efisiensi tersebut dibutuhkan algoritma penjadwalan yang tepat dalam penggunaan disk.

Terdapat berbagai macam algoritma penjadwalan disk, yaitu:
1. FCFS (First Come First Serve)
2. SSTF (Shortest-Seek-Time-First)
3. SCAN
4. C-SCAN (Circular SCAN)
5. LOOK
6. C-LOOK (Circular LOOK)

Untuk algoritma di SCAN, C-SCAN, LOOK dan C-LOOK, yang mengatur arah pergerakan disk arm adalah hardware controller. Hal ini membuat pengguna tidak terlibat di dalamnya.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

Bab 46. Manajemen Disk II

46.1. Pendahuluan

Beberapa aspek yang termasuk aspek penting dalam Manajemen Disk.

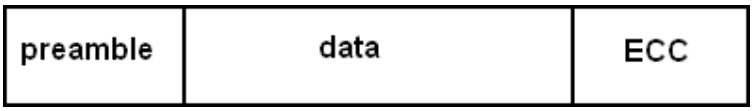
46.2. Komponen Disk

Format Disk

Disk adalah salah satu tempat penyimpanan data. Sebelum sebuah disk dapat digunakan, disk harus dibagi-bagi dalam beberapa sektor. Sektor-sektor ini yang kemudian akan dibaca oleh pengendali. Pembentukan sektor-sektor ini disebut *low level formatting* atau *physical formatting*. *Low level formatting* juga akan mengisi disk dgn beberapa struktur data penting seperti *header* dan *trailer*. *Header* dan *trailer* mempunyai informasi seperti nomor sektor, dan *Error Correcting Code* (ECC). ECC ini berfungsi sebagai *correcting code* karena mempunyai kemampuan untuk mendeteksi bit yang salah, menghitung nilai yang benar dan kemudian mengubahnya. Ketika proses penulisan, ECC di-update dengan menghitung bit di area data. Pada proses pembacaan, ECC dihitung ulang dan dicocokan dengan nilai ECC yang tersimpan saat penulisan. Jika nilainya berbeda maka dipastikan ada sektor yang terkorup.

Agar dapat menyimpan data, OS harus menyimpan struktur datanya dalam disk tersebut. Proses itu dilakukan dalam dua tahap, yaitu partisi dan *logical formatting*. Partisi akan membagi disk menjadi beberapa silinder yang dapat diperlakukan secara independen. *Logical formatting* akan membentuk sistem berkas disertai pemetaan disk. Terkadang sistem berkas ini dirasakan menggangu proses alokasi suatu data, sehingga diadakan sistem partisi lain yang tidak mengikutkan pembentukan sistem berkas, disebut *raw disk*.

Gambar 46.1. Format Sektor



Walaupun ukuran sektor dapat diatur pada proses formatting, namun kebanyakan disk memiliki sektor dengan ukuran 512 byte dengan alasan bahwa beberapa OS hanya dapat menangani ukuran sektor 512. Namun dimungkinkan juga untuk memformat sektor dengan ukuran misalnya 256 bytes atau 1024 bytes. Jika ukuran sektor semakin besar, itu artinya semakin sedikit jumlah sektor yang dapat dibuat, begitu pun sebaliknya. Bagian preamble mengandung bit-bit yang berisi informasi yang akan dikenali oleh hardware misalnya tentang informasi nomor silinder atau sektor.

Proses formatting sendiri sebenarnya terdiri dari dua proses utama, yaitu partisi dan logical formatting. Proses partisi akan membagi disk menjadi beberapa silinder sehingga silinder-silinder tersebut akan diperlakukan secara independen seolah- olah mereka adalah disk yang saling berbeda. Sedangkan proses logical formatting akan membentuk sebuah berkas system beserta pemetaan disk.

Perlu diingat bahwa beberapa OS memberlakukan berkas system khusus yang berbeda satu sama lain sehingga mungkin saja dirasakan mengganggu proses alokasi data. Hal ini dapat disiasati dengan sistem partisi lain yang tidak mengikutkan pembentukan file system, yang disebut raw disk (raw partition). Sebagai contoh, pada Windows XP ketika kita membuat sebuah partisi tanpa menyertakan suatu berkas system ke dalamnya maka partisi tersebut dapat disebut sebagai raw partition.

Boot Block

Saat sebuah komputer dijalankan, sistem akan mencari sebuah *initial program* yang akan memulai segala sesuatunya. *Initial program*-nya (*initial bootstrap*) bersifat sederhana dan akan menginisialisasi seluruh aspek yang diperlukan bagi komputer untuk beroperasi dengan baik seperti CPU registers, controller, dan yang terakhir adalah Sistem Operasinya.

Pada kebanyakan komputer, *bootstrap* disimpan di ROM (*read only memory*) karena letaknya yang tetap dan dapat langsung dieksekusi ketika pertama kali listrik dijalankan. Letak *bootstrap* di ROM juga menguntungkan karena sifatnya yang read only memungkinkan dia untuk tidak terinfeksi virus. Untuk melakukan tugasnya, *bootstrap* mencari *kernel* di disk dan me-*load kernel* ke memori dan kemudian loncat ke *initial address* untuk memulai eksekusi OS.

Untuk alasan praktis, *bootstrap* sering dibuat berbentuk kecil (*tiny loader*) dan diletakkan di ROM, yang kemudian akan me-*load full bootstrap* dari disk bagian disk yang disebut *boot block*. Perubahan menjadi bentuk simple ini bertujuan jika diadakan perubahan pada *bootstrap*, maka struktur ROM tidak perlu dirubah semuanya.

Konsep boot block sangat erat kaitannya dengan proses booting pada sebuah komputer. Ketika komputer dinyalakan, sistem akan mencari sebuah initial program yang akan memulai segala sesuatu yang berhubungan dengan proses booting. Program ini dikenal sebagai initial bootstrap dan akan menginisialisasi seluruh aspek yang diperlukan bagi komputer untuk beroperasi dengan baik seperti CPU registers, hardware controller, dan OS-nya.

Pada Pentium dan kebanyakan komputer, MBR terletak pada sektor 0 dan mengandung beberapa boot code beserta tabel partisi. Tabel partisi ini mengandung berbagai informasi misalnya letak sektor pertama pada tiap partisi dan ukuran partisi. Agar bisa melakukan boot dari hard disk, salah satu partisi yang terdaftar pada tabel partisi harus ditandai sebagai active partition.

Boot block tidak selalu mengandung kernel. Bisa saja ia berupa sebuah boot loader, misalnya LILO (Linux Loader) atau GRUB (GRand Unified Bootloader).

Contoh konfigurasi sebuah boot loader misalnya dapat ditemukan pada Linux Ubuntu 5.04 yang berada pada file "/boot/grub/menu.lst". Sedangkan pada sistem Windows XP, konfigurasinya dapat ditemukan pada berkas "C:\boot.ini" dengan catatan bahwa "C:\" adalah volume atau partisi yang di-set sebagai active partition.

Bad Block

Bad block adalah satu atau lebih sektor yang cacat atau rusak. Kerusakan ini dapat dapat diakibatkan karena kerentanan disk jika sering dipindah-pindah atau kemasukan benda asing. Akibatnya, data di dalam bad block menjadi tidak terbaca.

Pada disk sederhana seperti IDE controller, bad block akan ditangani secara manual seperti dengan perintah format pada MS-DOS yang akan mencari bad block dan menulis nilai spesial ke FAT entry agar tidak mengalokasikan branch routine ke blok tersebut. SCSI mengatasi bad block dengan cara yang lebih baik. Daftar bad block-nya dipertahankan oleh controller saat low-level formatting, dan terus diperbaharui selama disk itu digunakan. Low level formatting akan memindahkan bad sector itu ke tempat lain yang kosong dengan algoritma sector sparing (sector forwarding). Sector sparing dijalankan dengan cara ECC mendeteksi bad sector dan melaporkannya ke OS, sehingga saat sistem dijalankan sekali lagi, controller akan menggantikan bad sector tersebut dengan sektor kosong. Lain halnya dengan algoritma sector slipping. Ketika sebuah bad sector terdeteksi, sistem akan meng-copy semua isi sektor ke sektor selanjutnya secara bertahap satu per satu sampai ditemukan sektor kosong. Misal bad sector ditemukan di sektor 7 dan sektor-sektor yang kosong selanjutnya hanyalah sektor 30 dan seterusnya. Maka isi sektor 30 akan di-copy ke ruang kosong sementara itu sektor 29 ke sektor 30, 28 ke 29, 27 ke 28 dan seterusnya hingga sektor 8 ke sektor 9. Dengan demikian, sektor 8 menjadi kosong sehingga sektor 7 (bad sector) dapat di-map ke sektor 8.

Perlu diingat bahwa kedua algoritma di atas juga tidak selalu berhasil karena data yang telah tersimpan di dalam bad block biasanya sudah tidak bisa terbaca.

46.3. Manajemen Ruang *Swap*

Manajemen ruang *swap* adalah salah satu *low level task* dari OS. Memori virtual menggunakan ruang disk sebagai perluasan dari memori utama. Tujuan utamanya adalah untuk menghasilkan output yang baik. Namun di lain pihak, penggunaan disk akan memperlambat akses karena akses dari memori jauh lebih cepat.

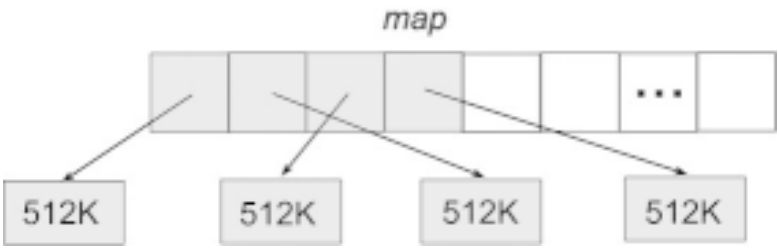
Penggunaan Ruang *Swap*

Ruang swap digunakan dalam beberapa cara tergantung penerapan algoritma. Sebagai contoh, sistem yang menggunakan swapping dapat menggunakan ruang swap untuk memegang seluruh proses pemetaan termasuk data dan segmen. Jumlah dari ruang swap yang dibutuhkan tergantung dari jumlah memori fisik, jumlah dari memori virtual yang dijalankan, cara penggunaan memori virtual tersebut. Beberapa sistem operasi seperti UNIX menggunakan banyak ruang swap, yang biasa diletakkan pada disk terpisah. Ketika kita akan menentukan besarnya ruang swap, sebaiknya kita tidak terlalu banyak atau terlalu sedikit. Namun perlu diketahui bahwa akan lebih aman jika mengalokasikan lebih banyak ruang swap. Jika sistem dijalankan dan ruang swap terlalu sedikit, maka proses akan dihentikan dan mungkin akan merusak sistem. Sebaliknya jika terlalu banyak juga akan mengakibatkan lambatnya akses dan pemborosan ruang disk, tetapi hal itu tidak menimbulkan resiko terhentinya proses. Sebagai contoh, Linux memperbolehkan penggunaan banyak ruang swap yang tersebar pada disk terpisah.

Lokasi Ruang *Swap*

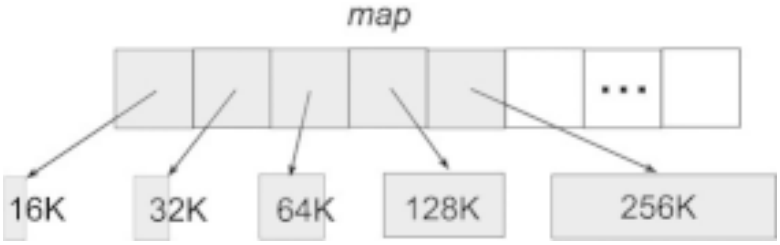
Ruang swap dapat diletakkan di sistem berkas normal atau dapat juga berada di partisi yang terpisah. Beberapa OS (Linux misalnya) dapat memiliki dua ruang swap sekaligus, yakni pada berkas biasa dan partisi terpisah. Jika ruang swap berukuran besar dan diletakkan di sistem berkas normal, routine-nya dapat menciptakan, menamainya dan menentukan besar ruang. Walaupun lebih mudah dijalankan, cara ini cenderung tidak efisien. Pengaksesannya akan sangat memakan waktu dan akan meningkatkan fragmentasi karena pencarian data yang berulang terus selama proses baca atau tulis.

Gambar 46.2. Manajemen Ruang Swap: Pemetaan Swap Segmen Teks 4.3 BSD



Ruang swap yang diletakkan di partisi disk terpisah (raw partition), menggunakan manajer ruang swap terpisah untuk melakukan pengalokasian ruang. Manajer ruang swap tersebut menggunakan algoritma yang mengutamakan peningkatan kecepatan daripada efisiensi. Walaupun fragmentasi masih juga terjadi, tapi masih dalam batas-batas toleransi mengingat ruang swap sangat sering diakses. Dengan partisi terpisah, alokasi ruang swap harus sudah pasti. Proses penambahan besar ruang swap dapat dilakukan hanya dengan partisi ulang atau penambahan dengan lokasi yang terpisah.

Gambar 46.3. Manajemen Ruang Swap: Pemetaan Swap Segmen Data 4.3 BSD



46.4. Struktur RAID

Disk memiliki resiko untuk mengalami kerusakan. Kerusakan ini dapat berakibat turunnya kinerja atau pun hilangnya data. Meski pun terdapat backup data, tetap saja ada kemungkinan data yang hilang karena adanya perubahan yang terjadi setelah terakhir kali data di-backup dan belum sempat untuk dibackup kembali. Karenanya reliabilitas dari suatu disk harus dapat terus ditingkatkan. Selain itu perkembangan kecepatan CPU yang begitu pesat mendorong perlunya peningkatan kecepatan kinerja disk karena jka tidak kecepatan CPU yang besar itu akan menjadi sia-sia.

Berbagai macam cara dilakukan untuk meningkatkan kinerja dan juga reliabilitas dari disk. Biasanya untuk meningkatkan kinerja, dilibatkan banyak disk sebagai satu unit penyimpanan. Tiap-tiap blok data dipecah ke dalam beberapa subblok, dan dibagi-bagi ke dalam disk-disk tersebut (striping). Ketika mengirim data disk-disk tersebut bekerja secara paralel, sehingga dapat meningkatkan kecepatan transfer dalam membaca atau menulis data. Ditambah dengan sinkronisasi pada rotasi masing- masing disk, maka kinerja dari disk dapat ditingkatkan. Cara ini dikenal sebagai RAID -- Redundant Array of Independent (atau Inexpensive) Disks. Selain masalah kinerja RAID juga dapat meningkatkan reliabilitas dari disk dengan jalan menggunakan disk tambahan (redundant) untuk menyimpan paritas bit/blok ataupun sebagai mirror dari disk-disk data yang ada.

Tiga karakteristik umum dari RAID ini, yaitu [Stallings2001]:
1. RAID adalah sebuah sebuah set dari beberapa physical drive yang dipandang oleh sistem operasi sebagai sebuah logical drive.
2. Data didistribusikan ke dalam array dari beberapa *physical drive*.
3. Kapasitas disk yang berlebih digunakan untuk menyimpan informasi paritas, yang menjamin data dapat diperbaiki jika terjadi kegagalan pada salah satu disk.

Peningkatan Kehandalan dan Kinerja

Peningkatan Kehandalan dan Kinerja dari disk dapat dicapai melalui dua cara:
1. **Redundansi.** Peningkatan kehandalan disk dapat dilakukan dengan redundansi, yaitu menyimpan informasi tambahan yang dapat dipakai untuk membentuk kembali informasi yang hilang jika suatu disk mengalami kegagalan. Salah satu teknik untuk redundansi ini adalah dengan cara mirroring atau shadowing, yaitu dengan membuat duplikasi dari tiap-tiap disk. Jadi, sebuah disk logical terdiri dari 2 disk physical, dan setiap penulisan dilakukan pada kedua disk, sehingga jika salah satu disk gagal, data masih dapat diambil dari disk yang lainnya, kecuali jika disk kedua gagal sebelum kegagalan pada disk pertama diperbaiki. Pada cara ini, berarti diperlukan media penyimpanan yang dua kali lebih besar daripada ukuran data sebenarnya. Akan tetapi, dengan cara ini pengaksesan disk yang dilakukan untuk membaca menjadi 2 kali karena pembacaan bisa di bagi ke kedua disk. Cara lain yang digunakan adalah paritas blok interleaved, yaitu menyimpan blok-blok data pada beberapa disk dan blok paritas pada sebuah (atau sebagian kecil) disk. Dengan cara ini maka pengaksesan sebuah blok data akan sama saja, tetapi pengaksesan beberapa blok data bisa lebih cepat karena dapat diakses secara paralel.
2. **Paralelisme.** Peningkatan kinerja dapat dilakukan dengan mengakses banyak disk secara paralel. Pada *disk mirroring*, di mana pengaksesan disk untuk membaca data menjadi dua kali lipat karena permintaan dapat dilakukan pada kedua disk, tetapi kecepatan transfer data pada setiap disk tetap sama. Kita dapat meningkatkan kecepatan transfer ini dengan cara melakukan

data *striping* ke dalam beberapa disk. Data *striping*, yaitu menggunakan sekelompok disk sebagai satu kesatuan unit penyimpanan, menyimpan bit data dari setiap *byte* secara terpisah pada beberapa disk (paralel).

Level RAID

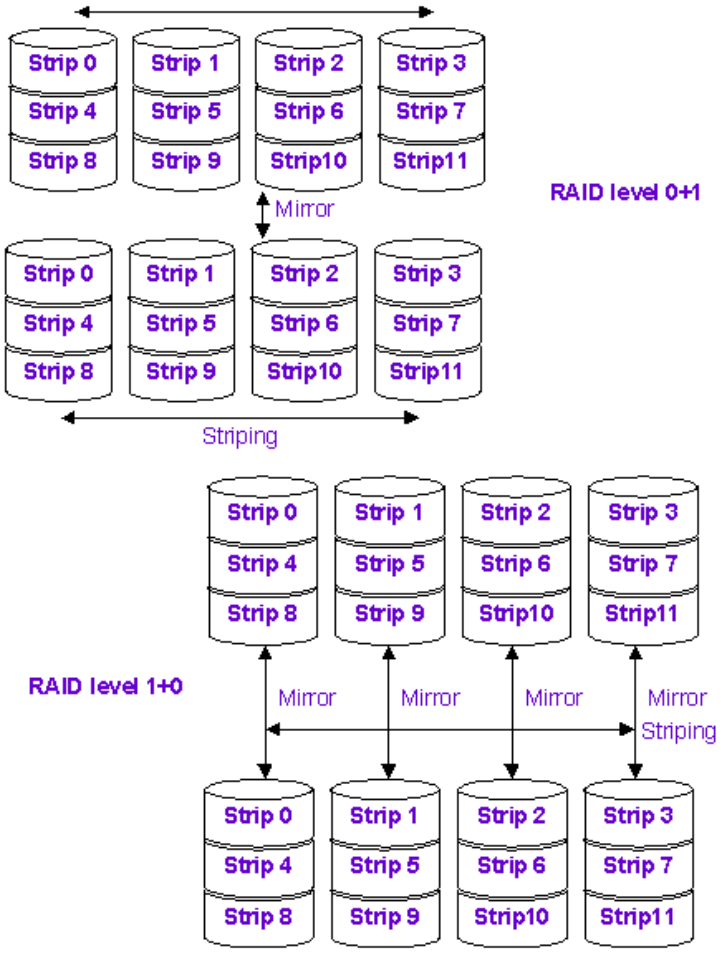
RAID terdiri dapat dibagi menjadi enam level yang berbeda:

1. **RAID level 0.** Menggunakan kumpulan disk dengan striping pada level blok, tanpa redundansi. Jadi hanya melakukan striping blok data ke dalam beberapa disk. Kelebihan dari level ini antara lain akses beberapa blok bisa dilakukan secara parallel sehingga bisa lebih cepat. Kekurangannya antara lain akses per blok sama saja seperti biasa tidak ada peningkatan, kehandalan kurang karena tidak adanya pemback-upan data dengan redundancy. Berdasarkan definisi RAID sebagai redundancy array maka level ini sebenarnya tidak termasuk ke dalam kelompok RAID karena tidak menggunakan redundansi untuk peningkatan kinerjanya.
2. **RAID level 1.** Merupakan disk mirroring, menduplikat setiap disk tanpa striping. Cara ini dapat meningkatkan kinerja disk, tetapi jumlah disk yang dibutuhkan menjadi dua kali lipat. Kelebihannya antara lain memiliki kehandalan (reliabilitas) yang baik karena memiliki back-up untuk tiap disk dan perbaikan disk yang rusak dapat dengan cepat dilakukan karena ada mirrornya. Kekurangannya antara lain biaya yang menjadi sangat mahal karena memerlukan jumlah disk 2 kali lipat dari kebutuhan sebenarnya.
3. **RAID level 2.** Merupakan pengorganisasian dengan error-correcting-code (ECC). Seperti pada memori di mana pendeteksian terjadinya error menggunakan paritas bit. Sebagai contoh, misalnya setiap byte data mempunyai sebuah paritas bit yang bersesuaian yang merepresentasikan jumlah bit "1" di dalam byte data tersebut di mana paritas bit=0 jika jumlah bit genap atau paritas=1 jika ganjil. Jadi, jika salah satu bit pada data berubah, paritas berubah dan tidak sesuai dengan paritas bit yang tersimpan. Dengan demikian, apabila terjadi kegagalan pada salah satu disk, data dapat dibentuk kembali dengan membaca error-correction bit pada disk lain. Kelebihannya antara lain kehandalan yang bagus karena dapat membentuk kembali data yang rusak dengan ECC tadi dan jumlah bit redundancy yang diperlukan lebih sedikit dibanding level 1 (mirroring). Kelemahannya antara lain perlu adanya perhitungan paritas bit sehingga untuk penulisan/perubahan data memerlukan waktu yang lebih lama dibandingkan dengan yang tanpa menggunakan paritas bit, level ini memerlukan disk khusus untuk penerapannya yang harganya cukup mahal
4. **RAID level 3.** Merupakan pengorganisasian dengan paritas bit interleaved. Pengorganisasian ini hampir sama dengan RAID level 2, perbedaannya adalah RAID level 3 ini hanya memerlukan sebuah disk redundan, berapa pun jumlah kumpulan disk-nya, hal ini dapat dilakukan karena disk controller dapat memeriksa apakah sebuah sector itu dibaca dengan benar atau tidak (mengalami kerusakan atau tidak). Jadi tidak menggunakan ECC, melainkan hanya menggunakan sebuah bit paritas untuk sekumpulan bit yang mempunyai posisi yang sama pada setiap disk yang berisi data. Selain itu juga menggunakan data striping dan mengakses disk-disk secara paralel. Kelebihannya antara lain kehandalan (reliabilitas) bagus, akses data lebih cepat karena pembacaan tiap bit dilakukan pada beberapa disk (paralel), hanya butuh 1 disk redundan yang tentunya lebih menguntungkan dibandingkan level 1 dan 2. kelemahannya antara lain perlu adanya perhitungan dan penulisan parity bit akibatnya performanya lebih rendah dibanding yang tidak menggunakan paritas.
5. **RAID level 4.** Merupakan pengorganisasian dengan paritas blok interleaved, yaitu menggunakan striping data pada level blok, menyimpan sebuah paritas blok pada sebuah disk yang terpisah untuk setiap blok data pada disk-disk lain yang bersesuaian. Jika sebuah disk gagal, blok paritas tersebut dapat digunakan untuk membentuk kembali blok-blok data pada disk yang gagal tadi. Kelebihannya antara lain sama seperti pada level 0 yaitu akses ke beberapa blok data bisa lebih cepat karena bisa parallel dan kehandalannya juga bagus karena adanya paritas blok. Kelemahannya antara lain akses perblok seperti biasa seperti pengunaan 1 disk, bahkan untuk penulisan ke 1 blok memerlukan 4 pengaksesan disk, yaitu 2 pengaksesan untuk membaca ke disk data yang bersangkutan dan paritas disk, dan 2 lagi untuk penulisan ke 2 disk itu pula (read-modify-write).
6. **RAID level 5.** Merupakan pengorganisasian dengan paritas blok interleaved tersebar. Data dan paritas disebar pada semua disk termasuk sebuah disk tambahan. Pada setiap blok, salah satu dari disk menyimpan paritas dan disk yang lainnya menyimpan data. Sebagai contoh, jika terdapat kumpulan dari 5 disk, paritas blok ke n akan disimpan pada disk (n mod 5) + 1, blok ke n dari empat disk yang lain menyimpan data yang sebenarnya dari blok tersebut. Sebuah paritas blok tidak disimpan pada disk yang sama dengan blok-blok data yang bersangkutan, karena kegagalan disk tersebut akan menyebabkan data hilang bersama dengan paritasnya dan data tersebut tidak

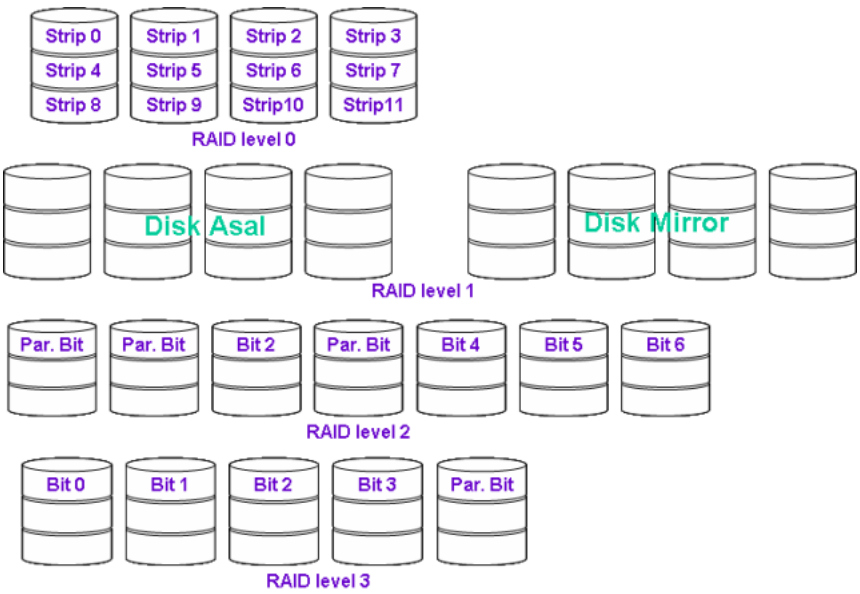
dapat diperbaiki. Kelebihannya antara lain seperti pada level 4 ditambah lagi dengan penyebaran paritas seperti ini dapat menghindari penggunaan berlebihan dari sebuah paritas disk seperti pada RAID level 4. Kelemahannya antara lain perlunya mekanisme tambahan untuk penghitungan lokasi dari paritas sehingga akan mempengaruhi kecepatan dalam pembacaan blok maupun penulisannya.

7. **RAID level 6.** Disebut juga redundansi P+Q, seperti RAID level 5, tetapi menyimpan informasi redundan tambahan untuk mengantisipasi kegagalan dari beberapa disk sekaligus. RAID level 6 melakukan dua perhitungan paritas yang berbeda, kemudian disimpan di dalam blok-blok yang terpisah pada disk-disk yang berbeda. Jadi, jika disk data yang digunakan sebanyak n buah disk, maka jumlah disk yang dibutuhkan untuk RAID level 6 ini adalah n+2 disk. Keuntungan dari RAID level 6 ini adalah kehandalan data yang sangat tinggi, karena untuk menyebabkan data hilang, kegagalan harus terjadi pada tiga buah disk dalam interval rata-rata untuk perbaikan data *Mean Time To Repair* (MTTR). Kerugiannya yaitu penalti waktu pada saat penulisan data, karena setiap penulisan yang dilakukan akan mempengaruhi dua buah paritas blok.
8. **RAID level 0+1 dan 1+0.** Ini merupakan kombinasi dari RAID level 0 dan 1. RAID level 0 memiliki kinerja yang baik, sedangkan RAID level 1 memiliki kehandalan. Namun, dalam kenyataannya kedua hal ini sama pentingnya. Dalam RAID 0+1, sekumpulan disk di-strip, kemudian strip tersebut di-mirror ke disk-disk yang lain, menghasilkan strip- strip data yang sama. Kombinasi lainnya yaitu RAID 1+0, di mana disk-disk di-mirror secara berpasangan, dan kemudian hasil pasangan mirrornya di-strip. RAID 1+0 ini mempunyai keuntungan lebih dibandingkan dengan RAID 0+1. Sebagai contoh, jika sebuah disk gagal pada RAID 0+1, seluruh strip-nya tidak dapat diakses, hanya strip dari mirrornya saja yang dapat diakses, sedangkan pada RAID 1+0, disk yang gagal tersebut tidak dapat diakses tetapi pasangan stripnya yang lain masih bisa, dan pasangan mirror-nya masih dapat diakses untuk menggantikannya sehingga disk-disk lain selain yang rusak masih bisa digunakan.

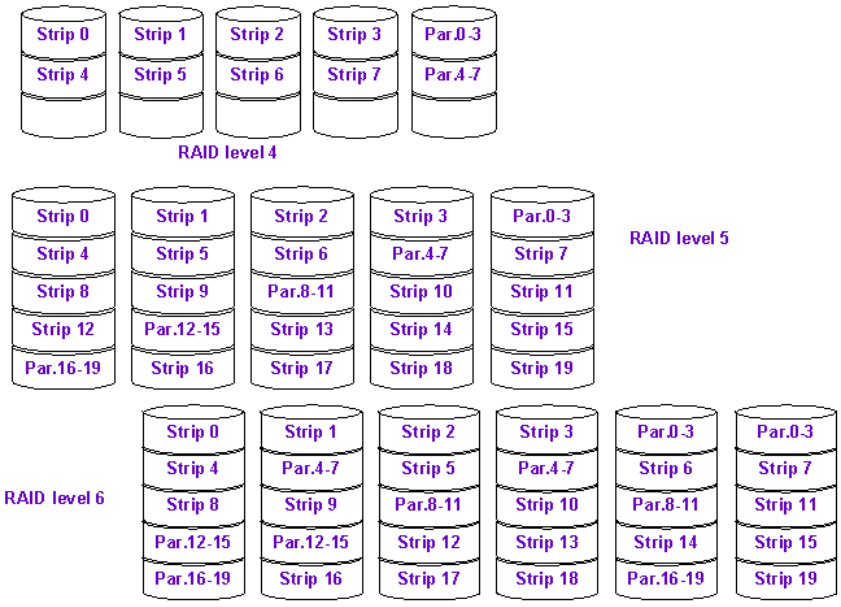
Gambar 46.4. RAID 0 + 1 dan 1 + 0



Gambar 46.5. RAID level 0, 1, 2, 3



Gambar 46.6. RAID level 4, 5, 6



46.5. *Host-Attached Storage*

Host attached storage merupakan sistem penyimpanan yang terhubung langsung dengan suatu komputer. Ada beberapa macam interface yang menghubungkan storage dengan komputer. Yang pertama adalah IDE atau ATA. Interface ini umumnya digunakan pada komputer rumahan, karenaharganya yang relatif murah. Interface yang kedua adalah SCSI. Biasa digunakan untuk server. Lebih cepat daripada interface IDE namun lebih mahal. Selanjutnya adalah fibre channel. Memiliki kecepatan yang sangat tinggi dan sering digunakan untuk *Storage-Area Network* (SAN), yang akan dibahas kemudian.

Dalam implementasinya pada jaringan, Host-Attached storage disebut juga Server-Attached Storage karena sistem penyimpanannya terdapat didalam server itu.

46.6. NAS: *Network-Attached Storage*

Network-Attached Storage device

Network-attached storage (NAS) adalah suatu konsep penyimpanan bersama pada suatu jaringan. NAS berkomunikasi menggunakan *Network File Sistem* (NFS) untuk UNIX, *Common Internet File System* (CIFS) untuk Microsoft Windows, FTP, http, dan protokol networking lainnya. NAS membawa kebebasan platform dan meningkatkan kinerja bagi suatu jaringan, seolah-olah adalah suatu dipasang peralatan. NAS device biasanya merupakan *dedicated single-purpose machine*. NAS dimaksudkan untuk berdiri sendiri dan melayani kebutuhan penyimpanan yang spesifik dengan sistem operasi mereka dan perangkat keras/perangkat lunak yang terkait. NAS mirip dengan alat plug-and-play, akan tetapi manfaatnya adalah untuk melayani kebutuhan penyimpanan. NAS cocok digunakan untuk melayani network yang memiliki banyak *client*, *server*, dan operasi yang mungkin menangani task seperti *web cache* dan *proxy*, *firewall*, *audio-video streeming*, *tape backup*, dan penyimpanan data dengan *file serving*.

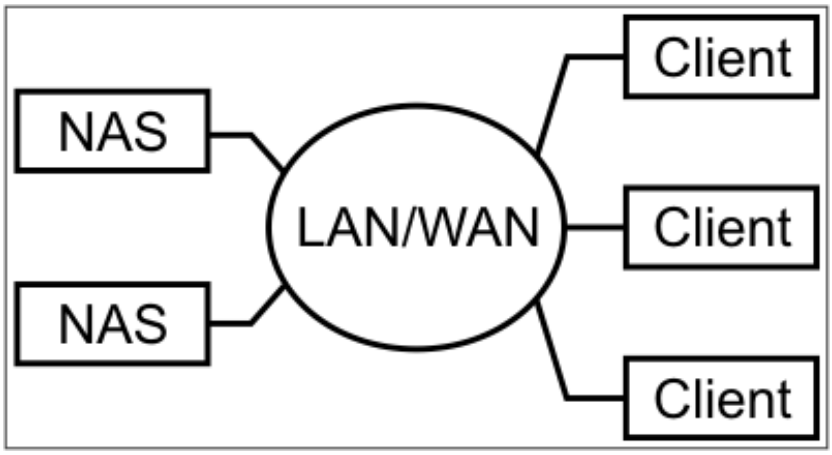
NAS: Memikirkan Pengguna Jaringan

NAS adalah *network-centric*. Biasanya digunakan Untuk konsolidasi penyimpanan client pada suatu LAN, NAS lebih disukai dalam solusi kapasitas penyimpanan untuk memungkinkan *client* untuk mengakses berkas dengan cepat dan secara langsung. Hal ini menghapuskan *bottleneck* user ketika mengakses berkas dari suatu *general-purpose server*.

NAS menyediakan keamanan dan melaksanakan semua berkas dan storage service melalui protokol standard network, menggunakan TCP/IP untuk transfer data, Ethernet Dan Gigabit Ethernet untuk media akses, dan CIFS, http, dan NFS untuk *remote file service*. Sebagai tambahan, NAS dapat melayani UNIX dan Microsoft Windows user untuk berbagi data yang sama antar arsitektur yang berbeda. Untuk user client, NAS adalah teknologi pilihan untuk menyediakan penyimpanan dengan akses *unen-cumbered* ke berkas.

Walaupun NAS menukar kinerja untuk manajemen dan kesederhanaan, hal ini bukanlah lazy technology. Gigabit ethernet mengijinkan NAS untuk melakukan kinerja yang tinggi serta latensi yang rendah, sehingga memungkinkan untuk mendukung banyak client pada suatu antarmuka tunggal. Banyak NAS device yang yang mendukung berbagai antarmuka dan dapat mendukung berbagai jaringan pada waktu yang sama.

Gambar 46.7. NAS



46.7. SAN: *Storage-Area Network*

SAN: Ruang Penyimpanan Komputer (*Back-End*)

Storage-Area Network (SAN) adalah suatu konsep penyimpanan yang bersifat data-centric, artinya SAN adalah jaringan khusus untuk transfer data. SAN terpisah dengan jaringan standar TCP/IP sehingga dapat menghindari hambatan yang terjadi pada TCP/IP. SAN biasa menggunakan fibre channel sebagai interface untuk meningkatkan kinerja dan mengurangi latency. Tidak seperti NAS yang bisa diakses oleh semua komputer yang terhubung dengan jaringan, SAN hanya bisa diakses oleh server yang terhubung langsung dengannya melalui fibre channel.

SAN menggunakan *gateway*, *switch*, dan *router* untuk memudahkan pergerakan data antar sarana penyimpanan dan server yang heterogen. Ini mengijinkan untuk menghubungkan kedua jaringan dan potensi untuk *semi-remote storage* (memungkinkan hingga jarak 10km) ke *storage manajement effort*. Arsitektur SAN optimal untuk memindahkan *storage block*. Di dalam ruang komputer, SAN adalah pilihan yang lebih disukai untuk menujukan isu *bandwidth* dan data aksesibilitas seperti halnya untuk menangani konsolidasi.

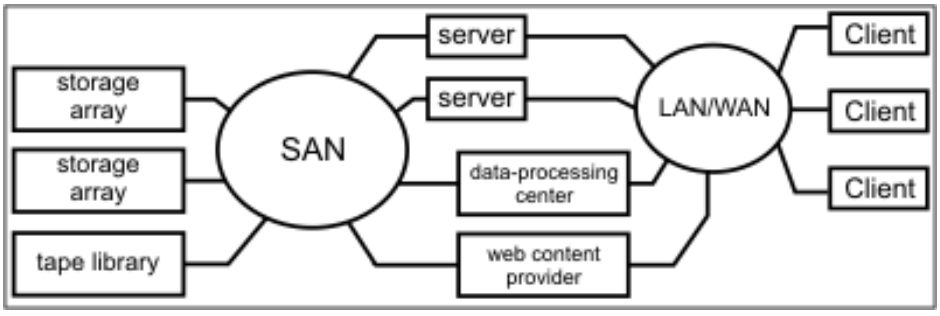
Dalam kaitan dengan teknologi dan tujuan mereka yang berbeda,salah satu maupun kedua-duanya dapat digunakan untuk kebutuhan penyimpanan. Dalam kenyataannya, batas antara keduanya samar sedikit menurut Kelompok Penilai, Analis Inc.. Sebagai contoh, dalam aplikasinya anda boleh memilih untuk mem-backup NAS *device* anda dengan SAN, atau menyertakan NAS *device* secara langsung ke SAN untuk mengijinkan *non-bottlenecked access* segera ke storage. (Sumber: An Overview of Network-Attached Storage, ¨ 2000, Evaluator Group, Inc.)

NAS vs SAN

NAS dan *Storage-Area Network* (SAN) memiliki sejumlah atribut umum. Kedua-Duanya menyediakan konsolidasi optimal, penyimpanan data yang dipusatkan, dan akses berkas yang efisien. Kedua-Duanya mengijinkan untuk berbagi storage antar host, mendukung berbagai sistem operasi yang berbeda pada waktu yang sama, dan memisahkan *storage* dari server aplikasi. Sebagai tambahan, kedua- duanya dapat menyediakan ketersediaan data yang tinggi dan dapat memastikan integritas dengan banyak komponen dan *Redundant Arrays of Independent Disk* (RAID). Banyak yang berpendapat bahwa NAS adalah saingan dari SAN, akan tetapi keduanya dalam kenyataannya dapat bekerja dengan cukup baik ketika digunakan bersama.

NAS dan SAN menghadirkan dua teknologi penyimpanan yang berbeda dan menghubungkan jaringan pada tempat yang sangat berbeda. NAS berada diantar server aplikasi dan sistem berkas. SAN berada diantara sistem berkas dan mendasari physical storage. SAN memiliki jaringan sendiri, menghubungkan semua storage dan semua server. Sementara NAS menggunakan jaringan TCP/IP standar. NAS bisa diakses oleh semua komputer yang terhubung dengan jaringan, sementara SAN hanya bisa diakses oleh server yang terhubung langsung dengannya melalui fibre channel. NAS dapat digunakan uleh berbagai arsitektur OS yang berbeda. Namun SAN merupakan operating system-dependent. Karena pertimbangan ini, masing-masing mendukung kebutuhan penyimpanan dari area bisnis yang berbeda.

Gambar 46.8. SAN



46.8. Implementasi Penyimpanan Stabil

Pada bagian sebelumnya, kita sudah membicarakan mengenai write-ahead log, yang membutuhkan ketersediaan sebuah storage yang stabil. Berdasarkan definisi, informasi yang berada di dalam stable storage tidak akan pernah hilang. Untuk mengimplementasikan storage seperti itu, kita perlu mereplikasi informasi yang dibutuhkan ke banyak peralatan storage (biasanya disk-disk) dengan failure modes yang independen. Kita perlu mengkoordinasikan penulisan update-update dalam sebuah cara yang menjamin bila terjadi kegagalan selagi meng-update tidak akan membuat semua kopi yang ada menjadi rusak, dan bila sedang recover dari sebuah kegagalan, kita dapat memaksa semua kopi yang ada ke dalam keadaan yang bernilai benar dan konsisten, bahkan bila ada kegagalan lain yang terjadi ketika sedang recovery. Untuk selanjutnya, kita akan membahas bagaimana kita dapat mencapai kebutuhan kita.

Terkadang terjadi kegagalan pada storage. Beberapa sector mungkin saja tiba-tiba menjadi bad sector. RAID mencoba melindungi data yang ada pada storage. Namun RAID tidak bisa menangani kegagalan yang terjadi ketika proses penulisan yang pertama kali. RAID tidak mampu melindungi storage ketika kegagalan terjadi ditengah proses penulisan.

Sebuah disk seharusnya bisa bekerja dengan baik sepanjang waktu. Sayangnya, hal itu tidak mungkin dilakukan. Ketika dilakukan proses penulisan, proses tersebut akan berhasil atau tidak sama sekali. Hal inilah yang disebut implementasi penyimpanan stabil (stable storage). Untuk selanjutnya akan dibahas bagaimana implementasi dari stable storage.

Sebuah disk write menyebabkan satu dari tiga kemungkinan:
1. *successful completion*
2. *partial failure*
3. *total failure*

Kita memerlukan, kapan pun sebuah kegagalan terjadi ketika sedang menulis ke sebuah blok, sistem akan mendeteksinya dan memanggil sebuah prosedur recovery untuk me-restore blok tersebut ke sebuah keadaan yang konsisten. Untuk melakukan itu, sistem harus menangani dua blok physical untuk setiap blok logical. Sebuah operasi output dieksekusi seperti berikut:

Ketika kegagalan terjadi ketika penulisan ke sebuah blok sedang berlangsung, sistem akan mendeteksinya dan melakukan prosedur recovery untuk mengembalikan blok ke keadaan yang konsisten. Untuk dapat melakukan hal ini, kita membutuhkan dua buah blok physical untuk sebuah blok logical dengan melakukan operasi berikut.
1. Tulis informasinya ke blok physical yang pertama.
2. Ketika penulisan pertama berhasil, tulis informasi yang sama ke blok *physical* yang kedua.
3. Operasi dikatakan berhasil hanya jika penulisan kedua berhasil.

Pada saat perbaikan dari sebuah kegagalan, setiap pasang blok physical diperiksa. Jika keduanya sama dan tidak terdeteksi adanya kesalahan, tetapi berbeda dalam isi, maka kita mengganti isi dari blok yang pertama dengan isi dari blok yang kedua. Prosedur recovery seperti ini memastikan bahwa sebuah penulisan ke stable storage akan sukses atau tidak ada perubahan sama sekali.

Kita dapat menambah fungsi prosedur ini dengan mudah untuk memboleh kan penggunaan dari kopi yang banyak dari setiap blok pada stable storage. Meski pun sejumlah besar kopi semakin mengurangi kemungkin an untuk terjadinya sebuah kegagalan, maka biasanya wajar untuk men simulasi stable storage hanya dengan dua kopi. Data di dalam stable storage dijamin aman kecuali sebuah kegagalan menghancurkan semua kopi yang ada.

Setelah proses di atas dilakukan, setiap pasang blok physical diperiksa. Terdapat tiga kemungkinan. Pertama, pada kedua blok tidak terjadi kesalahan dan memiliki isi yang sama. Maka sistem tidak melakukan apapun karena proses penulisan telah berhasil. Kedua, salah satu blok mengalami kegagalan. Ketika ini terjadi, data dari blok yang lain disalin ke blok yang rusak. Ketiga, tidak terjadi kesalahan pada kedua blok, tapi berbeda dalam ini. Jika ini terjadi maka isi dari blok kedua disalin ke blok pertama. Prosedur ini dilakukan untuk memastikan data berhasil ditulis atau tidak sama sekali.

Pada prosedur ini dapat ditambahkan fungsi untuk membolehkan penggunaan salinan yang banyak pada setiap blok pada stable storage. Semakin besar jumlah salinan maka akan semakin mengurangi kemungkinan kegagalan. Namun biasanya wajar untuk mensimulasi stable storage hanya dengan

dua buah salinan. Data di dalam stable storage dijamin aman kecuali sebuah kegagalan menghancurkan seluruh salinan yang ada.

46.9. Rangkuman

Aspek-aspek penting mengenai manajemen ruang swap, yaitu:

1. **Pengunaan ruang Swap.** Penggunaan ruang swap tergantung pada penerapan algoritma.
2. **Lokasi ruang swap.** Ruang swap dapat diletakkan di:
   a. sistem berkas normal
   b. partisi yang terpisah

RAID (Redundant Array of Independent Disks) merupakan salah satu cara untuk meningkatkan kinerja dan reliabilitas dari disk. Peningkatan Kehandalan dan Kinerja dari disk dapat dicapai melalui:

1. **Redudansi.** Dengan cara menyimpan informasi tambahan yang dapat dipakai untuk mengembalikan informasi yang hilang jika suatu disk mengalami kegagalan.
2. **Paralelisme.** Dengan cara mengakses banyak disk secara paralel.

Host-Attached Storage merupakan sistem penyimpanan yang terhubung secara langsung dengan komputer. Dalam implementasinya, Host-Attached Storage dapat disebut juga dengan Server-Attached Storage karena sistem penyimpanannya terdapat di dalam server.

Sedangkan Network-Attached Storage adalah suatu konsep penyimpanan bersama pada suatu jaringan. Network-Attached Storage mengutamakan pengguna jaringan. Berbeda dengan Storage-Area networks yang lebih mengutamakan mengenai kebutuhan ruang penyimpanan komputer.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

[WEBWirzOjaStafWe2004] Lars Wirzenius, Joanna Oja, dan StephenAlex StaffordWeeks. 2004. *The Linux System Administrator's Guide – The boot process in closer look http://www.tldp.org/ LDP/ sag/ html/ boot-process.html* . Diakses 7 Agustus 2006.

Bab 47. Penyimpanan Tersier

47.1. Pendahuluan

Karakteristik dari perangkat penyimpanan tersier pada dasarnya adalah menggunakan *removable media* yang tentu saja berdampak pada biaya produksi yang lebih murah. Sebagai contoh: sebuah VCR dengan banyak kaset akan lebih murah daripada sebuah VCR yang hanya dapat memainkan satu kaset saja.

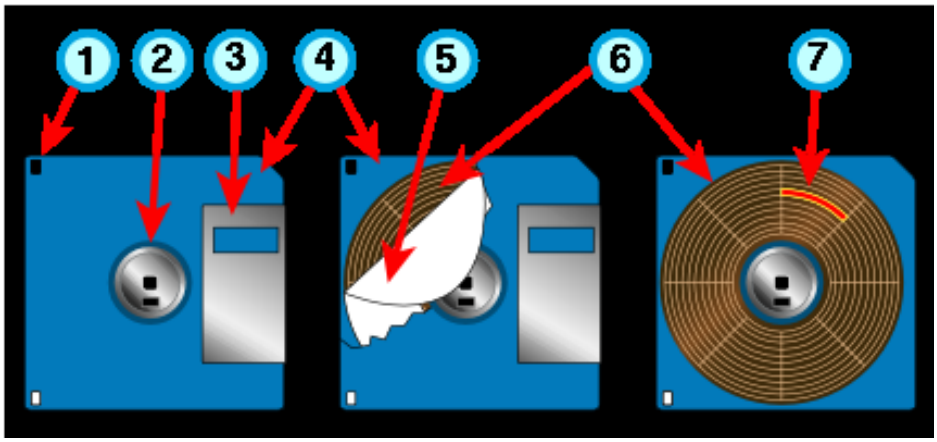
Dewasa ini perangkat penyimpanan tersier berkembang pesat dan sangat beragam yang dibuat dengan dengan removable media. Beberapa contohnya adalah Floppy disk, Tapes, CD, DVD, Flash Memori.

47.2. Jenis Struktur Penyimpanan Tersier

Floppy Disk

Floppy disk (disket) terbuat dari cakram tipis, fleksibel yang dilapisi bahan yang bersifat magnetik dan terbungkus atau dilindungi oleh plastik. Kebanyakan Floppy disk hanya mampu menampung data sekitar 1-2 Mb aja, Tetapi sekarang Floppy disk dapat menyimpan data hingga 1 Gb. Meskipun kecepatan akses datanya lebih lambat daripada Harddisk dan lebih rentan terhadap kerusakan permukaan disknya, floppy disk dulu sangat disukai karena harganya yang lebih murah daripada removable disk lainnya dan dapat ditulis berkali-kali.

Gambar 47.1. Komponen internal dasar floppy disk 3.5 inch



1. Write-protect tab; 2. Hub; 3. Shutter; 4. Plastic housing; 5. Paper ring; 6. Magnetic disk; 7. Disk sector. Sumber: "Floppy disk", Wikipedia, dari Wikimedia Commons – di bawah Creative Commons ShareAlike 1.0 License.

Magneto-optic disk

Magneto-optic disk adalah salah contoh dari Removable disk. Teknologi penyimpanan data pada Magneto-optic disk adalah dengan cara menyinari permukaan disk dengan sinar laser yang ditembakkan dari disk head. Tempat yang terkena sinar laser ini kemudian digunakan untuk menyimpan bit data. Untuk mengakses data yang telah disimpan, head mengakses data tersebut dengan bantuan Kerr effect. Cara kerja Kerr effect adalah ketika suatu sinar laser dipantulkan dari sebuah titik magnetik, polarisasinya akan diputar secara atau berlawanan dengan arah jarum jam, tergantung dari orientasi medan magnetiknya. Rotasi inilah yang dibaca oleh head disk sebagai sebuah bit data.

Gambar 47.2. Magneto-Optical Disc



Sumber: "Magneto-optical drive", Wikipedia, dari Wikimedia Commons -- public domain.

Optical disk

Optical disk tidak menggunakan bahan yang bersifat magnetik sama sekali. Optical disk menggunakan bahan spesial yang dapat diubah oleh sinar laser menjadi memiliki spot-spot yang relatif gelap atau terang. Contoh dari optical disk ini adalah CD-RW dan DVD-RW.

Teknologi Optical disk ini dibagi menjadi dua yaitu:

a. **Phase-change disk.** Disk dilapisi oleh bahan yang dapat mengkristal (beku) menjadi crystalline (serpihan-serpihan kristal) atau menjadi amorphous state (bagian yang tak berbentuk). Bagian crystalline ini lebih transparan, karenanya tembakan laser yang mengenainya akan lebih terang ketika melintasi bahan dan memantul dari lapisan pemantul. Drive phase-change disk ini menggunakan sinar laser dengan kekuatan yang berbeda. Sinar laser dengan kekuatan tinggi digunakan melelehkan disknya ke dalam amorphous state, sehingga dapat digunakan untuk menulis data lagi. Sinar laser dengan kekuatan sedang dipakai untuk menghapus data dengan cara melelehkan permukaan disknya dan membekukannya kembali ke dalam keadaan crystalline, sedangkan sinar laser dengan kekuatan lemah digunakan untuk membaca data yang telah disimpan.

b. **Dye-polimer disk.** Dye-polimer merekam data dengan membuat bump (gelembung). Disk dilapisi dengan bahan yang dapat menyerap sinar laser. Sinar laser ini membakar spot hingga spot ini memuai dan membentuk bump (gelembung). Bump ini dapat dihilangkan atau didatarkan kembali dengan cara dipanasi lagi dengan sinar laser.

Gambar 47.3. DVD-RW disc pada sebuah gelendong



Sumber: "DVD-RW", Wikipedia -- di bawah GNU Free Document License.

Write Once Read Many-times (WORM)

Sifat dari WORM ini adalah hanya dapat ditulis sekali dan data yang telah ditulis bisa tahan lama. WORM ini terbuat dari sebuah aluminium film yang dilapisi plastik di bagian bawah maupun di bagian atasnya. Cara penyimpanan data pada WORM ini adalah dengan cara memanfaatkan sinar laser untuk membuat lubang pada bagian aluminiumnya. Data yang telah disimpan tidak rusak atau tahan terhadap pengaruh medan magnet. Contoh dari WORM ini adalah CD-R dan DVD-R.

Gambar 47.4. CD-R



WORM ini dianggap tahan banting dan paling terpercaya karena lapisan metalnya dilindungi dengan aman oleh lapisan plastiknya dan juga datanya tidak dapat dirusak dengan pengaruh medan magnet.

Gambar 47.5. DVD-R



Kebanyakan *removable-disk* lebih lambat dari *non-removable-disk* karena kinerja mereka juga dipengaruhi oleh waktu yang dibutuhkan untuk menulis data. Waktu ini dipengaruhi oleh waktu rotasi, dan juga kadang-kadang *seek time*.

Read Only Disk

Read only disk menggunakan teknologi yang mirip dengan optical disk dan WORM, tetapi penyimpanan bit-bit dilakukan dengan cara pressed bukan dengan cara burned seperti proses penyimpanan pada optical disk maupun WORM. Datanya sudah direkam dari pabrik yang membuatnya dan datanya tahan lama. Contoh dari Read only disk adalah CD-ROM dan DVD-ROM.

Gambar 47.6. CDROM Drive



Tapes

Harga tapes drive memang lebih mahal daripada magnetic disk drive, tetapi harga catridge sebuah tape lebih murah dari pada equivalent penyimpanan data pada magnetic disk. Sebuah tapes dapat menyimpan data lebih banyak daripada optical disk maupun magnetic disk. Tape drive dan Disk drive hampir sama dalam kecepatan transfer data, tetapi dalam akses data secara random, tape jauh lebih lambat, karena membutuhkan operasi fast-forward atau rewind. Jadi tape kurang efektif dalam pengaksesan data secara random. Tapes banyak digunakan di supercomputer center dimana data yang ditampung sangat banyak dan besar dan tidak membutuhkan operasi random akses yang cepat. Untuk mengganti tape dalam library dalam skala besar secara otomatis, biasanya digunakan robotic tape changers.

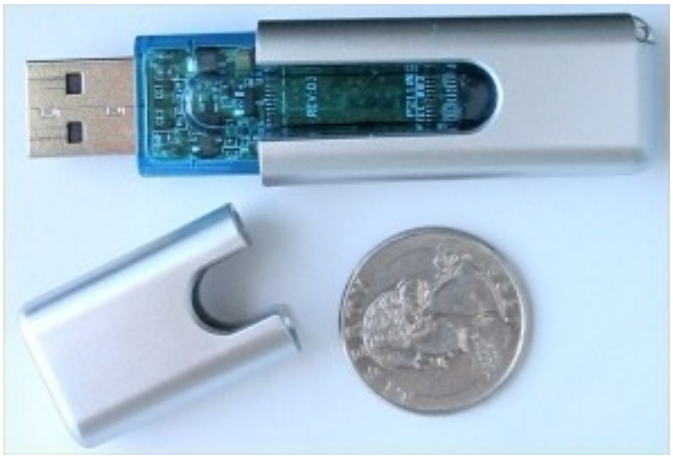
Gambar 47.7. DDS Tape Drives



Flash Memory

Flash Memori adalah sejenis EEPROM (Electrically-Erasable Programmable Read-Only Memori) yang mengizinkan akses pada lokasi memori untuk dihapus atau ditulis dalam satu operasi pemrograman. Istilah awamnya flash memori adalah suatu bentuk dari chip memori yang dapat ditulis, tidak seperti chip Random access memori, dan dapat menyimpan datanya meskipun tanpa daya listrik (non-volatile). Memori ini biasanya digunakan dalam kartu memori, USB flash drive (flash disk), pemutar MP3, kamera digital, dan telepon genggam.

Gambar 47.8. USB Drive



47.3. *Future Technology*

Penyimpanan *Holographic*

Teknologi penyimpanan sangat penting di masa yang akan datang, salah satunya adalah Holographic storage. Teknologi Holographic Storage menggunakan sinar laser untuk menyimpan foto hologram di media khusus. Hologram mempunyai pixel (titik), di mana setiap bit merepresentasikan satu yang mana warna hitam merepresentasikan bit 0 dan warna putih merepresentasikan bit 1. Kecepatan Transfer datanya melebihi kecepatan optic disk karena dalam proses pemindahan, Holograpic Storage ini mengizinkan jutaan pixel (titik) pada hologram dipindahkan semua dalam satu tembakan sinar laser.

Microelectronic Mechanical Systems (MEMS)

Teknologi ini digunakan untuk mengaplikasikan ide pabrikan yang ingin memproduksi media penyimpanan data yang fisiknya kecil, bersifat non-volatile dengan kecepatan yang lebih cepat dan lebih murah dari semiconductor DRAM.

47.4. Aplikasi Antarmuka

Sistem operasi tidak menangani *tapes* sebagaimana sistem operasi menangani *removable disk* maupun *fixed disk*. Suatu aplikasi tidak membuka suatu berkas pada *tape*, melainkan membuka *tape drive* secara keseluruhan sebagai *raw device*.

Biasanya *tape drive* disediakan untuk penggunaan aplikasi tersebut secara eksklusif, sampai aplikasi tersebut berakhir atau aplikasi tersebut menutup *tape device*. Eksklusivitas ini masuk akal, karena *random access* pada *tape* dapat memakan waktu yang lama, sehingga membiarkan beberapa aplikasi melakukan *random access* pada *tape* dapat menyebabkan *thrashing*.

Sistem operasi tidak menyediakan sistem berkas sehingga aplikasi harus memutuskan bagaimana cara menggunakan blok-blok array. Tiap aplikasi membuat peraturannya masing-masing tentang bagaimana mengatur *tape* supaya suatu *tape* yang penuh terisi dengan data hanya dapat digunakan oleh program yang membuatnya.

Tape drive mempunyai set operasi-operasi dasar yang berbeda dengan *disk* drive. Sebagai pengganti operasi *seek* (sebagaimana yang digunakan pada *disk drive*), *tape drive* menggunakan operasi *locate*. Operasi *locate* ini lebih akurat dibandingkan dengan operasi *seek* karena operasi ini memposisikan *tape* ke *logical block* yang spesifik.

Sebagian besar tape drive mempunyai operasi *read* position yang berfungsi memberitahu posisi *tape head* dengan menunjukkan nomor *logical blok*. Selain itu banyak juga *tape drive* yang menyediakan operasi *space* yang berfungsi memindahkan posisi *tape head*. Misalnya operasi *space* akan memindahkan posisi *tape head* sejauh dua blok ke belakang.

Untuk sebagian jenis *tape drive*, menulis pada blok mempunyai efek samping menghapus apa pun yang berada pada posisi sesudah posisi penulisan. Hal ini menunjukkan bahwa *tape drive* adalah *append-only devices*, maksudnya adalah apabila kita meng-*update* blok yang ada di tengah berarti kita akan menghapus semua data yang terletak sesudah blok tersebut. Untuk mencegah hal ini terjadi maka digunakan tanda EOT (*end-of-tape*) yang diletakkan pada posisi sesudah posisi blok yang ditulis. Drive menolak untuk mencari lokasi sesudah tanda EOT, tetapi adalah suatu hal yang penting untuk mencari lokasi EOT kemudian mulai menulis menulis data. Cara ini menyebabkan tanda EOT yang lama tertimpa, lalu tanda yang baru diletakkan pada posisi akhir dari blok yang baru saja ditulis.

Tugas utama dari sistem operasi adalah mengatur *physical device* (*hardware*) menjadi mesin virtual untuk menjalankan aplikasi-aplikasi. Bagaimanakah sistem operasi menangani media penyimpanan yang *removable*?

Kebanyakan sistem operasi menangani Removable disk seperti halnya Fixed Disk. Berbeda dengan Removable disk atau fixed disk, Sistem operasi biasanya menampilkan tapes sebagai media

penyimpanan mentah (raw storage medium). Suatu aplikasi tidak membuka suatu berkas pada tape, tetapi membuka tape drive secara keseluruhan sebagai Raw Device. Biasanya aplikasi yang menggunakan Tape drive secara eksklusif, aplikasi itu terus-menerus menggunakannya sampai aplikasi itu menutup device tape tersebut. Hal ini dilakukan untuk menghindari thrashing apabila beberapa aplikasi melakukan random access pada tape. Sistem operasi juga tidak menyediakan sistem berkas sehingga aplikasi harus memutuskan bagaimana cara menggunakan blok-blok array pada tapes. Tiap aplikasi masing- masing mengatur bagaimana cara mengatur tape agar suatu tape menyimpan data dan data tersebut hanya dapat digunakan oleh program yang membuatnya saja.

Operasi-operasi dasar tape drive:
1. **Operasi locate.** Operasi locate ini lebih akurat daripada operasi seek karena operasi ini mengatur posisi head tape ke logical block yang lebih spesifik.
2. **Operasi read position.** Operasi ini berfungsi untuk memberitahu posisi head dengan cara menunjukkan nomor logical block.
3. **Operasi Space.** Operasi ini memindahkan posisi head tape ke posisi satu blok atau lebih ke belakang maupun ke depan dari posisi head semula.

Sistem penyimpanan tape bersifat Append-only device atau menghapus apapun yang berada pada posisi sesudah penulisan, artinya apabila data yang berada di tengah, di-update maka data yang berada setelah blok tersebut akan terhapus. Untuk mencegah hal itu terjadi maka digunakan tanda EOT (end-of-Tape) yang diletakkan pada posisi sesudah posisi blok yang ditulis. Tape drive akan menolak untuk mengakses setelah tanda EOT tersebut, jadi untuk melakukan pengaksesan, tape drive harus mencari tanda EOT tersebut dan menimpanya, kemudian tanda yang baru diletakkan pada posisi akhir dari blok yang baru ditulis.

Penamaan Berkas

Penamaan berkas pada fixed disk tidaklah sulit, tetapi untuk penamaan berkas pada removable media cukup sulit terutama pada saat kita mau menulis data pada suatu komputer, kemudian menggunakan atau mengakses data tersebut pada komputer lain. Masalah yang paling sedikit muncul apabila kedua tipe komputer dan drivenya adalah sama, dan masalah yang paling banyak muncul apabila tipe komputer dan drivenya berbeda. Pada tipe mesin yang berbeda, penyimpanan urutan bytes, encoding untuk binary number maupun string mungkin berbeda, Sehingga dapat menimbulkan masalah pada penamaan atau pengaksesan berkas.

Sistem operasi sekarang banyak yang membiarkan masalah berkas naming pada removable media dan menyerahkan masalah tersebut pada pengguna maupun pada aplikasi untuk menyelesaikan masalah akses data tersebut. Tetapi ada beberapa jenis removable media sudah distandarkan sehingga semua komputer dapat menggunakan atau mengakses dengan cara sama. Beberapa contohnya adalah CD musik, CD data dan DVD.

Manajemen Penyimpanan Hierarkis

Ruang lingkup Sistem penyimpanan hierarkis ini sangat luas, mencakup memori primer maupun memori sekunder dan membentuk penyimpanan tersier. Penyimpanan tersier biasanya diimplementasikan sebagai jokebox dari tapes maupun removable disk. Robotic jukebox memungkinkan komputer untuk mengganti removable cartridge di tapes atau disk drive tanpa campur tangan manusia. Penggunaan utama dari teknologi ini adalah untuk kepentingan backup dan sistem penyimpanan hirarkis.

Pengambilan data dari jokebox membutuhkan waktu yang sangat lama dan memerlukan waktu yang lama untuk demand paging dan bentuk lain dari penggunaan virtual memori sehingga penyimpanan tersier dengan menggunakan sistem virtual memori tidak efektif.

Biasanya penyimpanan tersier dimasukkan dengan cara memberikannya ke berkas system. Berkas yang kapasitasnya kecil dan sering digunakan dibiarkan di dalam disk, sementara berkas yang kapasitasnya besar, sudah lama dan jarang digunakan akan diarsipkan di jokebox. Untuk menjaga berkas tetap exist, dalam sistem pengarsipan berkas akan dimasukkan ke dalam direktori, tetapi isi berkas tidak lagi berada di penyimpanan sekunder. Jika suatu aplikasi mencoba untuk membuka berkas, system call open akan ditunda sampai isi berkas terkirim ke penyimpanan tersier. Setelah proses pengiriman sudah selesai, maka operasi open akan mengembalikan control kepada aplikasi.

Biasanya yang menggunakan manajemen penyimpanan hierarkis ini adalah perusahaan atau instalasi besar dengan menggunakan sistem supercomputing untuk mengakses atau memproses data yang besar.

47.5. Masalah Kinerja

Sebagai salah satu komponen dari sistem operasi, penyimpanan tersier mempunyai tiga aspek utama dalam kinerja, yaitu: kecepatan, keandalan, dan biaya. Tiga aspek utama dari kinerja penyimpanan tersier berdasarkan [Silberschatz2002]

Kecepatan

Kecepatan dari penyimpanan tersier memiliki dua aspek: *bandwidth* dan *latency*. Menurut Silberschatz et. al. [Silberschatz2002], *Sustained bandwidth* adalah rata-rata tingkat data pada proses transfer, yaitu jumlah byte dibagi dengan waktu transfer. *Effective bandwidth* menghitung rata-rata pada seluruh waktu I/O, termasuk waktu untuk *seek* atau *locate*. Istilah *bandwidth* dari suatu *drive* sebenarnya adalah *sustained bandwidth*.

Kecepatan penyimpanan tersier memiliki dua aspek yaitu bandwidth dan latency.

Bandwidth diukur dalam byte per detik. Sustained bandwidth adalah rata-rata laju data pada proses transfer (jumlah byte dibagi dengan lamanya proses transfer). Perhitungan rata-rata pada seluruh waktu input atau output, termasuk waktu untuk pencarian disebut dengan Effective bandwidth. Untuk bandwidth suatu drive, umumnya yang dimaksud adalah sustained bandwidth.

Untuk removable disk, bandwidth berkisar dari beberapa megabyte per detik untuk yang paling lambat sampai melebihi 40 MB per detik untuk yang paling cepat. Tape memiliki kisaran bandwidth yang serupa, dari beberapa megabyte per detik sampai melebihi 30 MB per detik.

Aspek kedua kecepatan adalah access latency (waktu akses). Dengan ukuran kinerja ini, disk lebih cepat daripada tape. Penyimpanan disk secara esensial berdimensi dua--semua bit berada di luar di bukaan. Akses disk hanya memindahkan arm ke silinder yang dipilih dan menunggu rotational latency, yang bisa memakan waktu kurang daripada 5 milidetik. Sebaliknya, penyimpanan tape berdimensi tiga. Pada sembarang waktu, hanya bagian kecil tape yang terakses ke head, sementara sebagian besar bit terkubur di bawah ratusan atau ribuan lapisan tape (pita) yang menyelubungi reel (alat penggulung). Random access pada tape memerlukan pemutaran tape reel sampai blok yang dipilih mencapai tape head, yang bisa memakan waktu puluhan atau ratusan detik. Jadi, kita bisa secara umum mengatakan bahwa random access dalam sebuah tape cartridge lebih dari seribu kali lebih lambat daripada random access pada disk.

Kehandalan

Removable magnetic disk tidak begitu dapat diandalkan dibandingkan dengan *fixed hard-disk* karena *cartridge* lebih rentan terhadap lingkungan yang berbahaya seperti debu, perubahan besar pada temperatur dan kelembaban, dan gangguan mekanis seperti tekukan. *Optical disks* dianggap sangat dapat diandalkan karena lapisan yang menyimpan bit dilindungi oleh plastik transparan atau lapisan kaca.

Removable magnetic disk sedikit kurang andal daripada fixed hard disk karena cartridge-nya lebih berkemungkinan terpapar kondisi lingkungan yang merusak, seperti debu, perubahan besar dalam suhu dan kelembaban, dan tenaga mekanis seperti kejutan (shock) dan tekukan (bending). Optical disk dianggap sangat andal, karena lapisan yang menyimpan bit dilindungi oleh plastik transparan atau lapisan kaca. Keandalan magnetic tape sangat bervariasi, tergantung pada jenis drive-nya. Beberapa drive yang tidak mahal membuat tape tidak bisa dipakai lagi setelah digunakan beberapa lusin kali; jenis lainnya cukup lembut sehingga memungkinkan penggunaan ulang jutaan kali. Dibandingkan dengan magnetic-disk head, head dalam magnetic-tape drive merupakan titik lemah. Disk head melayang di atas media, tetapi tape head kontak langsung dengan tape-nya. Aksi penyapuan (scrubbing) dari tape bisa membuat head tidak bisa dipakai lagi setelah beberapa ribu atau puluhan ribu jam.

Ringkasnya, kita katakan bahwa fixed disk drive cenderung lebih andal daripada removable disk drive atau tape drive, dan optical disk cenderung lebih andal daripada magnetic disk atau tape. Namun, fixed magnetic disk memiliki satu kelemahan. Head crash (hantaman head) pada hard disk umumnya menghancurkan data, sedangkan, kalau terjadi kegagalan (failure) tape drive atau optical disk drive, seringkali data cartridge-nya tidak apa-apa.

Harga

Terjadi kecenderungan biaya per megabyte untuk memori DRAM, magnetic hard disk, dan tape drive. Harga dalam grafik adalah harga terendah yang ditemukan dalam iklan di berbagai majalah komputer dan di World Wide Web pada akhir tiap tahun. Harga-harga ini mencerminkan ruang pasar komputer kecil pembaca majalah-majalah ini, yang harganya rendah jika dibandingkan dengan pasar mainbingkai atau minikomputer. Dalam hal tape, harganya adalah untuk drive dengan satu tape. Biaya keseluruhan tape storage menjadi lebih rendah kalau lebih banyak tape yang dibeli untuk digunakan dengan drive itu, karena harga sebuah tape sangat rendah dibandingkan dengan harga drive. Namun, dalam sebuah tape library raksasa yang memuat ribuan cartridge, storage cost didominasi oleh biaya tape cartridge. Pada tahun 2004, biaya per GB tape cartridge dapat didekati sekitar kurang dari $2.

Sejak tahun 1997, harga per gigabyte tape drive yang tidak mahal telah berhenti turun secara dramatis, walaupun harga teknologi tape kisaran menengah (seperti DAT/DDS) terus turun dan sekarang mendekati harga drive yang tidak mahal. Harga tape-drive tidak ditunjukkan sebelum tahun 1984, karena, seperti telah disebutkan, majalah yang digunakan dalam menelusuri harga ditargetkan untuk ruang pasar komputer kecil, dan tape drive tidak secara luas digunakan bersama komputer kecil sebelum tahun 1984.

Harga per megabyte juga telah turun jauh lebih cepat untuk disk drive daripada untuk tape drive. Pada kenyataannya, harga per megabyte magnetic disk drive mendekati sebuah tape cartridge tanpa tape drive. Akibatnya, tape library yang berukuran kecil dan sedang memiliki storage cost lebih tinggi daripada sistem disk dengan kapasitas setara.

Kejatuhan dramatis dalam harga disk telah membuat penyimpanan tersier usang. Kita tidak lagi memiliki teknologi penyimpanan tersier yang sangat lebih murah daripada magnetic disk. Tampaknya kebangkitan penyimpanan tersier harus menunggu terobosan teknologi yang revolusioner. Sementara, tape storage penggunaannya kebanyakan akan terbatas untuk keperluan seperti sebagai backup disk drive dan penyimpanan arsip dalam tape pustaka yang sangat besar yang jauh melebihi kapasitas penyimpanan praktis kumpulan disk besar.

47.6. Rangkuman

Penyimpanan tersier dibangun dengan removable media. Penyimpanan tersier biasanya diimplementasikan sebagai jukebox dari tape atau removable disk.

Kebanyakan sistem operasi menangani removable disk seperti fixed disk. Sementara itu, tape biasanya ditampilkan sebagai media penyimpanan mentah (raw storage medium), sistem berkas tidak disediakan.

Tiga aspek kinerja yang utama adalah kecepatan, kehandalan dan biaya. Random access pada tape jauh lebih lama daripada disk. Keandalan removable magnetic disk masih kurang karena masih rentan terhadap lingkungan yang berbahaya seperti debu, temperatur, dan kelembaban. Optical disk lebih andal daripada magnetic media karena memiliki lapisan pelindung dari plastik transparan atau kaca.

Rujukan

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

[WEBWiki2006h] From Wikipedia, the free encyclopedia. 2006. *Keydrive – http://en.wikipedia.org/ wiki/ Keydrive* . Diakses 09 Agustus 2006.

[WEBWiki2006i] From Wikipedia, the free encyclopedia. 2006. *Tape drive – http://en.wikipedia.org/ wiki/ Tape_drive* . Diakses 09 Agustus 2006.

[WEBWiki2006j] From Wikipedia, the free encyclopedia. 2006. *CD-ROM – http://en.wikipedia.org/ wiki/ CD-ROM* . Diakses 09 Agustus 2006.

[WEBWiki2006k] From Wikipedia, the free encyclopedia. 2006. *DVD – http://en.wikipedia.org/ wiki/ DVD* . Diakses 09 Agustus 2006.

[WEBWiki2006l] From Wikipedia, the free encyclopedia. 2006. *CD – http://en.wikipedia.org/ wiki/ CD* . Diakses 09 Agustus 2006.

[WEBWiki2006m] From Wikipedia, the free encyclopedia. 2006. *DVD-RW – http://en.wikipedia.org/ wiki/ DVD-RW* . Diakses 09 Agustus 2006.

[WEBWiki2006n] From Wikipedia, the free encyclopedia. 2006. *Magneto-optical drive – http://en.wikipedia.org/ wiki/ Magneto-optical_drive* . Diakses 09 Agustus 2006.

[WEBWiki2006o] From Wikipedia, the free encyclopedia. 2006. *Floppy disk – http://en.wikipedia.org/ wiki/ Floppy_disk* . Diakses 09 Agustus 2006.

Bab 48. Keluaran/Masukan Linux

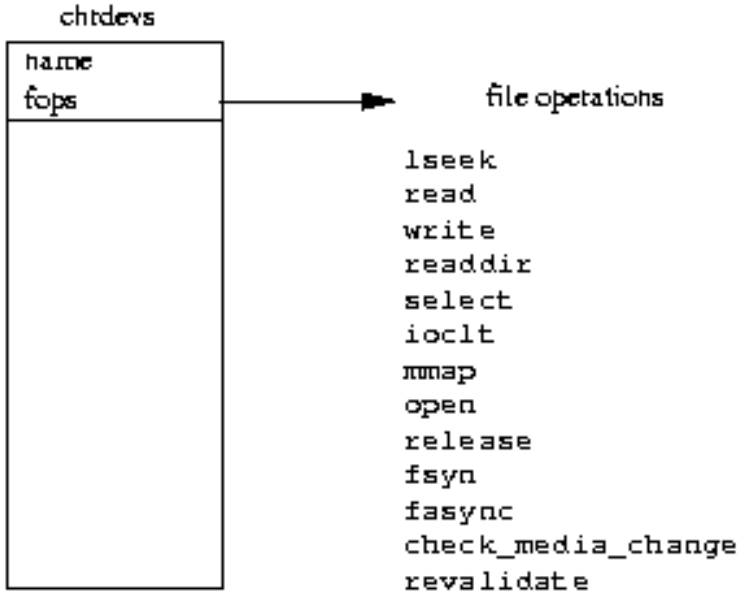
48.1. Pendahuluan

Salah satu tujuan sistem operasi adalah menyembunyikan kerumitan device perangkat keras dari para penggunanya. Umpamanya, sistem berkas virtual menyamakan tampilan sistem berkas yang dimount tanpa memperdulikan devices fisik yang berada di bawahnya. Bab ini akan menjelaskan bagaimana kernel Linux mengatur device fisik di sistem.

Salah satu fitur yang mendasar adalah kernel mengabstraksi penanganan device. Semua device hardware terlihat seperti berkas pada umumnya: mereka dapat dibuka, ditutup, dibaca, dan ditulis menggunakan calls sistem yang sama dan standar untuk memanipulasi berkas. Setiap device di sistem direpresentasikan oleh sebuah file khusus device, contohnya disk IDE yang pertama di sistem direpresentasikan dengan /dev/hda. Devices blok (disk) dan karakter dibuat dengan perintah mknod dan untuk menjelaskan device tersebut digunakan nomor devices besar dan kecil. Devices jaringan juga direpresentasikan dengan berkas khusus device, tapi berkas ini dibuat setelah sistem menemukan dan menginisialisasi pengontrol-pengontrol jaringan di sistem. Semua device yang dikontrol oleh driver device yang sama memiliki nomor device besar yang umum. Nomor devices kecil digunakan untuk membedakan antara device-device yang berbeda dan pengontrol-pengontrol mereka, contohnya setiap partisi di disk IDE utama punya sebuah nomor device kecil yang berbeda. Jadi, /dev/hda2, yang merupakan partisi kedua dari disk IDE utama, punya nomor besar 3 dan nomor kecil yaitu 2. Linux memetakan berkas khusus device yang diteruskan ke system call (katakanlah melakukan mount ke sistem berkas device blok) pada driver device dengan menggunakan nomor device besar dan sejumlah tabel sistem, contohnya tabel device karakter, chrdevs. Linux membagi devices ke tiga kelas: devices karakter, devices blok dan devices jaringan.

48.2. *Device Karakter*

Gambar 48.1. CharDev.



Device karakter, device paling sederhana dari Linux, diakses sebagai berkas. Aplikasi menggunakan *system calls* standar untuk membukanya, membacanya dan menulisnya dan menutupnya persis seolah devices adalah berkas. Memang benar, meski pun devices ini merupakan modem yang

sedang digunakan oleh PPP daemon untuk menghubungkan sistem Linux ke jaringan. Saat sebuah device karakter diinisialisasi, driver devicenya mendaftarkan pada kernel Linux dengan menambahkan sebuah entry ke vektor chrdevs dari struktur data device_struct. Pengenal utama devicenya digunakan sebagai indeks ke vektor ini. Pengenal utama untuk suatu device tidak pernah berubah.

Setiap entry di vektor chrdevs, sebuah struk data device_struct, mengandung dua elemen: sebuah penunjuk nama dari driver devices yang terdaftar dan sebuah penunjuk ke operasi-operasi berkas seperti buka, baca, tulis, dan tutup. Isi dari /proc/devices untuk devices karakter diambil dari vektor chrdevs.

Saat sebuah berkas khusus karakter yang merepresentasikan sebuah devices karakter (contohnya /dev/cua0) dibuka, kernelnya harus mengatur beberapa hal sehingga routine operasi berkas yang benar dari driver devices karakter akan terpanggil.

Seperti sebuah berkas atau direktori pada umumnya, setiap berkas khusus device direpresentasikan dengan sebuah inode VFS. Inode VFS untuk sebuah berkas khusus karakter tersebut, sebenarnya untuk semua berkas yang berada dibawahnya, contohnya EXT2. Hal ini terlihat dari informasi di berkas yang sebenarnya ketika nama berkas khusus device dilihat.

Setiap inode VFS memiliki keterkaitan dengan seperangkat operasi berkas dan operasi-operasi ini berbeda tergantung pada obyek sistem berkas yang direpresentasikan oleh inode tersebut. Kapan pun sebuah VFS yang merepsentasikan berkas khusus karakter dibuat, operasi-operasi berkasnya diset ke operasi device karakter default.

VFS inode memiliki hanya satu operasi berkas, yaitu operasi membuka berkas. Saat berkas khusus karakter dibuka oleh sebuah aplikasi, operasi buka berkas yang umum atau generik menggunakan pengenal utama dari device tersebut. Pengenal ini digunakan sebagai index ke vektor chrdevs untuk memperoleh blok operasi berkas untuk device tertentu ini. Ia juga membangun struk data berkas yang menjelaskan berkas khusus karakter ini, yang membuat penunjuk operasi berkas menunjuk ke driver device itu. Setelah itu semua aplikasi dari operasi-operasi berkas aplikasi akan dipetakan untuk memanggil perangkat devices karakter dari operasi berkas itu.

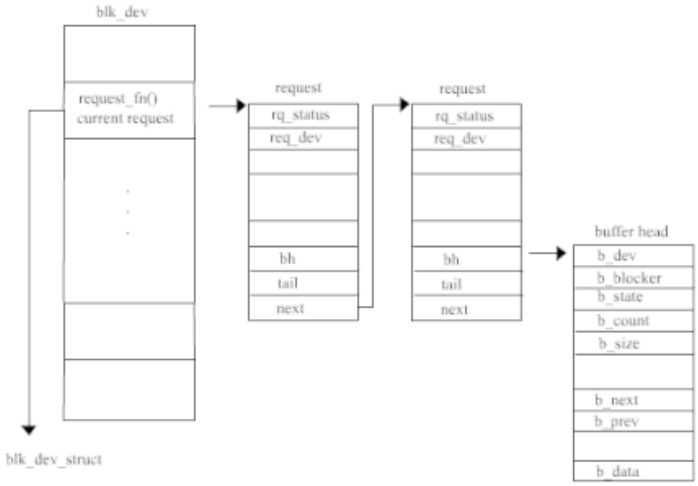
48.3. *Device* Blok

Device ini diakses seperti berkas. Mekanisme untuk menyediakan perangkat operasi berkas yang benar bagi berkas khusus blok yang terbuka sama seperti devices karakter. Linux memelihara operasi dari perangkat device blok yang terdaftar sebagai vektor blkdevs. Vektor ini, seperti halnya vektor chrdevs, diindeks dengan menggunakan nomor device besar dari sang device. Entrynya juga merupakan struk data device_struct. Tidak seperti devices karakter, ada sejumlah kelas yang dimiliki device blok. Device-device SCSI adalah salah satu kelasnya dan device IDE adalah kelas lainnya. Kelaslah yang mendaftarkan dirinya sendiri pada kernel Linux dan menyediakan operasi berkas kepada kernel. Driver-driver device untuk sebuah kelas device blok menyediakan interface khusus kelas kepada kelas tersebut. Jadi, contohnya, sebuah driver device SCSI harus menyediakan interface untuk subsistem SCSI agar dapat menyediakan operasi berkas bagi devices ini ke kernel.

Setiap driver device blok harus menyediakan sebuah interface ke cache buffernya, demikian pula interface operasi umum berkas. Setiap driver device blok mengisi entrynya di vektor blk_dev dari struk data blk_dev_struct. Indeksnya ke vektor ini, lagi-lagi, nomor utama devicenya. Struk data blk_dev_struct mengandung alamat routine permintaan dan sebuah penunjuk ke sekumpulan struk data request,yang masing-masingnya merepresentasikan sebuah request dari cache buffernya untuk driver untuk membaca atau menulis atau menulis satu blok data.

Setiap kali cache buffer ingin membaca dari, atau pun menuliskan satu blok data ke device terdaftar, ia menambahkan struk data request kedalam blk_dev_struct nya. Gambar di atas ini menunjukkan bahwa setiap request memiliki *pointer* (penunjuk) ke satu atau lebih struk data buffer_head. Masing-masingnya merupakan suatu request untuk membaca atau menulis sebuah blok data. Struk buffer_head tersebut dikunci (oleh cache buffer) dan mungkin ada suatu proses yang menunggu buffer ini selesai di operasi blok tersebut. Setiap struk request dialokasikan dari suatu daftar yang statik, yaitu daftar all_request. Jika proses tersebut sedang dimasukkan sebuah ke list request yang kosong, fungsi request dari drivernya akan dipanggil agar memulai proses antrian request. Jika tidak driver tersebut hanya akan memproses setiap request di daftar request.

Gambar 48.2. Buffer



Sekali driver device telah menyelesaikan sebuah request, ia harus membuang setiap stuk buffer_request dari struk requestnya, kemudian mencapnya *up to date* dan membuka kuncinya. Pembukaan kunci buffer_head akan membangunkan proses apa pun yang tidur akibat menunggu operasi blok selesai. Contoh dari kasus ini misalnya dimana sebuah nama berkas sedang ditangani dan sistem berkas EXT2 harus membaca blok data yang mengandung entry direktori EXT2 berikutnya dari device blok yang menyimpan sistem berkas tersebut. Proses ini tidur di buffer_head yang akan mengandung entri direktorinya sampai driver devicenya membangunkannya. Struk data request tersebut ditandai bebas sehingga ia dapat digunakan di request blok lainnya.

48.4. *Device* Jaringan

Device jaringan merupakan sebuah entity yang mengirimkan dan menerima paket-paket data. Biasanya ia merupakan device fisik seperti kartu ethernet. Beberapa devices jaringan bagaimana pun hanyalah software, seperti device loopback yang digunakan untuk mengirimkan data ke Anda. Setiap device direpresentasikan dengan struk data device. Driver device jaringan mendaftarkan device-device yang ia kontrol pada Linux selama inisialisasi jaringan yaitu saat kernel melakukan booting. Struk data device tersebut berisi informasi mengenai device dan alamat fungsi-fungsi yang memungkinkan bermacam-macam protokol jaringan menggunakan layanan dari device tersebut. Fungsi-fungsi ini kebanyakan terkait dengan mentransmisikan data dengan menggunakan device jaringan. Device tersebut menggunakan mekanisme pendukung jaringan standar untuk melewatkan data yang diterima sampai ke lapisan protokol yang semestinya. Semua data jaringan atau paket yang ditransmisikan dan diterima, direpresentasikan dengan struk-struk data sk_buff. Struk-struk data yang bersifat fleksibel ini memungkinkan header-header protokol jaringan menjadi mudah ditambahkan dan dibuang. Bagian ini hanya memfokuskan pada struk data device serta bagaimana jaringan ditemukan dan diinsialisasi.

Struk data device ini mengandung informasi tentang device jaringan berikut.

Nama

Berbeda dengan device karakter dan blok yang menggunakan berkas khusus device yang dibuat dengan perintah mknod, berkas khusus device terlihat sekilas seperti device jaringan sistem yang ditemukan dan diinsialisasi. Nama mereka standar, yaitu setiap nama merepsentasikan jenis device masing-masing. Device multiple dari jenis yang sama dinomori lebih besar dari 0. Oleh sebab itu device-device ethernet dikenal sebagai /dev/eth0, /dev/eth1, /dev/eth2 dan seterusnya.

Beberapa device jaringan yang umum adalah
- /dev/ethN Device ethernet
- /dev/slN Device SLIP

- /dev/pppN Device PPP
- /dev/lo Device Loopback

Informasi Bus

Berikut ini adalah informasi yang driver device butuhkan untuk mengontrol devicenya. Nomor irq merupakan interrupt yang digunakan oleh device ini. Alamat basisnya adalah alamat dari segala register status dan control dari device yang ada di memori M/K. Channel DMA adalah nomor DMA yang device jaringan ini gunakan. Semua informasi ini diset pada waktu booting, yaitu saat device ini diinisialisasi.

Flags Interface

Hal-hal berikut ini akan menjelaskan karakteristik dan kemampuan dari device jaringan:
- IFF_UP Interface bangkit dan berjalan,
- IFF_BROADCAST Alamat broadcast di device adalah sah
- IFF_DEBUG Penghilangan error dinyalakan
- IFF_LOOPBACK Merupakan device loopback
- IFF_POINTTOPOINT Merupakan link point to point (SLIP dan PPP)
- IFF_NOTRAILERS Tidak ada pengangkut jaringan
- IFF_RUNNING Sumberdaya yang dialokasikan
- IFF_NOARP Tidak mendukung protokol ARP
- IFF_PROMISC Device di mode penerimaan acak, ia akan menerima semua paket tanpa memperdulikan kemana paket-paket ini dialamatkan
- IFF_ALLMULTI Menerima seluruh frame multicast IP
- IFF_MULTICAST Dapat menerima frame multicast IP

Informasi Protokol

Setiap device menjelaskan bagaimana ia digunakan oleh lapisan protokol jaringan.

MTU

Ukuran paket terbesar yang jaringan dapat kirim, tidak termasuk header lapisan link yang ia perlu tambahkan.

Keluarga

Keluarga ini menandakan bahwa keluarga protokol yang dapat didukung oleh device tersebut. Keluarga untuk seluruh device jaringan Linux adalah AF_INET, keluarga alamat internet.

Jenis

Jenis menjelaskan media di mana device jaringan terpasang. Ada banyak jenis media yang didukung oleh device jaringan Linux. Termasuk diantaranya adalah Ethernet, X.25, Token Ring, Slip, PPP dan Apple Localtalk.

Alamat

Struk data device tersebut memiliki sejumlah alamat yang relevan bagi device jaringan ini, termasuk alamat-alamat IP-nya.

Antrian Paket

Merupakan antrian paket-paket sk_buff yang antri menunggu untuk dikirmkan lewat device jaringan ini.

Fungsi Pendukung

Setiap device menyediakan seperangkat routine standar yang lapisan-lapisan protokol sebut sebagai bagian dari interface mereka ke lapisan link device ini. Hal ini termasuk pembuatannya dan routine-routine pengirim frame dan routine-routine penambah header standar dan pengumpul statistik. Statistik ini bisa dilihat dengan memakai perintah ifconfig.

48.5. Rangkuman

Dasar dari elemen perangkat keras yang terkandung pada M/K adalah *bus*, *device controller*, dan M/K itu sendiri. Kinerja kerja pada data yang bergerak antara device dan memori utama di jalankan oleh CPU, di program oleh M/K atau mungkin DMA *controller*. Modul kernel yang mengatur *device* adalah *device driver*. *System-call interface* yang disediakan aplikasi dirancang untuk menghandle beberapa dasar kategori dari perangkat keras, termasuk *block devices, character devices, memory mapped files, network sockets,* dan *programmed interval timers*.

Subsistem M/K kernel menyediakan beberapa servis. Diantaranya adalah *I/O schedulling, buffering, spooling, error handling,* dan *device reservation*. Salah satu servis dinamakan *translation*, untuk membuat koneksi antara perangkat keras dan nama file yang digunakan oleh aplikasi.

I/O system calls banyak dipakai oleh CPU, dikarenakan oleh banyaknya lapisan dari perangkat lunak antara *physical device* dan aplikasi. Lapisan ini mengimplikasikan *overhead* dari *context switching* untuk melewati *kernel's protection boundary*, dari sinyal dan *interrupt handling* untuk melayani *I/O devices*.

Disk drives adalah *major secondary-storage I/O device* pada kebanyakan komputer. Permintaan untuk disk M/K digenerate oleh sistem file dan sistem virtual memori. Setiap permintaan menspesifikasikan alamat pada disk untuk dapat direferensikan pada *form* di *logical block number*.

Algoritma *disk schedulling* dapat meningkatkan efektifitas *bandwidth*, *average response time*, dan *variance response time*. Algoritma seperti SSTF, SCAN, C-SCAN, LOOK dan C-LOOK didesain untuk membuat perkembangan dengan menyusun ulang antrian disk untuk meningkatkan total waktu pencarian.

Performa dapat rusak karena *external fragmentation*. Satu cara untuk menyusun ulang disk untuk mengurangi fragmentasi adalah untuk *back up* dan *restore* seluruh disk atau partisi. Blok-blok dibaca dari lokasi yang tersebar, me-*restore* tulisan mereka secara berbeda. Beberapa sistem mempunyai kemampuan untuk men-*scan* sistem file untuk mengidentifikasi file terfragmentasi, lalu menggerakan blok-blok mengelilingi untuk meningkatkan fragmentasi. Mendefragmentasi file yang sudah di fragmentasi (tetapi hasilnya kurang optimal) dapat secara signifikan meningkatkan performa, tetapi sistem ini secara umum kurang berguna selama proses defragmentasi sedang berjalan. Sistem operasi me-*manage* blok-blok pada disk. Pertama, disk baru di format secara *low level* untuk menciptakan sektor pada perangkat keras yang masih belum digunakan. Lalu, disk dapat di partisi dan sistem file diciptakan, dan blok-blok boot dapat dialokasikan. Terakhir jika ada blok yang terkorupsi, sistem harus mempunyai cara untuk me-*lock out* blok tersebut, atau menggantikannya dengan cadangan.

Tertiary storage di bangun dari disk dan *tape drives* yang menggunakan media yang dapat dipindahkan. Contoh dari *tertiary storage* adalah *magnetic tape*, *removable magnetic*, dan *magneto-optic disk*.

Untuk *removable disk*, sistem operasi secara general menyediakan servis penuh dari sistem file *interface*, termasuk *space management* dan *request-queue schedulling* . Untuk tape, sistem operasi secara general hanya menyediakan *interface* yang baru. Banyak sistem operasi yang tidak memiliki *built-in support* untuk *jukeboxes*. *Jukebox support* dapat disediakan oleh *device driver*.

Setiap aplikasi yang dijalankan di linux mempunyai pengenal yang disebut sebagai process identification number (PID). PID disimpan dalam 32 bit dengan angka berkisar dari 0-32767 untuk menjamin kompatibilitas dengan unix. Dari nomor PID inilah linux dapat mengawasi dan mengatur proses-proses yang terjadi didalam system. Proses yang dijalankan atau pun yang baru dibuat mempunyai struktur data yang disimpan di task_struct. Linux mengatur semua proses di dalam sistem melalui pemeriksaan dan perubahan terhadap setiap struktur data task_struct yang dimiliki

setiap proses. Sebuah daftar pointer ke semua struktur data task_struct disimpan dalam task vector. Jumlah maksimum proses dalam sistem dibatasi oleh ukuran dari task vector. Linux umumnya memiliki task vector dengan ukuran 512 entries. Saat proses dibuat, task_struct baru dialokasikan dari memori sistem dan ditambahkan ke task vector. Linux juga mendukung proses secara real time. Proses semacam ini harus bereaksi sangat cepat terhadap event eksternal dan diperlakukan berbeda dari proses biasa lainnya oleh penjadual.

Obyek-obyek yang terdapat di sistem berkas linux antara lain file, inode, file sistem dan nama inode. Sedangkan macam-macam sistem berkas linux antar lain : ext2fs, ext3fs, reiser, x, proc dan tiga tambahan : sistem berkas web, sistem berkas transparent cryptographic dan sistem berkas steganographic

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

Bagian VIII. Topik Lanjutan

Sebagai penutup, akan dibahas empat topik lanjutan, yaitu: Sistem Waktu Nyata dan Multimedia, Sistem Terdistribusi, Keamanan Sistem, serta Perancangan dan Pemeliharaan.

Bab 49. Waktu Nyata dan Multimedia

49.1. Pendahuluan

Sistem yang memiliki persyaratan tertentu, tentunya memiliki tujuan yang berbeda dengan yang selama ini kita pelajari. Seperti halnya pula dengan sistem waktu nyata, dimana sistem ini mempersyaratkan bahwa komputasi yang dihasilkan benar tapi juga harus sesuai dengan waktu yang dikehendaki. Oleh karena itulah algoritma penjadwalan yang tradisional haruslah dimodifikasi sehingga dapat memenuhi persyaratan deadline yang diminta. Hal ini pula yang dipersyaratkan oleh sistem multimedia yang tidak hanya memiliki data konvensional (seperti berkas teks, program, dll), tetapi juga memiliki data multimedia. Hal ini disebabkan karena data multimedia terdiri dari continous-media data (audio dan video), seperti contohnya adalah frame video, yang mempersyaratkan juga pengirimannya dalam batas waktu tertentu, misalnya 30 frame/detik. Untuk dapat memenuhi permintaan ini, dibutuhkan perubahan yang cukup signifikan pada struktur sistem operasi, yang kebanyakan pada memori, data dan juga manajemen jaringan.

49.2. Kernel Waktu Nyata

Sebelum memasuki lebih jauh tentang kernel waktu nyata, kita perlu tahu apa yang dimaksud dengan waktu nyata. Waktu nyata merujuk pada bentuk aplikasi yang mengontrol proses dimana masalah waktu merupakan hal yang sangat penting. Sistem waktu nyata digunakan ketika ada persyaratan waktu yang ketat pada operasi di prosesor atau flow dari data; yang sering digunakan sebagai alat kontrol yang pada aplikasi yang terpisah. Atau dengan kata lain, sebuah sistem waktu nyata tidak hanya perlu untuk menjalankan software melalui proses dengan benar, tapi juga perlu untuk menjalankannnya dalam waktu yang tepat, kalau tidak sistem akan gagal.

Pada sistem operasi yang mendukung sistem waktu nyata, tidak dibutuhkan fitur yang penting untuk standar desktop dan server system. Hal ini dikarenakan:

1. Kebanyakan sistem waktu nyata melayani untuk sebuah tujuan saja, sehingga tidak perlu membutuhkan banyak fitur pada desktop PC. Sistem waktu nyata tertentu juga tidak memasukkan notion pada user karena sistem hanya mendukung sejumlah kecil task saja, yang sering menunggu input dari peralatan H/W.
2. Fitur yang didukung oleh standar desktop dan server system tidak memungkinkan untuk menyediakan prosesor yang cepat dan memori yang banyak. Kekurangan space, menyebabkan sistem waktu nyata tidak dapat untuk mendukung drive disk yang peripheral atau mendisplay grafik.
3. Mendukung fitur yang biasa ada pada standar desktop dan server system akan sangat meningkatkan cost dari sistem waktu nyata.

Fitur-fitur minimal yang dibutuhkan oleh sistem operasi yang mendukung sistem yang real time adalah:

1. Penjadwalan berdasarkan prioritas dan preemptif
2. Kernel preemptif
3. Latency yang minimal

Fitur yang dihilangkan pada daftar di atas adalah dukungan jaringan. Hal ini dikarenakan hanya dipersyaratkan pada sistem komputer yang memang berinteraksi dengan sistem komputer lain melalui jaringan.

49.3. Penjadwalan Berdasarkan Prioritas

Algoritma penjadwalan berdasarkan prioritas dan preemptif mampu menjalankan proses berdasarkan tingkat kepentingannya, dimana proses yang sedang berjalan akan dipreemptifkan apabila sebuah proses berprioritas tinggi menjadi tersedia untuk dijalankan. Oleh karena itulah algoritma ini dibutuhkan, mengingat fitur utama yang paling dibutuhkan oleh sistem yang waktu nyata adalah mampu merespon proses yang real time secepat proses membutuhkan CPU.

Sistem waktu nyata dibagi menjadi dua, yaitu sistem waktu nyata keras dan sistem waktu nyata

lembut. Algoritma ini adalah hanya menjamin fungsionalitas waktu nyata lembut. Hal ini dikarenakan sistem waktu nyata lembut memberikan aturan yang kurang, sehingga memungkinkan proses yang kritis untuk mendapatkan prioritas belakangan. Walaupun menambahkan fungsionalitas waktu nyata lembut pada sistem baru dapat menyebabkan alokasi sumber daya yang tidak adil dan dapat pula menyebabkan penundaan yang lebih lama, atau bahkan kelaparan, namun setidaknya hal tersebut untuk beberapa proses mungkin untuk dicapai.

Hasilnya adalah sebuah sistem bertujuan umum yang dapat mendukung multimedia berupa grafik interaktif berkecepatan tinggi dan berbagai jenis pekerjaan yang secara fungsi tidak akan diterima pada lingkungan yang tidak mendukung perhitungan waktu nyata lembut. Algoritma ini tidak mampu mendukung sistem waktu nyata keras, karena kedepannya sistem waktu nyata keras harus memiliki jaminan bahwa task waktu nyata akan dilayani sesuai dengan persyaratan waktu tenggangnya. Pernyataan tentang jumlah waktu yang dibutuhkan untuk penyelesaian proses, disubmit bersamaan dengan proses itu sendiri. Kemudian penjadual akan memberikan ijin bagi proses tersebut, memberikan jaminan bahwa proses tersebut dapat diselesaikan tepat waktu, dan apabila tidak memungkinkan untuk diselesaikan tepat waktu, maka akan ditolak.

Jaminan semacam ini tidak akan mungkin dilakukan pada sistem dengan secondary storage atau memori virtual, karena subsistem ini dapat menyebabkan hal-hal yang tidak dapat dicegah dan tidak dapat diperkirakan pada sejumlah waktu untuk mengeksekusi proses tertentu. Oleh karenanyalah, sistem waktu nyata keras disusun dari software yang bertujuan khusus yang berjalan pada hardware yang khusus dipersembahkan bagi proses yang kritis, dan membutuhkan fungsionalitas yang penuh dari komputer modern dan sistem operasi.

49.4. Kernel Preemptif

Kernel yang non-preemptif tidak mengizinkan preemption pada proses yang berjalan pada mode kernel, dimana proses yang mode kernel akan berjalan sampai dengan keluar dari mode kernel, blok atau voluntarily dari yields control dari CPU. Sebaliknya, kernel yang preemptif mengizinkan preemption dari task untuk berjalan pada mode kernel.

Untuk dapat memenuhi persyaratan waktu dari sistem waktu nyata keras, adanya kernel preemptif menjadi penting. Kalau tidak maka task waktu nyata akan dapat menunggu dalam waktu yang panjang padahal task yang lain aktif dalam kernel.

Ada beberapa cara untuk membuat kernel yang dapat preemptif. Salah satunya adalah dengan memasukkan preemption point pada system call berdurasi panjang, dimana preemption point akan mengecek apakah proses dengan berprioritas tinggi perlu untuk dijalankan. Jika demikian maka context switch yang berperan. Maka, ketika proses berprioritas tinggi terminasi, proses yang diinterupsi akan melanjutkan system call. Preemption point akan ditempatkan hanya pada lokasi aman pada kernel, yaitu hanya pada saat dimana struktur data kernel yang belum dimodifikasi. Strategi kedua adalah dengan membuat sebuah kernel yang dapat preemptif melalui penggunaan mekanisme sinkronisasi. Dengan metodologi ini, kernel dapat selalu dipreemptifkan karena date kernel tertentu yang diupdate yang akan diproteksi dari proses berprioritas tinggi. Cara seperti ini digunakan di Solaris 2.

Fase konflik dari keterlambatan kehadiran memiliki dua komponen, yaitu:
1. Preemption dari berbagai proses terjadi di kernel
2. Pelepasan oleh proses berprioritas rendah yang sumber dayanya

dibutuh oleh prioritas tinggi. Sebagai contoh pada Solaris 2 dengan meniadakan kemampuan untuk preemption, keterlambatan berada di atas 100 ms; dengan memungkinkan untuk preemption, maka akan berkurang menjadi 2 ms.

49.5. Mengurangi Latency

Event latency merupakan sejumlah waktu yang berlalu, mulai dari ketika sebuah event terjadi sampai dengan ketika event tersebut dilayani. Biasanya event yang berbeda memiliki requirement latency yang berbeda. Performa dari sistem waktu nyata dipengaruhi oleh dua jenis keterlambatan berikut ini.

Interrupt latency

Interrupt latency merujuk kepada periode waktu dari kedatangan interupsi sampai dengan pada CPU sampai dengan dimulainya rutin dimana service diinterupsi. Ketika interupsi terjadi, sistem operasi pertama kali harus melengkapi instruksi yang dieksekusinya dan menentukan jenis dari interupsi yang terjadi. Kemudian harus disimpan state dari proses saat ini sebelum melayani interupsi menggunakan interrupt service routine (ISR) tertentu. Sebenarnya merupakan hal yang krusial bagi sistem operasi real time untuk mengurangi interrupt latency untuk menjamin bahwa real time task menerima perhatian yang cepat.

Satu faktor yang penting dalam berkontribusi untuk interrupt latency adalah jumlah waktu interupsi dapat di disable ketika kernel struktur data sedang diperbaiki. Sistem operasi real time membutuhkan interupsi untuk didisable untuk periode waktu yang sangat pendek. Walaupun demikian untuk sistem waktu nyata keras, interrupt latency tidak boleh hanya diminimalisir. Teknik yang paling efektif untuk menjaga dispatch latency, yaitu yang terdiri dari dua komponen yang merupakan tahapan konflik pada dispatch latency:

1. Preemptif pada setiap proses berjalan dikernel.
2. Dikeluarkan oleh sumber daya dari proses berprioritas rendah.

Dispatch latency

Satu isu yang dapat menyebabkan dispatch latency bertambah ketika proses berprioritas rendah butuh untuk membaca atau memodifikasi data kernel yang saat itu sedang diakses oleh oleh proses prioritas rendah- atau sebuah rantai proses berprioritas rendah. Sebagai kernel data yang dilindungi dengan sebuah lock, sebuah proses berprioritas tinggi harus menunggu yang prioritasnya rendah dalam menyelesaikan sumber dayanya. Situasi menjadi lebih rumit ketika proses prioritas rendah preemptif dengan proses lain yang prioritasnya lebih tinggi. Misalnya ada tiga proses L, M, H yang prioritasnya sebagai berikut L < M < H. Diasumsikan proses H membutuhkan sumber daya R, yang sedang diakses oleh proses L. Sekarang misalkan proses M dapat berjalan, menyelak proses L. Secara tidak langsung, proses dengan prioritas rendah, yaitu proses M, yang telah menyebabkan proses H semakin lama menunggu L untuk menyerahkan sumberdaya R.

Masalah ini dikenal dengan inversi prioritas, yang dapat dipecahkan dengan protokol priority-inheritance. Dengan adanya protokol ini, ketika ada sebuah proses yang sedang menggunakan sumber daya, kemudian ada proses lain dengan prioritas yang lebih tinggi yang juga membutuhkan sumber daya tersebut, maka proses tersebut akan mewariskan prioritasnya. Hal ini terjadi sampai dengan sumber daya selesai dieksekusi. Untuk contoh kasus diatas, maka L akan mewarisi prioritas H. Hal ini menyebabkan H tidak dapat menyelak untuk mengakses sumber daya R. Nilai dari L akan dikembalikan seperti semula sampai dengan R selesai digunakan. Hal ini menyebabkan H dapat langsung mengakses R, setelahnya.

49.6. Penjadual Proses

Penjadwalan untuk sistem pada sistem waktu nyata lembut tidak memberikan jaminan kapan proses yang kritis akan dijadualkan, akan tetapi memberikan jaminan bahwa proses tersebut akan didahulukan daripada proses yang tidak kritis. Untuk sistem waktu nyata keras, persyaratan penjadwalan lebih ketat, yaitu berdasarkan deadline. Sebuah task harus dilayani berdasarkan deadline-nya. Sehingga ketika deadline sudah kadaluarsa (melebihi deadline), maka task tidak akan mendapat pelayanan.

Karakteristik dari proses yang ada pada sistem waktu nyata adalah proses tersebut dianggap periodik, karena membutuhkan CPU pada interval yang konstan (periode). Setiap proses berperiode memiliki waktu pemrosesan yang fix yaitu w, setiap kali mendapatkan CPU, juga memiliki sebuah deadline d ketika harus dilayani oleh CPU, dan sebuah periode p, yang dapat diekspresikan sebagai $0 < t < d < p$. Rate dari task berpriodik adalah 1/p.

Hal yang tidak biasa dari bentuk penjadwalan ini adalah, proses akan mengumumkan deadlinenya pada penjadual. Kemudian dengan algoritma admission-control, penjadual akan menyatakan bahwa proses akan diselesaikan tepat waktu atau ditolak permohonannya karena tidak dapat menjamin bahwa task akan dapat dilayani sesuai deadline.

Berikut merupakan algoritma penjadwalan untuk sistem waktu nyata keras.

Penjadwalan *Rate-Monotonic*

Algoritma ini menjadualkan task berpriodik berdasarkan ketentuan prioritas statik dengan preemption. Jika proses berprioritas rendah sedang jalan dan prioritas tinggi siap untuk jalan, maka akan didahulukan proses dengan prioritas rendah.Untuk memasuki sistem, setiap task berpriodik mendapatkan prioritas dengan inversi dari periodenya. Semakin rendah periodenya makan akan semakin tinggi prioritasnya, dan demikian pula sebaliknya. Ketentuan ini sebenarnya memprioritaskan proses yang lebih sering menggunakan CPU.

Sebagai contoh adalah proses P1 dan P2, dimana periode P1 yaitu 50 (p1=50) dan periode dari P2 adalah 100 (p2=100). Sedangkan waktu pemrosesannya adalah t1=20 dan t2=35. Deadline dari proses mempersyaratkan untuk menyelesaikan CPU burst-nya pada awal dari periode berikutnya. Utilisasi CPU dari proses P1 yang merupakan rasio dari t1/p1, adalah 20/50=0,40 dan utilisasi CPU dari P2 adalah 35/100=0,35 sehingga total utilisasi CPU-nya adalah 0,75%. Dengan ini, tampaknya dapat memenuhi deadline dan menyisakan burst time.

Dengan menggunakan penjadwalan rate-monotonic maka P1 akan mendapat prioritas lebih tinggi dari P2, karena P1 lebih pendek daripada P2. P1 mulai terlebih dahulu dan menyelesaikan CPU burst pada waktu 20 (memenuhi deadline pertama). Kemudian dilanjutkan dengan P2 sampai dengan waktu 50, dimana tersisa 5 ms pada CPU burst-nya. P1 melanjutkan sampai waktu 70 (memenuhi deadline kedua), kemudian P2 menyelesaikan CPU burst-nya pada waktu 75 (memenuhi deadline pertama dari P2).

Algoritma rate-monotonic dianggap optimal apabila sebuah set proses tidak dapat dijadualkan oleh algoritma ini, serta tidak dapat pula dijadualkan oleh algoritma lain yang menggunakan prioritas statik. Sebagai contoh, diasumsikan proses P1 memiliki periode p1=50 dan CPU burst t1=25. Selain itu, ada proses P2 yang memiliki nilai p2=80 dan t2=35. Maka, proses P1 akan mendapat prioritas lebih besar. Total utilisasi CPU dari 2 proses diatas adalah (25/50) + (35/80)=0,94. Nampaknya, secara logikal keduanya dapat dijadualkan dan masih meninggalkan CPU dengan 6% waktu tersedia. Awalnya, P1 berjalan sampai dengan menyelesaikan CPU burst-nya pada waktu 25. Kemudian dilanjutkan dengan proses P2 yang berjalan sampai dengan waktu 50. P2 masih menyelesaikan burstnya 10. Kemudian dilanjutkan lagi oleh P1 sampai dengan waktu 75. Namun, P1 kehilangan deadline-nya untuk menyelesaikan burst-nya, yaitu pada waktu 80.

Selain menjadi optimal, penjadual rate-monotonic memiliki batasan utilisasi CPU yang terbatas dan tidak selalu mungkin untuk memaksimalkan secara penuh sumber daya CPU. Kemungkinan buruk utilisasi CPU untuk penjadwalan N proses adalah

$$\frac{1/n}{2(2^{1/n} - 1)}$$

EDF: *Earliest-Deadline-First*

Penjadwalan dilakukan berdasarkan deadline, yaitu semakin dekat deadline-nya maka semakin tinggi prioritasnya, dan demikian pula sebaliknya.Ketentuan yang berlaku, ketika proses akan mulai jalan, maka proses akan mengumumkan syarat deadline-nya pada sistem. Prioritas harus ditentukan untuk merefleksikan deadline dari proses yang baru dapat berjalan.

Sebagai contoh adalah kasus yang tidak dapat diselesaikan dengan penjadual rate-monotonic. Pada kasus ini, dengan EDF maka P1 akan mendapat prioritas awal lebih tinggi dari P2 karena P1 lebih mendekati deadline dari P2. Kemudian dilanjutkan oleh P2 sampai dengan akhir burst time dari P1.

Apabila rate-monotonic membiarkan P1 untuk melanjutkan kembali, maka pada EDF, P2 (dateline pada 80) yang melanjutkan karena lebih dekat dengan deadline daripada P1 (pada 100). Pada waktu 85 kedua proses sudah menyelesaikan deadline-nya masing-masing. Kemudian berlanjut lagi dengan P2, sampai dengan 100 maka P1 didahulukan kembali karena dateline-nya lebih awal dari P2.

Berbeda dengan algoritma rate-monotonic adalah, penjadual EDF tidak membutuhkan proses untuk periodik, dan tidak juga harus membutuhkan jumlah waktu CPU per burst yang konstan. Syarat satu-satunya adalah proses mengumumkan deadline-nya pada penjadual ketika dapat jalan. Secara teoritis, algoritma ini optimal, yaitu dapat memenuhi semua dateline dari proses dan juga dapat menjadikan utilisasi CPU menjadi 100%. Namun dalam kenyataan hal tersebut sulit terjadi karena cost dari context switching antara proses dan interrupt handler.

Penjadwalan *Proportional Share*

Penjadual ini akan mengalokasikan T bagian di antara semua aplikasi. Sebuah aplikasi dapat menerima N bagian waktu, yang menjamin bahwa aplikasi akan memiliki N/T dari total waktu prosesor. Sebagai contoh, diasumsikan ada total dari T=100 bagian untuk dibagi diantara tiga proses, yaitu A, B, dan C. A mendapatkan 50 bagian, B mendapat 15 bagian dan C mendapat 20 bagian. Hal ini menjamin bahwa A akan mendapat 50% dari total proses, B mendapar 15% dari total proses dan C mendapat 20% dari total proses.

Penjadwalan proportional share harus bekerja dengan memasukkan ketentuan admission control untuk menjamin bahwa aplikasi mendapatkan alokasi pembagian waktunya. Ketentuan admission control hanya akan menerima permintaan client terhadap sejumlah bagian apabila bagian yang diinginkan tersedia.

49.7. Penjadual Disk

Penjadwalan disk yang telah kita pelajari pada bab sebelumnya memfokuskan untuk menangani data yang konvensional, yang sasarannya adalah fairness dan troughput. Sedangkan pada penjadwalan waktu nyata dan multimedia yang menjadi tujuan adalah lebih kepada untuk mengatasi hambatan yang tidak dimiliki oleh data konvensional, yaitu: timing deadline dan rate requirement, sehingga dapat memiliki jaminan QoS. Namun sayangnya, kedua hambatan tersebut sering berkonflik. Berkas yang continous-media tipikalnya membutuhkan disk bandwidth rate yang sangat tinggi untuk memenuhi requirement data rate mereka. Karena disk memiliko transfer rate yang relatif rendah dan latency rate yang relatif tinggi maka penjadual disk harus mengurangi waktu latency untuk menjamin bandwidth yang tinggi. Bagaimanapun, mengurangi waktu latency dapat menyebabkan polisi dari penjadwalan yang tidak memberikan prioritas pada deadline.

Berikut merupakan algoritma penjadwalan yang memenuhi persyaratan QoS untuk sistem continous-media:

EDF: *Earliest Deadline first*

Penjadwalan EDF yang digunakan pada penjadwalan proses pada subbab sebelumnya, dapat digunakan pula untuk melakukan penjadwalan disk. EDF mirip dengan shortest-seek-time-first (SSTF), kecuali melayani permintaan terdekat dengan silinder saat itu karena EDF melayani permintaan yang terdekat dengan deadline. Algoritma ini cocok untuk karakter multimedia yang butuh respon secepat mungkin.

Masalah yang dihadapi dari pendekatan ini adalah pelayanan permintaan yang kaku berdasarkan deadline akan memiliki seek time yang tinggi, karena head dari disk harus secara random mencari posisi yang tepat tanpa memperhatikan posisinya saat ini.

Sebagai contoh adalah disk head pada silinder 75 dan antrian dari silinder (diurutkan berdasarkan deadline) adalah 98, 183, 105. Dengan EDF, maka head akan bergerak dari 75, ke 98, ke 183, dan balik lagi ke 105 (head melewati silinder 105 ketika berjalan dari 98 ke 183). Hal ini memungkinkan penjadual disk telah dapat melayani permintaan silinder 105 selama perjalanan ke silinder 183 dan masuh dapat menjaga persyaratan deadline dari silinder 183.

Penjadwalan SCAN-EDF

Masalah dasar dari penjadwalan EDF yang kaku adalah mengabaikan posisi dari read-write head dari disk; ini memungkinkan pergerakan head melayang secara liar ke dan dari disk, yang akan berdampak pada seek time yang tidak dapat diterima, sehingga berdampak negatif pada throughput dari disk. Hal ini pula yang dialami oleh penjadwalan FCFS dimana akhirnya dimunculkan penjadwalan SCAN, yang menjadi solusi.

SCAN-EDF merupakan algoritma hibrida dari kombinasi penjadwalan EDF dengan penjadwalan SCAN. SCAN-EDF dimulai dengan EDF ordering tetapi permintaan pelayanan dengan deadline yang sama menggunakan SCAN order. Apa yang terjadi apabila beberapa permintaan memiliki deadline yang berbeda yang relatif saling tertutup? Pada kasus ini, SCAN-EDF akan menumpuk permintaan, menggunakan SCAN ordering untuk melayani permintaan pelayanan yang ada dalam satu tumpukan. Ada banyak cara menumpuk permintaan dengan deadline yang mirip;satu-satunya syarat adalah reorder permintaan pada sebuah tumpukan tidak boleh menghalangi sebuah permintaan untuk dilayani berdasarkan deadlinenya. Apabila deadline tersebar merata, tumpukan dapat diatur pada grup pada ukuran tertentu. Pendekatan yang lain adalah dengan menumpuk permintaan yang deadlinenya jatuh pada treshold waktu yang diberikan, misalnya 10 permintaan pertumpukan.

Pendekatan lain adalah dengan menumpuk permintaan yang deadline-nya jatuh pada threshold waktu yang diberikan, misalnya 100 ms.

49.8. Manajemen Berkas

Manajemen berkas merupakan salah satu komponen dalam sebuah sistem operasi. Sebuah komputer dapat menyimpan informasi di dalam media yang berbeda-beda, seperti magnetic tape, disk, dan drum. Setiap perangkat tersebut memiliki karakteristik yang berbeda-beda.

Untuk kenyamanan dalam penggunaan sistem komputer, sistem operasi menyeragamkan penyajian suatu informasi kepada penggunanya karena seperti yang kita ketahui bahwa informasi yang disimpan dalam media penyimpanan sebenarnya tidaklah berwujud seperti apa yang kita lihat. Adalah tugas sistem operasi untuk melakukan mapping atas informasi yang tersimpan di dalam sebuah media penyimpanan ke dalam perangkat fisik sebagai perangkat akhir dimana pengguna dapat memperoleh informasi tersebut.

Berkas merupakan kumpulan informasi yang berhubungan (sesuai dengan tujuan pembuatan berkas tersebut). Sebuah berkas juga dapat memiliki struktur yang bersifat hirarkis (direktori, volume, dan lain-lain).

Sebuah sistem operasi memiliki tanggung jawab pada berkas sebagai berikut:
1. Pembuatan dan penghapusan sebuah berkas
2. Pembuatan dan penghapusan sebuah direktori
3. Pemanipulasian sebuah berkas atau direktori
4. Mapping berkas ke secondary storage
5. Melakukan back-up sebuah berkas ke media penyimpanan yang bersifat permanen (nonvolatile)

Karakteristik sistem multimedia:
1. Berkas multimedia biasanya memiliki karakteristik memiliki ukuran yang besar, untuk itu dalam mengatur atau memanajemen berkas mulitimedia diperlukan alokasi tempat dalam storage yang cukup besar pula. Selain itu untuk menjamin sebuah berkas multimedia dapat diakses dan dieksekusi dengan baik, diperlukan ukuran memori yang cukup serta spesifikasi lain dari komputer.
2. Memerlukan data rates yang sangat tinggi. Misalnya dalam sebuah video dijital, dimana frame dari video yang ingin ditampilkan beresolusi 800 x 600. Apabila kita menggunakan 24 bits untuk merepresentasikan warna pada tiap piksel, tiap frame berarti membutuhkan 800 x 600 x 24 = 11.520.000 bits data. Jika frame-frame tersebut ditampilkan pada kecepatan 30 frame/detik, maka bandwidth yang diperlukan adalah lebih dari 345 Mbps.
3. Aplikasi mutlimedia sensitif terhadap timing delay selama pemutaran ulang. Setiap kali berkas countinous-media dikirim kepada klien, pengiriman harus kontinu pada kecepatan tertentu selama pemutaran media tersebut. Jika tidak demikian, pendengar atau penonton dari berkas multimedia tersebut akan terganggu dan cenderung untuk melakukan pause dari presentasi berkas

multimedia tersebut.

49.9. Manajemen Jaringan

Sistem terdistribusi adalah sekumpulan prosesor yang tidak berbagi memori atau clock. Setiap prosesor mempunyai memori dan clock sendiri. Prosesor-prosesor tersebut terhubung melalui jaringan komunikasi sistem terdistribusi yang menyediakan akses pengguna ke bermacam- macam sumber daya sistem. Akses tersebut menyebabkan peningkatan kecepatan komputasi dan meningkatkan kemampuan penyediaan data.

Fungsi utama dari jaringan itu sendiri merupakan komunikasi. Model acuan dalam mempelajari jaringan komputer pada buku ini adalah dengan OSI dimana terdapat lapisan jaringan yang mengatur pengiriman paket data pada subnet. Fungsi utama dari lapisan jaringan ini adalah:
1. Menyediakan service terhadap lapisan transport
2. Menentukan rute pengiriman paket, apakah connection atau connectionless.
3. Mengontrol dan memanajemen jaringan, dengan menjaga:
   a. kestabilan, tidak macet dalam pengiriman sebuah paket.
   b. mengontrol arus paket (flow control) dalam jaringan.

Ketika suatu data dikirim melalui jaringan, prosesi transmisi yang berlangsung pasti mengalami hambatan atau keterlambatan yang disebabkan oleh lalu lintas jaringan yang begitu padat. Dalam kaitannya dengan multimedia, pengiriman data dalam sebuah jaringan harus memperhatikan masalah waktu. Yakni penyampaian data kepada klien harus tepat waktu atau paling tidak dalam batas waktu yang masih bisa ditoleransi.

Sebuah protokol yang dapat memperhatikan masalah waktu tersebut adalah real-time transport protocol (RTP). RTP adalah sebuah standar internet untuk melakukan pengiriman data secara real-time, yang meliputi audio dan video.

49.10. Unicasting dan Multicasting

Secara umum, terdapat tiga metode untuk melakukan pengiriman suatu data dari server ke klien dalam sebuah jaringan. Ketiga metode tersebut adalah:
1. **Unicasting.** Server mengirim data ke klien tunggal. Apabila data ingin dikirim ke lebih dari satu klien, maka server harus membangun sebuah unicast yang terpisah untuk masing-masing klien.
2. **Broadcasting.** Server mengirim data ke semua klien yang ada meskipun tidak semua klien meminta/membutuhkan data yang dikirim oleh server.
3. **Multicasting.** Server mengirim data ke suatu grup penerima data (klien) yang menginginkan data tersebut. Metode ini merupakan metode yang berada di pertengahan metode unicasting dan broadcasting.

49.11. Real-Time Streaming Protocol

Cara kerja dari media yang di-stream dapat dikirim oleh seorang klien dari sebuah standar web server adalah dengan pendekatan yang menggunakan hypertext transport protocaol atau HTTP yang merupakan protokol yang biasa digunakan untuk mengirim dokumen dari web browser. Biasanya, seorang klien menggunakan media player, seperti QuickTime, RealPlayer, atau Windows Media Player untuk memutar kembali media yang di-stream dari sebuah web server. Biasanya, pertama-tama klien akan meminta metafile, yang meliputi keterangan alamat letak data yang di-stream berada (URL), dari data yang di-stream. Metafile ini nantinya akan dikirim ke web browser klien, kemudian browser akan membuka berkas yang dimaksud dengan memilah media player yang sesuai dengan jenis media yang dispesifikasikan oleh metafile.

Masalah yang dihadapi dari pengiriman data yang di-stream dari standar web server adalah bahwa HTTP merupakan protokol yang tidak memiliki status. Sehingga, web server tidak dapat menjaga status dari koneksinya dengan klien. Keadaan ini mengakibatkan kesulitan yang dialami klien manakala ia ingin melakukan pause saat pengiriman data yang distream. Hal ini dikarenakan pelaksanaan pause membutuhkan pengetahuan (biasanya status) dari web browser, sehingga ketika klien ingin memulai kembali mengirim data melalui streaming web server dapat dengan mudah memutar kembali berdasarkan status yang disimpan tersebut.

Strategi alternatif yang dapat dilakukan untuk menanggulangi hal diatas adalah dengan menggunakan server streaming khusus yang didesain untuk men-streaming media, yaitu real-time streaming protocol (RTSP). RTSP didesain untuk melakukan komunikasi antara server yang melakukan streaming dengan media player. Keuntungan RTSP adalah bahwa protokol ini menyediakan koneksi yang memiliki status antara server dan klien, yang dapat mempermudah klien ketika ingin melakukan pause atau mencari posisi random dalam stream ketika memutar kembali data.

RTSP memiliki empat buah perintah. Perintah ini dikirim dari klien ke sebuah server streaming RTSP. Keempat perintah tersebut adalah:
1. **SETUP.** Server mengalokasikan sumber daya kepada client session.
2. **PLAY.** Server mengirim sebuah stream ke client session yang telah dibangun dari perintah SETUP sebelumnya.
3. **PAUSE.** Server menunda pengiriman stream namun tetap menjaga sumber daya yang telah dialokasikan.
4. **TEARDOWN.** Server memutuskan koneksi dan membebastugaskan sumber daya yang sebelumnya telah digunakan.

49.12. Kompresi

Kompresi merupakan pengurangan ukuran suatu berkas menjadi ukuran yang lebih kecil dari aslinya. Pengompresian berkas ini sangat menguntungkan manakala terdapat suatu berkas yang berukuran besar dan data di dalamnya mengandung banyak pengulangan karakter. Adapun teknik dari kompresi ini adalah dengan mengganti karakter yang berulang-ulang tersebut dengan suatu pola tertentu sehingga berkas tersebut dapat meminimalisasi ukurannya.

Misalnya terdapat kata "Hari ini adalah hari Jumat. Hari Jumat adalah hari yang menyenangkan". Jika kita telaah lagi, kalimat tersebut memiliki pengulangan karakter seperti karaktek pembentuk kata hari, hari Jum'at, dan adalah. Dalam teknik sederhana kompresi pada perangkat lunak, kalimat di atas dapat diubah menjadi pola sebagai berikut

```
# ini $ %. % $ # ya@ menyena@kan.
```

dimana dalam kalimat diatas, karakter pembentuk hari diubah menjadi karakter #, hari Jum'at menjadi %, adalah menjadi $, ng menjadi @. Saat berkas ini akan dibaca kembali, maka perangkat lunak akan mengembalikan karakter tersebut menjadi karakter awal dalam kalimat. Pengubangan karakter menjadi lebih singkat hanya digunakan agar penyimpanan kalimat tersebut dalam memory komputer tidak memakan tempat yang banyak.

Implementasi kompresi dalam personal computer (PC) dibagi menjadi tiga cara, yaitu:
1. **Pengkompresian berkas berdasarkan kegunaannya.** Utility-based file compression, merupakan jenis kompresi yang melakukan kompresi per berkas dengan menggunakan utilitas kompresi. Untuk melihat berkas yang telah dikompresi, berkas tersebut harus didekompres dengan menggunakan utilitas dekompresi. Dalam jenis ini, sistem operasi tidak tahu menahu mengenai aktivitas kompresian atau dekompresi sebuah berkas. Contoh dari sistem kompresi yang cukup terkenal adalah PKZIP, WinZip, WinRar, dan lain-lain.
2. **Pengkompresian berkas pada sistem operasi.** Operating system file compression, beberapa sistem operasi memiliki sistem kompresi di dalamnya. Contoh dari sistem operasi yang memiliki sistem kompresi di dalamnya adalah Windows NT yang menggunakan sistem berkas NTFS. Dengan menggunakan sistem kompresi seperti ini, sistem operasi secara otomatis dapat mendekompres berkas yang telah dikompresi manakala berkas tersebut ingin digunakan oleh sebuah program.
3. **Pengkompresian Isi.** Volume compression, dengan pengkompresian ini, kita dapat menghemat penggunaan space pada disk tanpa harus mengkompresi tiap berkas di dalamnya secara individual. Setiap berkas yang dikopi ke dalam volume compression akan dikompresi secara otomatis dan akan didekompresi secara otomatis apabila berkas tersebut dibutuhkan oleh sebuah program.

Dalam kaitannya dengan mutltimedia, kompresi sangat menguntungkan karena dapat menghemat

tempat penyimpanan serta dapat mempercepat proses pengiriman data kepada klien, sebab pengiriman berkas dengan ukuran yang lebih kecil lebih cepat daripada berkas yang memiliki ukuran besar. Kompresi juga penting manakala suatu data di-stream melalui sebuah jaringan.

Algoritma kompresi diklasifikasikan menjadi dua buah, yaitu:

1. **Algoritma kompresi lossy.** Keuntungan dari algoritma ini adalah bahwa rasio kompresi (perbandingan antara ukuran berkas yang telah dikompresi dengan berkas sebelum dikompresi) cukup tinggi. Namun algoritma ini dapat menyebabkan data pada suatu berkas yang dikompresi hilang ketika didekompresi. Hal ini dikarenakan cara kerja algoritma lossy adalah dengan mengeliminasikan beberapa data dari suatu berkas. Namun data yang dieliminasikan biasanya adalah data yang kurang diperhatikan atau diluar jangkauan manusia, sehingga pengeliminasian data tersebut kemungkinan besar tidak akan mempengaruhi manusia yang berinteraksi dengan berkas tersebut. Contohnya pada pengkompresian berkas audio, kompresi lossy akan mengeleminasi data dari berkas audio yang memiliki frekuensi sangat tinggi/rendah yang berada diluar jangkauan manusia. Beberapa jenis data yang biasanya masih dapat mentoleransi algoritma lossy adalah gambar, audio, dan video.
2. **Algoritma kompresi lossless.** Berbeda dengan algoritma kompresi lossy, pada algoritma kompresi lossless, tidak terdapat perubahan data ketika mendekompresi berkas yang telah dikompresi dengan kompresi lossless ini. Algoritma ini biasanya diimplementasikan pada kompresi berkas teks, seperti program komputer (berkas zip, rar, gzip, dan lain-lain).

Contoh lain dari penerapan algoritma kompresi lossy dalam kehidupan sehari-hari adalah pada berkas berformat MPEG (Moving Picture Experts Group). MPEG merupakan sebuah set dari format berkas dan standar kompresi untuk video digital.

49.13. Rangkuman

Waktu nyata dapat dibagi dua, yaitu waktu nyata keras dan waktu nyata lembut.Sistem waktu nyata keras menjamin pekerjaan yang kritis akan diselesaikan dengan tepat waktu. Sedangkan sistem yang dikatakan seba- gai sistem waktu nyata lembut adalah sistem yang batasannya kurang, dimana pekerjaan yang kritis mendapat prioritas setelah pekerjaan yang lain.

Algoritma Penjadual Waktu Nyata diantaranya adalah Earliest Deadline first (EDF) dan SCAN-EDF Schedulling

Manajemen berkas merupakan salah satu komponen dalam sebuah sistem operasi. Sebuah sistem operasi memiliki tanggung jawab pada berkas sebagai seperti dalam pembuatan dan penghapusan sebuah berkas, pembuatan dan penghapusan sebuah direktori, pemanipulasian terhadap sebuah berkas atau direktori, memetakan berkas ke secondary storage, serta melakukan back-up sebuah berkas ke media penyimpanan yang bersifat permanen (non-volitile).

Fungsi utama dari jaringan adalah untuk komunikasi. Secara umum, terdapat tiga metode untuk melakukan pengiriman suatu data dari server ke klien dalam sebuah jaringan. Ketiga metode tersebut adalah unicasting, broadcasting, dan multicasting. Real-stream streaming protocol memiliki empat perintah, yaitu setup, play, pause dan teardown.

Kompresi merupakan pengurangan ukuran suatu berkas menjadi ukuran yang lebih kecil dari aslinya. Implementasi kompresi dalam personal computer (PC) dibagi menjadi tiga cara, yaitu pengkompresian berkas berdasarkan kegunaannya (Utility-based file compression), pengkompresian berkas pada sistem operasi (Operating system file compression), dam pengkompresian isi (Volume compression). Algoritma kompresi yang cukup dikenal adalah algoritma lossy dan lossless

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[WEBKozierok2005] Charles M Kozierok. 2005. *Reference Guide – Hard Disk Drives http://www.storagereview.com/ guide/* . Diakses 9 Agustus 2006.

Bab 50. Sistem Terdistribusi

50.1. Pendahuluan

Pengguna tidak sadar akan multiplisitas perangkat. Akses terhadap sumber daya jarak jauh sama dengan sumber daya lokal.

1. Remote loging ke dalam perangkat jarak jauh yang sesuai.
2. Migrasi Data dengan memindahkan seluruh data atau hanya mentransfer file yang diperlukan untuk task langsung.
3. Computation Migration, memindahkan komputasi bukannya data, melalui sistem.

Process Migration: Mengeksekusi seluruh proses arat bagian daripadanya pada situs/tempat yang berbeda.

1. Load balancing
2. Computation Speedup
3. Hardware preference
4. Software Preference
5. Data Access

Tipe-tipe koneksi

1. LAN = di rancang untuk mencakup area yang kecil.

2. WAN = menghubungkan situs-situs yang terpisah secara geografi

Aplikasi

1. Transmisi dari sebuah paket network antar host dalam sebuah jaringan ethernet.

2. Setiap host mempunyai sebuah IP unik dan sebuah alamat ethernet yang berkorespondensi

3. Komunikasi membutuhkan alamat keduanya.

4. DNS yang dapat digunakan untuk mendapatkan alamat IP.

Sistem berkas

Sebuah implementasi dari model klasik time sharing dari sebuah sistem file, dimana banyak user dapat berbagi file dan penyimpanan resource.

Struktur sistem berkas

1. Servis: Software entity yang berjalan pada sebuah atau lebih hardware dan menyediakan sebuah tipe fungsi partikular pada client yang tidak dikenal
2. Server: Software servis yang berjalan pada sebuah mesin
3. Client: Proses yang dapat mengivoke sebuah servis mengunakan set operasi yang membentuk client interfacenya

Replikasi

1. Replika dari file yang sama di dalam mesin yang failure-independent
2. Memperbaiki ketersediaan dan dapat mempersingkat waktu servis
3. Schema penamaan memetakan sebuah nama file yang di replikasi pada sebuah replika yang partikulir.
4. Update, sebuah update terhadap replika apapun harus direfleksikan pada replika lainnya.

Asumsi

Sistem terdiri dari n proses, tiap proses berada di dalam processor yang berbeda Setiap proses mempunyai critical section yang membutuhkan mutual exclusion

Requirement

Jika sebuah proses sedang mengeksekusi critical section, maka tidak ada proses lain yang mengeksekusi critical sectionnya. Terdapat dua pendekatan yaitu:

1. **Centralized Approach.**
   a. Satu dari proses dalam sistem dipilih untuk mengkoordinasikan entry terhadap critical section.
   b. Sebuah proses yang ingin memasuki critical section meminta coordinator dengan mengirimkan pesan.
   c. Koordinator memutuskan proses mana yang dapat memasuki critical section, dengan mengirimkan pesan balasan.
   d. Ketika sebuah proses menerima pesan dari koordinator maka ia memasuki critical section.
   e. Setelah mengakhiri critical sectionnya, sebuah proses mengirimkan pesan lagi kepada koordinator. Lalu keluar dari C. S.
2. **Fully Distributed Approach.**
   a. Ketika sebuah proses ingin memasuki critical sectionya, ia membangkitkan sebuah timestamp baru dan mengirimkan pesan permohonan kepada seluruh proses lain.
   b. Ketika proses menerima pesan permohonan, ia dapat langsung membalas atau mengirim reply back.
   c. Ketika pesan tersebut menerima reply dari seluruh proese lain, ia dapat memasuki critical section.
   d. Ketika ia mengakhiri C.S. proses mengirimkan reply message.

50.2. Topologi Jaringan

Situs dalam sistem dapat terkoneksi secara fisik, dalam berbagai cara, yang dibandingkan terhadap berbagai kriteria tertentu.

1. Biaya dasar, yaitu berapa biaya untuk menghubungkan berbagai situs dalam sistem.
2. Biaya komunikasi, yiatu berapa waktu yang dibutuhkan untuk membawa sebuah pesan dari situs A ke situs B.
3. Relibilitas, jika sebuah link dalam sebuah situs terputus, apakan situs-situs yang lain dapat berfungsi dengan normal.

50.3. Isu Lainnya

- Mutex
- Sistem Berkas
- Replikasi Berkas
- Mutex
- *Middleware*
- Aplikasi
- Kluster

50.4. Rangkuman

Ini merupakan rangkuman perihal sistem terdistribusi.

Rujukan

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

Bab 51. Keamanan Sistem

51.1. Pendahuluan

Pertama-tama kita harus mengetahui perbedaan antara keamanan dan proteksi. Proteksi menyangkut mengenai faktor-faktor internal suatu sistem komputer. Sedangkan keamanan mempertimbangkan faktor-faktor eksternal (lingkungan) di luar sistem dan faktor proteksi terhadap sumber daya sistem. Melihat perbedaan ini, terlihat jelas bahwa keamanan mencakup hal yang lebih luas dibanding dengan proteksi.

Bagaimana suatu sistem dapat dikatakan aman? Suatu sistem baru dapat dikatakan aman apabila resource yang digunakan dan diakses sesuai dengan kehendak user dalam berbagai keadaan. Sayangnya, tidak ada satu sistem komputer pun yang memiliki sistem keamanan yang sempurna. Data atau informasi penting yang seharusnya tidak dapat diakses oleh orang lain mungkin dapat diakses, dibaca ataupun diubah oleh orang lain.

Oleh karena itu dibutuhkan suatu keamanan sistem untuk menanggulangi kemungkinan dimana informasi penting dapat diakses oleh orang lain. Diatas dijelaskan bahawa tidak ada satu sistem komputer yang memiliki sistem keamanan yang sempurna. Akan tetapi, setidaknya kita harus mempunyai suatu mekanisme yang membuat pelanggaran semacam itu jarang terjadi.

Dalam bab ini kita akan membahas hal-hal yang menyangkut mengenai suatu keamanan sistem, dengan mempelajarinya diharapkan akan membantu kita mengurangi pelanggaran-pelanggaran yang dapat terjadi.

51.2. Manusia dan Etika

Berbicara mengenai manusia dan etika, kita mengetahui bahwa di muka bumi ini terdapat bermacam-macam karakter orang yang berbeda-beda. Sebagian besar orang memiliki hati yang baik dan selalu mencoba untuk menaati peraturan. Akan tetapi, ada beberapa orang jahat yang ingin menyebabkan kekacauan. Dalam konteks keamanan, orang-orang yang membuat kekacauan di tempat yang tidak berhubungan dengan mereka disebut intruder. Ada dua macam intruder, yaitu:

1. **Passive intruder.** Intruder yang hanya ingin membaca berkas yang tidak boleh mereka baca.
2. **Active intruder.** Lebih berbahaya dari passive intruder. Mereka ingin membuat perubahan yang tidak diijinkan (unauthorized) pada data.

Ketika merancang sebuah sistem yang aman terhadap intruder, penting untuk mengetahui sistem tersebut akan dilindungi dari intruder macam apa. Empat contoh kategori:

1. **Keingintahuan seseorang tentang hal-hal pribadi orang lain.** Banyak orang mempunyai PC yang terhubung ke suatu jaringan dan beberapa orang dalam jaringan tersebut akan dapat membaca e-mail dan file-file orang lain jika tidak ada 'penghalang' yang ditempatkan. Sebagai contoh, sebagian besar sistem UNIX mempunyai default bahwa semua file yang baru diciptakan dapat dibaca oleh orang lain.
2. **Penyusupan oleh orang-orang dalam.** Pelajar, programmer sistem, operator, dan teknisi menganggap bahwa mematahkan sistem keamanan komputer lokal adalah suatu tantangan. Mereka biasanya sangat ahli dan bersedia mengorbankan banyak waktu untuk usaha tersebut.
3. **Keinginan untuk mendapatkan uang.** Beberapa programmer bank mencoba mencuri uang dari bank tempat mereka bekerja dengan cara-cara seperti mengubah software untuk memotong bunga daripada membulatkannya, menyimpan uang kecil untuk mereka sendiri, menarik uang dari account yang sudah tidak digunakan selama bertahun-tahun, untuk memeras ("Bayar saya, atau saya akan menghancurkan semua record bank anda").
4. **Espionase komersial atau militer.** Espionase adalah usaha serius yang diberi dana besar oleh saingan atau negara lain untuk mencuri program, rahasia dagang, ide-ide paten, teknologi, rencana bisnis, dan sebagainya. Seringkali usaha ini melibatkan wiretaping atau antena yang diarahkan pada suatu komputer untuk menangkap radiasi elektromagnetisnya.

Perlindungan terhadap rahasia militer negara dari pencurian oleh negara lain sangat berbeda dengan perlindungan terhadap pelajar yang mencoba memasukkan message-of-the-day pada suatu sistem. Terlihat jelas bahwa jumlah usaha yang berhubungan dengan keamanan dan proteksi tergantung

pada siapa "musuh"nya.

51.3. Kebijaksanaan Pengamanan

Kebijaksanaan pengamanan yang biasa digunakan yaitu yang bersifat sederhana dan umum. Dalam hal ini berarti tiap pengguna dalam sistem dapat mengerti dan mengikuti kebijaksanaan yang telah ditetapkan. Isi dari kebijaksanaan itu sendiri berupa tingkatan keamanan yang dapat melindungi data-data penting yang tersimpan dalam sistem. Data-data tersebut harus dilindungi dari tiap pengguna yang menggunakan sistem tersebut.

Beberapa hal yang perlu dipertimbangkan dalam menentukan kebijaksanaan pengamanan adalah: siapa sajakah yang memiliki akses ke sistem, siapa sajakah yang diizinkan untuk menginstall program ke dalam sistem, siapa sajakah memiliki data-data tertentu, perbaikan terhadap kerusakan yang mungkin terjadi, dan penggunaan yang wajar dari sistem.

51.4. Keamanan Fisik

Lapisan keamanan pertama yang harus diperhitungkan adalah keamanan secara fisik dalam sistem komputer. Keamanan fisik menyangkut tindakan mengamankan lokasi adanya sistem komputer terhadap intruder yang bersenjata atau yang mencoba menyusup ke dalam sistem komputer.

Pertanyaan yang harus dijawab dalam menjamin keamanan fisik antara lain:
1. Siapa saja yang memiliki akses langsung ke dalam sistem?
2. Apakah mereka memang berhak?
3. Dapatkah sistem terlindung dari maksud dan tujuan mereka?
4. Apakah hal tersebut perlu dilakukan?

Banyak keamanan fisik yang berada dalam sistem memiliki ketergantungan terhadap anggaran dan situasi yang dihadapi. Apabila pengguna adalah pengguna rumahan, maka kemungkinan keamanan fisik tidak banyak dibutuhkan. Akan tetapi, jika pengguna bekerja di laboratorium atau jaringan komputer, banyak yang harus dipikirkan.

Saat ini, banyak komputer pribadi memiliki kemampuan mengunci. Biasanya kunci ini berupa socket pada bagian depan casing yang bisa dimasukkan kunci untuk mengunci ataupun membukanya. Kunci casing dapat mencegah seseorang untuk mencuri dari komputer, membukanya secara langsung untuk memanipulasi ataupun mencuri perangkat keras yang ada.

51.5. Keamanan Perangkat Lunak

Contoh dari keamanan perangkat lunak yaitu BIOS. BIOS merupakan perangkat lunak tingkat rendah yang mengkonfigurasi atau memanipulasi perangkat keras tertentu. BIOS dapat digunakan untuk mencegah penyerang mereboot ulang mesin dan memanipulasi sistem LINUX.

Contoh dari keamanan BIOS dapat dilihat pada LINUX, dimana banyak PC BIOS mengizinkan untuk mengeset password boot. Namun, hal ini tidak banyak memberikan keamanan karena BIOS dapat direset, atau dihapus jika seseorang dapat masuk ke case. Namun, mungkin BIOS dapat sedikit berguna. Karena jika ada yang ingin menyerang sistem, untuk dapat masuk ke case dan mereset ataupun menghapus BIOS akan memerlukan waktu yang cukup lama dan akan meninggalkan bekas. Hal ini akan memperlambat tindakan seseorang yang mencoba menyerang sistem.

51.6. Keamanan Jaringan

Pada dasarnya, jaringan komputer adalah sumber daya (*resources*) yang dibagi dan dapat digunakan oleh banyak aplikasi dengan tujuan berbeda. Kadang-kadang, data yang ditransmisikan antara aplikasi- aplikasi merupakan rahasia, dan aplikasi tersebut tentu tidak mau sembarang orang membaca data tersebut.

Sebagai contoh, ketika membeli suatu produk melalui internet, pengguna (*user*) memasukkan nomor

kartu kredit ke dalam jaringan. Hal ini berbahaya karena orang lain dapat dengan mudah menyadap dan membaca data tsb pada jaringan. Oleh karena itu, user biasanya ingin mengenkripsi (encrypt) pesan yang mereka kirim, dengan tujuan mencegah orang-orang yang tidak diizinkan membaca pesan tersebut.

51.7. Kriptografi

Dasar enkripsi cukup sederhana. Pengirim menjalankan fungsi enkripsi pada pesan plaintext, ciphertext yang dihasilkan kemudian dikirimkan lewat jaringan, dan penerima menjalankan fungsi dekripsi (decryption) untuk mendapatkan plaintext semula. Proses enkripsi/dekripsi tergantung pada kunci (key) rahasia yang hanya diketahui oleh pengirim dan penerima. Ketika kunci dan enkripsi ini digunakan, sulit bagi penyadap untuk mematahkan ciphertext, sehingga komunikasi data antara pengirim dan penerima aman.

Kriptografi macam ini dirancang untuk menjamin privacy: mencegah informasi menyebar luas tanpa ijin. Akan tetapi, privacy bukan satu-satunya layanan yang disediakan kriptografi. Kriptografi dapat juga digunakan untuk mendukung authentication (memverifikasi identitas user) dan integritas (memastikan bahwa pesan belum diubah).

Kriptografi digunakan untuk mencegah orang yang tidak berhak untuk memasuki komunikasi, sehingga kerahasiaan data dapat dilindungi. Secara garis besar, kriptografi digunakan untuk mengirim dan menerima pesan. Kriptografi pada dasarnya berpatokan pada key yang secara selektif telah disebar pada komputer-komputer yang berada dalam satu jaringan dan digunakan untuk memproses suatu pesan.

51.8. Operasional

Keamanan operasional (*operations security*) adalah tindakan apa pun yang menjadikan sistem beroperasi secara aman, terkendali, dan terlindung. Yang dimaksud dengan sistem adalah jaringan, komputer, lingkungan. Suatu sistem dinyatakan operasional apabila sistem telah dinyatakan berfungsi dan dapat dijalankan dengan durasi yang berkesinambungan, yaitu dari hari ke hari, 24 jam sehari, 7 hari seminggu.

Manajemen Administratif (*Administrative Management*) adalah penugasan individu untuk mengelola fungsi-fungsi keamanan sistem. Beberapa hal yang terkait:

1. **Pemisahan Tugas (*Separation of Duties*).** Menugaskan hal-hal yang menyangkut keamanan kepada beberapa orang saja. Misalnya, yang berhak menginstall program ke dalam sistem komputer hanya admin, user tidak diberi hak tersebut.
2. **Hak Akses Minimum (*Least Privilege*).** Setiap orang hanya diberikan hak akses minimum yang dibutuhkan dalam pelaksanaan tugas mereka.
3. **Keingin-tahuan (*Need to Know*).** Yang dimaksud dengan need to know adalah pengetahuan akan informasi yang dibutuhkan dalam melakukan suatu pekerjaan.

Kategori utama dari kontrol keamanan operasional antara lain:

1. **Kendali Pencegahan (*Preventative Control*).** Untuk mencegah error dan intruder memasuki sistem. Misal, kontrol pencegahan untuk mencegah virus memasuki sistem adalah dengan menginstall antivirus.
2. **Kontrol Pendeteksian (*Detective Control*).** Untuk mendeteksi error yang memasuki sistem. Misal, mencari virus yang berhasil memasuki sistem.
3. **Kontrol Perbaikan (*Corrective/Recovery Control*).** Membantu mengembalikan data yang hilang melalui prosedur recovery data. Misal, memperbaiki data yang terkena virus.

Kategori lainnya mencakup:

1. **Kendali Pencegahan (*Deterrent Control*).** Untuk menganjurkan pemenuhan (*compliance*) dengan kontrol eksternal.
2. **Kendali Aplikasi (*Application Control*).** Untuk memperkecil dan mendeteksi operasi-operasi perangkat lunak yang tidak biasa.
3. **Kendali Transaksi (*Transaction Control*).** Untuk menyediakan kendali di berbagai tahap transaksi (dari inisiasi sampai keluaran, melalui kontrol testing dan kontrol perubahan).

51.9. BCP/DRP

Berdasarkan pengertian, BCP atau Business Continuity Plan adalah rencana bisnis yang berkesinambungan, sedangkan DRP atau Disaster Recovery Plan adalah rencana pemulihan dari kemungkinan kerusakan-kerusakan yang terjadi.

Aspek yang terkandung di dalam suatu rencana bisnis yang berkesinambungan yaitu rencana pemulihan dari kemungkinan kerusakan-kerusakan yang terjadi. Dengan kata lain, DRP terkandung di dalam BCP.

Rencana untuk pemulihan dari kerusakan, baik yang disebabkan oleh alam maupun manusia, tidak hanya berdampak pada kemampuan proses komputer suatu perusahaan, tetapi juga akan berdampak pada operasi bisnis perusahaan tersebut. Kerusakan-kerusakan tersebut dapat mematikan seluruh sistem operasi. Semakin lama operasi sebuah perusahaan mati, maka akan semakin sulit untuk membangun kembali bisnis dari perusahaan tersebut.

Konsep dasar pemulihan dari kemungkinan kerusakan-kerusakan yang terjadi yaitu harus dapat diterapkan pada semua perusahaan, baik perusahaan kecil maupun perusahaan besar. Hal ini tergantung dari ukuran atau jenis prosesnya, baik yang menggunakan proses manual, proses dengan menggunakan komputer, atau kombinasi dari keduanya.

Pada perusahaan kecil, biasanya proses perencanaannya kurang formal dan kurang lengkap. Sedangkan pada perusahaan besar, proses perencanaannya formal dan lengkap. Apabila rencana tersebut diikuti maka akan memberikan petunjuk yang dapat mengurangi kerusakan yang sedang atau yang akan terjadi.

51.10. Proses Audit

Audit dalam konteks teknologi informasi adalah memeriksa apakah sistem komputer berjalan semestinya. Tujuh langkah proses audit:
1. Implementasikan sebuah strategi audit berbasis manajemen risiko serta control practice yang dapat disepakati semua pihak.
2. Tetapkan langkah-langkah audit yang rinci.
3. Gunakan fakta/bahan bukti yang cukup, handal, relevan, serta bermanfaat.
4. Buatlah laporan beserta kesimpulannya berdasarkan fakta yang dikumpulkan.
5. Telaah apakah tujuan audit tercapai.
6. Sampaikan laporan kepada pihak yang berkepentingan.
7. Pastikan bahwa organisasi mengimplementasikan managemen risiko serta control practice.

Sebelum menjalankan proses audit, tentu saja proses audit harus direncanakan terlebih dahulu. Audit planning (perencanaan audit) harus secara jelas menerangkan tujuan audit, kewenangan auditor, adanya persetujuan managemen tinggi, dan metode audit. Metodologi audit:
1. **Audit subject.** Menentukan apa yang akan diaudit.
2. **Audit objective.** Menentukan tujuan dari audit.
3. **Audit Scope.** Menentukan sistem, fungsi, dan bagian dari organisasi yang secara spesifik/khusus akan diaudit.
4. **Preaudit Planning.** Mengidentifikasi sumber daya dan SDM yang dibutuhkan, menentukan dokumen-dokumen apa yang diperlukan untuk menunjang audit, menentukan lokasi audit.
5. **Audit procedures dan steps for data gathering.** Menentukan cara melakukan audit untuk memeriksa dan menguji kendali, menentukan siapa yang akan diwawancara.
6. **Evaluasi hasil pengujian dan pemeriksaan.** Spesifik pada tiap organisasi.
7. **Prosedur komunikasi dengan pihak manajemen.** Spesifik pada tiap organisasi.
8. **Audit Report Preparation.** Enentukan bagaimana cara mereview hasil audit, yaitu evaluasi kesahihan dari dokumen-dokumen, prosedur, dan kebijakan dari organisasi yang diaudit.

Struktur dan isi laporan audit tidak baku, tapi umumnya terdiri atas:
- **Pendahuluan.** Tujuan, ruang lingkup, lamanya audit, prosedur audit.
- **Kesimpulan umum dari auditor.** -
- **Hasil audit.** Apa yang ditemukan dalam audit, apakah prosedur dan kontrol layak atau tidak.
- **Rekomendasi.** Tanggapan dari manajemen (bila perlu).
- **Exit interview.** Interview terakhir antara auditor dengan pihak manajemen untuk membicarakan

temuan-temuan dan rekomendasi tindak lanjut. Sekaligus meyakinkan tim manajemen bahwa hasil audit sahih.

51.11. Rangkuman

Data atau informasi penting yang seharusnya tidak dapat diakses oleh orang lain mungkin dapat diakses, baik dibaca ataupun diubah oleh orang lain. Kita harus mempunyai suatu mekanisme yang membuat pelanggaran jarang terjadi. Ketika merancang sebuah sistem yang aman terhadap intruder, penting untuk mengetahui sistem tersebut akan dilindungi dari intruder macam apa. Untuk menjaga sistem keamanan sebuah komputer dapat dicapai dengan berbagai cara, antara lain:

- **Keamanan Fisik.** Hal ini tergantung oleh anggaran dan situasi yang dihadapi.
- **Keamanan Perangkat Lunak.** Contoh dari keamanan perangkat lunak yaitu BIOS.
- **Keamanan Jaringan.** Yaitu dengan cara kriptografi.

DRP (Disaster Recovery Plan) terkandung di dalam BCP (Business Continuity Plan). Konsep dasar DRP harus dapat diterapkan pada semua perusahaan. Proses audit bertujuan untuk memeriksa apakah sistem komputer berjalan dengan semestinya.

Rujukan

[KrutzVines2001] Ronald L Krutz dan Russell D Vines. 2001. *The CISSP Prep Guide Mastering the Ten Domains of Computer Security*. John Wiley & Sons.

[PeterDavie2000] Larry L Peterson dan Bruce S Davie. 2000. *Computer Networks A Systems Approach*. Second Edition. Morgan Kaufmann.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

Bab 52. Perancangan dan Pemeliharaan

52.1. Pendahuluan

Merancang sebuah sistem operasi merupakan hal yang sulit. Merancang sebuah sistem sangat berbeda dengan merancang sebuah algoritma. Hal tersebut disebabkan karena keperluan yang dibutuhkan oleh sebuah sistem sulit untuk didefinisikan secara tepat, lebih kompleks dan sebuah sistem memiliki struktur internal dan antarmuka internal yang lebih banyak serta ukuran dari kesuksesan dari sebuah sistem sangat abstrak.

Masalah pertama dalam merancang sistem operasi adalah mendefinisikan tujuan dan spesifikasi sistem. Pada level tertinggi, rancangan sistem akan dipengaruhi oleh pemilihan perangkat keras serta jenis sistem seperti *batch, time shared, single user, multiuser, distributed, real time* atau tujuan umum.

Berdasarkan tingkatan rancangan tertinggi, kebutuhan sistem akan lebih sulit untuk dispesifikasi. Kebutuhan sistem dapat dibagi menjadi dua kelompok utama, yaitu user goal dan sistem goal. User menginginkan properti sistem yang pasti seperti: sistem harus nyaman dan mudah digunakan, mudah dipelajari, reliable, aman dan cepat.

Sekumpulan kebutuhan dapat juga didefinisikan oleh orang-orang yang harus mendesain, membuat, memelihara dan mengoperasikan sistem operasi seperti: sistem operasi harus mudah didesain, diimplementasikan dan dipelihara, sistm harus fleksibel, reliable, bebas eror dan efisien.

Yang harus diperhatikan ketika merancang sebuah sistem yang baik adalah apakah sistem tersebut memenuhi tiga kebutuhan: fungsionalitas: apakah sistem tersebut bekerja dengan baik?, kecepatan: apakah sistem tersebut cukup cepat?, dan fault-tolerance: apakah sistem tersebut dapat terus bekerja?. Berikut,akan dijabarkan prinsip-prinsip dalam merancang sistem operasi tersebut.

Extensibility

Extensibility terkait dengan kapasitas sistem operasi untuk tetap mengikuti perkembangan teknologi komputer, sehingga setiap perubahan yang terjadi dapat difasilitasi setiap waktu, pengembang sistem operasi modern menggunakan arsitektur berlapis, yaitu struktur yang modular. Karena struktur yang modular tersebut, tambahan subsystem pada sistem operasi dapat ditambahkan tanpa mempengaruhi subsystem yang sudah ada.

Portability

Suatu sistem operasi dikatakan portable jika dapat dipindahkan dari arsitektur hardware yang satu ke yang lain dengan perubahan yang relatif sedikit. Sistem operasi modern dirancang untuk portability. Keseluruhan bagian sistem ditulis dalam bahasa C dan C++. Semua kode prosesor diisolasi di DLL (Dynamic Link Library) disebut dengan abstraksi lapisan hardware.

Reliability

Adalah kemampuan sistem operasi untuk mengatasi kondisi eror, termasuk kemampuan sistem operasi untuk memproteksi diri sendiri dan pengggunanya dari software yang cacat. Sistem operasi modern menahan diri dari serangan dan cacat dengan menggunakan proteksi perangkat keras untuk memori virtual dan mekanisme proteksi perangkat lunak untuk sumber daya sistem operasi.

Security

Sistem operasi harus memberikan keamanan terhadap data yang disimpan dalam semua drive.

High Performance

Sistem operasi dirancang untuk memberikan kinerja tinggi pada sistem desktop, server sistem multi-thread yang besar dan multiprosesor. untuk memenuhi kebutuhan kinerja, sistem operasi menggunakan variasi teknik seperti asynchronous I/O, optimized protocols untuk jaringan, grafik berbasis kernel, dan caching data sistem file.

52.2. Perancangan Antarmuka

Merancang antarmuka merupakan bagian yang paling penting dari merancang sistem. Biasanya hal tersebut juga merupakan bagian yang paling sulit, karena dalam merancang antarmuka harus memenuhi tiga persyartan: sebuah antarmuka harus sederhana, sebuah antarmuka harus lengkap, dan sebuah antarmuka harus memilki kinerja yang cepat.

Alasan utama mengapa antarmuka sulit untuk dirancang adalah karena setiap antarmuka adalah sebuah bahasa pemrograman yang kecil: antarmuka menjelaskan sekumpulan obyek-obyek dan operasi-operasi yang bisa digunakan untuk memanipulasi obyek.

52.3. Implementasi

Rancangan Sistem

Desain sistem memiliki masalah dalam menentukan tujuan dan spesifikasi sistem. Pada level paling tinggi, desain sistem akan dipengaruhi oleh pilihan perangkat keras dan jenis sistem. Kebutuhannya akan lebih sulit untuk dispesifikasikan. Kebutuhan terdiri dari target user dan target sistem. User menginginkan sistem yang nyaman digunakan, mudah dipelajari, dapat dipercaya, aman, dan cepat. Namun itu semua tidaklah signifikan untuk desain sistem. Orang yang mendesain ingin sistem yang mudah didesain, diimplementasikan, fleksibel, dapat dipercaya, bebas eror, efisien. Sampai saat ini belum ada solusi yang pas untuk menentukan kebutuhan dari sistem operasi. Lain lingkungan, lain pula kebutuhannya.

Mekanisme dan Kebijakan

Mekanisme menentukan bagaimana melakukan sesuatu. Kebijakan menentukan apa yang akan dilakukan. Pemisahan antara mekanisme dan kebijakan sangatlah penting untuk fleksibilitas. Perubahan kebijakan akan membutuhkan definisi ulang pada beberapa parameter sistem, bahkan bisa mengubah mekanisme yang telah ada. Sistem operasi Microkernel-based menggunakan pemisahan mekanisme dan kebijakan secara ekstrim dengan mengimplementasikan perangkat dari primitive building blocks. Semua aplikasi mempunyai antarmuka yang sama karena antarmuka dibangun dalam kernel. Kebijakan penting untuk semua alokasi sumber daya dan penjadwalan problem. Perlu atau tidaknya sistem mengalokasikan sumber daya, kebijakan yang menentukan. Tapi bagaimana dan apa, mekanismelah yang menentukan.

Rancangan Sistem

Desain sistem memiliki masalah dalam menentukan tujuan dan spesifikasi sistem. Pada level paling tinggi, desain sistem akan dipengaruhi oleh pilihan perangkat keras dan jenis sistem. Kebutuhannya akan lebih sulit untuk dispesifikasikan. Kebutuhan terdiri dari target user dan target sistem. User menginginkan sistem yang nyaman digunakan, mudah dipelajari, dapat dipercaya, aman, dan cepat. Namun itu semua tidaklah signifikan untuk desain sistem. Orang yang mendesain ingin sistem yang mudah didesain, diimplementasikan, fleksibel, dapat dipercaya, bebas eror, efisien. Sampai saat ini belum ada solusi yang pas untuk menentukan kebutuhan dari sistem operasi. Lain lingkungan, lain pula kebutuhannya.

Mekanisme dan Kebijakan

Mekanisme menentukan bagaimana melakukan sesuatu. Kebijakan menentukan apa yang akan dilakukan.Pemisahan antara mekanisme dan kebijakan sangatlah penting untuk fleksibilitas.

Perubahan kebijakan akan membutuhkan definisi ulang pada beberapa parameter sistem, bahkan bisa mengubah mekanisme yang telah ada. Sistem operasi *Microkernel-based* menggunakan pemisahan mekanisme dan kebijakan secara ekstrim dengan mengimplementasikan perangkat dari *primitive building blocks*. Semua aplikasi mempunyai antarmuka yang sama karena antarmuka dibangun dalam kernel.

Kebijakan penting untuk semua alokasi sumber daya dan penjadwalan problem. Perlu atau tidaknya sistem mengalokasikan sumber daya, kebijakan yang menentukan. Tapi bagaimana dan apa, mekanismelah yang menentukan.

Umumnya sistem operasi ditulis dalam bahasa rakitan, tapi sekarang ini sering ditulis dalam bahasa tingkat tinggi. Keuntungannya adalah kodenya bisa ditulis lebih cepat, lebih padat, mudah dimengerti dan di-debug. Sistem operasi mudah diport (dipindahkan ke perangkat keras lain). Kerugiannya adalah mengurangi kecepatan dan membutuhkan tempat penyimpanan yang lebih banyak.

Pemberian Alamat

Sebelum masuk ke memori, suatu proses harus menunggu. Hal ini disebut antrian masukan. Proses-proses ini akan berada dalam beberapa tahapan sebelum dieksekusi. Alamat-alamat yang dibutuhkan mungkin saja direpresentasikan dalam cara yang berbeda dalam tahapan-tahapan ini. Alamat dalam kode program masih berupa simbolik. Alamat ini akan diikat oleh kompilator ke alamat memori yang dapat diakses. Kemudian *linkage editor* dan *loader*, akan mengikat alamat fisiknya. Setiap pengikatan akan memetakan suatu ruang alamat ke lainnya. Penjilidan alamat dapat terjadi pada 3 saat, yaitu:

1. **Waktu Kompilasi.** Jika diketahui pada waktu kompilasi, dimana proses ditempatkan di memori. Untuk kemudian kode absolutnya dapat dibuat. Jika kemudian alamat awalnya berubah, maka harus dikompilasi ulang.
2. **Waktu Pemanggilan.** Jika tidak diketahui dimana poses ditempatkan di memori, maka kompilator harus membuat kode yang dapat dialokasikan. Dalam kasus pengikatan akan ditunda sampai waktu pemanggilan. Jika alamat awalnya berubah, kita hanya perlu menempatkan ulang kode, untuk menyesuaikan dengan perubahan.
3. **Waktu Eksekusi.** Jika proses dapat dipindahkan dari suatu segmen memori ke lainnya selama dieksekusi. Pengikatan akan ditunda sampai run-time.

Umumnya sistem operasi ditulis dalam bahasa rakitan, tapi sekarang ini sering ditulis dalam bahasa tingkat tinggi. Keuntungannya adalah kodenya bisa ditulis lebih cepat, lebih padat, mudah dimengerti dan di-*debug*. Sistem operasi mudah diport (dipindahkan ke perangkat keras lain). Kerugiannya adalah mengurangi kecepatan dan membutuhkan tempat penyimpanan yang lebih banyak.

52.4. Kinerja

Kinerja sebuah sistem ditentukan oleh komponen-komponen yang membangun sistem tersebut. Kinerja yang paling diinginkan ada pada sebuah sistem adalah bebas eror, cepat dan fault-tolerance.

52.5. Pemeliharaan Sistem

Pemeliharaan sistem operasi berkaitan erat dengan pemeliharaan komputer, karena bagaimanapun sebuah komputer tidak akan berguna tanpa adanya sebuah sistem operasi di dalamnya. Sistem operasi juga bertindak sebagai manajer bagi semua komponen pada arsitektur komputer. Oleh karena itu, pemeliharaan sistem operasi juga dikaitkan dengan pemeliharaan komponen komputer/perangkat keras dan perangkat lunak lainnya.

Adapun beberapa hal yang dapat dilakukan daalm rangka memelihara sistem operasi antara lain:

1. Pastikan untuk selalu melakukan shutdown pada sistem operasi sebelum power switch dimatikan. Hal ini penting untuk melindungi cacat pada hard drive yang disebabkan oleh kontak antara head pada hard drive dengan permukaan drive disc, dan juga menghindari kehilangan data-data penting. Pengecualian adalah ketika komputer di-lock dan hard-drive tidak berjalan. Pada situasi ini komputer dapat dimatikan tanpa ada efek membahayakan pada hard drive.

2. Usahakan untuk selalu memback-up data yang sangat penting ke dalam sedikitnya dua physical drive yang terpisah. Jadi backup data bisa dilakukan ke floppy, zip disks, Cd-Rw, dll.
3. Jalankan Scandisk dan defragment minimal sekali dalam satu bulan. Hal ini berguna agar hard drive tetap dalam kondisi baik dan menghindari terjadinya crash.
4. Sisakan minimal 100 MB pada drive C: untuk digunakan oleh sistem operasi. Jika tidak, sistem operasi akan men-delete data-data pada hard-drive, atau sistem operasi menjadi sangat lambat. gunakan Add/delete untuk mendelete program yang tidak lagi digunakan.
5. Jangan membiarkan banyak program di-load saat men-start komputer. program-program tersebut menggunakan memori yang banyak sehingga membuat komputer menjadi lambat.
6. Lakukan pengecekan virus secara rutin.

52.6. *Tuning*

Adalah mungkin untuk mendesign, mengkode, dan megimplementasikan sebuah sistem operasi khusus untuk satu mesin di suatu *site*. Pada umumnya sistem operasi dibuat untuk berjalan pada beberapa kelas mesin di berbagai *site* dan berbagai konfigurasi *peripheral*. Kemudian, sistem dikonfigurasikan untuk masing-masing komputer, untuk *site* yang spesifik. Proses ini terkadang disebut sebagai ***System Generation***.

Sistem program membaca dari berkas yang diberikan atau mungkin bertanya pada operator tentang informasi yang berhubungan dengan perangkat keras tersebut, antara lain adalah sebagai berikut:
- CPU apa yang digunakan, pilihan yang diinstall?
- Berapa banyak memori yang tersedia?
- Peralatan yang tersedia?
- Pilihan Sistem operasi apa yang diinginkan atau parameter yang digunakan?

Satu kali informasi didapat, bisa digunakan dengan berbagai cara.

52.7. Trend

Trend sistem operasi pada tahun-tahun mendatang akan berkaitan dengan sistem operasi yang bersifat Open Source yang lebih dikenal sebagai Linux. Hal ini sesuai dengan kemampuan yang ada pada Linux, yang sangat diharapkan oleh para pengguna di kalangan bisnis, yaitu unjuk kerja yang tinggi, sekuriti, kestabilan yang tinggi, handal dan murah. Ditambah dengan dukungan aplikasi dan vendor kelas enterprise yang kini telah tersedia. Sehingga dapat dikatakan bahwa Linux telah siap untuk digunakan sebagai solusi bisnis yang bisa diandalkan. Para vendor inipun telah siap menyediakan dukungan teknis untuk Linux sehingga hal ini diharapkan dapat menghapus keragu-raguan dunia bisnis untuk memanfaatkan Linux sebagai platform operasi mereka.

52.8. Rangkuman

Merancang sebuah sistem sangat berbeda dengan merancang sebuah algorithma. Yang harus diperhatikan ketika merancang sebuah sistem yang baik adalah apakah sistem tersebut memenuhi tiga kebutuhan, yaitu fungsionalitas, kecepatan dan fault-tolerance. Orang yang mendesain ingin sistem yang mudah didesain, diimplementasikan, fleksibel, dapat dipercaya, bebas eror, efisien.

Trend perancangan sistem operasi di masa depan lebih kepada penggunaan sistem operasi yang berbasis open source, dalam hal ini adalah Linux. Pada umumnya trend demikian berkembang karena Linux menawarkan kemampuan-kemampuan dan performance yang lebih baik dari sistem operasi lainnya. Ditambah dengan stausnya yang Open Source.

Rujukan

[WEBIBM1997] IBM Coorporation. 1997. *General Programming Concepts: Writing and Debugging Programs – Threads Scheduling http://www.unet.univie.ac.at/ aix/ aixprggd/ genprogc/ threads_sched.htm* . Diakses 1 Juni 2006.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[WEBOSRLampson1983] Butler W Lampson. “Hints for Computer System Design – http://research.microsoft.com/ copyright/ accept.asp? path=/~lampson/ 33-Hints/ Acrobat.pdf&pub=acm ”. Diakses 10 Agustus 2006. *Operating Systems Review*. 15. 5. Oct 1983.

[WEBQuirke2004] Chris Quirke. 2004. *What is a Maintenance OS? – http://cquirke.mvps.org/ whatmos.htm* . Diakses 11 Agustus 2006.

Daftar Rujukan Utama

[CC2001] 2001. *Computing Curricula 2001*. Computer Science Volume. ACM Council. IEEE-CS Board of Governors.

[Deitel2005] Harvey M Deitel dan Paul J Deitel. 2005. *Java How To Program*. Sixth Edition. Prentice Hall.

[Hariyanto1997] Bambang Hariyanto. 1997. *Sistem Operasi* . Buku Teks Ilmu Komputer . Edisi Kedua. Informatika. Bandung.

[HenPat2002] John L Hennessy dan David A Patterson. 2002. *Computer Architecture* . A Quantitative Approach . Third Edition. Morgan Kaufman. San Francisco.

[Hyde2003] Randall Hyde. 2003. *The Art of Assembly Language*. First Edition. No Strach Press.

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[KrutzVines2001] Ronald L Krutz dan Russell D Vines. 2001. *The CISSP Prep Guide Mastering the Ten Domains of Computer Security*. John Wiley & Sons.

[Kusuma2000] Sri Kusumadewi. 2000. *Sistem Operasi* . Edisi Dua. Graha Ilmu. Yogyakarta.

[Love2005] Robert Love. 2005. *Linux Kernel Development* . Second Edition. Novell Press.

[Morgan1992] K Morgan. “The RTOS Difference”. *Byte*. August 1992. 1992.

[PeterDavie2000] Larry L Peterson dan Bruce S Davie. 2000. *Computer Networks A Systems Approach*. Second Edition. Morgan Kaufmann.

[SariYansen2005] Riri Fitri Sari dan Yansen. 2005. *Sistem Operasi Modern* . Edisi Pertama. Andi. Yogyakarta.

[Sidik2004] Betha Sidik. 2004. *Unix dan Linux*. Informatika. Bandung.

[Silberschatz2002] Abraham Silberschatz, Peter Galvin, dan Greg Gagne. 2002. *Applied Operating Systems*. Sixth Edition. John Wiley & Sons.

[Silberschatz2005] Avi Silberschatz, Peter Galvin, dan Grag Gagne. 2005. *Operating Systems Concepts*. Seventh Edition. John Wiley & Sons.

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

[Tanenbaum1997] Andrew S Tanenbaum dan Albert S Woodhull. 1997. *Operating Systems Design and Implementation*. Second Edition. Prentice-Hall.

[Tanenbaum2001] Andrew S Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

[UU2000030] RI. 2000. *Undang-Undang Nomor 30 Tahun 2000 Tentang Rahasia Dagang*.

[UU2000031] RI. 2000. *Undang-Undang Nomor 31 Tahun 2000 Tentang Desain Industri*.

[UU2000032] RI. 2000. *Undang-Undang Nomor 32 Tahun 2000 Tentang Desain Tata Letak Sirkuit Terpadu*.

[UU2001014] RI. 2001. *Undang-Undang Nomor 14 Tahun 2001 Tentang Paten*.

[UU2001015] RI. 2001. *Undang-Undang Nomor 15 Tahun 2001 Tentang Merek*.

[UU2002019] RI. 2002. *Undang-Undang Nomor 19 Tahun 2002 Tentang Hak Cipta*.

[Venners1998] Bill Venners. 1998. *Inside the Java Virtual Machine*. McGraw-Hill.

[WEBAmirSch2000] Yair Amir dan Theo Schlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBArpaciD2005] Andrea C Arpaci-Dusseau dan Remzi H Arpaci-Dusseau. 2005. *CS 537: Introduction to Operating Systems – File System: User Perspective – http://www.cs.wisc.edu/ ~remzi/ Classes/537/ Fall2005/ Lectures/ lecture18.ppt* . Diakses 8 Juli 2006.

[WEBBabicLauria2005] G Babic dan Mario Lauria. 2005. *CSE 660: Introduction to Operating Systems – Files and Directories – http://www.cse.ohio-state.edu/ ~lauria/ cse660/ Cse660.Files.04-08-2005.pdf* . Diakses 8 Juli 2006.

[WEBBraam1998] Peter J Braam. 1998. *Linux Virtual File System – http://www.coda.cs.cmu.edu/ doc/ talks/ linuxvfs/* . Diakses 25 Juli 2006.

[WEBCACMF1961] John Fotheringham. " Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store – http://www.eecs.harvard.edu/ cs261/ papers/ frother61.pdf ". Diakses 29 Juni 2006. *Communications of the ACM* . 4. 10. October 1961.

[WEBCarter2004] John Carter. 2004. *CS 5460 Operating Systems – Lecture 19: File System Operations and Optimizations – http://www.cs.utah.edu/ classes/ cs5460/ lectures/ lecture19.pdf* . Diakses 29 Mei 2006.

[WEBChung2005] Jae Chung. 2005. *CS4513 Distributed Computer Systems – File Systems – http://web.cs.wpi.edu/ ~goos/ Teach/ cs4513-d05/ slides/ fs1.ppt* . Diakses 7 Juli 2006.

[WEBCook2006] Tony Cook. 2006. *G53OPS Operating Systems – Directories – http://www.cs.nott.ac.uk/ ~acc/ g53ops/ lecture14.pdf* . Diakses 7 Juli 2006.

[WEBCornel2005] Cornel Computer Science Department. 2005. *Classic Sync Problems Monitors – http://www.cs.cornell.edu/ Courses/ cs414/ 2005fa/ docs/ cs414-fa05-06-semaphores.pdf* . Diakses 13 Juni 2006.

[WEBDrake96] Donald G Drake. April 1996. *Introduction to Java threads – A quick tutorial on how to implement threads in Java – http://www.javaworld.com/ javaworld/ jw-04-1996/ jw-04-threads.html* . Diakses 29 Mei 2006.

[WEBEgui2006] Equi4 Software. 2006. *Memory Mapped Files – http://www.equi4.com/ mkmmf.html* . Diakses 3 Juli 2006.

[WEBFasilkom2003] Fakultas Ilmu Komputer Universitas Indonesia. 2003. *Sistem Terdistribusi – http://telaga.cs.ui.ac.id/ WebKuliah/ sisdis2003/* . Diakses 29 Mei 2006.

[WEBFSF1991a] Free Software Foundation. 1991. *GNU General Public License – http://gnui.vLSM.org/ licenses/ gpl.txt* . Diakses 29 Mei 2006.

[WEBFSF2001a] Free Software Foundation. 2001. *Definisi Perangkat Lunak Bebas – http://gnui.vlsm.org/ philosophy/ free-sw.id.html* . Diakses 29 Mei 2006.

[WEBFSF2001b] Free Software Foundation. 2001. *Frequently Asked Questions about the GNU GPL – http://gnui.vlsm.org/ licenses/ gpl-faq.html* . Diakses 29 Mei 2006.

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

[WEBGolmFWK2002] Michael Golm, Meik Felser, Christian Wawersich, dan Juerge Kleinoede. 2002. *The JX Operating System – http://www.jxos.org/ publications/ jx-usenix.pdf* . Diakses 31 Mei 2006.

[WEBGooch1999] Richard Gooch. 1999. *Overview of the Virtual File System –*

http://www.atnf.csiro.au/ people/ rgooch/ linux/ docs/ vfs.txt . Diakses 29 Mei 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBHarris2003] Kenneth Harris. 2003. *Cooperation: Interprocess Communication – Concurrent Processing – http://people.cs.uchicago.edu/ ~mark/ 51081/ LabFAQ/ lab5/ IPC.html* . Diakses 2 Juni 2006.

[WEBHP1997] Hewlett-Packard Company. 1997. *HP-UX Memory Management – Overview of Demand Paging – http://docs.hp.com/ en/ 5965-4641/ ch01s10.html* . Diakses 29 Juni 2006.

[WEBHuham2005] Departemen Hukum dan Hak Asasi Manusia Republik Indonesia. 2005. *Kekayaan Intelektual – http://www.dgip.go.id/ article/ archive/ 2* . Diakses 29 Mei 2006.

[WEBIBMNY] IBM Corporation. NY. *General Programming Concepts – Writing and Debugging Programs – http://publib16.boulder.ibm.com/ pseries/ en_US/ aixprggd/ genprogc/ ls_sched_subr.htm* . Diakses 1 Juni 2006.

[WEBIBM1997] IBM Coorporation. 1997. *General Programming Concepts: Writing and Debugging Programs – Threads Scheduling http://www.unet.univie.ac.at/ aix/ aixprggd/ genprogc/ threads_sched.htm* . Diakses 1 Juni 2006.

[WEBIBM2003] IBM Corporation. 2003. *System Management Concepts: Operating System and Devices – http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690 / IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm* . Diakses 29 Mei 2006.

[WEBInfoHQ2002] InfoHQ. 2002. *Computer Maintenance Tips – http://www.infohq.com/ Computer/ computer_maintenance_tip.htm* . Diakses 11 Agustus 2006.

[WEBITCUV2006] IT& University of Virginia. 2006. *Mounting File Systems (Linux) – http://www.itc.virginia.edu/ desktop/ linux/ mount.html* . Diakses 20 Juli 2006.

[WEBJeffay2005] Kevin Jeffay. 2005. *Secondary Storage Management – http://www.cs.unc.edu/ ~jeffay/ courses/ comp142/ notes/ 15-SecondaryStorage.pdf* . Diakses 7 Juli 2006.

[WEBJonesSmith2000] David Jones dan Stephen Smith. 2000. *85349 – Operating Systems – Study Guide – http://www.infocom.cqu.edu.au/ Courses/ aut2001/ 85349/ Resources/ Study_Guide/ 85349.pdf* . Diakses 20 Juli 2006.

[WEBJones2003] Dave Jones. 2003. *The post-halloween Document v0.48 (aka, 2.6 - what to expect) – http://zenii.linux.org.uk/ ~davej/ docs/ post-halloween-2.6.txt* . Diakses 29 Mei 2006.

[WEBJupiter2004] Jupitermedia Corporation. 2004. *Virtual Memory – http://www.webopedia.com/ TERM/ v/ virtual_memory.html* . Diakses 29 Juni 2006.

[WEBKaram1999] Vijay Karamcheti. 1999. *Honors Operating Systems – Lecture 15: File Systems – http://cs.nyu.edu/ courses/ spring99/ G22.3250-001/ lectures/ lect15.pdf* . Diakses 5 Juli 2006.

[WEBKessler2005] Christhope Kessler. 2005. *File System Interface – http://www.ida.liu.se/ ~TDDB72/ slides/ 2005/ c10.pdf* . Diakses 7 Juli 2006.

[WEBKozierok2005] Charles M Kozierok. 2005. *Reference Guide – Hard Disk Drives http://www.storagereview.com/ guide/* . Diakses 9 Agustus 2006.

[WEBLee2000] Insup Lee. 2000. *CSE 380: Operating Systems – File Systems – http://www.cis.upenn.edu/ ~lee/ 00cse380/ lectures/ ln11b-fil.ppt* . Diakses 7 Juli 2006.

[WEBLindsey2003] Clark S Lindsey. 2003. *Physics Simulation with Java – Thread Scheduling and Priority – http://www.particle.kth.se/ ~fmi/ kurs/ PhysicsSimulation/ Lectures/ 10A/*

schedulePriority.html . Diakses 1 Juni 2006.

[WEBMassey2000] Massey University. May 2000. *Monitors & Critical Regions – http://www-ist.massey.ac.nz/ csnotes/ 355/ lectures/ monitors.pdf* . Diakses 29 Mei 2006.

[WEBMooreDrakos1999] Ross Moore dan Nikos Drakos. 1999. *Converse Programming Manual – Thread Scheduling Hooks – http://charm.cs.uiuc.edu/ manuals/ html/ converse/ 3_3Thread_ Scheduling_Hooks.html* . Diakses 1 Juni 2006.

[WEBOCWEmer2005] Joel Emer dan Massachusetts Institute of Technology. 2005. *OCW – Computer System Architecture – Fall 2005 – Virtual Memory Basics – http://ocw.mit.edu/ NR/ rdonlyres/ Electrical -Engineering -and -Computer -Science/ 6 -823Computer -System -ArchitectureSpring2002/ C63EC0D0 -0499 -474F -BCDA -A6868A6827C4/ 0/ lecture09.pdf* . Diakses 29 Juni 2006.

[WEBOSRLampson1983] Butler W Lampson. "Hints for Computer System Design – http://research.microsoft.com/ copyright/ accept.asp? path=/~lampson/ 33-Hints/ Acrobat.pdf&pub=acm ". Diakses 10 Agustus 2006. *Operating Systems Review*. 15. 5. Oct 1983.

[WEBQuirke2004] Chris Quirke. 2004. *What is a Maintenance OS? – http://cquirke.mvps.org/ whatmos.htm* . Diakses 11 Agustus 2006.

[WEBRamam2005] B Ramamurthy. 2005. *File Management – http://www.cse.buffalo.edu/ faculty/ bina/ cse421/ spring2005/ FileSystemMar30.ppt* . Diakses 5 Juli 2006.

[WEBRamelan1996] Rahardi Ramelan. 1996. *Hak Atas Kekayaan Intelektual Dalam Era Globalisasi http://leapidea.com/ presentation?id=6* . Diakses 29 Mei 2006.

[WEBRegehr2002] John Regehr dan University of Utah. 2002. *CS 5460 Operating Systems – Demand Halamand Virtual Memory – http://www.cs.utah.edu/ classes/ cs5460-regehr/ lecs/ demand_paging.pdf* . Diakses 29 Juni 2006.

[WEBRobbins2003] Steven Robbins. 2003. *Starving Philosophers: Experimentation with Monitor Synchronization – http://vip.cs.utsa.edu/ nsf/ pubs/ starving/ starving.pdf* . Diakses 29 Mei 2006.

[WEBRusQuYo2004] Rusty Russell, Daniel Quinlan, dan Christopher Yeoh. 2004. *Filesystem Hierarchy Standard – http://www.pathname.com/fhs/* . Diakses 27 Juli 2006.

[WEBRustling1997] David A Rusling. 1997. *The Linux Kernel – The EXT2 Inode – http://www.science.unitn.it/ ~fiorella/ guidelinux/ tlk/ node96.html* . Diakses 1 Agustus 2006.

[WEBRyan1998] Tim Ryan. 1998. *Java 1.2 Unleashed – http://utenti.lycos.it/ yanorel6/ 2/ ch52.htm* . Diakses 31 Mei 2006.

[WEBSamik2003a] Rahmat M Samik-Ibrahim. 2003. *Pengenalan Lisensi Perangkat Lunak Bebas – http://rms46.vlsm.org/ 1/ 70.pdf* . vLSM.org. Pamulang. Diakses 29 Mei 2006.

[WEBSamik2005a] Rahmat M Samik-Ibrahim. 2005. *IKI-20230 Sistem Operasi - Kumpulan Soal Ujian 2002-2005 – http://rms46.vlsm.org/ 1/ 94.pdf* . vLSM.org. Pamulang. Diakses 29 Mei 2006.

[WEBSchaklette2004] Mark Shacklette. 2004. *CSPP 51081 Unix Systems Programming: IPC – http://people.cs.uchicago.edu/ ~mark/ 51081/ LabFAQ/ lab5/ IPC.html* . Diakses 29 Mei 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

[WEBStallman1994a] Richard M Stallman. 1994. *Mengapa Perangkat Lunak Seharusnya Tanpa Pemilik – http://gnui.vlsm.org/ philosophy/ why-free.id.html* . Diakses 29 Mei 2006.

[WEBVolz2003] Richard A Volz. 2003. *Real Time Computing – Thread and Scheduling Basics – http://linserver.cs.tamu.edu/ ~ravolz/ 456/ Chapter-3.pdf* . Diakses 1 Juni 2006.

[WEBWalton1996] Sean Walton. 1996. *Linux Threads Frequently Asked Questions (FAQ) – http://linas.org/ linux/ threads-faq.html* . Diakses 29 Mei 2006.

[WEBWIPO2005] World Intellectual Property Organization. 2005. *About Intellectual Property – http://www.wipo.int/ about-ip/ en/* . Diakses 29 Mei 2006.

[WEBWirzOjaStafWe2004] Lars Wirzenius, Joanna Oja, dan StephenAlex StaffordWeeks. 2004. *The Linux System Administrator's Guide – The boot process in closer look http://www.tldp.org/ LDP/ sag/ html/ boot-process.html* . Diakses 7 Agustus 2006.

[WEBWiki2005a] From Wikipedia, the free encyclopedia. 2005. *Intellectual property – http://en.wikipedia.org/ wiki/ Intellectual_property* . Diakses 29 Mei 2006.

[WEBWiki2006a] From Wikipedia, the free encyclopedia. 2006. *Title – http://en.wikipedia.org/ wiki/ Zombie_process* . Diakses 2 Juni 2006.

[WEBWiki2006b] From Wikipedia, the free encyclopedia. 2006. *Atomicity– http://en.wikipedia.org/ wiki/ Atomicity* . Diakses 6 Juni 2006.

[WEBWiki2006c] From Wikipedia, the free encyclopedia. 2006. *Memory Management Unit – http://en.wikipedia.org/ wiki/ Memory_management_unit* . Diakses 30 Juni 2006.

[WEBWiki2006d] From Wikipedia, the free encyclopedia. 2006. *Page Fault – http://en.wikipedia.org/ wiki/ Page_fault* . Diakses 30 Juni 2006.

[WEBWiki2006e] From Wikipedia, the free encyclopedia. 2006. *Copy on Write – http://en.wikipedia.org/ wiki/ Copy_on_Write* . Diakses 03 Juli 2006.

[WEBWiki2006f] From Wikipedia, the free encyclopedia. 2006. *Page replacement algorithms – http://en.wikipedia.org/ wiki/ Page_replacement_algorithms* . Diakses 04 Juli 2006.

[WEBWiki2006g] From Wikipedia, the free encyclopedia. 2006. *File system – http://en.wikipedia.org/ wiki/ File_system* . Diakses 04 Juli 2006.

[WEBWiki2006h] From Wikipedia, the free encyclopedia. 2006. *Keydrive – http://en.wikipedia.org/ wiki/ Keydrive* . Diakses 09 Agustus 2006.

[WEBWiki2006i] From Wikipedia, the free encyclopedia. 2006. *Tape drive – http://en.wikipedia.org/ wiki/ Tape_drive* . Diakses 09 Agustus 2006.

[WEBWiki2006j] From Wikipedia, the free encyclopedia. 2006. *CD-ROM – http://en.wikipedia.org/ wiki/ CD-ROM* . Diakses 09 Agustus 2006.

[WEBWiki2006k] From Wikipedia, the free encyclopedia. 2006. *DVD – http://en.wikipedia.org/ wiki/ DVD* . Diakses 09 Agustus 2006.

[WEBWiki2006l] From Wikipedia, the free encyclopedia. 2006. *CD – http://en.wikipedia.org/ wiki/ CD* . Diakses 09 Agustus 2006.

[WEBWiki2006m] From Wikipedia, the free encyclopedia. 2006. *DVD-RW – http://en.wikipedia.org/ wiki/ DVD-RW* . Diakses 09 Agustus 2006.

[WEBWiki2006n] From Wikipedia, the free encyclopedia. 2006. *Magneto-optical drive – http://en.wikipedia.org/ wiki/ Magneto-optical_drive* . Diakses 09 Agustus 2006.

[WEBWiki2006o] From Wikipedia, the free encyclopedia. 2006. *Floppy disk – http://en.wikipedia.org/ wiki/ Floppy_disk* . Diakses 09 Agustus 2006.

Lampiran A. *GNU Free Documentation License*

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

A.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor

acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties -- for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

A.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

A.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Lampiran B. Kumpulan Soal Ujian

Berikut merupakan kumpulan soal Ujian Tengah Semester (UTS) dan Ujian Akhir Semester (UAS) antara 2003 dan 2005 untuk Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia. Waktu pengerjaan setiap soal [kecuali Bagian B.1, “Pasangan Konsep (2003-2005)”] ialah 30 menit.

B.1. Pasangan Konsep (2003-2005)

Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:

a. *OS View: "Resource Allocator" vs. "Control Program".*

b. *"Graceful Degradation" vs. "Fault Tolerant".*

c. *Dual Mode Operation: "User Mode" vs. "Monitor Mode".*

d. *Operating System Goal: "Convenient" vs. "Efficient".*

e. *"System Components" vs. "System Calls".*

f. *"Operating System Components" vs. "Operating System Services".*

g. *"Symetric Multiprocessing" vs. "Asymetric Multiprocessing".*

h. *"Distributed Systems' vs. "Clustered Systems".*

i. *"Client Server System" vs. "Peer-to-peer system".*

j. *"Microkernels" vs. "Virtual Machines".*

k. *"Random Access Memory" vs. "Magnetic Disk".*

l. *"Hard Real-time" vs "Soft Real-time".*

m. *Job: "Batch system" vs. "Time-Sharing System".*

n. *System Design: "Mechanism" vs. "Policy".*

o. *Burst Cycle: "I/O Burst" vs. "CPU Burst".*

p. *Process Bound: "I/O Bound" vs. "CPU Bound".*

q. *"Process State" vs. "Process Control Block".*

r. *"Waiting Time" vs. "Response Time".*

s. *Process Type: "Lightweight" vs. "Heavyweight". .*

t. *Multithread Model: "One to One" vs. "Many to Many".*

u. *Scheduling Process: "Short Term" vs. "Long Term".*

v. *Scheduling Algorithm: "FCFS (First Come First Serve)" vs. "SJF (Shortest Job First)".*

w. *"Preemptive Shortest Job First" vs. "Non-preemptive Shortest Job First".*

x. *"Preemptive Shortest Job First" vs. "Non-preemptive Shortest Job First".*

y. *Inter Process Communication: "Direct Communication" vs. "Indirect Communication".*

z. *Process Synchronization: "Monitor" vs. "Semaphore".*

aa. *"Deadlock Avoidance" vs. "Deadlock Detection".*

ab. *"Deadlock" vs. "Starvation".*

ac. *Address Space: "Logical" vs. "Physical".*

ad. *Dynamic Storage Allocation Strategy: "Best Fit" vs. "Worse Fit".*

ae. *Virtual Memory Allocation Strategy: "Global" vs. "Local Replacement".*

af. *File Operations: "Deleting" vs. "Truncating".*

ag. *Storage System: "Volatile" vs. "Non-volatile".*

ah. *File Allocation Methods: "Contiguous" vs. "Linked".*

ai. *Disk Management: "Boot Block" vs. "Bad Block".*

aj. *I/O Data-Transfer Mode: "Character" vs. "Block".*

ak. *I/O Access Mode: "Sequential" vs. "Random".*

al. *I/O Transfer Schedulle: "Synchronous" vs. "Asynchronous".*

am *I/O Sharing: "Dedicated" vs. "Sharable".* .

an. *I/O direction: "Read only" vs. "Write only".*

ao. *"I/O Structure" vs. "Storage Structure".*

ap. *Software License: "Free Software" vs. "Copyleft".*

B.2. GNU/Linux (2003)

a. Sebutkan perbedaan utama antara kernel linux versi 1.X dan versi 2.X !

b. Terangkan, apa yang disebut dengan "Distribusi (distro) Linux"? Berikan empat contoh distro!

B.3. Perangkat Lunak Bebas (2005)

a. Terangkan ke-empat (3+1) definisi Perangkat Lunak Bebas (PLB) menurut *Free Software Foundation* (*FSF*).

b. Terangkan perbedaan dan persamaan antara PLB dan *Open Source Software*.

c. Terangkan perbedaan dan persamaan antara PLB dan Perangkat Lunak "*Copyleft*".

d. Berikan contoh/ilustrasi Perangkat Lunak Bebas yang bukan "*Copyleft*".

e. Berikan contoh/ilustrasi Perangkat Lunak Bebas "*Copyleft*" yang bukan *GNU Public License*.

B.4. Konsep Sistem Operasi (2005)

a. Terangkan/jabarkan sekurangnya empat komponen utama dari sebuah Sistem Operasi.

b. Terangkan/jabarkan peranan/pengaruh dari keempat komponen di atas terhadap sebuah Sistem Operasi Waktu Nyata (*Real Time System*).

c. Terangkan/jabarkan peranan/pengaruh dari keempat komponen di atas terhadap sebuah Sistem Prosesor Jamak (*Multi Processors System*).

d. Terangkan/jabarkan peranan/pengaruh dari keempat komponen di atas terhadap sebuah Sistem Operasi Terdistribusi (*Distributed System*).

e. Terangkan/jabarkan peranan/pengaruh dari keempat komponen di atas terhadap sebuah Sistem Operasi Telepon Seluler (*Cellular Phone*).

B.5. Kernel Linux 2.6.X (=KL26) (2004)

a. Terangkan, apa yang dimaksud dengan Perangkat Lunak Bebas (PLB) yang berbasis lisensi GNU GPL (*General Public Licence*)!

b. KL26 diluncurkan Desember 2003. Terangkan mengapa hingga kini (Januari 2005), belum juga dibuka cabang pengembangan Kernel Linux versi 2.7.X!

c. KL26 lebih mendukung sistem berskala kecil seperti Mesin Cuci, Kamera, Ponsel, mau pun PDA. Terangkan, bagaimana kemampuan (*feature*) opsi tanpa MMU (*Memory Management Unit*) dapat mendukung sistem berskala kecil.

d. KL26 lebih mendukung sistem berskala sangat besar seperti "*Enterprise System*". Terangkan sekurangnya dua kemampuan (*feature*) agar dapat mendukung sistem berskala sangat besar.

e. KL26 lebih mendukung sistem interaktif seperti "*Work Station*". Terangkan sekurangnya satu kemampuan (*feature*) agar dapat mendukung sistem interaktif.

B.6. Rancangan Sistem (2002)

Rancang sebuah sistem yang secara rata-rata:

- sanggup melayani secara bersamaan (*concurrent*) hingga 1000 pengguna (*users*).
- hanya 1% dari pengguna yang aktif mengetik pada suatu saat, sedangkan sisanya (99%) tidak mengerjakan apa-apa (*idle*).
- kecepatan mengetik 10 karakter per detik.
- setiap ketukan (ketik) menghasilkan *response CPU burst* dengan ukuran 10000 instruksi mesin.
- setiap instruksi mesin dijalankan dalam 2 (dua) buah siklus mesin (*machine cycle*).
- utilisasi CPU 100%.

a. Gambarkan GANTT chart dari proses-proses tersebut di atas. Lengkapi gambar dengan yang dimaksud dengan *burst time* dan *response time*!

b. Berapa lama, durasi sebuah *CPU burst* tersebut?

c. Berapa lama, kasus terbaik (*best case*) *response time* dari ketikan tersebut?

d. Berapa lama, kasus terburuk (*worse case*) *response time* dari ketikan tersebut?

e. Berapa MHz. *clock-rate CPU* pada kasus butir tersebut di atas?

B.7. Tabel Proses I (2003)

Berikut merupakan sebagian dari keluaran menjalankan perintah ``**top b n 1**" pada sebuah sistem GNU/Linux yaitu "**bunga.mhs.cs.ui.ac.id**" pada tanggal 10 Juni 2003 yang lalu.

```
16:22:04 up 71 days, 23:40,  8 users,  load average: 0.06, 0.02, 0.00
58 processes: 57 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  15.1% user,   2.4% system,   0.0% nice,  82.5% idle
Mem:    127236K total,   122624K used,     4612K free,     2700K buffers
Swap:   263160K total,     5648K used,   257512K free,    53792K cached

  PID USER   PRI NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
    1 root     0  0   112   72    56 S     0.0  0.0   0:11 init
    2 root     0  0     0    0     0 SW    0.0  0.0   0:03 kflushd
    4 root     0  0     0    0     0 SW    0.0  0.0 156:14 kswapd
    ...
14953 root     0  0   596  308   236 S     0.0  0.2  19:12 sshd
31563 daemon   0  0   272  256   220 S     0.0  0.2   0:02 portmap
 1133 user1   18  0  2176 2176  1752 R     8.1  1.7   0:00 top
 1112 user1    0  0  2540 2492  2144 S     0.0  1.9   0:00 sshd
 1113 user1    7  0  2480 2480  2028 S     0.0  1.9   0:00 bash
30740 user2    0  0  2500 2440  2048 S     0.0  1.9   0:00 sshd
30741 user2    0  0  2456 2456  2024 S     0.0  1.9   0:00 bash
30953 user3    0  0  2500 2440  2072 S     0.0  1.9   0:00 sshd
30954 user3    0  0  2492 2492  2032 S     0.0  1.9   0:00 bash
 1109 user3    0  0  3840 3840  3132 S     0.0  3.0   0:01 pine
...
 1103 user8    0  0  2684 2684  1944 S     0.0  2.1   0:00 tin
```

a. Jam berapakah program tersebut di atas dijalankan?

b. Berapa waktu sebelumnya (dari tanggal 10 Juni tersebut), server "bunga.mhs.cs.ui.ac.id" terakhir kali *(re)boot*?

c. Apakah yang dimaksud dengan "*load average*"?

d. Sebutkan nama dari sebuah proses di atas yang statusnya "*running*"!

e. Sebutkan nama dari sebuah proses di atas yang statusnya "*waiting*"!

B.8. Tabel Proses II (2003)

Berikut merupakan sebagian dari keluaran hasil eksekusi perintah ``**top b n 1**" pada sebuah sistem GNU/Linux yaitu "**bunga.mhs.cs.ui.ac.id**" beberapa saat yang lalu.

a. Berapakah nomer *Process Identification* dari program "*top*" tersebut?

b. Siapakah yang mengeksekusi program "*top*" tersebut?

c. Sekitar jam berapakah, program tersebut dieksekusi?

d. Sudah berapa lama sistem GNU/Linux tersebut hidup/menyala?

e. Berapa pengguna yang sedang berada pada sistem tersebut?

f. Apakah yang dimaksud dengan "*load average*"?

g. Apakah yang dimaksud dengan proses "*zombie*" ?

```
15:34:14 up 28 days, 14:40, 53 users,  load average: 0.28, 0.31, 0.26
265 processes: 264 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:   5.9% user,   1.8% system,   0.1% nice,  92.2% idle
Mem:    126624K total,   113548K used,    13076K free,      680K buffers
Swap:   263160K total,    58136K used,   205024K free,    41220K cached

PID   USER  PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
    1 root    8   0   460  420   408 S     0.0  0.3   0:56 init
    2 root    9   0     0    0     0 SW    0.0  0.0   0:02 keventd
    3 root   19  19     0    0     0 SWN   0.0  0.0   0:02 ksoftirqd_CPU0
.....
17353 user1   9   0  2500 2004  2004 S     0.0  1.5   0:00 sshd
17354 user1   9   0  1716 1392  1392 S     0.0  1.0   0:00 bash
17355 user1   9   0  2840 2416  2332 S     0.0  1.9   0:00 pine
12851 user2   9   0  2500 2004  2004 S     0.0  1.5   0:00 sshd
12852 user2   9   0  1776 1436  1436 S     0.0  1.1   0:00 bash
13184 user2   9   0  1792 1076  1076 S     0.0  0.8   0:00 vi
13185 user2   9   0   392  316   316 S     0.0  0.2   0:00 grep
22272 user3   9   0  2604 2592  2292 S     0.0  2.0   0:00 sshd
22273 user3   9   0  1724 1724  1396 S     0.0  1.3   0:00 bash
22283 user3  14   0   980  980   660 R    20.4  0.7   0:00 top
19855 user4   9   0  2476 2048  1996 S     0.0  1.6   0:00 sshd
19856 user4   9   0  1700 1392  1392 S     0.0  1.0   0:00 bash
19858 user4   9   0  2780 2488  2352 S     0.0  1.9   0:00 pine
.....
```

B.9. Tabel Proses III (2004)

Berikut merupakan sebagian dari keluaran hasil eksekusi perintah ``**top b n 1**" pada sebuah sistem GNU/Linux yaitu "**rmsbase.vlsm.org**" beberapa saat yang lalu.

a. Berapakah nomor *Process Identification* dari program "*top*" tersebut?

b. Sekitar jam berapakah, program tersebut dieksekusi?

c. Apakah yang dimaksud dengan proses "*nice*" ?

d. Dalam sistem Linux, "*process*" dan "*thread*" berbagi "*process table*" yang sama. Identifikasi/tunjukkan (nomor *Process Identification*) dari salah satu *thread*. Terangkan alasannya!

e. Terangkan, mengapa sistem yang 46.6% idle dapat memiliki "*load average*" yang tinggi!

```
top - 17:31:56 up 10:14 min,  1 user,  load average: 8.64, 5.37, 2.57
Tasks:     95 total,     2 running, 93 sleeping,  0 stopped,   0 zombie
Cpu(s):  14.1% user,  35.7% system,   3.6% nice,  46.6% idle
Mem:    256712k total,   252540k used,     4172k free,    13772k buffers
Swap:   257032k total,     7024k used,   250008k free,   133132k cached

PID USER  PR NI  VIRT  RES  SHR S %CPU %MEM  TIME+  COMMAND
809 root  19 19  6780 6776 6400 S 42.2  2.6 1:02.47 rsync
709 root  20 19  6952 6952  660 R 29.3  2.7 1:46.72 rsync
710 root  19 19  6492 6484 6392 S  0.0  2.5 0:02.12 rsync
818 rms46 13  0   880  880  668 R  7.3  0.3 0:00.10 top
...
660 rms46  9  0  1220 1220  996 S  0.0  0.5 0:00.00 bash
661 rms46  9  0  1220 1220  996 S  0.0  0.5 0:00.01 bash
...
712 rms46  9  0  9256 9256 6068 S  0.0  3.6 0:06.82 evolution
781 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:02.59 evolution-mail
803 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:00.41 evolution-mail
804 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:00.00 evolution-mail
805 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:07.76 evolution-mail
806 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:00.02 evolution-mail
766 rms46  9  0  5624 5624 4572 S  0.0  2.2 0:01.01 evolution-calen
771 rms46  9  0  4848 4848 3932 S  0.0  1.9 0:00.24 evolution-alarm
788 rms46  9  0  5544 5544 4516 S  0.0  2.2 0:00.55 evolution-addre
792 rms46  9  0  4608 4608 3740 S  0.0  1.8 0:01.08 evolution-execu
...
713 rms46  9  0 23580  23m  13m S  0.0  9.2 0:04.33 firefox-bin
763 rms46  9  0 23580  23m  13m S  0.0  9.2 0:00.57 firefox-bin
764 rms46  9  0 23580  23m  13m S  0.0  9.2 0:00.00 firefox-bin
796 rms46  9  0 23580  23m  13m S  0.0  9.2 0:00.18 firefox-bin
```

B.10. Status Proses I (2003)

a. Gambarkan sebuah model bagan status proses (*process state diagram*) dengan minimum lima (5) status.

b. Sebutkan serta terangkan semua nama status proses (*process states*) tersebut.

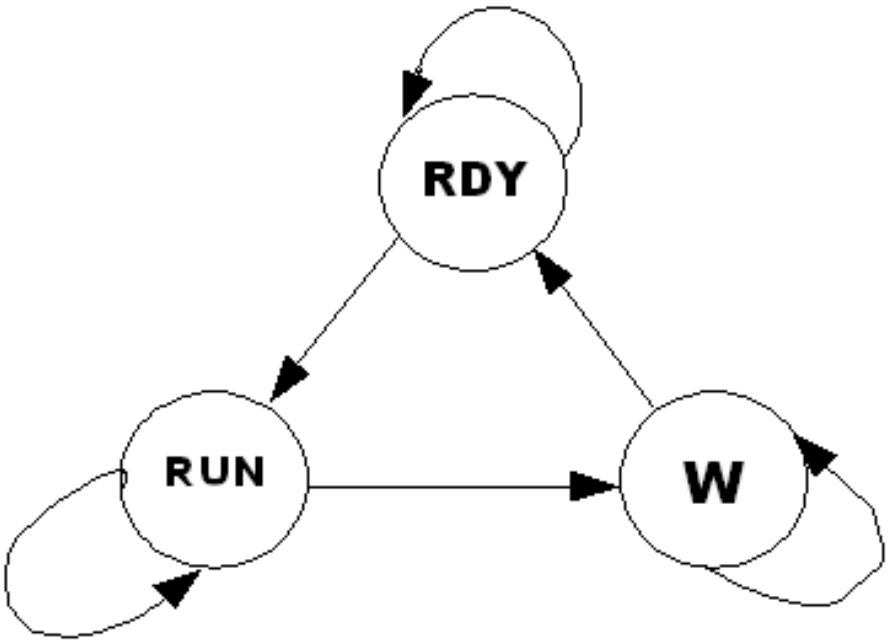
c. Sebutkan serta terangkan semua nama kejadian (*event*) yang menyebabkan perubahan status proses.

d. Terangkan perbedaan antara proses "*I/O Bound*" dengan proses "*CPU Bound*" berdasarkan bagan status proses tersebut.

B.11. Status Proses II (2005)

Diketahui empat proses (P_1, P_2, P_3, P_4) yang pada t_0 berada pada status "*RDY*" (*READY*). Pada satu saat, hanya satu proses yang boleh memiliki status "*RUN*". Status "*W*" (*Wait*) dan "*RDY*" dapat dimiliki beberapa proses setiap saat. Peralihan status proses dari "*RDY*" ke "*RUN*" diatur sebagai berikut:

- Prioritas diberikan kepada proses yang paling lama berada di "RDY" (bukan kumulatif).
- Proses yang tiba di "RDY" dapat langsung transit ke "RUN".
- Utamakan ID yang lebih kecil, jika proses-proses memiliki prioritas yang sama.

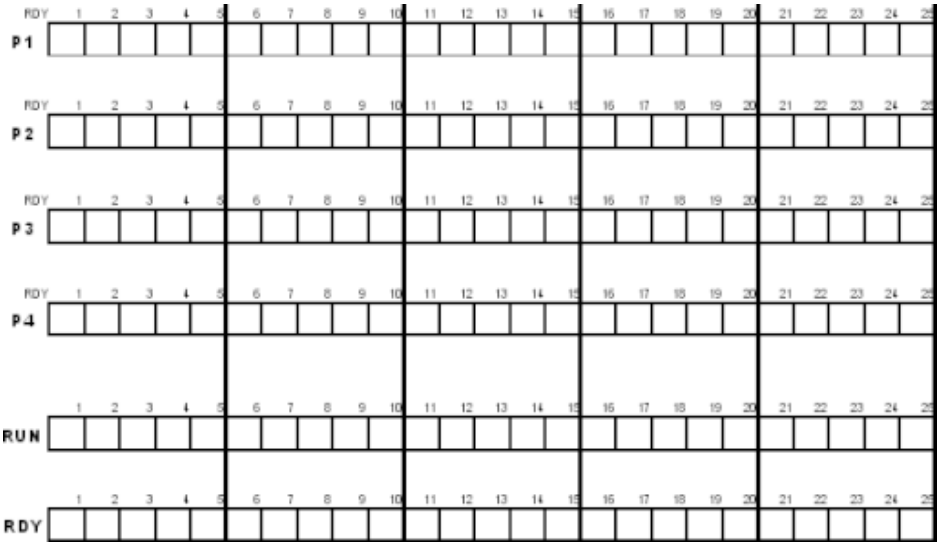


Pola *RUN*/*Wait* dari P_1 bergantian sebagai berikut: (3, 9, 3, 9, 3, 9, ...). Sedangkan pola *RUN*/*Wait*, berturut-turut: P_2 (2, 6, 2, 6, 2, 6, ...), P_3 (1, 6, 1, 6, 1, 6, ...), P_4 (1, 8, 1, 8, 1, 8, ...).

a. Gambarkan *Gantt Chart* selama 25 satuan waktu selanjut, untuk setiap Proses, serta status *RUN*/*CPU* dan *RDY*.

b. Berapa % utilitasi dari *CPU*?

c. Berapakah rata-rata *load RDY*?



B.12. *Fork* Proses I (2005)

Silakan menelusuri program C berikut ini. Diasumsikan bahwa PID dari program tersebut (baris 17) ialah 5000, serta tidak ada proses lain yang terbentuk kecuali dari `fork()` program ini.

a. Tuliskan keluaran dari program tersebut.

b. Ubahlah MAXLEVEL (baris 04) menjadi "5"; lalu kompail ulang dan jalankan kembali! Tuliskan bagian keluaran dari modifikasi program tersebut.

c. Jelaskan asumsi pemilihan PID pada butir "b" di atas!

```
01 #include <sys/types.h>
02 #include <stdio.h>
03 #include <unistd.h>
04 #define  MAXLEVEL 4
06 char*  turunan[]=
07        {"", "pertama","kedua","ketiga","keempat","kelima"};

08 main()
09 {
10    int   idx     = 1;
11    int   putaran = 0;
12    int   deret0  = 0;
13    int   deret1  = 1;
14    int   tmp;
15    pid_t pid;
16
17    printf("PID INDUK %d\n", (int) getpid());
18    printf("START deret Fibonacci... ");
      printf(" %d... %d...\n", deret0, deret1);
20    while (putaran < MAXLEVEL)
21    {
22       tmp=deret0+deret1;
23       deret0=deret1;
24       deret1=tmp;
25
26       pid = fork();             /* FORK     */
27
28       if (pid > 0)              /* Induk?   */
29       {
30          wait(NULL);
31          printf("INDUK %s selesai menunggu ", turunan[idx]);
32          printf("PID %d...\n", (int) pid);
33          putaran++;
34       } else if (pid==0) {      /* Turunan? */
35          printf("Deret Fibonacci selanjutnya...");
            printf(" %d...\n", deret1);
36          idx++;
37          exit (0);
38       } else {                  /* Error?   */
39          printf("Error...\n");
40          exit (1);
41       }
42    };
43    exit (0);
44 }
```

B.13. *Fork* Proses II (2005)

Silakan menelusuri program "*multifork*" berbahasa C berikut ini. Diasumsikan bahwa *PID* dari program tersebut (baris 14) ialah 5000, serta tidak ada proses lain yang terbentuk kecuali dari `fork()` program ini. Tuliskan keluaran dari program tersebut!

```
001 /*********************************************************/
002 /* multifork (c) 2005 Rahmat M. Samik-Ibrahim, GPL-like */
003 /*********************************************************/
004
005 #include <sys/types.h>
006 #include <stdio.h>
007 #include <unistd.h>
008
009 /********************************************** main ***/
010 main()
011 {
012    pid_t pid1, pid2, pid3;
013
014    printf("PID_INDUK   ******* %5.5d  ******* ******\n",
015                                        (int) getpid());
016
017    pid1 = fork();
018    wait(NULL);
019    pid2 = fork();
020    wait(NULL);
021    pid3 = fork();
022    wait(NULL);
023    printf("PID1(%5.5d) -- PID2(%5.5d) -- PID3(%5.5d)\n",
024                       (int) pid1, (int) pid2, (int) pid3);
025 }
027 /*********************************************************/
```

B.14. Penjadualan Proses I (2001)

Diketahui lima (5) PROSES dengan nama berturut-turut:

- $P_1(0,9)$
- $P_2(2,7)$
- $P_3(4,1)$
- $P_4(6,3)$
- $P_5(8,2)$

Angka dalam kurung menunjukkan: ("*arrival time*", "*burst time*"). Setiap peralihan proses, selalu akan diperlukan waktu-alih (*switch time*) sebesar satu (1) satuan waktu (*unit time*).

a. Berapakah rata-rata *turn-around time* dan *waiting time* dari kelima proses tersebut, jika diimplementasikan dengan algoritma penjadualan FCFS (*First Come, First Served*)?

b. Bandingkan *turnaround time* dan *waiting time* tersebut, dengan sebuah algoritma penjadualan dengan ketentuan sebagai berikut:

- *Pre-emptive*: pergantian proses dapat dilakukan kapan saja, jika ada proses lain yang memenuhi syarat. Namun durasi setiap proses dijamin minimum dua (2) satuan waktu, sebelum boleh diganti.
- Waktu alih (*switch-time*) sama dengan di atas, yaitu sebesar satu (1) satuan waktu (*unit time*).
- Jika proses telah menunggu >= 15 satuan waktu:
  - dahulukan proses yang telah menunggu paling lama
  - lainnya: dahulukan proses yang menunggu paling sebentar.
- Jika kriteria yang terjadi seri: dahulukan proses dengan nomor urut yang lebih kecil (umpama: P_1 akan didahulukan dari P_2).

B.15. Penjadualan Proses II (2002)

Lima proses tiba secara bersamaan pada saat "t_0" (awal) dengan urutan P_1, P_2, P_3, P_4, dan P_5. Bandingkan (rata-rata) *turn-around time* dan *waiting time* dari ke lima proses tersebut di atas; jika mengimplementasikan algoritma penjadualan seperti FCFS (*First Come First Served*), SJF (*Shortest Job First*), dan RR (*Round Robin*) dengan kuantum 2 (dua) satuan waktu. Waktu *context switch* diabaikan.

a. Burst time kelima proses tersebut berturut-turut (10, 8, 6, 4, 2) satuan waktu.

b. Burst time kelima proses tersebut berturut-turut (2, 4, 6, 8, 10) satuan waktu.

B.16. Penjadualan Proses III (2004)

Diketahui tiga (3) proses *preemptive* dengan nama berturut-turut $P_1(0)$, $P_2(2)$, dan $P_3(4)$. Angka dalam kurung menunjukkan waktu tiba ("*arrival time*"). Ketiga proses tersebut memiliki *burst time* yang sama yaitu 4 satuan waktu (*unit time*). Setiap memulai/peralihan proses, selalu diperlukan waktu-alih (*switch time*) sebesar satu (1) satuan waktu.

Berapakah rata-rata *turn-around time* dan *waiting time* dari ketiga proses tersebut, jika diimplementasikan dengan algoritma penjadualan:

- *Shortest Waiting First*: mendahulukan proses dengan waiting time terendah.
- *Longest Waiting First*: mendahulukan proses dengan waiting time tertinggi.

Jika kriteria penjadualan seri, dahulukan proses dengan nomor urut yang lebih kecil (umpama: P_1 akan didahulukan dari P_2). Jangan lupa membuat *Gantt Chart*-nya!

B.17. *Deadlock* I (2003)

Gambarkan graf pada urutan T_0, T_1,... dan seterusnya, hingga semua permintaan sumber-daya terpenuhi dan dikembalikan. Sebutkan, jika terjadi kondisi "*unsafe*"!

Diketahui:

- set P yang terdiri dari dua (2) proses; P = { P_1, P_2 }.
- set R yang terdiri dari dua (2) sumber-daya (*resources*); dengan berturut-turut lima (5) dan dua

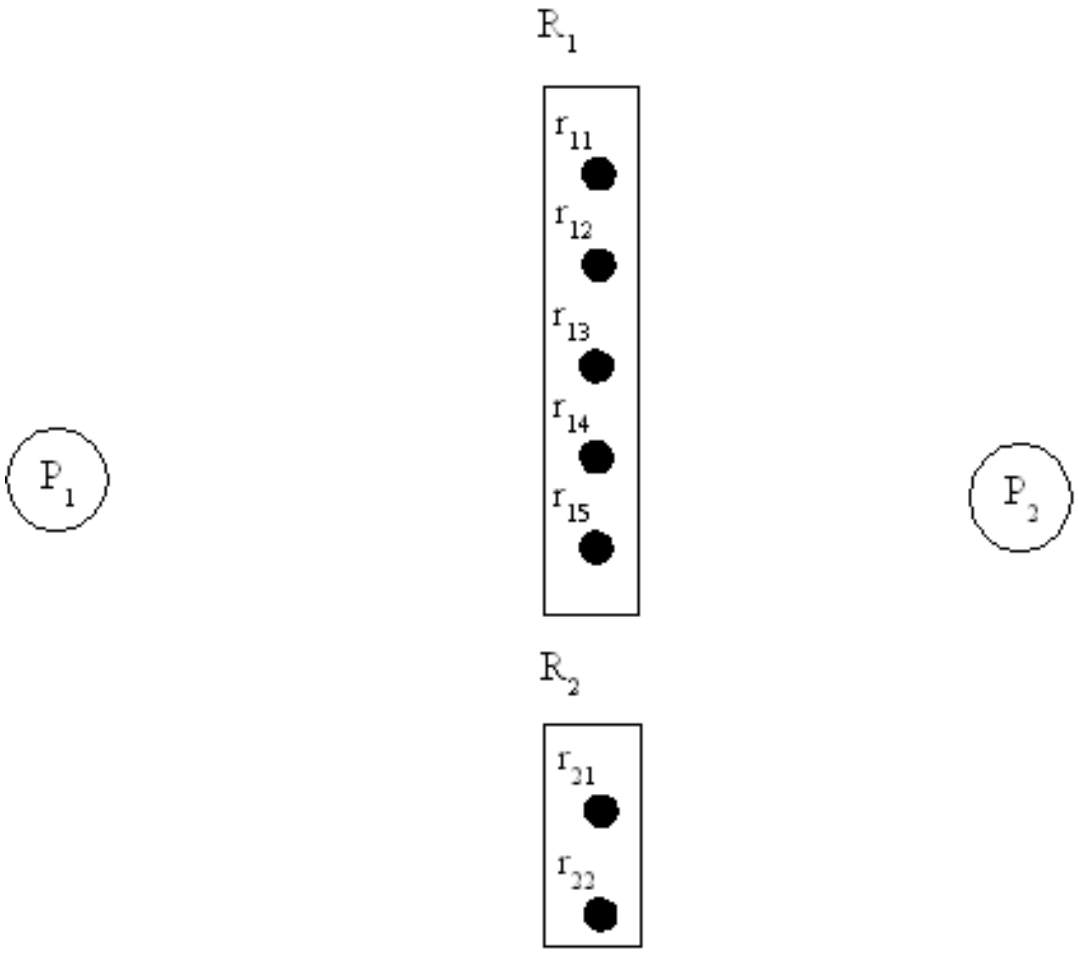
(2) *instances*; R = { R_1, R_2 } = { {r_{11}, r_{12}, r_{13}, r_{14}, r_{15} }, {r_{21}, r_{22} } }.

- Plafon (jatah maksimum) sumber-daya untuk masing-masing proses ialah:

	r_1	**r_2**
p_1	5	1
p_2	3	1

- Pencegahan *deadlock* dilakukan dengan *Banker's Algorithm*.
- Alokasi sumber-daya yang memenuhi kriteria *Banker's Algorithm* di atas, akan diprioritaskan pada proses dengan indeks yang lebih kecil.
- Setelah mendapatkan semua sumber-daya yang diminta, proses akan mengembalikan SELURUH sumber-daya tersebut.
- Pada saat T_0, "Teralokasi" serta "Permintaan" sumber-daya proses ditentukan sebagai berikut:

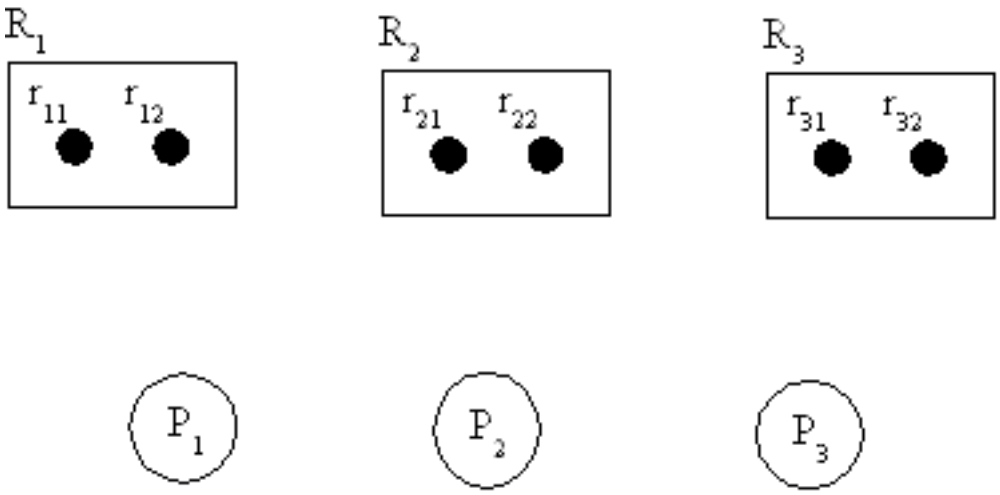
	TERALOKASI		**PERMINTAAN**	
	R_1	**R_2**	**R_1**	**R_2**
p_1	2	0	2	1
p_2	2	0	1	1



B.18. *Deadlock* II (2003)

Diketahui:

- set P yang terdiri dari tiga (3) proses; P = { P_1, P_2, P_3 }.
- set R yang terdiri dari tiga (3) sumber-daya (*resources*); masing-masing terdiri dari dua (2) instan (*instances*); R = { R_1, R_2, R_3 } = { {r_{11}, r_{12} }, {r_{21}, r_{22} }, {r_{31}, r_{32} } }.
- Prioritas alokasi sumber-daya akan diberikan pada proses dengan indeks yang lebih kecil.
- Jika tersedia: permintaan alokasi sumber-daya pada T_N akan dipenuhi pada urutan berikutnya T_{N+1}.
- Proses yang telah dipenuhi semua permintaan sumber-daya pada T_M ; akan melepaskan semua sumber-daya tersebut pada urutan berikutnya T_{M+1}.
- Pencegahan *deadlock* dilakukan dengan menghindari *circular wait*.
- Pada saat T_0, set E_0 = { } (atau kosong), sehingga gambar graf-nya sebagai berikut:



Jika set E pada saat T_1 menjadi: E_1 = { P_1 –> R_1, P_1 –> R_2, P_2 –> R_1, P_2 –> R_2, P_3 –> R_1, P_3 –> R_2, P_3 –> R_3 }, gambarkan graf pada urutan T_1, T_2,... serta (E_2, E_3, ...) berikutnya hingga semua permintaan sumber-daya terpenuhi dan dikembalikan.

B.19. *Deadlock* III (2005)

a. Terangkan/jabarkan secara singkat, keempat kondisi yang harus dipenuhi agar terjadi *Deadlock*! Gunakan graf untuk menggambarkan keempat kondisi tersebut!

b. Terangkan/jabarkan secara singkat, apakah akan selalu terjadi *Deadlock* jika keempat kondisi tersebut dipenuhi?!

B.20. Problem *Reader/Writer* I (2001)

Perhatikan berkas "`ReaderWriterServer.java`" berikut ini (*source-code* terlampir):

a. Ada berapa *object class* "`Reader`" yang terbentuk? Sebutkan nama-namanya!

b. Ada berapa *object class* "`Writer`" yang terbentuk? Sebutkan nama-namanya!

c. Modifikasi kode program tersebut (cukup baris terkait), sehingga akan terdapat 6 (enam) "`Reader`" dan 4 (empat) "`Writer`".

d. Modifikasi kode program tersebut, dengan menambahkan sebuah (satu!) *object thread* baru yaitu "`janitor`". Sang "`janitor`" berfungsi untuk membersihkan (*cleaning*). Setelah membersihkan, "`janitor`" akan tidur (*sleeping*). Pada saat bangun, "`janitor`" kembali akan membersihkan. Dan seterusnya... Pada saat "`janitor`" akan membersihkan, tidak boleh ada "`reader`" atau "`writer`" yang aktif. Jika ada, "`janitor`" harus menunggu. Demikian pula, "`reader`" atau "`writer`" harus menunggu "`janitor`" hingga selesai membersihkan.

```
001 // Gabungan ReaderWriterServer.java Reader.java Writer.java
002 //          Semaphore.java Database.java
003 // (c) 2000 Gagne, Galvin, Silberschatz
004
005 public class ReaderWriterServer {
006    public static void main(String args[]) {
007       Database server = new Database();
008       Reader[] readerArray = new Reader[NUM_OF_READERS];
009       Writer[] writerArray = new Writer[NUM_OF_WRITERS];
010       for (int i = 0; i < NUM_OF_READERS; i++) {
011          readerArray[i] = new Reader(i, server);
012          readerArray[i].start();
013       }
014       for (int i = 0; i < NUM_OF_WRITERS; i++) {
015          writerArray[i] = new Writer(i, server);
016          writerArray[i].start();
017       }
018    }
019    private static final int NUM_OF_READERS = 3;
020    private static final int NUM_OF_WRITERS = 2;
021 }
022
023 class Reader extends Thread {
024    public Reader(int r, Database db) {
025       readerNum = r;
026       server = db;
027    }
028    public void run() {
029      int c;
030      while (true) {
031        Database.napping();
032        System.out.println("reader " + readerNum
                                      + " wants to read.");
033        c = server.startRead();
034        System.out.println("reader " + readerNum +
035             " is reading. Reader Count = " + c);
036        Database.napping();
037        System.out.print("reader " + readerNum +
                " is done reading. ");
038        c = server.endRead();
039       }
040    }
041    private Database  server;
042    private int       readerNum;
043 }
```

```
045 class Writer extends Thread {
046    public Writer(int w, Database db) {
047       writerNum = w;
048       server = db;
049    }
050    public void run() {
051      while (true) {
052        System.out.println("writer " + writerNum + " is sleeping.");
053        Database.napping();
054        System.out.println("writer " + writerNum + " wants to write.");
055        server.startWrite();
056        System.out.println("writer " + writerNum + " is writing.");
057        Database.napping();
058        System.out.println("writer " + writerNum + " is done writing.");
059        server.endWrite();
060       }
061    }
062    private Database  server;
063    private int       writerNum;
064 }
065
066 final class Semaphore {
067    public Semaphore() {
068       value = 0;
069    }
070    public Semaphore(int v) {
071       value = v;
072    }
073    public synchronized void P() {
074       while (value <= 0) {
075          try { wait(); }
076          catch (InterruptedException e) { }
077       }
078       value--;
079    }
080    public synchronized void V() {
081       ++value;
082       notify();
083    }
084    private int value;
085 }
086
087 class Database {
088    public Database() {
089       readerCount = 0;
090       mutex = new Semaphore(1);
091       db = new Semaphore(1);
092    }
093    public static void napping() {
094      int sleepTime = (int) (NAP_TIME * Math.random() );
095      try { Thread.sleep(sleepTime*1000); }
096      catch(InterruptedException e) {}
097    }
098    public int startRead() {
099       mutex.P();
100       ++readerCount;
101       if (readerCount == 1) {
102          db.P();
103       }
104       mutex.V();
105       return readerCount;
106    }
```

```
107    public int endRead() {
108       mutex.P();
109       --readerCount;
110       if (readerCount == 0) {
111          db.V();;
112       }
113       mutex.V();
114       System.out.println("Reader count = " + readerCount);
115       return readerCount;
116    }
117    public void startWrite() {
118       db.P();
119    }
120    public void endWrite() {
121       db.V();
122    }
123    private int readerCount;
124    Semaphore mutex;
125    Semaphore db;
126    private static final int NAP_TIME = 15;
127 }
128
129 // The Class java.lang.Thread
130 // When a thread is created, it is not yet active; it begins
131 // to run when method start is called. Invoking the start
132 // method causes this thread to begin execution; by calling
    // the run method.
133 // public class Thread implements Runnable {
134 //     ...
135 //     public void run();
136 //     public void start()
137 //        throws IllegalThreadStateException;
138 //     ...
139 // }
```

B.21. Problem *Reader/Writer* II (2002)

Perhatikan berkas "`ReaderWriterServer.java`" pada soal yang lalu, yang merupakan gabungan berbagai berkas seperti "`ReaderWriterServer.java`", "`Reader.java`", "`Writer.java`", "`Semaphore.java`", "`Database.java`", oleh Gagne, Galvin, dan Silberschatz. Terangkan berdasarkan berkas tersebut:

a. akan terbentuk berapa *thread*, jika menjalankan program *class* "`ReaderWriterServer`" ini? Apa yang membedakan antara sebuah *thread*, dengan *thread*, lainnya?

b. mengapa: jika ada "`Reader`" yang sedang membaca, tidak ada "`Writer`" yang dapat menulis; dan mengapa: jika ada "`Writer`" yang sedang menulis, tidak ada "`Reader`" yang dapat membaca?

c. mengapa: jika ada "`Reader`" yang sedang membaca, boleh ada "`Reader`" lainnya yang turut membaca?

d. modifikasi kode program tersebut (cukup mengubah baris terkait), sehingga akan terdapat 5 (lima) "`Reader`" dan 4 (empat) "`Writer`"!

Modifikasi kode program tersebut (cukup mengubah `method` terkait), sehingga pada saat **RAJA** (Reader 0) ingin membaca, tidak boleh ada **RAKYAT** (`Reader` lainnya) yang sedang/akan membaca. **JANGAN MEMPERSULIT DIRI SENDIRI**: jika **RAJA** sedang membaca, **RAKYAT** boleh turut membaca.

B.22. Problem *Reader/Writer* III (2004)

Perhatikan berkas program java pada halaman berikut ini.

1. Berapa jumlah *thread class* `Reader` yang akan terbentuk?
2. Berapa jumlah *thread class* `Writer` yang akan terbentuk?
3. Perkirakan bagaimana bentuk keluaran (*output*) dari program tersebut!
4. Modifikasi program agar nap rata-rata dari *class* `Reader` lebih besar daripada *class* `Writer`.

```
001 /*************************************************************
002  * Gabungan/Modif: Factory.java Database.java RWLock.java
003  * Reader.java Semaphore.java SleepUtilities.java Writer.java
004  * Operating System Concepts with Java - Sixth Edition
005  * Gagne, Galvin, Silberschatz Copyright John Wiley & Sons-2003.
006  */
007
008 public class Factory
009 {
010     public static void main(String args[])
011     {
012         System.out.println("INIT Thread...");
013         Database server   = new Database();
014         Thread   readerX  = new Thread(new Reader(server));
015         Thread   writerX  = new Thread(new Writer(server));
016         readerX.start();
017         writerX.start();
018         System.out.println("Wait...");
019     }
020 }
022 // Reader // ************************************************
023 class Reader implements Runnable
024 {
025     public Reader(Database db) { server = db; }
026
027     public void run() {
028         while (--readercounter > 0)
029          {
030             SleepUtilities.nap();
031             System.out.println("readerX: wants to read.");
032             server.acquireReadLock();
033             System.out.println("readerX: is reading.");
034             SleepUtilities.nap();
035             server.releaseReadLock();
036             System.out.println("readerX: done...");
037         }
038     }
039
040     private Database server;
041     private int      readercounter = 3;
042 }
043
```

```
044 // Writer // ************************************************
045 class Writer implements Runnable
046 {
047     public Writer(Database db) { server = db; }
049     public void run() {
050         while (writercounter-- > 0)
051         {
052            SleepUtilities.nap();
053             System.out.println("writerX: wants to write.");
054             server.acquireWriteLock();
055             System.out.println("writerX: is writing.");
056              SleepUtilities.nap();
057             server.releaseWriteLock();
058             System.out.println("writerX: done...");
059         }
060     }
062     private Database server;
063     private int      writercounter = 3;
064 }
065 // Semaphore // ********************************************
066 class Semaphore
067 {
068     public Semaphore()        { value = 0; }
069     public Semaphore(int val) { value = val; }
070     public synchronized void acquire() {
071         while (value == 0) {
072             try { wait(); }
073             catch (InterruptedException e) { }
074         }
075         value--;
076     }
077
078     public synchronized void release() {
079         ++value;
080         notifyAll();
081     }
082     private int value;
083 }
084
085 // SleepUtilities // ****************************************
086 class SleepUtilities
087 {
088     public static void nap() { nap(NAP_TIME); }
089
090     public static void nap(int duration) {
091         int sleeptime = (int) (duration * Math.random() );
092           try { Thread.sleep(sleeptime*1000); }
093         catch (InterruptedException e) {}
094     }
095     private static final int NAP_TIME = 3;
096 }
097
098 // Database // **********************************************
099 class Database implements RWLock
100 {
101     public Database()              { db = new Semaphore(1); }
102     public void acquireReadLock()  { db.acquire(); }
103     public void releaseReadLock()  { db.release(); }
104     public void acquireWriteLock() { db.acquire(); }
105     public void releaseWriteLock() { db.release(); }
106     Semaphore                      db;
107 }
108 // An interface for reader-writer locks. // *********************
109 interface RWLock
110 {
111     public abstract void acquireReadLock();
112     public abstract void releaseReadLock();
113     public abstract void acquireWriteLock();
114     public abstract void releaseWriteLock();
115 }
```

B.23. *Bounded Buffer* (2003)

Perhatikan berkas "`BoundedBufferServer.java`" pada halaman berikut.

a. Berapakah ukuran penyangga (*buffer*)?

b. Modifikasi program (sebutkan nomor barisnya) agar ukuran penyangga menjadi 6 (enam).

c. Tuliskan/perkirakan keluaran (*output*) 10 baris pertama, jika menjalankan program ini.

d. Jelaskan fungsi dari ketiga *semaphore* (*mutex*, *full*, *empty*) pada program tersebut.

e. Tambahkan (sebutkan nomor barisnya) sebuah *thread* dari *class* Supervisor yang berfungsi:

   i. pada awal dijalankan, melaporkan ukuran penyangga.

   ii. secara berkala (acak), melaporkan jumlah pesan (*message*) yang berada dalam penyangga.

f. *Semaphore* mana yang paling relevan untuk modifikasi butir "e" di atas?

```
001 // Authors: Greg Gagne, Peter Galvin, Avi Silberschatz
002 // Slightly Modified by: Rahmat M. Samik-Ibrahim
003 // Copyright 2000 by Greg Gagne, Peter Galvin, Avi Silberschatz
004 // Applied Operating Systems Concepts-John Wiley & Sons, Inc.
005 //
006 // Class "Date":
007 //     Allocates a Date object and initializes it so that
008 //     it represents the time at which it was allocated,
009 //     (E.g.): "Wed Apr 09 11:12:34 JAVT 2003"
010 // Class "Object"/ method "notify":
011 //     Wakes up a single thread that is waiting on this
    //     object's monitor.
012 // Class "Thread"/ method "start":
013 //     Begins the thread execution and calls the run method
    //     of the thread.
014 // Class "Thread"/ method "run":
015 //     The Runnable object's run method is called.
016
017 import java.util.*;
018 // main *******************************************************
019 public class BoundedBufferServer
020 {
021     public static void main(String args[])
022     {
023         BoundedBuffer server         = new BoundedBuffer();
024         Producer      producerThread = new Producer(server);
025         Consumer      consumerThread = new Consumer(server);
026         producerThread.start();
027         consumerThread.start();
028     }
029 }
030
```

```
031 // Producer ********************************************************
032 class Producer extends Thread
033 {
034     public Producer(BoundedBuffer b)
035     {
036          buffer = b;
037     }
038
039     public void run()
040     {
041         Date message;
042         while (true)
043         {
044             BoundedBuffer.napping();
045
046             message = new Date();
047             System.out.println("P: PRODUCE  " + message);
048             buffer.enter(message);
049         }
050     }
051     private  BoundedBuffer buffer;
052 }
053
054 // Consumer ********************************************************
055 class Consumer extends Thread
056 {
057    public Consumer(BoundedBuffer b)
058    {
059       buffer = b;
060    }
061    public void run()
062    {
063          Date message;
064          while (true)
065          {
066              BoundedBuffer.napping();
067              System.out.println("C: CONSUME  START");
068              message = (Date)buffer.remove();
069          }
070     }
071     private  BoundedBuffer buffer;
072 }
074 // BoundedBuffer.java ********************************************
075 class BoundedBuffer
076 {
077    public BoundedBuffer()
078    {
079         count  = 0;
080         in     = 0;
081         out    = 0;
082         buffer = new Object[BUFFER_SIZE];
083         mutex  = new Semaphore(1);
084         empty  = new Semaphore(BUFFER_SIZE);
085         full   = new Semaphore(0);
086    }
087    public static void napping()
088    {
089         int sleepTime = (int) (NAP_TIME * Math.random() );
090         try { Thread.sleep(sleepTime*1000); }
091         catch(InterruptedException e) { }
092    }
```

```
093    public void enter(Object item)
094    {
095         empty.P();
096         mutex.P();
097         ++count;
098         buffer[in] = item;
099         in = (in + 1) % BUFFER_SIZE;
100         System.out.println("P: ENTER    " + item);
101         mutex.V();
102         full.V();
103    }
104    public Object remove()
105    {
106         Object item;
107         full.P();
108         mutex.P();
109         --count;
110         item = buffer[out];
111         out = (out + 1) % BUFFER_SIZE;
112         System.out.println("C: CONSUMED " + item);
113         mutex.V();
114         empty.V();
115         return item;
116     }
117     public  static final int NAP_TIME    = 5;
118     private static final int BUFFER_SIZE = 3;
119     private Semaphore        mutex;
120     private Semaphore        empty;
121     private Semaphore        full;
122     private int              count, in, out;
123     private Object[]         buffer;
124 }
125
126 // Semaphore.java ************************************************
128 final class Semaphore
129 {
130     public Semaphore()
131     {
132         value = 0;
133     }
134     public Semaphore(int v)
135     {
136        value = v;
137     }
138     public synchronized void P()
139     {
140         while (value <= 0)
141         {
142             try { wait(); }
143             catch (InterruptedException e) { }
144         }
145         value --;
146     }
147     public synchronized void V()
148     {
149         ++value;
150         notify();
151     }
152     private int value;
153 }
```

B.24. Sinkronisasi I (2005)

a. Terangkan peranan/fungsi dari semafor-semafor pada program Java berikut ini!

b. Tuliskan keluaran dari program tersebut!

c. Modifikasi program (baris mana?), agar object proses dengan index tinggi mendapat prioritas didahulukan dibandingkan proses dengan index rendah.

d. Terangkan kelemahan dari program ini! Kondisi bagaimana yang mengakibatkan semafor tidak berperan seperti yang diinginkan!

```
 0 /*************************************************************
 1  * SuperProses (c) 2005 Rahmat M. Samik-Ibrahim, GPL-like   */
 2
 3 // ********* SuperProses *
 4 public class SuperProses {
 5    public static void main(String args[]) {
 6       Semafor[] semafor1 = new Semafor[JUMLAH_PROSES];
 7       Semafor[] semafor2 = new Semafor[JUMLAH_PROSES];
 8       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
 9          semafor1[ii] = new Semafor();
10          semafor2[ii] = new Semafor();
11       }
12
13       Thread superp=new Thread(new SuperP(semafor1,semafor2,JUMLAH_PROSES
14       superp.start();
15
16       Thread[] proses= new Thread[JUMLAH_PROSES];
17       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
18          proses[ii]=new Thread(new Proses(semafor1,semafor2,ii));
19          proses[ii].start();
20       }
21    }
22
23    private static final int JUMLAH_PROSES = 5;
24 }
25
26 // ** SuperP ********************
27 class SuperP implements Runnable {
28    SuperP(Semafor[] sem1, Semafor[] sem2, int jmlh) {
29       semafor1      = sem1;
30       semafor2      = sem2;
31       jumlah_proses = jmlh;
32    }
33
34    public void run() {
35       for (int ii = 0; ii < jumlah_proses; ii++) {
36          semafor1[ii].kunci();
37       }
38       System.out.println("SUPER PROSES siap...");
39       for (int ii = 0; ii < jumlah_proses; ii++) {
40          semafor2[ii].buka();
41          semafor1[ii].kunci();
42       }
43    }
44
45    private Semafor[] semafor1, semafor2;
46    private int       jumlah_proses;
47 }// ** Proses ********************
```

```
50 class Proses implements Runnable {
51    Proses(Semafor[] sem1, Semafor[] sem2, int num) {
52       num_proses = num;
53       semafor1   = sem1;
54       semafor2   = sem2;
55    }
57    public void run() {
58       semafor1[num_proses].buka();
59       semafor2[num_proses].kunci();
60       System.out.println("Proses " + num_proses + " siap...");
61       semafor1[num_proses].buka();
62    }
64    private Semafor[] semafor1, semafor2;
65    private int       num_proses;
66 }
68 // ** Semafor *
69 class Semafor {
70    public Semafor()        { value = 0;   }
71    public Semafor(int val) { value = val; }
72
73    public synchronized void kunci() {
74       while (value == 0) {
75          try   { wait(); }
76          catch (InterruptedException e) { }
77       }
78       value--;
79    }
80
81    public synchronized void buka() {
82       value++;
83       notify();
84    }
85
86    private int value;
87 }
```

B.25. Sinkronisasi II(2005)

Silakan menelusuri program Java "Hompimpah" pada lampiran berikut ini.

a. Berapa jumlah pemain "Hompimpah" tersebut?

b. Sebuatkan nama *object* `semafor` yang digunakan untuk melindungi *Critical Section*? Sebutkan baris berapa saja yang termasuk *Critical Section* tersebut.

c. Tuliskan salah satu kemungkinan keluaran dari program ini.

d. Terangkan fungsi/peranan dari metoda-metoda berikut ini: `syncPemainBandar()`; `syncBandar()`; `syncPemain()`; `syncBandarPemain()`.

```
001 /*********************************************************/
002 /* Hompimah (c) 2005 Rahmat M. Samik-Ibrahim, GPL-like  */
003 /* Nilai Tangan:TRUE="telapak" FALSE="punggung tangan"  */
004 /*********************************************************/

006 // ************************************* Hompimpah ***
007 public class Hompimpah {
008    public static void main(String args[]) {
009       Gambreng gamserver = new Gambreng(JUMLAH_PEMAIN);
010       Thread[] pemain    = new Thread[JUMLAH_PEMAIN];
011       for (int ii = 0; ii < JUMLAH_PEMAIN; ii++) {
012          pemain[ii]=new Thread(new Pemain(gamserver,ii));
013          pemain[ii].start();
014       }
015       gamserver.bandarGambreng();
016    }

017    // ***************************************************
018    private static final int JUMLAH_PEMAIN = 6;
019 }

021 // ************************************** Pemain ***
022 class Pemain implements Runnable {
023    Pemain(Gambreng gserver, int nomer) {
024       gamserver  = gserver;
025       no_pemain  = nomer;
026    }
027    // ********************************** Pemain.run ***
028    public void run() {
029       gamserver.pemainGambreng(no_pemain);
030    }
031    // ***************************************************
032    private Gambreng gamserver;
033    private int      no_pemain;
034 }
036 // ************************************** Gambreng ***
037 class Gambreng {
038    public Gambreng(int jumlah) {
039       bandar          = new Semafor[jumlah];
040       pemain          = new Semafor[jumlah];
041       for (int ii=0; ii<jumlah; ii++) {
042          bandar[ii]   = new Semafor();
043          pemain[ii]   = new Semafor();
044       }
045       mutex           = new Semafor(1);
046       jumlahPemain    = jumlah;
047       iterasiGambreng = 0;
048       resetGambreng();
049    }
050    // ********************** Gambreng.bandarGambreng ***
051    public void bandarGambreng() {
052       syncBandar();
053       while(! menangGambreng()) {
054          resetGambreng();
055          syncPemainBandar();
056          hitungGambreng();
057          iterasiGambreng++;
058       }
059       syncPemain();
060       System.out.println("Nomor Peserta Pemain [0] - [" +
061          (jumlahPemain-1) + "]  Pemenang Pemain Nomor[" +
062          nomorPemenang + "]  Jumlah Iterasi[" +
063          iterasiGambreng + "]");
064    }
```

```
065    // ************************ Gambreng.pemainGambreng ***
066    public void pemainGambreng(int nomor) {
067       syncBandarPemain(nomor);
068       while(! menangGambreng()) {
069          mutex.kunci();
070          // TRUE="telapak" FALSE="punggung tangan" *******
071          if ((int)(Math.random()*2)==1) {
072             truePemain=nomor;
073             trueCount++;
074          } else {
075             falsePemain=nomor;
076             falseCount++;
077          }
078          mutex.buka();
079          syncBandarPemain(nomor);
080       }
081    }
082    // ************************* Gambreng.resetGambreng ***
083    private void resetGambreng() {
084       mutex.kunci();
085       adaPemenang   = false;
086       truePemain    = 0;
087       trueCount     = 0;
088       falsePemain   = 0;
089       falseCount    = 0;
090       mutex.buka();
091    }
092    // ************************ Gambreng.menangGambreng ***
093    private boolean menangGambreng() {
094       return adaPemenang;
095    }
096    // ************************ Gambreng.hitungGambreng ***
097    private void hitungGambreng() {
098       mutex.kunci();
099       if (trueCount == 1) {
100          adaPemenang=true;
101          nomorPemenang=truePemain;
102       } else if (falseCount == 1) {
103          adaPemenang=true;
104          nomorPemenang=falsePemain; }
106       mutex.buka();
107    }
108    // ******************** Gambreng.syncPemainGambreng ***
109    private void syncPemainBandar() {
110       for (int ii=0; ii<jumlahPemain; ii++) {
111          pemain[ii].buka();
112          bandar[ii].kunci();
113       }
114    }
115    // *************************** Gambreng.syncBandar ***
116    private void syncBandar() {
117       for (int ii=0; ii<jumlahPemain; ii++)
118          bandar[ii].kunci();
119    }
120    // *************************** Gambreng.syncPemain ***
121    private void syncPemain() {
122       for (int ii=0; ii<jumlahPemain; ii++)
123          pemain[ii].buka();
124    }
125    // ********************** Gambreng.syncBandarPemain ***
126    private void syncBandarPemain(int ii) {
127       bandar[ii].buka();
128       pemain[ii].kunci();
129    }
```

```
130    // ************************************************
131    private boolean   adaPemenang;
132    private int       truePemain, trueCount, iterasiGambreng;
133    private int       falsePemain, falseCount;
134    private int       nomorPemenang, jumlahPemain;
135    private Semafor[] bandar, pemain;
136    private Semafor   mutex;
137 }
139 // ************************************** Semafor ***
140 class Semafor {
141    public Semafor()        { value = 0;   }
142    public Semafor(int val) { value = val; }
143    // ********************************* Semafor.kunci ***
144    public synchronized void kunci() {
145       while (value == 0) {
146          try   { wait(); }
147          catch (InterruptedException e) { }
148       }
149       value--;
150    }
151    // ********************************** Semafor.buka ***
152    public synchronized void buka() {
153       value++;
154       notify();
155    }
156    // ************************************************
157    private int value;
158 }
160 // ************************************************
```

B.26. IPC (2003)

Perhatikan berkas program java berikut ini:

a. Berapakah jumlah *object* dari "`Worker Class`" yang akan terbentuk?

b. Sebutkan nama-nama *object* dari "`Worker Class`" tersebut!

c. Tuliskan/perkirakan keluaran (*output*) 10 baris pertama, jika menjalankan program ini!

d. Apakah keluaran pada butir "c" di atas akan berubah, jika parameter CS_TIME diubah menjadi dua kali NON_CS_TIME? Terangkan!

e. Apakah keluaran pada butir "c" di atas akan berubah, jika selain parameter CS_TIME diubah menjadi dua kali NON_CS_TIME, dilakukan modifikasi NN menjadi 10? Terangkan!

```
001 /* Gabungan Berkas:
002  * FirstSemaphore.java, Runner,java, Semaphore.java, Worker.java.
003  * Copyright (c) 2000 oleh Gagne, Galvin, Silberschatz.
004  * Applied Operating Systems Concepts-John Wiley & Sons, Inc.
005  * Slightly modified by Rahmat M. Samik-Ibrahim.
006  *
007  * Informasi Singkat (RMS46):
008  * Threat.start()   --> memulai thread yang memanggil Threat.run().
009  * Threat.sleep(xx) --> thread akan tidur selama xx milidetik.
010  * try {...} catch(InterruptedException e){} --> terminasi program.
011  */
012
```

```
013 public class FirstSemaphore
014 {
015     public static void main(String args[]) {
016         Semaphore  sem = new Semaphore(1);
017         Worker[]  bees = new Worker[NN];
018         for (int ii = 0; ii < NN; ii++)
019             bees[ii] = new Worker(sem, ii);
020         for (int ii = 0; ii < NN; ii++)
021             bees[ii].start();
022     }
023     private final static int NN=4;
024 }
025
026 // Worker ========================================================
027 class Worker extends Thread
028 {
029     public Worker(Semaphore sss, int nnn) {
030         sem     = sss;
031         wnumber = nnn;
032          wstring = WORKER + (new Integer(nnn)).toString();
033     }
034

035     public void run() {
036         while (true) {
037             System.out.println(wstring + PESAN1);
038             sem.P();
039             System.out.println(wstring + PESAN2);
040             Runner.criticalSection();
041             System.out.println(wstring + PESAN3);
042             sem.V();
043             Runner.nonCriticalSection();
044        }
045     }
046     private Semaphore sem;
047     private String    wstring;
048     private int       wnumber;
049     private final static String PESAN1=
                              " akan  masuk   ke  Critical Section.";
050     private final static String PESAN2=
                              " berada  di  dalam Critical Section.";
051     private final static String PESAN3=
                              " telah keluar dari Critical Section.";
052     private final static String WORKER=
                                                      "PEKERJA ";
053 }
054
055 // Runner ========================================================
056 class Runner
057 {
058     public static void criticalSection() {
059         try {
060             Thread.sleep( (int) (Math.random()*CS_TIME * 1000));
061         }
062         catch (InterruptedException e) { }
063     }
064
065     public static void nonCriticalSection() {
066         try {
067             Thread.sleep( (int) (Math.random()*NON_CS_TIME*1000));
068         }
069         catch (InterruptedException e) { }
070     }
071     private final static int     CS_TIME = 2;
072     private final static int NON_CS_TIME = 2;
073 }
074
```

```
075 // Semaphore ==========================================================
076 final class Semaphore
077 {
078     public Semaphore() {
079         value = 0;
080     }
081
082     public Semaphore(int v) {
083         value = v;
084     }
085
086     public synchronized void P() {
087         while (value <= 0) {
088             try {
089                 wait();
090             }
091             catch (InterruptedException e) { }
092         }
093         value --;
094     }
095
096     public synchronized void V() {
097         ++value;
098         notify();
099     }
100
101     private int value;
102 }
103
104 // END ============================================================
```

B.27. Status Memori I (2004)

Berikut merupakan sebagian dari keluaran hasil eksekusi perintah "**top b n 1**" pada sebuah sistem GNU/Linux yaitu "**rmsbase.vlsm.org**" beberapa saat yang lalu.

```
top - 10:59:25 up  3:11,  1 user,  load average: 9.18, 9.01, 7.02
Tasks: 122 total,   3 running, 119 sleeping,   0 stopped,   0 zombie
Cpu(s):  14.5% user,  35.0% system,   1.4% nice,  49.1% idle
Mem:    256712k total,   253148k used,     3564k free,    20148k buffers
Swap:   257032k total,    47172k used,   209860k free,    95508k cached

  PID USER   VIRT  RES  SHR %MEM  PPID SWAP CODE DATA nDRT COMMAND
    1 root    472  432  412  0.2     0   40   24  408    5 init
    4 root      0    0    0  0.0     1    0    0    0    0 kswapd
   85 root      0    0    0  0.0     1    0    0    0    0 kjournald
  334 root    596  556  480  0.2     1   40   32  524   19 syslogd
  348 root    524  444  424  0.2     1   80   20  424    5 gpm
  765 rms46  1928  944  928  0.4     1  984   32  912   23 kdeinit
  797 rms46  6932 5480 3576  2.1   765 1452   16 5464  580 kdeinit
  817 rms46  1216 1144 1052  0.4   797   72 408  736   31 bash
 5441 rms46   932  932  696  0.4   817    0   44  888   59 top
  819 rms46  1212 1136 1072  0.4   797   76 404  732   32 bash
27506 rms46   908  908  760  0.4   819    0 308  600   37 shsh
27507 rms46   920  920  808  0.4 27506    0 316  604   38 sh
 5433 rms46  1764 1764  660  0.7 27507    0 132 1632  282 rsync
 5434 rms46  1632 1628 1512  0.6  5433    4 124 1504  250 rsync
 5435 rms46  1832 1832 1524  0.7  5434    0 140 1692  298 rsync
27286 rms46 24244  23m  14m  9.4   765    0  52  23m 2591 firefox-bin
27400 rms46 24244  23m  14m  9.4 27286    0  52  23m 2591 firefox-bin
27401 rms46 24244  23m  14m  9.4 27400    0  52  23m 2591 firefox-bin
27354 rms46 17748  17m 7948  6.9     1    0 496  16m 2546 evolution-mail
27520 rms46 17748  17m 7948  6.9 27354    0 496  16m 2546 evolution-mail
27521 rms46 17748  17m 7948  6.9 27520    0 496  16m 2546 evolution-mail
```

a. Berapakah ukuran total, memori fisik dari sistem tersebut di atas?

b. Terangkan, apa yang dimaksud dengan: "VIRT", "RES", "SHR", "PPID", "SWAP", "CODE", "DATA", "nDRT".

c. Bagaimanakah, hubungan (rumus) antara "RES" dengan parameter lainnya?

d. Bagaimanakah, hubungan (rumus) antara "VIRT", dengan parameter lainnya?

B.28. Status Memori II (2005)

Diketahui, keluaran dari perintah "**swapon -s**" atau isi "`/proc/swaps`" sebagai berikut:

```
Filename                Type      Size    Used  Priority
/dev/hda3               partition 257032  10636 -1
/extra/.swap/swapfile   file      1047544 0     -2
```

SEBAGIAN keluaran dari perintah "**top b n 1**" (dengan modifikasi `.toprc`) sebagai berikut:

```
top - 22:04:32 up  3:08, 20 users,  load average: 0.04, 0.08, 0.06
Tasks: 131 total,   3 running, 128 sleeping,   0 stopped,   0 zombie
Cpu(s):  3.5% us,  0.4% sy,  0.0% ni, 94.5% id,  1.6% wa,  0.1% hi,  0.0% si
Mem:    516064k total,   509944k used,     6120k free,    47708k buffers
Swap:  1304576k total,    10636k used,  1293940k free,   117964k cached

  PID  PPID  UID  VIRT SWAP  RES  SHR CODE DATA %MEM nFLT nDRT COMMAND
    1     0    0  1560 1028  532  460   28  240  0.1   14    0 init
 7915     1    0  2284 1532  752  672   20  240  0.1    0    0 inetd
 8281     1 1000  2708 1292 1416  888   64  644  0.3    1    0 gam_server
 9450     1 1000  128m  68m  60m  21m   60  86m 12.0  243    0 firefox-bin
11744 11743 1017 95972  54m  39m  17m   60  61m  7.7   36    0 firefox-bin
11801 11800 1024 88528  53m  32m  15m   60  57m  6.5    0    0 firefox-bin
11844 11843 1003 96844  54m  40m  16m   60  63m  8.0    5    0 firefox-bin
 8168     1    0 12096 7528 4568 3180  352 1692  0.9   17    0 apache2
 8213  8168   33 12096 7516 4580 3180  352 1692  0.9    0    0 apache2
 8214  8168   33 12096 7516 4580 3180  352 1692  0.9    0    0 apache2
 8215  8168   33 12096 7516 4580 3180  352 1692  0.9    0    0 apache2
 8216  8168   33 12096 7516 4580 3180  352 1692  0.9    0    0 apache2
```

a. Terangkan secara singkat; apa yang dimaksud dengan: PID, PPID, UID, VIRT, SWAP, RES, SHR, CODE, DATA, %MEM, nFLT, nDRT, COMMAND.

b. Berapa ukuran total dari sistem "*swap*"?

c. Berapa besar bagian "*swap*" yang sedang digunakan?

d. Berapa besar ukuran bagian "*swap*" yang berada dalam partisi terpisah?

e. Berapa besar ukuran bagian "*swap*" yang berbentuk berkas biasa?

B.29. Managemen Memori dan Utilisasi CPU (2004)

a. Terangkan bagaimana pengaruh derajat "*multiprogramming*" (MP) terhadap utilisasi CPU. Apakah peningkatan MP akan selalu meningkatkan utilisasi CPU? Mengapa?

b. Terangkan bagaimana pengaruh dari "*page-fault*" memori terhadap utilisasi CPU!

c. Terangkan bagaimana pengaruh ukuran memori (*RAM size*) terhadap utilisasi CPU!

d. Terangkan bagaimana pengaruh memori virtual (VM) terhadap utilisasi CPU!

e. Terangkan bagaimana pengaruh teknologi "*copy on write*" terhadap utilisasi CPU!

f. Sebutkan Sistem Operasi berikut mana saja yang telah mengimplementasi teknologi "*copy on write*": Linux 2.4, Solaris 2, Windows 2000.

B.30. Memori I (2002)

a. Terangkan, apa yang dimaksud dengan algoritma penggantian halaman *Least Recently Used* (LRU)!

b. Diketahui sebuah *reference string* berikut: " 1 2 1 7 6 7 3 4 3 5 6 7 ". Jika proses mendapat alokasi tiga *frame*; gambarkan pemanfaatan *frame* tersebut menggunakan *reference string* tersebut di atas menggunakan algoritma LRU.

c. Berapa *page fault* yang terjadi?

d. Salah satu implementasi LRU ialah dengan menggunakan *stack*; yaitu setiap kali sebuah halaman memori dirujuk, halaman tersebut diambil dari *stack* serta diletakkan ke atas (*TOP of*) *stack*. Gambarkan urutan penggunaan *stack* menggunakan *reference string* tersebut.

B.31. Memori II (2002)

Diketahui spesifikasi sistem memori virtual sebuah proses sebagai berikut:

- *page replacement* menggunakan algoritma LRU (*Least Recently Used*).
- alokasi memori fisik dibatasi hingga 1000 *bytes* (per proses).
- ukuran halaman (*page size*) harus tetap (*fixed*, minimum 100 bytes).
- usahakan, agar terjadi *page fault* sesedikit mungkin.
- proses akan mengakses alamat berturut-turut sebagai berikut:

```
1001, 1002, 1003, 2001, 1003, 2002, 1004, 1005, 2101, 1101,
2099, 1001, 1115, 3002, 1006, 1007, 1008, 1009, 1101, 1102
```

a. Tentukan ukuran halaman yang akan digunakan.

b. Berapakah jumlah *frame* yang dialokasikan?

c. Tentukan *reference string* berdasarkan ukuran halaman tersebut di atas!

d. Buatlah bagan untuk algoritma LRU!

e. Tentukan jumlah *page-fault* yang terjadi!

B.32. Memori III (2003)

Sebuah proses secara berturut-turut mengakses alamat memori berikut:

```
1001, 1002, 1003, 2001, 2002, 2003, 2601, 2602, 1004, 1005,
1507, 1510, 2003, 2008, 3501, 3603, 4001, 4002, 1020, 1021.
```

Ukuran setiap halaman (page) ialah 500 bytes.

a. Tentukan "*reference string*" dari urutan pengaksesan memori tersebut.

b. Gunakan algoritma "*Optimal Page Replacement*". Tentukan jumlah "*frame*" minimum yang diperlukan agar terjadi "*page fault*" minimum! Berapakah jumlah "*page fault*" yang terjadi? Gambarkan dengan sebuah bagan!

c. Gunakan algoritma "*Least Recently Used (LRU)*". Tentukan jumlah "*frame*" minimum yang diperlukan agar terjadi "*page fault*" minimum! Berapakah jumlah "*page fault*" yang terjadi? Gambarkan dengan sebuah bagan!

d. Gunakan jumlah "*frame*" hasil perhitungan butir "b" di atas serta alrgoritma LRU. Berapakah jumlah "*page fault*" yang terjadi? Gambarkan dengan sebuah bagan!

B.33. Memori Virtual (2005)

Diketahui sebuah sistem memori dengan ketentuan sebagai berikut:

- Ukuran "*Logical Address Space*" ialah 16 *bit*.
- Ukuran sebuah "*Frame*" memori fisik ialah 512 *byte*.
- Menggunakan "*Single Level Page Table*".
- Setiap "*Page Table Entry*" terdiri dari 1 *bit* "*Valid/Invalid Page*" (*MSB*) dan 7 *bit* "*Frame Pointer Number*" (total=8 *bit*).
- "*Page Table*" diletakkan mulai alamat 0 (nol) pada memori fisik.
- "*Frame #0*" diletakkan langsung setelah "*Page Table*" berakhir. Demikian seterusnya, "*Frame #1*", "*Frame #2*", ... hingga "*Frame #7*".

a. Berapa *byte*, kapasitas maksimum dari "*Virtual Memory*" dengan "*Logical Address Space*" tersebut?

b. Gambarkan pembagian "*Logical Address Space*" tersebut: berapa *bit* untuk PT/"*Page Table*", serta berapa *bit* untuk alokasi *offset*?



c. Berapa *byte* yang diperlukan untuk "*Page Table*" tersebut di atas?

d. Berapa *byte* ukuran total dari memori fisik (semua *Frame* dan *Page Table*)

B.34. *Multilevel Paging Memory* I (2003)

Diketahui sekeping memori berukuran 32 *byte* dengan alamat fisik "00" - "1F" (Heksadesimal) - yang digunakan secara "*multilevel paging*" - serta dialokasikan untuk keperluan berikut:

- "*Outer Page Table*" ditempatkan secara permanen (*non-swappable*) pada alamat "00" - "07" (Heks).
- Terdapat alokasi untuk dua (2) "*Page Table*", yaitu berturut-turut pada alamat "08" - "0B" dan "0C" - "0F" (Heks). Alokasi tersebut dimanfaatkan oleh semua "*Page Table*" secara bergantian (*swappable*) dengan algoritma "LRU".
- Sisa memori "10" - "1F" (Heks) dimanfaatkan untuk menempatkan sejumlah "*memory frame*".

Keterangan tambahan perihal memori sebagai berikut:

- Ukuran "*Logical Address Space*" ialah tujuh (7) bit.
- Ukuran data ialah satu *byte* (8 bit) per alamat.
- "*Page Replacement*" menggunakan algorithma "LRU".
- "*Invalid Page*" ditandai dengan bit pertama (MSB) pada "*Outer Page Table*"/ "*Page Table*" diset menjadi "1".
  - sebaliknya, "*Valid Page*" ditandai dengan bit pertama (MSB) pada "*Outer Page Table*"/ "*Page Table*" diset menjadi "0", serta berisi alamat awal (*pointer*) dari "*Page Table*" terkait.

Pada suatu saat, isi keping memori tersebut sebagai berikut:

address	**isi**		***address***	**isi**		***address***	**isi**		***address***	**isi**
00H	08H		08H	10H		10H	10H		18H	18H
01H	0CH		09H	80H		11H	11H		19H	19H
02H	80H		0AH	80H		12H	12H		1AH	1AH
03H	80H		0BH	18H		13H	13H		1BH	1BH
04H	80H		0CH	14H		14H	14H		1CH	1CH
05H	80H		0DH	1CH		15H	15H		1DH	1DH
06H	80H		0EH	80H		16H	16H		1EH	1EH
07H	80H		0FH	80H		17H	17H		1FH	1FH

a. Berapa *byte*, kapasitas maksimum dari "*Virtual Memory*" dengan "*Logical Address Space*" tersebut?

b. Gambarkan pembagian "*Logical Address Space*" tersebut: berapa bit untuk P1/ "*Outer Page Table*", berapa bit untuk P2/ "*Page Table*", serta berapa bit untuk alokasi *offset*?



c. Berapa *byte*, ukuran dari sebuah "*memory frame*" ?

d. Berapa jumlah total dari "*memory frame*" pada keping tersebut?

e. Petunjuk: Jika terjadi "*page fault*", terangkan juga apakah terjadi pada "*Outer Page Table*" atau pada "*Page Table*". Jika tidak terjadi "*page fault*", sebutkan isi dari *Virtual Memory Address* berikut ini:

   i. *Virtual Memory Address*: 00H

   ii. *Virtual Memory Address*: 3FH

   iii. *Virtual Memory Address*: 1AH

B.35. *Multilevel Paging Memory* II (2004)

Diketahui sekeping memori berukuran 32 *byte* dengan alamat fisik "00" - "1F" (Heksadesimal) - yang digunakan secara "*multilevel paging*" - serta dialokasikan untuk keperluan berikut:

- "*Outer Page Table*" ditempatkan secara permanen (*non-swappable*) pada alamat "00" - "03" (Heks).
- Terdapat alokasi untuk tiga (3) "*Page Table*", yaitu berturut-turut pada alamat "04" - "07", "08" - "0B" dan "0C" - "0F" (Heks).
- Sisa memori "10" - "1F" (Heks) dimanfaatkan untuk menempatkan sejumlah "*memory frame*".

Keterangan tambahan perihal memori sebagai berikut:

- Ukuran "*Logical Address Space*" ialah tujuh (7) bit.
- Ukuran data ialah satu *byte* (8 bit) per alamat.
- "*Page Replacement*" menggunakan algorithma "LRU".
- "*Invalid Page*" ditandai dengan bit pertama (MSB) pada "*Outer Page Table*"/ "*Page Table*" diset menjadi "1".
  - sebaliknya, "*Valid Page*" ditandai dengan bit pertama (MSB) pada "*Outer Page Table*"/ "*Page Table*" diset menjadi "0", serta berisi alamat awal (*pointer*) dari "*Page Table*" terkait.

Pada suatu saat, isi keping memori tersebut sebagai berikut:

address	**isi**		***address***	**isi**		***address***	**isi**		***address***	**isi**
00H	80H		08H	80H		10H	10H		18H	18H
01H	04H		09H	80H		11H	11H		19H	19H
02H	08H		0AH	80H		12H	12H		1AH	1AH
03H	0CH		0BH	80H		13H	13H		1BH	1BH
04H	80H		0CH	80H		14H	14H		1CH	1CH
05H	10H		0DH	80H		15H	15H		1DH	1DH
06H	80H		0EH	80H		16H	16H		1EH	1EH
07H	80H		0FH	18H		17H	17H		1FH	1FH

a. Berapa *byte*, kapasitas maksimum dari "*Virtual Memory*" dengan "*Logical Address Space*" tersebut?

b. Gambarkan pembagian "*Logical Address Space*" tersebut: berapa bit untuk P1/ "*Outer Page Table*", berapa bit untuk P2/ "*Page Table*", serta berapa bit untuk alokasi *offset*?



c. Berapa *byte*, ukuran dari sebuah "*memory frame*" ?

d. Berapa jumlah total dari "*memory frame*" pada keping tersebut?

e. Petunjuk: Jika terjadi "*page fault*", terangkan juga apakah terjadi pada "*Outer Page Table*" atau pada "*Page Table*". Jika tidak terjadi "*page fault*", sebutkan isi dari *Virtual Memory Address* berikut ini:

   i. *Virtual Memory Address*: 00H

   ii. *Virtual Memory Address*: 28H

   iii. *Virtual Memory Address*: 55H

   iv. *Virtual Memory Address*: 7BH

B.36. *Multilevel Paging Memory* III (2005)

[1 k = 2^{10}; 1 M = 2^{20}; 1 G = 2^{30}]

a. Sebuah sistem komputer menggunakan ruang alamat logika (*logical address space*) 32 bit dengan ukuran halaman (*page size*) 4 kbyte. Jika sistem menggunakan skema tabel halaman satu tingkat (*single level page table*); perkirakan ukuran memori yang diperlukan untuk tabel halaman tersebut! Jangan lupa: setiap masukan tabel halaman memerlukan satu bit ekstra sebagai *flag*!

b. Jika sistem menggunakan skema tabel halaman dua tingkat (*two level page table*) dengan ukuran *outer-page* 10 bit; tentukan bagaimana konfigurasi minimum tabel yang diperlukan (minimum berapa *outer-page table* dan minimum berapa *page table*)? Perkirakan ukuran memori yang diperlukan untuk konfigurasi minimum tersebut?

c. Terangkan keuntungan dan kerugian skema tabel halaman satu tingkat tersebut!

d. Terangkan keuntungan dan kerugian skema tabel halaman dua tingkat tersebut!

e. Terangkan mengapa skema tabel halaman bertingkat kurang cocok untuk ruang alamat yang lebih besar dari 32 bit? Bagaimana cara mengatasi hal tersebut?

B.37. FHS *(File Hierarchy Standards)* (2002)

a. Sebutkan tujuan dari FHS.

b. Terangkan perbedaan antara "*shareable*" dan "*unshareable*".

c. Terangkan perbedaan antara "*static*" dan "*variable*".

d. Terangkan/berikan ilustrasi sebuah direktori yang "*shareable*" dan "*static*".

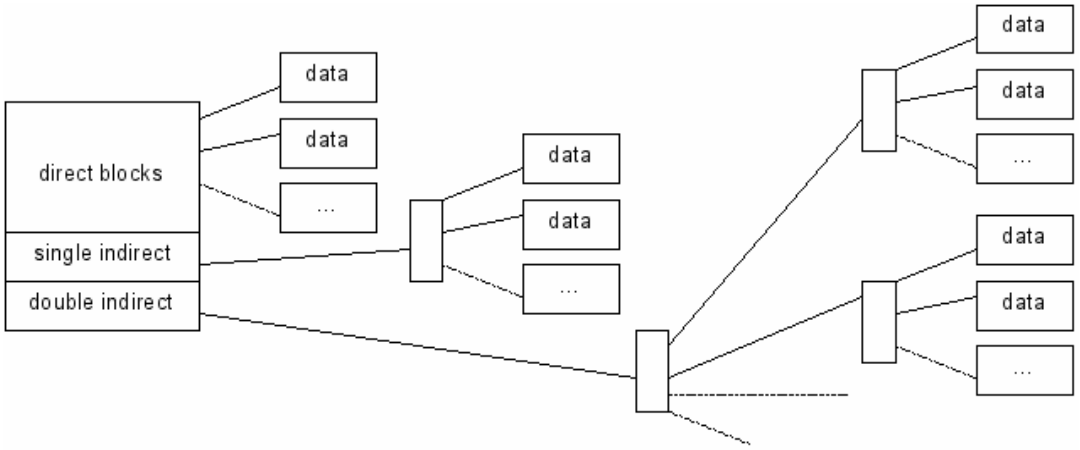
e. Terangkan/berikan ilustrasi sebuah direktori yang "*shareable*" dan "*variable*".

f. Terangkan/berikan ilustrasi sebuah direktori yang *unshareable*". dan *static*".

g. Terangkan/berikan ilustrasi sebuah direktori yang *unshareable*". dan *variable*".

B.38. Sistem Berkas I (2002)

Sebuah sistem berkas menggunakan metoda alokasi serupa *i-node* (*unix*). Ukuran *pointer* berkas (*file pointer*) ditentukan 10 *bytes*. *Inode* dapat mengakomodir 10 *direct blocks*, serta masing-masing sebuah *single indirect block* dan sebuah *double indirect block*.



a. Jika ukuran blok = 100 *bytes*, berapakah ukuran maksimum sebuah berkas?

b. Jika ukuran blok = 1000 *bytes*, berapakah ukuran maksimum sebuah berkas?

c. Jika ukuran blok = N *bytes*, berapakah ukuran maksimum sebuah berkas?

B.39. Sistem Berkas II (2003)

Pada saat merancang sebuah situs web, terdapat pilihan untuk membuat link berkas yang absolut atau pun relatif.

a. Berikan sebuah contoh, link berkas yang absolut.

b. Berikan sebuah contoh, link berkas yang relatif.

c. Terangkan keunggulan dan/atau kekurangan jika menggunakan link absolut.

d. Terangkan keunggulan dan/atau kekurangan jika menggunakan link relatif.

B.40. Sistem Berkas III (2004)

a. Terangkan persamaan dan perbedaan antara operasi dari sebuah sistem direktori dengan operasi dari sebuah sistem berkas (*filesystem*).

b. Silberschatz et. al. mengilustrasikan sebuah model sistem berkas berlapis enam (6 *layers*), yaitu "*application programs*", "*logical file system*", "*file-organization module*", "*basic file system*", "*kendali M/K*", "*devices*". Terangkan lebih rinci serta berikan contoh dari ke-enam lapisan tersebut!

c. Terangkan mengapa pengalokasian blok pada sistem berkas berbasis *FAT* (*MS DOS*) dikatakan efisien! Terangkan pula kelemahan dari sistem berkas berbasis *FAT* tersebut!

d. Sebutkan dua fungsi utama dari sebuah *Virtual File Sistem* (secara umum atau khusus Linux).

B.41. Sistem Berkas IV (2005)

a. Terangkan kedua fungsi dari sebuah VFS (*Virtual File System*).

b. Bandingkan implementasi sistem direktori antara *Linier List* dan *Hash Table*. Terangkan kelebihan/kekurangan masing-masing!

B.42. Sistem Berkas "*ReiserFS*" (2003)

a. Terangkan secara singkat, titik fokus dari pengembangan sistem berkas "*reiserfs*": apakah berkas berukuran besar atau kecil, serta terangkan alasannya!

b. Sebutkan secara singkat, dua hal yang menyebabkan ruangan (*space*) sistem berkas "*reiserfs*" lebih efisien!

c. Sebutkan secara singkat, manfaat dari "*balanced tree*" dalam sistem berkas "*reiserfs*"!

d. Sebutkan secara singkat, manfaat dari "*journaling*" pada sebuah sistem berkas!

e. Sistem berkas "*ext2fs*" dilaporkan 20% lebih cepat jika menggunakan blok berukuran 4 *kbyte* dibandingkan 1 *kbyte*. Terangkan mengapa penggunaan ukuran blok yang besar dapat meningkatkan kinerja sistem berkas!

f. Para pengembang sistem berkas "*ext2fs*" merekomendasikan blok berukuran 1 *kbyte* dari pada yang berukuran 4 *kbyte*. Terangkan, mengapa perlu menghindari penggunaan blok berukuran besar tersebut!

B.43. Sistem Berkas "*NTFS*" (2005)

Keunggulan *NTFS* diantaranya dalam banyak hal seperti: "*data recovery*", "*security*", "*fault tolerance*", "*very large file system* / *very large file size*", "*multiple data streams*", "*UNICODE names*", "*sparse file*", "*encryption*", "*journaling*", "*file compression*", dan "*shadow copies*". Jabarkan secara lebih rinci, sekurangnya lima (5) keunggulan tersebut di atas.

B.44. RAID (*Redudant Array of I* Disks*) (2004)

a. Terangkan dan ilustrasikan: apa yang dimaksud dengan RAID level 0.

b. Terangkan dan ilustrasikan: apa yang dimaksud dengan RAID level 1.

c. Terangkan dan ilustrasikan: apa yang dimaksud dengan RAID level 0 + 1.

d. Terangkan dan ilustrasikan: apa yang dimaksud dengan RAID level 1 + 0.

B.45. *Mass Storage System* I (2002)

Bandingkan jarak tempuh (dalam satuan silinder) antara penjadualan FCFS (*First Come First Served*), SSTF (*Shortest-Seek-Time-First*), dan LOOK. Isi antrian permintaan akses berturut-turut untuk silinder:

```
100, 200, 300, 101, 201, 301.
```

Posisi awal *disk head* pada silinder 0.

B.46. *Mass Storage System* II (2003)

Posisi awal sebuah "disk head " pada silinder 0. Antrian permintaan akses berturut-turut untuk silinder:

```
100, 200, 101, 201.
```

a. Hitunglah jarak tempuh (dalam satuan silinder) untuk algoritma penjadualan "*First Come First Served*" (FCFS).

b. Hitunglah jarak tempuh (dalam satuan silinder) untuk algoritma penjadualan "*Shortest Seek Time First*" (SSTF).

B.47. *Mass Storage System* III (2003)

Pada sebuah PC terpasang sebuah disk IDE/ATA yang berisi dua sistem operasi: *MS Windows 98 SE*dan *Debian GNU/Linux Woody 3.0 r1*. Informasi "*fdisk*" dari perangkat disk tersebut sebagai berikut:

```
# fdisk /dev/hda
=====================================================
   Device Boot Start   End    Blocks  Id  System
               (cylinders)   (kbytes)
-----------------------------------------------------
/dev/hda1    *     1   500   4000000  0B  Win95 FAT32
/dev/hda2        501   532    256000  82  Linux swap
/dev/hda3        533  2157  13000000  83  Linux
/dev/hda4       2158  2500   2744000  83  Linux
```

Sedangkan informasi berkas "*fstab*" sebagai berikut:

```
# cat /etc/fstab

# ============================================================
# <file system> <mount point> <type> <options> <dump> <pass>
# ------------------------------------------------------------
/dev/hda1       /win98        vfat   defaults    0      2
/dev/hda2       none          swap   sw          0      0
/dev/hda3       /             ext2   defaults    0      0
/dev/hda4       /home         ext2   defaults    0      2
# ------------------------------------------------------------
```

Gunakan pembulatan 1 Gbyte = 1000 Mbytes = 1000000 kbytes dalam perhitungan berikut ini:

a. Berapa *Gbytes* kapasitas disk tersebut di atas?

b. Berapa jumlah silinder disk tersebut di atas?

c. Berapa *Mbytes* terdapat dalam satu silinder?

d. Berapa *Mbytes* ukuran partisi dari direktori "`/home`"?

Tambahkan disk ke dua (`/dev/hdc`) dengan spesifikasi teknis serupa dengan disk tersebut di atas (`/dev/hda`). Bagilah disk kedua menjadi tiga partisi:

- 4 *Gbytes* untuk partisi *Windows FAT32* (Id: 0B)
- *256 Mbytes* untuk partisi *Linux Swap* (Id: 82)
- Sisa disk untuk partisi "`/home`" yang baru (Id: 83).

Partisi "`/home`" yang lama (disk pertama) dialihkan menjadi "`/var`".

e. Bagaimana bentuk infomasi "*fdisk*" untuk "`/dev/hdc`" ini?

f. Bagaimana seharusnya isi berkas "`/etc/fstab`" setelah penambahan disk tersebut?

B.48. *I/O Interface* (2003)

Bandingkan perangkat disk yang berbasis *IDE/ATA* dengan yang berbasis *SCSI*:

a. Sebutkan kepanjangan dari *IDE/ATA*.

b. Sebutkan kepanjangan dari *SCSI*.

c. Berapakah kisaran harga kapasitas disk *IDE/ATA* per satuan *Gbytes*?

d. Berapakah kisaran harga kapasitas disk *SCSI* per satuan *Gbytes*?

e. Bandingkan beberapa parameter lainnya seperti unjuk kerja, jumlah perangkat, penggunaan *CPU*, dst.

B.49. *I/O* dan *USB* (2004)

a. Sebutkan sedikitnya sepuluh (10) kategori perangkat yang telah berbasis *USB*!

b. Standar *IEEE 1394b (FireWire800)* memiliki kinerja tinggi, seperti kecepatan alih data 800 MBit per detik, bentangan/jarak antar perangkat hingga 100 meter, serta dapat menyalurkan catu daya hingga 45 Watt. Bandingkan spesifikasi tersebut dengan *USB 1.1* dan *USB 2.0*.

c. Sebutkan beberapa keunggulan perangkat *USB* dibandingkan yang berbasis standar *IEEE 1394b* tersebut di atas!

d. Sebutkan dua trend perkembangan teknologi perangkat *I/O* yang saling bertentangan (konflik).

e. Sebutkan dua aspek dari sub-sistem *I/O* kernel yang menjadi perhatian utama para perancang Sistem Operasi!

f. Bagaimana *USB* dapat mengatasi trend dan aspek tersebut di atas?

B.50. Struktur Keluaran/Masukan Kernel (*I/O*) (2004)

a. Buatlah sebuah bagan yang menggambarkan hubungan/relasi antara lapisan-lapisan (*layers*) kernel, subsistem M/K (*I/O*), *device driver*, *device controller*, dan *devices*.

b. Dalam bagan tersebut, tunjukkan dengan jelas, bagian mana yang termasuk perangkat keras, serta bagian mana yang termasuk perangkat lunak.

c. Dalam bagan tersebut, berikan contoh sekurangnya dua *devices*!

d. Terangkan apa yang dimaksud dengan *devices*!

e. Terangkan apa yang dimaksud dengan *device controller*!

f. Terangkan apa yang dimaksud dengan *device driver*!

g. Terangkan apa yang dimaksud dengan subsistem M/K (*I/O*)!

h. Terangkan apa yang dimaksud dengan kernel!

B.51. Masukan/Keluaran (2005)

a. Terangkan secara singkat, sekurangnya enam prinsip/cara untuk meningkatkan efisiensi M/K (Masukan/Keluaran)!

b. Diketahui sebuah model M/K yang terdiri dari lapisan-lapisan berikut: Aplikasi, Kernel, *Device-Driver*, *Device-Controller*, *Device*. Terangkan pengaruh pemilihan lapisan tersebut untuk pengembangan sebuah aplikasi baru. Diskusikan aspek-aspek berikut ini: Jumlah waktu pengembangan, Efisiensi, Biaya Pengembangan, Abstraksi, dan Fleksibilitas.

B.52. *CDROM* (2005)

a. Terangkan, apa bedanya antara kepingan CD *Audio*, CD-ROM, CD-R, dan CD-RW!

b. Pada awalnya sekeping CD *Audio* (650 MB) memiliki durasi 74 menit. Hitung, berapa kecepatan transfer sebuah *CD-ROM reader* dengan kecepatan "37 x".

B.53. *HardDisk* I (2001)

Diketahui sebuah perangkat DISK dengan spesifikasi berikut ini:

- Kapasitas 100 *Gbytes* (asumsi 1*Gbytes* = 1000 *Mbytes*).
- Jumlah lempengan (*plate*) ada dua (2) dengan masing-masing dua (2) sisi permukaan (*surface*).
- Jumlah silinder = 2500 (Revolusi: 6000 *RPM*)
- Pada suatu saat, hanya satu *HEAD* (pada satu sisi) yang dapat aktif.

a. Berapakah waktu latensi maksimum dari perangkat DISK tersebut?

b. Berapakah rata-rata latensi dari perangkat DISK tersebut?

c. Berapakah waktu minimum (tanpa latensi dan *seek*) yang diperlukan untuk mentransfer satu juta (1 000 000) *byte* data?

B.54. *HardDisk* II (2003)

Diketahui sebuah perangkat DISK dengan spesifikasi:

- Dua (2) permukaan (*surface* #0, #1).
- Jumlah silinder: 5000 (*cyl*. #0 - #4999).
- Kecepatan Rotasi: 6000 *rpm*.
- Kapasitas Penyimpanan: 100 *Gbyte*.
- Jumlah sektor dalam satu trak: 1000 (*sec*. #0 - #999).
- Waktu tempuh *seek* dari *cyl*. #0 hingga #4999 ialah 10 mS.
- Pada T=0, *head* berada pada posisi *cyl* #0, *sec*. #0.
- Satuan M/K terkecil untuk baca/tulis ialah satu (1) sektor.
- Akan menulis data sebanyak 5010 *byte* pada *cyl*. #500, *surface* #0, *sec*. #500.
- Untuk memudahkan, 1 *kbyte* = 1000 *byte*; 1 *Mbyte* = 1000 *kbyte*; 1 *Gbyte* = 1000 *Mbyte*.

a. Berapakah kecepatan *seek* dalam satuan *cyl*/*ms* ?

b. Berapakah *rotational latency* (*max*.) dalam satuan *ms* ?

c. Berapakah jumlah (*byte*) dalam satu sektor ?

d. Berapa lama (*ms*) diperlukan *head* untuk mencapai *cyl*. #500 dari *cyl*. #0, *sec*. #0 ?

e. Berapa lama (*ms*) diperlukan head untuk mencapai *cyl*. #500, *sec*. #500 dari *cyl*. #0, *sec*. #0?

f. Berapa lama (*ms*) diperlukan untuk menulis kedalam satu sektor ?

g. Berdasarkan butir (e) dan (f) di atas, berapa kecepatan transfer efektif untuk menulis data sebanyak 5010 *byte* ke dalam disk tersebut dalam satuan *Mbytes*/detik?

B.55. *HardDisk* III (2004)

Diketahui sebuah disk dengan spesifikasi berikut ini:

- Dua (2) permukaan (muka #0 dan #1).
- Jumlah silinder: 5000 (silinder #0 - #4999).
- Kecepatan Rotasi: 6000 *rpm*.
- Kapasitas Penyimpanan: 100 *Gbytes*.
- Jumlah sektor dalam satu trak: 1000 (sektor #0 - #999).

- Waktu tempuh hingga stabil antar trak yang berurutan: 1 *mS* (umpama dari trak #1 ke trak #2).
- Pada setiap saat, hanya satu muka yang *head*-nya aktif (baca/tulis). Waktu alih antar muka (dari muka #0 ke muka #1) dianggap 0 *mS*.
- Algoritma pergerakan *head*: *First Come First Served*.
- Satuan M/K (*I/O*) terkecil untuk baca/tulis ialah satu (1) sektor.
- Pada T=0, *head* berada pada posisi silinder #0, sektor #0.
- Untuk memudahkan, 1 *kbyte* = 1000 *byte*; 1 *Mbyte* = 1000 *kbyte*; 1 *Gbyte* = 1000 *Mbyte*.

a. Berapa kapasitas (*kbyte*) dalam satu sektor?

b. Berapakah *rotational latency* maksimum (*mS*) ?

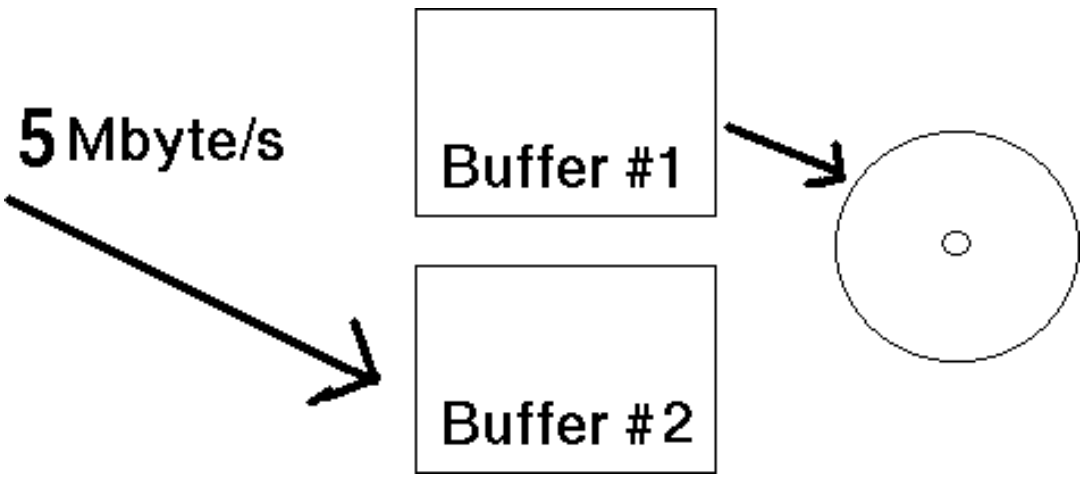
c. Berapakah waktu tempuh (*mS*) dari muka #0, trak #0, sektor #0 ke muka #0, trak #0, sektor #999 ([0,0,0] –> [0,0,999])?

d. Berapakah waktu tempuh (*mS*) dari [0,0,0] –> [0,0,999] –> [0,1,500] –> [0,1,999] –> [0,1,0] –> [0,1,499]?

e. Berapakah waktu tempuh (mS) dari [0,0,0] –> [0,0,999] –> [0,1,0] –> [0,1,999]?

B.56. *HardDisk* IV (2005)

- Sebuah disk (6000 *RPM*) yang setiap *track* terdiri dari 1000 sektor @ 10 *kbytes*.
- Penulisan disk didukung dengan dua *buffer* @ 10 *kbytes* yang diisi secara bergantian dengan kecepatan 5 *Mbytes/detik* oleh sistem.
- Umpamanya, jika *buffer* #1 penuh, maka proses penulisan ke sebuah sektor disk dimulai, sedangkan *buffer* #2 diisi oleh sistem. Jika proses penulisan dari *buffer* #1 selesai, maka *buffer* #2 di tulis ke disk, sedangkan *buffer* #1 diisi. Jika kedua *buffer* penuh, maka sistem menunda pengisian, hingga *buffer* dapat diisi kembali.
- Abaikan semua macam waktu *tunda/delay* seperti *buffer switch time*, *seek time*, kecuali "*Rotational Latency*". Pada saat t=0, *buffer* #1 penuh (siap tulis ke disk), dan *buffer* #2 kosong (siap diisi). Posisi *head* berada pada sektor #0 pada sebuah *track* tertentu.
- Gunakan *Gantt chart/timing-chart*, jika dianggap perlu!



a. Hitung, berapa lama diperlukan untuk menulis berturut-turut sebesar 30 *kbytes* ke sektor #0, #1, #2.

b. Berapa waktu minimum diperlukan untuk menulis 30 *kbytes* ke *track* tersebut.

c. Data tersebut, akan diisikan ke sektor berapa saja?

B.57. Waktu Nyata/Multimedia (2005)

a. Sebutkan sekurangnya empat ciri/karakteristik dari sebuah sistem waktu nyata. Terangkan secara singkat, maksud masing-masing ciri tersebut!

b. Sebutkan sekurangnya tiga ciri/karakteristik dari sebuah sistem multimedia. Terangkan secara singkat, maksud masing-masing ciri tersebut!

c. Terangkan perbedaan prinsip kompresi antara berkas dengan format *JPEG* dan berkas dengan format *GIF*.

d. Terangkan lebih rinci, makna keempat faktor QoS berikut ini: *throughput*, *delay*, *jitter*, *reliabilty*!

B.58. Tugas Kelompok/Buku Sistem Operasi (2004)

Bandingkan buku Sistem Operasi versi 1.3 (terbitan awal 2003) dengan versi 1.9 (terbitan akhir 2003):

a. Sebutkan beberapa perbaikan/kemajuan umum buku versi 1.9 ini, dibandingkan dengan versi sebelumnya.

b. Sebutkan hal-hal yang masih perlu mendapatkan perhatian/perbaikan.

c. Penulisan Pokok Bahasan mana yang terbaik untuk versi 1.9 ini? Sebutkan alasannya!

d. Sebutkan sebuah Sub-Pokok Bahasan (SPB) yang anda ingat/kuasai (tidak harus yang anda kerjakan). SPB tersebut merupakan bagian dari Pokok Bahasan yang mana?

e. Bandingkan SPB tersebut di butir "d" dengan SPB setara yang ada di buku-buku Silberschatz et. al.; Tanenbaum, dan Stalling. Dimana perbedaan/persamaannya?

Lampiran C. *UPDATE/WishList*

C.1. *WishList*

1. -

Indeks

A

Alamat
  Fisik, 249
  Frame, 249
  Logik, 72
  Logis, 249
  Virtual, 74
Algoritma
  Algoritma Bankir, 204
  Alokasi Fixed Alocation
    Equal Allocation, 285
    Proportional Allocation, 285
  Anomali Belady, 279
  Antrian, 278
  Bit Modifikasi, 280
  Circular Queue, 280
  Counting, 282
  Elevator, 398
  FIFO, 278
  Frame, 278
  Halaman, 278
    Kesalahan Halaman, 278
    Pemindahan Halaman, 278
  Least Recently Used, 279
  NFU, 280
  NRU, 280
  Optimal, 279
  Page Buffering
    Frame, 282
    Halaman, 282
    Pool, 282
  Penjadwalan, 239, 403
  Perkiraan LRU, 280
  Second-Chance, 280
Alokasi Frame
  Kesalahan Halaman, 285
  Penghalamanan, 285
  Pergantian Halaman, 285
  Permintaan Halaman, 285
Alokasi Global Lawan Lokal
  Alokasi Global, 285
  Alokasi Lokal, 285
  Pergantian Global, 285
  Pergantian Lokal, 285
  Throughput, 285
Alokasi Memori
  Berkesinambungan, 241
  Fragmentasi, 241
  Proteksi Memori, 241
  Tidak Berkesinambungan, 241
Anomali
  Anomali Belady, 278
Antarmuka, 71
  Aplikasi, 422
  Standar, 74
Aspek Permintaan Halaman
  Memori Virtual, 271
  Pembuatan Proses, 271
  Sistem Permintaan Halaman, 271
Atmel ARM, 183

B

Backup
  Restore, 363
Bahasa
  Assembly, 65
  Pemrograman, 65
Berkas, 62, 66, 69, 307
  Atribut, 307
  Jenis, 307, 307
  Operasi, 308
  Shareable, 339
  Static, 339
  Struktur, 309
  Unshareable, 339
  Variable, 339
Binary Semaphore, 189
Block
  Bad, 406
  Boot, 405
Blok, 74
Buffer, 165
  Bounded Buffer, 167
  Unbounded Buffer, 167
Buffering, 386

C

Cache, 386
  Perbedaan Dengan Buffer, 386
Cancellation
  Asynchronous Cancellation, 112
  Defered Cancellation, 112
Client, 168
Command Interpreter, 67
  Shell, 63
Compile, 88
Console, 71
Context Switch, 126
Copy Semantics, 386
Copy-On-Write
  Halaman Copy-On-Write, 271
  Halaman Zero-Fill-On, 271
  Proses Anak, 271
  Proses Induk, 271
  Sistem Pemanggilan Exec(), 271
  Sistem Pemanggilan Fork(), 271
  Zero-Fill-On-Demand, 271
Counting Semaphore, 189
Critical Section
  Problema
    Syarat Solusi Masalah, 174
critical section
  Solusi
    Algoritma Solusi Masalah, 177

D

Data, 61, 62
Deadlock, 199
Debug, 74
Demand Paging

Paging
Halaman (Page), 264
Device
Driver, 71
Kompleksitas, 392
Direct Virtual-Memory Access, 380
Direktori, 61
CP/M, 335
MS-DOS, 335
Operasi, 312
Struktur, 312
Asiklik, 314
Dua Tingkat, 313
Pohon, 313
Satu Tingkat, 312
Single Level, 312
Two Level, 313
Umum, 314
Unix, 336
Disk, 62
Floppy, 417
Format, 405
Manajemen, 395, 405
Raw, 405
Struktur, 395
WORM, 419
Djikstra, 186
DMA
Cara Implementasi, 380
Channel, 430
Definisi, 378
Handshaking, 380
Transfer, 378
Drum, 74

E

Effective Access Time, 267
Eksekusi, 65
Error, 65
Error Handling, 388

F

File, 307
Flag pada Thread di Linux
Tabel Fungsi Flag, 155
FTP
DFS
WWW, 322

G

Garbage-Collection Scheme, 314
Graf
Graf Alokasi Sumber Daya
Graf Tunggu, 207

H

Hak Kekayaan Intelektual, 39
HaKI, 39
Hirarki
/usr, 342
/var, 344

I

Identifier
Eksternal, 74
Internal, 74
Implementasi
Direktori, 334
Sistem Berkas, 331
Informasi, 69
Instruksi Atomik, 184
Interface
Pengguna, 74
Interupsi, 376
Interrupt Vector Dan Interrupt Chaining, 378
Mekanisme Dasar, 376
Penyebab, 378
Rutin Penanganan, 74
IPC
Send/Recives, 165
IRQ, 430

J

Jaringan, 62
Ethernet, 429
Struktur, 52
Java RMI, 171

K

Kernel, 71, 74
Kernel Linux
Deskriptor Proses, 152
Komponen Modul, 98
Manajemen Modul, 98
Modul, 98
Pemanggilan Modul, 98
Kernel-mikro, 74
Kernelmikro, 71
Kesalahan Halaman
Ambil (Fetch), 265
Bit Validasi, 265
Dekode, 265
Free-Frame, 265
Illegal Address Trap, 265
Interrupt Handler , 249
Page-Fault Trap, 249
Terminasi, 265
Tidak Valid, 265
Kinerja
Efisiensi, 360
Komputasi, 65
Komunikasi, 62, 65, 66, 69, 74, 74
Langsung, 165
Konsep Dasar
Algoritma Pemindahan Halaman, 277
Bit Tambahan, 277
Disk, 277
Frame, 277
Frame Yang Kosong, 277
Kesalahan Halaman, 277
Memori, 277
Memori Fisik, 277
Memori Logis, 277

Memori Virtual, 277
Overhead, 277
Pemindahan Halaman, 277
Perangkat Keras, 277
Permintaan Halaman, 277
Proses, 277
Proses Pengguna, 277
Ruang Pertukaran, 277
Rutinitas, 277
String Acuan, 277
Tabel, 277
Waktu Akses Efektif, 277
Konsep Dasar Pemindahan Halaman
Kesalahan Halaman, 277
Pemindahan Halaman, 277
Permintaan Halaman, 277
Throughput, 277
Utilisasi CPU, 277
Kontrol proses, 69
Kooperatif, 163

L

Lapisan, 71
M/K, 74
Manajemen Memori, 74
Penjadwal CPU, 74
Layanan, 74
Level, 74
Linear List
Hash Table, 334
Linking Dinamis, 301
Linking Statis, 301
Linux
Tux, 96
Load Device Drivers, 389
Loading, 66
Loopback, 429

M

M/K, 33
Aplikasi Antarmuka, 383
Blok, 385
Efisiensi, 392
Jam Dan Timer, 385
Kinerja, 391
M/K Blok, 384
M/K Jaringan, 384
Nonblok, 385
Penjadwalan, 385
Subsistem Kernel, 385
M/K Stream
Definisi, 390
Mailbox, 166
Mainframe, 374
Manajemen
Berkas, 62
Memori
Utama, 61
Penyimpanan Sekunder, 62
Proses, 61
Sistem
Masukan/Keluaran, 62
Manajemen Memori, 263
Masalah dalam Segmentasi
Fragmentasi, 261
Masalah Readers/Writers, 221
Masukan/Keluaran, 62, 65
Memori, 61, 62
Gambaran Memori, 239
M/K
Buffer, 239
Manajemen Memori, 233
Pengguna, 239
Program Counter, 233
Ready Queue, 239
Share, 65
Shared, 69
Memori Virtual
Byte, 263
Memory Synchronous, 184
Memory-Mapped Files
Akses Memori Rutin, 272
Berkas M/K, 272
Disk, 272
Memori Virtual, 272
Pemetaan Memori, 272
Sistem Pemanggilan Read(), 272
Sistem Pemanggilan Write, 272
Mesin Virtual
Mesin Virtual Java, 88
Perangkat Lunak, 88
Message Passing, 65, 69, 74
Message-Passing, 389
Metode
Akses, 309
Mirroring, 408
Motherboard, 374
Mount
Umount
Mount Point, 316
Mount Point, 317
Multi-tasking, 69
Multiprogramming
Swap-In, 264
Swap-Out, 264
Mutual Exclusion
Arti Bebas, 174

N

Nama Berkas
Komponen, 339
NFS, 318

O

On Disk
In Memory, 331
Overlays
Assembler, 235
Two-pass Assembler, 235
Owner
Group
Universe, 324

P

Page Cache
Disk Cache
Buffer Cache, 361
Page Fault
Tidak Valid, 264
Valid, 264
Paralelisme, 408
Paritas, 408
Blok Interleaved, 408
Partisi
Mounting, 332
Path
Mutlak, 313
Relatif, 313
Pemanggilan Dinamis
Disk, 234
Linkage Loader, 234
Routine, 234
Pemberian Alamat, 239
Antrian Masukan, 457
Pengikatan Alamat
Waktu Eksekusi, 457
Waktu Pemanggilan, 457
Waktu Pembuatan, 457
Waktu Eksekusi, 239
Waktu Pemanggilan, 239
Waktu Pembuatan, 239
Pemberian Halaman
Alamat
Index, 247
Nomor Halaman, 247
Dukungan Perangkat Keras, 247
Keuntungan dan Kerugian, 247
Metoda Dasar, 247
Penerjemahan Halaman, 247
Proteksi, 247
Penamaan, 165
Penanganan Sinyal
Penerima Sinyal, 113
Pola Penanganan Sinyal, 113
Pengecekan Rutin
Restore
Backup, 362
Pengendali
Perangkat, 374
Penghubungan Dinamis
Pustaka Bersama, 234
Penjadwal
Penjadwal Jangka Panjang, 125
Penjadwalan
C-LOOK, 401
C-SCAN, 399
First-Come-First-Serve, 396
SCAN, 398
Shortest-Seek-Time-First, 397
Penjadwalan Disk, 395
Disk Bandwith, 395
System Call, 395
Penjadwalan LOOK, 401
Penjadwalan proses, 123
Penukaran
Penyimpanan Sementara, 239
Roll Out, Roll In, 239
Penyimpanan
Implementasi Stabil, 414
Penyimpanan Sekunder, 62, 74
Perangkat Keras, 74
Memori Sekunder, 268
Tabel Halaman, 268
Perangkat Lunak Bebas, 39
Perangkat M/K
Klasifikasi Umum, 374
Prinsip-Prinsip, 374
Perangkat Penyimpanan Tersier
Kerr Effect, 418
Magneto-Optic Disk, 417
MEMS, 422
Optical Disk, 418
Penyimpanan Holographic, 422
WORM, 419
Peranti, 65, 69
Persyaratan /usr
Direktori, 342
PLB, 39
Politisi Busuk, 32
Polling, 375
PPP, 429
Prioritas, 141
Procedural Programming, 170
Process Control Block, 105
Processor Synchronous, 183
Produsen Konsumen, 163
Proses, 61
Definisi, 103
Kooperatif
Kecepatan Komputasi, 163
Kenyamanan, 163
Modularitas, 163
Pembagian Informasi, 163
Pembentukan, 103
Sinkronisasi, 74
Prosesor Jamak, 141
Proteksi, 65
Proteksi Perangkat Keras
Dual Mode Operation
Monitor/Kernel/System Mode, 55
User Mode, 55
Masukan/Keluaran, 56
Proteksi CPU, 57
Proteksi Memori, 56

R

Race Condition
Solusi, 174
RAID
Level, 409
Struktur, 408
Redundansi, 408
Remote Registry, 171
RPC, 170
Ruang Alamat
Alamat Fisik, 233
Alamat Logika, 233
Memory Management Unit, 233
Register, 233

S

Scheduler
  CPU Scheduler, 125
  Job Scheduler, 125
  Medium-term Scheduler, 125
  Short-term Scheduler, 125
Segmentasi
  Offset, 257
  Segmen, 257
  Unit Lojik, 257
Segmentasi dengan Pemberian Halaman
  Pemberian Halaman, 259
Semafor, 186
Server, 168
  Klien, 322
Shadowing, 408
Signal, 187
Single-tasking, 69
Sinkronisasi
  Blocking, 167
  NonBlocking, 167
Sisi
  Sisi Permintaan, 207
Sistem
  Desain, 456
  Proteksi, 62
  Terdistribusi, 62
Sistem Berkas, 65
  Alokasi Blok, 349
    Berindeks, 354
    Berkesinambungan, 349
    Linked, 351
  Bit Map, 358
  Bit Vector, 358
  Blk_dev, 428
  Blk_dev_struct, 428
  chrdevs, 427
  Chrdevs, 428
  Grouping
    Counting, 359
  IDE, 428
  Inode VFS, 428
  Khusus Device, 427
  Nomor Device, 427
  Persyaratan
    Direktori, 340
  Root
    Operasi, 340
  SCSI, 428
  System Call, 427
  Virtual, 427
Sistem Operasi, 65
  Definisi, 31
  GNU/Linux, 31
  Komponen
    Control Program, 33
    Internet Explorer, 32
    Resource Allocator, 33
  Layanan, 65
  Tradisional, 33
  Tujuan, 373
    Kenyamanan, 33
  Windows, 31
Sistem Waktu Nyata
  Hard Real-Time, 141
  Soft Real-Time, 141
Skleton, 171
Sleep On, 153
SLIP, 429
SO, 31
Socket, 168
Spooling, 387
  Cara Kerja, 387
Spooling Directory, 387
Status, 66
Status Proses
  New, 104
  Ready, 104
  Runing, 104
  Terminated, 104
  Waiting, 104
Stream
  Pembagian, 390
Struktur
  Hard Disk, 49
  Optical Disc, 49
  Organisasi Komputer, 47
  Sistem Komputer
    Keluaran/Masukan (M/K), 50
    Operasi Sistem Komputer, 48
Struktur Berkas
  Layered File System, 329
Struktur Disk
  Constant Angular Velocity, 395
  Constant Linear Velocity, 395
  Waktu Akses, 395
Stub, 171
Sumber Daya, 65
Swap
  Lokasi Ruang, 407
  Manajemen Ruang, 407
  Penggunaan Ruang, 407
System
  Calls, 69, 71
  Program, 66
System CallLinux
  Clone
    Perbedaan Fork dan Clone, 155
  Fork, 155
System Generation, 75, 458

T

Tabel Halaman
  Masukan, 249
  Nomor Halaman, 249
  Tabel Halaman Secara Inverted , 249
  Tabel Halaman Secara Multilevel , 249
Tabel Registrasi, 99
Task Interruptible, 152
Teknologi Informasi dan Komunikasi, 39
Thrashing
  Halaman, 287
  Kesalahan Halaman, 287

Multiprogamming, 287
Swap-In, 287
Swap-Out, 287
Utilitasi CPU, 287
Thread
Definisi, 109
Kernel Thread, 110
Keuntungan
Berbagi Sumber Daya, 110
Ekonomi, 110
Responsif, 110
Utilisasi Arsitektur, 110
Multithreading, 109
Multithreading Models
Many-to-Many Model, 110
Many-to-One Model, 110
One-to-One Model, 110
Pembuatan, 117
Pthreads, 114
Specific Data, 114
Thread Pools, 114
User Thread, 110
TIK, 39

V

Vertex
Proses, 207

W

Wait, 187
Working Set
Bit Referensi, 287
Delta, 287
Fixed Internal Timer Interrupt, 287
Interupsi, 287
Lokalitas, 287
Overlap, 287