Pengantar Sistem Operasi Komputer

Jilid Kedua

Masyarakat Digital Gotong Royong (MDGR)

Pengantar Sistem Operasi Komputer: Jilid Kedua

oleh Masyarakat Digital Gotong Royong (MDGR)

Tanggal Publikasi $Date: 2008-08-29 14:59:08 $

Catatan Revisi

Revisi 4.59	29-Agustus-2008	RMS46
Fop 0.9X, perbaiki ukuran gambar.		
Revisi 4.57	04-Agustus-2008	RMS46
Merapihkan ulang; memperbaiki sana-sini; gabung kembali.		
Revisi 4.52	04-Februari-2008	RMS46
Mengedit ulang+membenah jilid 1 dan 2. Menambah Soal Ujian.		
Revisi 4.27	31-Agustus-2007	RMS46
Merapihkan dan memecah menjadi dua jilid.		
Revisi 4.22	07-Agustus-2007	RMS46
Daftar dalam Pengantar, Mengisi Kerangka, Soal Ap. B., Urut ulang.		
Revisi 4.16	03-Februari-2007	RMS46
Kerangka Baru.		
Revisi 4.8	28-Desember-2006	RMS46
Reset, start mengerjakan kerangka bab.		
Revisi 4.7	18-November-2006	RMS46
Pemulaian persiapan revisi 5.0 (kapan?).		
Revisi 4.00	28-Agustus-2006	RMS46
Menganggap selesai revisi 4.0.		
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, 150 Semester Genap 2005/2006, 152–157 dan 181–186 Semester Ganjil 2006/2007, 192–198 Semester Genap 2006/2007, 217 Semester Ganjil 2007/2008, Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia (http://rms46.vlsm.org/2/150.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.

Kelompok 152-157 (2006). Kelompok-kelompok tersebut mulai mengerjakan perbaikan versi 5.0. Nama-nama mereka ialah: Muhammad Ibnu Naslin (Bab 5, 11, 48), Iis Ansari (Bab 5, 11, 48), Agung Firmansyah (Bab 6, 29, 36), Arawinda D (Bab 19, 22, 30), Arudea Mahartianto (Bab 17, 20, 32), Chandra Prasetyo U (Bab 31, 36, 42), Charles Christian (Bab 16, 27, 38), Dyta Anggraeni (Bab 18, 33, 35), Hansel Tanuwijaya (Bab 8, 28, 39), Haryadi Herdian (Bab 12, 39, 46), Laverdy Pramula (Bab 14, 41, 46), Motti Getarinta (Bab 19, 25, 44), Muhammad Haris (Bab 24, 29, 42), Nulad Wisnu Pambudi (Bab 21, 37, 43), Ricky Suryadharma (Bab 13, 16, 40), Rizki Mardian (Bab 28, 41, 43), Siti Fuaida Fithri (Bab 23, 33, 34), Sugianto Angkasa (Bab 9, 15, 27), Teddy (Bab 15, 26, 37), Andrew Fiade (Bab 7, 45, 47), Della Maulidiya (Bab 7, 45, 47), Elly Matul Imah (Bab 7, 45, 47), Ida Bgs Md Mahendra (Bab 7, 45, 47), Ni Kt D Ari Jayanti (Bab 7, 45, 47), Wikan Pribadi (Bab 7, 45, 47).

Kelompok 181 (2006). Kelompok ini mengerjakan latihan soal dari Apendiks B: Angelina Novianti, Grahita Prajna Anggana, Haryoguno Ananggadipa, Muhammad Aryo N.P., Steven Wongso.

Kelompok 182 (2006). Kelompok ini mengerjakan latihan soal serta menuliskan jawaban dari Apendiks B: Adeline Halim Kesuma, Bonifatio Hartono, Maulahikmah Galinium, Selvia Ettine, Tania Puspita.

Kelompok 183 (2006). Kelompok ini mengerjakan latihan soal serta menuliskan jawaban dari Apendiks B: Bunga, Burhan, Danny Laidi, Arinal Gunawan, Prio Romano.

Kelompok 184 (2006). Kelompok ini mengerjakan latihan soal serta menuliskan jawaban dari Apendiks B: Arra'di Nur Rizal, Erlangga Muhammad Akbar, Pradana Atmadiputra, Stella Maria, Yanuar Rizky.

Kelompok 185 (2006). Kelompok ini mengerjakan latihan soal serta menuliskan jawaban dari Apendiks B: Christopher S, Edwin Ardhian, Gabriel F, Marcories, Nancy M H.

Kelompok 186 (2006). Kelompok ini mengerjakan latihan soal serta menuliskan jawaban dari Apendiks B: Kristina R. Setiawan, Maynard L. Benyamin, Melvin Rubianto, Varian S. Cahyadi, Victor L. Budiarta.

Bagian II: Konsep Dasar Sistem Operasi versi 5.0 (Kelompok 192, 2007). Bab 05 (Komponen Sistem Operasi) ditulis ulang oleh: Muhammad Ilman Akbar, Sagi Arsyad. Bab 06 (System Calls & Programs) ditulis ulang oleh: Adhitya Novian Raidy, Ananta Dian P. Bab 07 (Struktur Sistem Operasi) ditulis ulang oleh: Andre Tampubolon. Bab 08 (Mesin Virtual) ditulis ulang oleh: Achmad Rohman, Rizal Fahlevi, Aulia Fitri. Bab 09 (GNU/Linux) ditulis ulang oleh: Bayu Distiawan T, Octo Alexandro.

Bagian III: Proses dan Penjadwalan versi 5.0 (Kelompok 193, 2007). Bab 10 (Konsep Proses) ditulis ulang oleh: Bobby Alexander W, Refly H Hadiwijaya. Bab 11 (Konsep Thread) ditulis ulang

oleh: Yohanes Immanuel, Suviyanto. Bab 12 (Thread Java) ditulis ulang oleh: Annisa Ihsani. Bab 13 (Konsep Penjadwalan) ditulis ulang oleh: Moehamad Ichsan, Mulyandra Pratama, Erwanto D. Bab 14 (Algoritma Penjadwal) ditulis ulang oleh: Diandra Aditya K, Fitriany Nadjib. Bab 15 (Penjadwalan Prosesor Jamak) ditulis ulang oleh: Akhmad Mubarok, A Sobari. Bab 16 (Evaluasi Algoritma) ditulis ulang oleh: Heninggar S, Lia Sadita.

Bagian IV: Proses dan Sinkronisasi versi 5.0 (Kelompok 194, 2007). Bab 17 (Konsep Interaksi) ditulis ulang oleh: Hanif Rasyidi, Muhamad Wahyudin. Bab 18 (Sinkronisasi) ditulis ulang oleh: Purniawan, Yenni N. Bab 19 (Masalah Critical Section) ditulis ulang oleh: Niko Adi Nugroho. Bab 20 (Perangkat Sinkronisasi) ditulis ulang oleh: Danu Widatama, Abdul Muttaqien. Bab 21 (Transaksi Atomik) ditulis ulang oleh: Clara Vania, Bernadia Puspasari. Bab 22 (Sinkronisasi Linux) ditulis ulang oleh: Suryanto Ang. Bab 23 (Deadlock) ditulis ulang oleh: M. Sidik. Bab 24 (Diagram Graf) ditulis ulang oleh: Puspa Setia P. Bab 25 (Bounded Buffer) ditulis ulang oleh: Laksmita Rahadianti. Bab 26 (Readers/Writers) ditulis ulang oleh: Muchamad Irvan G. Bab 27 (Sinkronisasi Dua Arah) ditulis ulang oleh: Evi Dwi Jayanti, Istiana S.

Bagian V: Memori versi 5.0 (Kelompok 195, 2007). Bab 28 (Manajemen Memori) ditulis ulang oleh: Mursal Rais – Pita Larasati F N. Bab 29 (Alokasi Memori) ditulis ulang oleh: Novi Indriyani. Bab 30 (Pemberian Halaman) ditulis ulang oleh: Meirna Asti R, Leonny Pramitasari. Bab 31 (Arsitektur Intel Pentium) ditulis ulang oleh: Meldi Harrosyid. Bab 32 (Memori Virtual) ditulis ulang oleh: Rina Violyta, Metti Zakaria W. Bab 33 (Algoritma Ganti Halaman) ditulis ulang oleh: Renggo Pribadi, Kemal Nasir. Bab 34 (Strategi Alokasi Bingkai) ditulis ulang oleh: Vinky Halim, Armando Yonathan. Bab 35 (Seputar Alokasi Bingkai) ditulis ulang oleh: Nur Asyiah. Bab 36 (Memori Linux) ditulis ulang oleh: M Yudha A, Rizkiansyah Za, Anugrah Ramadhani.

Bagian VI: Masukan/Keluaran versi 5.0 (Kelompok 196, 2007). Bab 37 (Sistem M/K) ditulis ulang oleh: Tiara Mulia Putri, Imairi Eitiveni. Bab 38 (Subsistem M/K Kernel) ditulis ulang oleh: Anna Yatia Putri. Bab 39 (M/K Linux) ditulis ulang oleh: Reizki Permana.

Bagian VII: Penyimpanan Masal versi 5.0 (Kelompok 197, 2007). Bab 40 (Sistem Berkas) ditulis ulang oleh: Bambang Adhi, Darwin Cuputra. Bab 41 (Struktur Direktori) ditulis ulang oleh: Dian Seprina, Yans Sukma Pratama. Bab 42 (FHS) ditulis ulang oleh: Mustafa Kamal, Risnal Diansyah. Bab 43 (Implementasi Sistem Berkas) ditulis ulang oleh: Asa Ramdhani, Anita Rahmawati, Theresia Liberatha S. Bab 44 (Metoda Alokasi Blok) ditulis ulang oleh: Elisabeth Martha K, Mira Melissa. Bab 45 (Aneka Aspek Sistem Berkas) ditulis ulang oleh: Ginanjar Ck,Fandy Permana. Bab 46 (Media Disk) ditulis ulang oleh: Bambang Adhi. Bab 47 (Sistem Penyimpanan Masal) ditulis ulang oleh: Jusni S Jadera, Jan Sarbunan. Bab 48 (Sistem Berkas Linux) ditulis ulang oleh: Kukuh Setiadi, Rizal Mulyadi.

Bagian VIII: Topik Lanjutan versi 5.0 (Kelompok 198, 2007). Bab 49 (Keamanan Sistem) ditulis ulang oleh: Purwanto, Andi Muhammad Rijal. Bab 50 (Sistem Terdistribusi) ditulis ulang oleh: Suci Lestarini N. Bab 51 (Waktu Nyata dan Multimedia) ditulis ulang oleh: Prajna Wira Basnur. Bab 52 (Perancangan dan Pemeliharaan) ditulis ulang oleh: Sri Krisna Karunia, Hari Prasetyo.

Kelompok 217 (Semester Ganjil 2007/2008). Perbaikan lanjut dilakukan oleh: Hisma Mulya S (Bab 7/Buku I), Tieta Antaresti (Bab 8/Buku I), Hilda Deborah (Bab 10/Buku I), Lucia Roly P (Bab 11/Buku I), Phina Lidyawati (Bab 12/Buku I), Ivonne Margi I (Bab 13/Buku I), Irvan Ferdiansyah (Bab 14/Buku I), Ronny (Bab 15/Buku I), Dimas Rahmanto (Bab 16/Buku I), Pomona Angela K M (Bab 17/Buku I), Rosalina (Bab 18/Buku I), Indah Chandra (Bab 19/Buku I), Anita Kosasih (Bab 20/ Buku I), Yuli Biena (Bab 21/Buku I), Deni Lukmanul Hakim (Bab 22/Buku I), Abe Mitsu Teru (Bab 23/Buku I), Angga Kho Meidy (Bab 24/Buku I), Antonius Hendra (Bab 25/Buku I), Randy Oktavianus H (Bab 26/Buku I), Ramadhan K Sagala (Bab 27/Buku I), Lucky Haryadi (Bab 1/Buku II), Ivo Bahar Nugroho (Bab 2/Buku II), Ragil Ari Yuswito (Bab 3/Buku II), Anita Rahmawati (Bab 4/Buku II), Moehammad Radif M E (Bab 5/Buku II), Arip Mulyanto (Bab 6/Buku II), Pomona Angela K M (Bab 7/Buku II), Lucky Haryadi (Bab 8/Buku II), Phina Lidyawati (Bab 9/Buku II), Hilda Deborah (Bab 10/Buku II), Andrew Fiade (Bab 11/Buku II), Rosalina (Bab 13/Buku II), Irvan Ferdiansyah (Bab 14/ Buku II), Indah Chandra (Bab 15/Buku II), Randy Oktavianus H (Bab 16/Buku II), Tieta Antaresti (Bab 17/Buku II), Ramadhan K Sagala (Bab 18/Buku II), Andrew Fiade (Bab 19/Buku II), Ivo Bahar Nugroho (Bab 21/Buku II).

Daftar Isi

Daftar Gambar

Daftar Tabel

Daftar Contoh

Kata Pengantar

1. Calon Revisi 5.0 (Kapan?)

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

– DRAFT – BELUM TERBIT – DRAFT –

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/.

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.

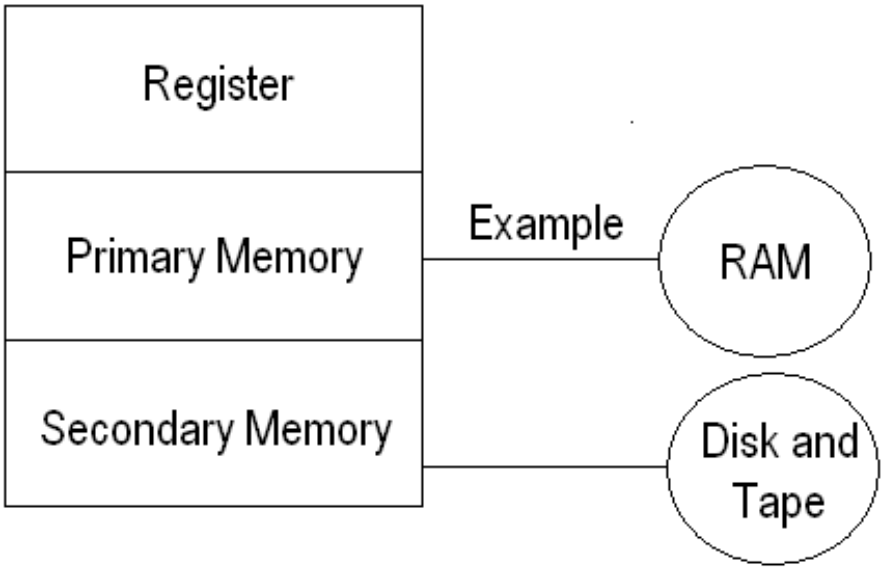
Bab 1. Konsep Dasar Memori

1.1. Pendahuluan

Memori merupakan bagian dari komputer yang berfungsi sebagai tempat penyimpanan informasi yang harus diatur dan dijaga sebaik-baiknya. Sebagian besar komputer memiliki hirarki memori yang terdiri atas tiga level, yaitu:

- *Register* di CPU, berada di level teratas. Informasi yang berada di register dapat diakses dalam satu clock cycle CPU.
- *Primary Memory (executable memory)*, berada di level tengah. Contohnya, RAM. *Primary Memory* diukur dengan satu byte dalam satu waktu, secara relatif dapat diakses dengan cepat, dan bersifat *volatile* (informasi bisa hilang ketika komputer dimatikan). CPU mengakses memori ini dengan instruksi *single load* dan *store* dalam beberapa clock cycle.
- *Secondary Memory*, berada di level bawah. Contohnya, disk atau tape. *Secondary Memory* diukur sebagai kumpulan dari bytes *(block of bytes)*, waktu aksesnya lambat, dan bersifat *non-volatile* (informasi tetap tersimpan ketika komputer dimatikan). Memori ini diterapkan di *storage device*, jadi akses meliputi aksi oleh *driver* dan *physical device*.

Gambar 1.1. Gambar Hirarki Memori



Komputer yang lebih canggih memiliki level yang lebih banyak pada sistem hirarki memorinya, yaitu *cache memory* dan bentuk lain dari *secondary memory* seperti *rotating magnetic memory*, *optical memory*, dan *sequntially access memory*. Akan tetapi, masing-masing level ini hanya sebuah penyempurnaan salah satu dari tiga level dasar yang telah dijelaskan sebelumnya.

Bagian dari sistem operasi yang mengatur hirarki memori disebut dengan *memory manager*. Di era *multiprogramming* ini, *memory manager* digunakan untuk mencegah satu proses dari penulisan dan pembacaan oleh proses lain yang dilokasikan di *primary memory*, mengatur *swapping* antara memori utama dan disk ketika memori utama terlalu kecil untuk memegang semua proses.

Tujuan dari manajemen ini adalah untuk:

- Meningkatkan utilitas CPU

- Data dan instruksi dapat diakses dengan cepat oleh CPU
- Efisiensi dalam pemakaian memori yang terbatas
- Transfer dari/ke memori utama ke/dari CPU dapat lebih efisien

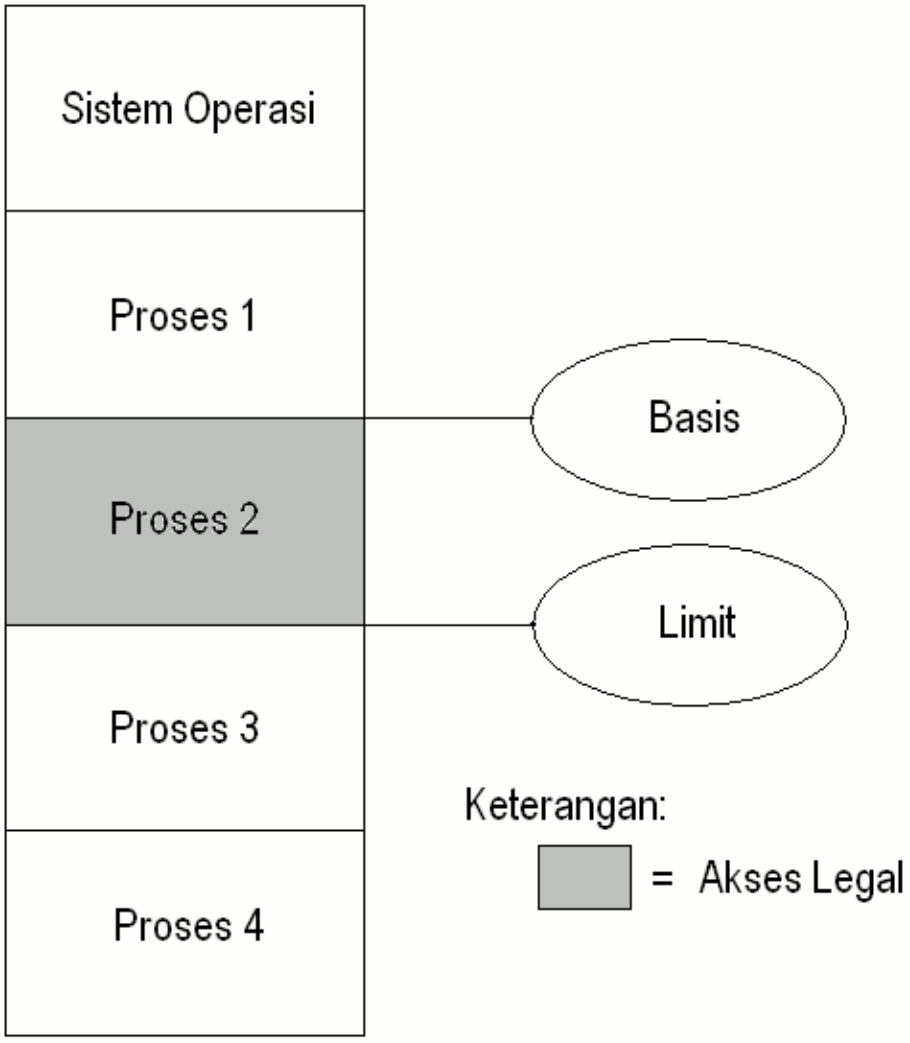
1.2. Proteksi Perangkat Keras

Pada saat suatu proses sedang berjalan, ada keadaan dimana *processor* berhenti. Hal ini menandakan tidak adanya lagi data yang diproses oleh *processor*. Oleh karena itu, *processor* pastinya akan mencari data ke dalam memori. Pengaksesan data tersebut memerlukan banyak *clock cycle*. Situasi ini tidak bisa ditoleransi sehingga membutuhkan perbaikan dalam kecepatan pengaksesan antara CPU dan memori utama. Tidak hanya peduli tentang kecepatan tersebut, tetapi juga memastikan operasi yang benar untuk melindungi pengaksesan sistem operasi dari proses lainnya, dan melindungi proses yang satu dari pengaksesan proses lainnya pula. Perlindungan atau proteksi ini disediakan oleh perangkat keras.

Kita harus memastikan bahwa masing-masing proses memiliki ruang memori yang terpisah. Caranya dengan menentukan jarak alamat yang dilegalkan dimana proses bisa mengakses dan memastikan bahwa proses tersebut hanya bisa mengakses pada alamat tersebut.

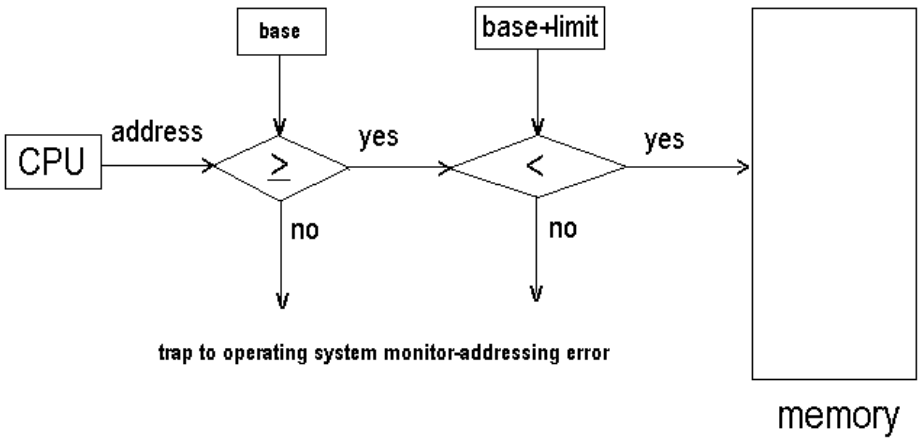
Proteksi di atas dilakukan oleh perangkat keras. Perangkat keras menyediakan dua register, yaitu *base register* dan *limit register*. *Base register* memegang alamat fisik terkecil yang dilegalkan, sedangkan *limit register* menentukan ukuran dari jarak alamat tersebut. Contohnya jika *base register* memegang 300040 dan *limit register* 120900, maka program bisa mengakses secara legal di semua alamat dari 300040 sampai 420940.

Gambar 1.2. Gambar *Base* dan Limit Register



Fungsi dari proteksi ini untuk mencegah *user program* dari kesengajaan memodifikasi kode/struktur data baik di sistem operasi atau *user* lainnya. Jika proteksi gagal, semua hal yang dilakukan oleh program executing di *user mode* untuk mengakses memori sistem operasi atau memori user lainnya akan terperangkap di sistem operasi dan bisa menyebabkan kesalahan yang fatal, yaitu *addressing error*.

Gambar 1.3. Gambar Proteksi Perangkat Keras dengan *base* dan *limit register*



1.3. *Address Binding*

Pengertian *address binding* adalah sebuah prosedur untuk menetapkan alamat fisik yang akan digunakan oleh program yang terdapat di dalam memori utama. *Address binding* yang dilakukan terhadap suatu program dapat dilakukan di 3 tahap yang berbeda, yaitu:

- **Compilation time**. Pada tahap ini sebuah program pada awalnya akan menghasilkan alamat berupa simbol-simbol, kemudian simbol-simbol ini akan langsung diubah menjadi alamat absolut atau alamat fisik yang bersifat statik. Bila suatu saat terjadi pergeseran alamat dari program tersebut maka untuk mengembalikan ke alamat yang seharusnya dapat dilakukan kompilasi ulang. Contoh : file bertipe .com yang merupakan hasil dari kompilasi program

- **Load time**. Pada tahap ini awalnya program menghasilkan alamat berupa simbol-simbol yang sifatnya acak (*relative address*), kemudian akan dilakukan penghitungan ulang agar program tersebut ditempatkan pada alamat yang dapat dialokasikan ulang (*relocateble address*). Singkatnya binding terjadi pada waktu program telah selesai di- *load*. Contoh: File bertipe .exe.

- **Execution time**. Alamat bersifat relatif, binding akan dilakukan pada saat *run time*. Pada saat run time dibutuhkan bantuan hardware yaitu MMU (*Memory Management Unit*).

1.4. Ruang Alamat Logika dan Fisik

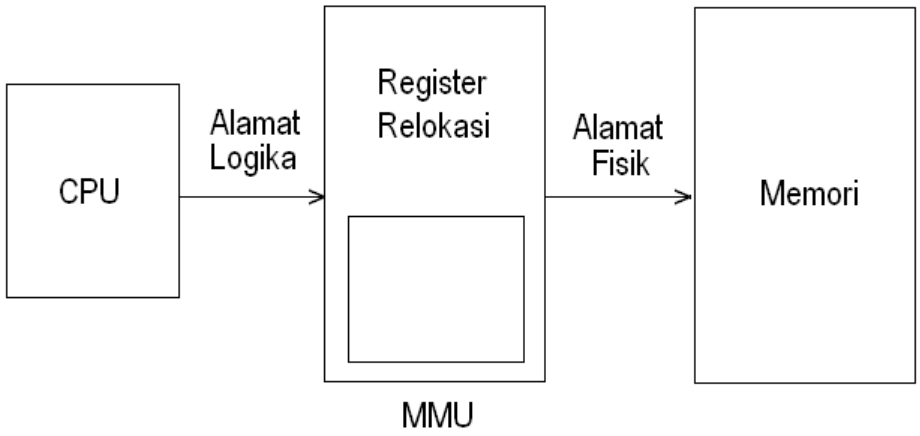
Alamat yang dihasilkan oleh CPU berupa alamat logika, sedangkan yang masuk ke dalam memori adalah alamat fisik. Pada *compile time* dan *load time*, alamat fisik dan logika identik. Sebaliknya, perbedaan alamat fisik dan logika terjadi pada *execution time*.

Kumpulan semua alamat logika yang dihasilkan oleh program adalah ruang alamat logika/ruang alamat virtual. Kumpulan semua alamat fisik yang berkorespondensi dengan alamat logika disebut ruang alamat fisik.

Pada saat program berada di CPU, program tersebut memiliki alamat logika, kemudian oleh MMU dipetakan menjadi alamat fisik yang akan disimpan di dalam memori.

Ilustrasinya sebagai berikut, nilai pada register ini akan ditambah dengan setiap alamat yang dibuat oleh *user process* yang kemudian dikirim ke memori. Contohnya register relokasi berada di 14000, alamat logika di 346, maka langsung dipetakan menjadi alamat fisik di 14346.

Gambar 1.4. Gambar Relokasi Dinamis dengan Menggunakan *Relocation* Register



1.5. Pemuatan Dinamis

Ukuran dari memori fisik terbatas. Supaya utilitas memori berjalan dengan baik, maka kita menggunakan pemuatan dinamis. Dengan cara ini, *routine-routine* hanya akan dipanggil jika dibutuhkan.

Ilustrasi sebagai berikut, semua *routine* disimpan di disk dalam format yang dapat dialokasikan ulang (*relocatable load format*). Program utama diletakkan di memori dan dieksekusi. Ketika sebuah *routine* memanggil *routine* yang lain, hal pertama yang dilakukan adalah mengecek apakah ada *routine* lain yang sudah di-*load*. Jika tidak, *relocatable linking loader* dipanggil untuk menempatkan *routine* yang dibutuhkan ke memori dan memperbaharui tabel alamat program. Lalu, kontrol diberikan pada *routine* baru yang dipanggil.

Keuntungan dari pemuatan dinamis adalah *routine* yang tidak digunakan tidak pernah dipanggil. Metode ini berguna pada kode yang berjumlah banyak, ketika muncul kasus seperti *routine* yang salah. Walaupun ukuran kode besar , porsi yang digunakan bisa jauh lebih kecil.

Sistem operasi tidak membuat mekanisme pemuatan dinamis, tetapi hanya menyediakan *routine-routine* untuk menerapkan mekanisme ini. *User*-lah yang merancang programnya sendiri agar programnya menggunakan sistem pemuatan dinamis.

1.6. Linking Dinamis

Pustaka bisa bersifat statik, dikenal dengan *archive* yang terdiri dari kumpulan *routine* yang diduplikasi ke sebuah program oleh *compiler, linker, atau binder*, sehingga menghasilkan sebuah aplikasi yang dapat dieksekusi (bersifat *stand alone* atau dapat berjalan sendiri). *Compiler* menyediakan *standard libraries*, misalnya C standard library, tetapi *programmer* bisa juga membuat pustakanya untuk digunakan sendiri atau disebarkan. Pustaka statis ini menyebabkan memori menjadi berat. Oleh karena itu, seiring dengan perkembangan teknologi, terdapat pustaka yang bersifat dinamis. Mekanismenya disebut linking dinamis, sedangkan pustakanya disebut *dynamically linked library*.

Linking Dinamis artinya data (kode) di pustaka tidak diduplikasi ke dalam program pada *compile time*, tapi tinggal di file terpisah di disk *Linker* hanya membutuhkan kerja sedikit pada *compile time*. Fungsi *linker* adalah mencatat apa yang dibutuhkan oleh pustaka untuk eksekusi dan nama indeks atau nomor. Kerja yang berat dari linking akan selesai pada *load time* atau selama *run time*. Kode penghubung yang diperlukan adalah *loader*. Pada waktu yang tepat, *loader* menemukan pustaka yang relevan di disk dan menambahkan data dari pustaka ke proses yang ada di ruang memori. Keuntungan dari linking dinamis adalah memori program tidak menjadi berat.

1.7. Pustaka Bersama

Satu pustaka dipakai bersama-sama oleh banyak program pada waktu yang bersamaan. Sekumpulan data dapat diperbaharui versinya dan semua program yang menggunakan pustaka tersebut secara otomatis menggunakan versi baru. Metode yang dipakai adalah linking dinamis. Tanpa adanya metode ini, semua program akan melakukan proses linking ulang untuk dapat mengakses pustaka yang baru, sehingga program tidak bisa langsung mengeksekusi yang baru, informasi versi terdapat di program dan pustaka. Lebih dari satu versi dari pustaka bisa masuk ke memori sehingga setiap program menggunakan informasi versinya untuk memutuskan versi mana yang akan digunakan dari salinan pustaka.

1.8. Rangkuman

Memori merupakan sumber daya yang paling penting untuk dijaga sebaik-baiknya karena merupakan pusat dari kegiatan di komputer. Terdapat proteksi perangkat keras yang dilakukan dengan menggunakan dua register, yaitu *base* dan *limit register* sehingga proses hanya bisa mengakses di alamat yang dilegalkan. Alamat yang dihasilkan oleh CPU disebut alamat logika yang kemudian dipetakan oleh MMU menjadi alamat fisik yang disimpan di memori. Untuk mendapatkan utilitas memori yang baik, maka diperlukan metode pemuatan dinamis, linking dinamis, dan pustaka bersama.

Rujukan

[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.

[Wikipedia2007] Anonim. 2007. *Shared Library – http://en.wikipedia.org/wiki/Shared_library* . Diakses 16 maret 2007.

[KUR2003] David A.S, Habib A.M, dan Endah W. 2003. *Makalah IF3191 Sistem Operasi: Manajemen Memori– kur2003.if.itb.ac.id/file/FKML-K1-07.pdf* . Diakses 16 maret 2007.

[Suny2005] Suny. 2005. *Address Binding– www.cs.binghamton.edu/~nael/classes/cs350/notes/4-lec15.pdf* . Diakses 16 maret 2007.

Bab 2. Alokasi Memori

2.1. Pendahuluan

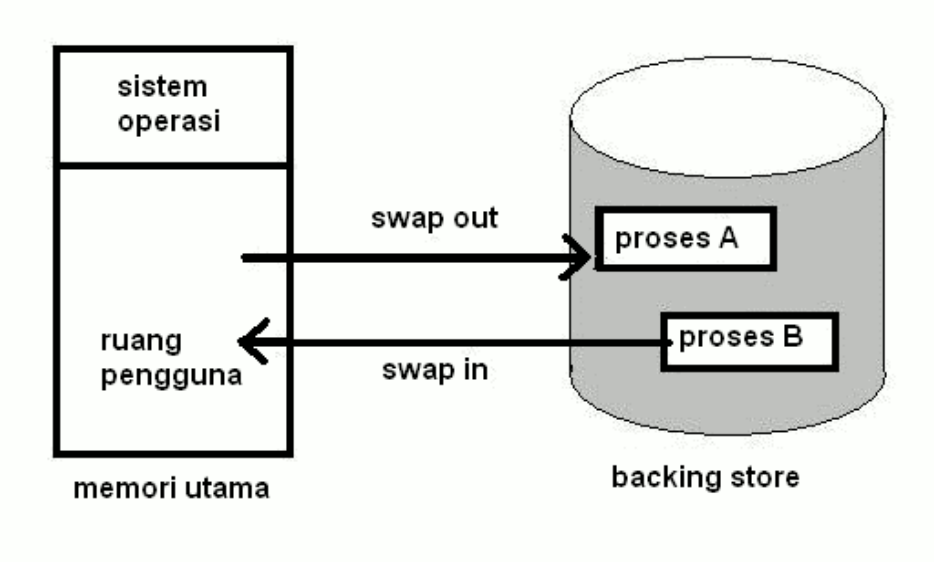
Memori merupakan salah satu sumber daya yang penting dalam pengeksekusian sebuah proses. Memori terdiri dari *array word* atau *byte* yang masing-masing memiliki alamat. Suatu proses dapat dieksekusi bila ia telah berada dalam memori sebelum CPU mengambil instruksi-instruksi pada alamat yang ditunjuk oleh *program counter*.

Bagian dari sistem yang bertugas untuk mengatur memori disebut *memory manager*. *Memory manager* mengatur bagian mana dari memori yang harus digunakan dan mana yang tidak pada suatu waktu, selain itu *memory manager* juga mengalokasikan memori untuk proses-proses yang membutuhkannya serta men-dealokasikannya kembali saat proses-proses tersebut tidak lagi membutuhkannya. Masalahnya adalah bagaimana jika memori tidak lagi cukup untuk menampung semua proses yang akan dieksekusi? Solusi untuk masalah ini adalah dengan teknik pemindahan proses dari memori ke dalam disk dan kembali memindahkannya ke memori pada saat hendak dieksekusi lagi atau yang lebih dikenal dengan istilah *swapping*.

2.2. *Swap*

Sebuah proses, sebagaimana telah diterangkan di atas, harus berada di memori sebelum dieksekusi. Proses *swapping* menukarkan sebuah proses keluar dari memori untuk sementara waktu ke sebuah penyimpanan sementara dengan sebuah proses lain yang sedang membutuhkan sejumlah alokasi memori untuk dieksekusi. Tempat penyimpanan sementara ini biasanya berupa sebuah *fast disk* dengan kapasitas yang dapat menampung semua salinan dari semua gambaran memori serta menyediakan akses langsung ke gambaran tersebut. Jika eksekusi proses yang dikeluarkan tadi akan dilanjutkan beberapa saat kemudian, maka ia akan dibawa kembali ke memori dari tempat penyimpanan sementara tadi. Bagaimana sistem mengetahui proses mana saja yang akan dieksekusi? Hal ini dapat dilakukan dengan *ready queue*. *Ready queue* berisikan semua proses yang terletak baik di penyimpanan sementara maupun memori yang siap untuk dieksekusi. Ketika penjadwal CPU akan mengeksekusi sebuah proses, ia lalu memeriksa apakah proses bersangkutan sudah ada di memori ataukah masih berada dalam penyimpanan sementara. Jika proses tersebut belum berada di memori maka proses *swapping* akan dilakukan seperti yang telah dijelaskan di atas.

Gambar 2.1. Proses *Swapping*



Sebuah contoh untuk menggambarkan teknik *swapping* ini adalah sebagai berikut: Algoritma *Round-Robin* yang digunakan pada *multiprogramming environment* menggunakan waktu kuantum (satuan

waktu CPU) dalam pengeksekusian proses-prosesnya. Ketika waktu kuantum berakhir, *memory manager* akan mengeluarkan (*swap out*) proses yang telah selesai menjalani waktu kuantumnya pada suatu saat serta memasukkan (*swap in*) proses lain ke dalam memori yang telah bebas tersebut. Pada saat yang bersamaan penjadwal CPU akan mengalokasikan waktu untuk proses lain dalam memori. Hal yang menjadi perhatian adalah, waktu kuantum harus cukup lama sehingga waktu penggunaan CPU dapat lebih optimal jika dibandingkan dengan proses penukaran yang terjadi antara memori dan disk.

Teknik *swapping roll out, roll in* menggunakan algoritma berbasis prioritas dimana ketika proses dengan prioritas lebih tinggi tiba maka *memory manager* akan mengeluarkan proses dengan prioritas yang lebih rendah serta me-*load* proses dengan prioritas yang lebih tinggi tersebut. Saat proses dengan prioritas yang lebih tinggi telah selesai dieksekusi maka proses yang memiliki prioritas lebih rendah dapat dimasukkan kembali ke dalam memori dan kembali dieksekusi.

Sebagian besar waktu *swapping* adalah waktu transfer. Sebagai contoh kita lihat ilustrasi berikut ini: sebuah proses pengguna memiliki ukuran 5 MB, sedangkan tempat penyimpanan sementara yang berupa *harddisk* memiliki kecepatan transfer data sebesar 20 MB per detiknya. Maka waktu yang dibutuhkan untuk mentransfer proses sebesar 5 MB tersebut dari atau ke dalam memori adalah sebesar 5000 KB / 20000 KBps = 250 ms

Perhitungan di atas belum termasuk waktu latensi, sehingga jika kita asumsikan waktu latensi sebesar 2 ms maka waktu *swap* adalah sebesar 252 ms. Oleh karena terdapat dua kejadian dimana satu adalah proses pengeluaran sebuah proses dan satu lagi adalah proses pemasukan proses ke dalam memori, maka total waktu *swap* menjadi 252 + 252 = 504 ms.

Agar teknik *swapping* dapat lebih efisien, sebaiknya proses-proses yang di- *swap* hanyalah proses-proses yang benar-benar dibutuhkan sehingga dapat mengurangi waktu *swap*. Oleh karena itulah, sistem harus selalu mengetahui perubahan apapun yang terjadi pada pemenuhan kebutuhan terhadap memori. Disinilah sebuah proses memerlukan fungsi *system call*, yaitu untuk memberitahukan sistem operasi kapan ia meminta memori dan kapan membebaskan ruang memori tersebut.

Jika kita hendak melakukan *swap*, ada beberapa hal yang harus diperhatikan. Kita harus menghindari menukar proses dengan M/K yang ditunda (asumsinya operasi M/K tersebut juga sedang mengantri di antrian karena peralatan M/Knya sedang sibuk). Contohnya seperti ini, jika proses P1dikeluarkan dari memori dan kita hendak memasukkan proses P2, maka operasi M/K yang juga berada di antrian akan mengambil jatah ruang memori yang dibebaskan P1 tersebut. Masalah ini dapat diatasi jika kita tidak melakukan *swap* dengan operasi M/K yang ditunda. Selain itu, pengeksekusian operasi M/K hendaknya dilakukan pada *buffer* sistem operasi.

Tiap sistem operasi memiliki versi masing-masing pada teknik *swapping* yang digunakannya. Sebagai contoh pada UNIX, *swapping* pada dasarnya tidak diaktifkan, namun akan dimulai jika banyak proses yang membutuhkan alokasi memori yang banyak. *Swapping* akan dinonaktifkan kembali jika jumlah proses yang dimasukkan berkurang. Pada sistem operasi *Microsoft Windows 3.1*, jika sebuah proses baru dimasukkan dan ternyata tidak ada cukup ruang di memori untuk menampungnya, proses yang lebih dulu ada di memori akan dipindahkan ke disk. Sistem operasi ini pada dasarnya tidak menerapkan teknik *swapping* secara penuh, hal ini disebabkan pengguna lebih berperan dalam menentukan proses mana yang akan ditukar daripada penjadwal CPU. Dengan ketentuan seperti ini proses-proses yang telah dikeluarkan tidak akan kembali lagi ke memori hingga pengguna memilih proses tersebut untuk dijalankan.

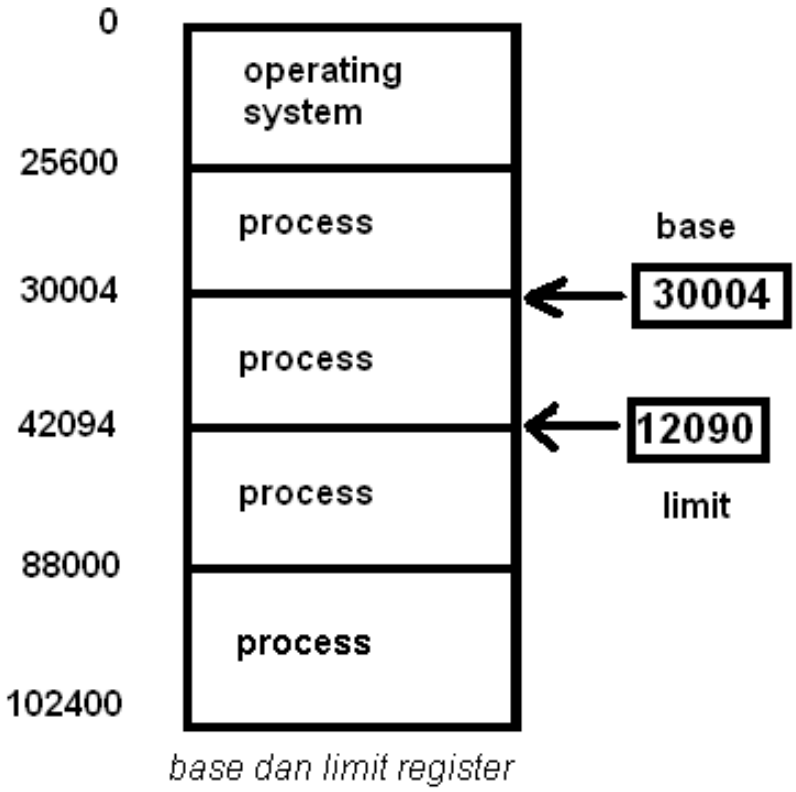
2.3. Pemetaan Memori

Sistem operasi dan berbagai proses pengguna terletak di dalam memori utama. Oleh karena itu, kita harus menjaga agar proses diantara keduanya tidak bercampur dengan cara mengalokasikan sejumlah bagian memori tersebut untuk sistem operasi dan proses pengguna. Memori ini biasanya dibagi menjadi 2 bagian. Satu untuk sistem operasi, dan satu lagi untuk proses pengguna.

Pemetaan memori (*memory mapping*) membutuhkan sebuah register relokasi. Sebagaimana yang telah dijelaskan pada bab sebelumnya, register relokasi merupakan *base register* yang ditambahkan ke

dalam tiap alamat proses pengguna pada saat dikirimkan ke memori. Pada pemetaan memori ini terdapat *limit register* yang terdiri dari rentang nilai alamat logika . Dengan adanya *limit register* dan register relokasi, tiap alamat logis haruslah lebih kecil dari *limit register*. Proses pemetaan dilakukan oleh MMU (*Memory Management Unit*) dengan menjumlahkan nilai register relokasi ke alamat logis. Alamat yang telah dipetakan ini lalu dikirim ke memori. Pada gambar dibawah ini kita dapat melihat bahwa sebuah proses yang memiliki *base register* 30004 dan *limit register* 12090 akan dipetakan ke memori fisik dengan alamat awalnya sesuai dengan *base register* (30004) dan berakhir pada alamat (30004 + 12090 = 42094).

Gambar 2.2. *Base dan Limit Register*



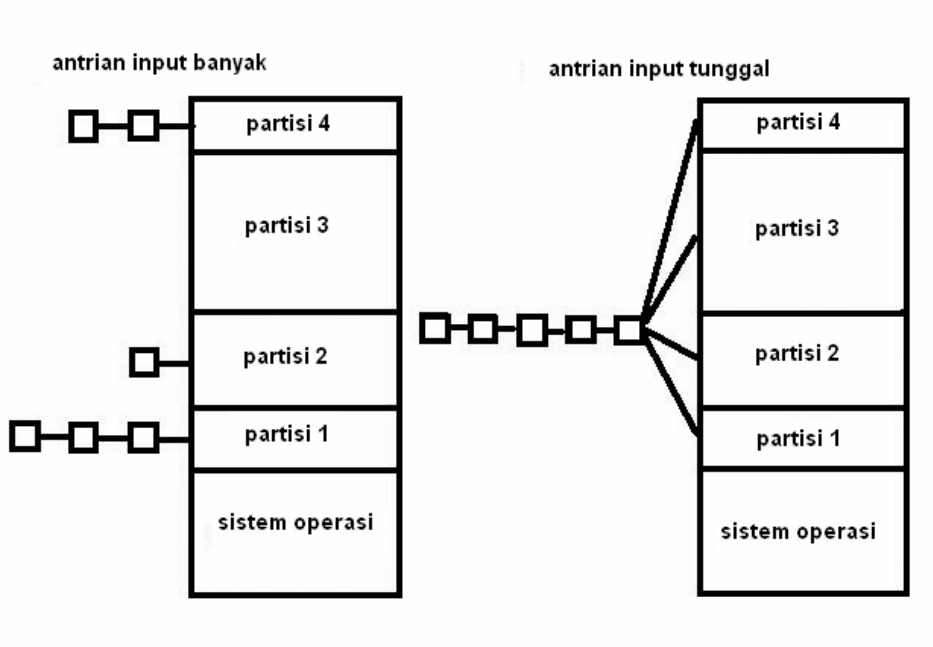
Ketika penjadwal CPU memilih suatu proses untuk dieksekusi, ia akan memasukkan register relokasi dan juga *limit register* -nya. Register relokasi memungkinkan sistem operasi untuk merubah ukuran partisinya pada memori secara dinamis. Contohnya, kode dan *buffer* yang dialokasikan untuk driver peralatan pada sistem operasi dapat dihapus dari memori jika peralatan tersebut jarang digunakan. Kode semacam ini disebut kode sistem operasi yang *transient*, oleh karena kode ini dapat "datang dan pergi" dari memori tergantung kapan ia dibutuhkan. Sehingga penggunaan kode *transient* ini dapat merubah ukuran sistem operasi selama eksekusi program berlangsung.

2.4. Partisi Memori

Memori harus di diatur agar penempatan proses-proses tersebut dapat tersusun dengan baik. Hal tersebut berkaitan dengan banyaknya jumlah proses yang berada di memori pada suatu saat/waktu. Cara yang paling mudah adalah dengan membagi memori ke dalam beberapa partisi dengan ukuran yang tetap. Cara ini memungkinkan pembagian yang tidak sama rata. Tiap partisi dapat terdiri dari hanya satu buah proses. Sehingga derajat *multiprogramming*-nya dibatasi oleh jumlah partisi tersebut.

Gambar 2.3 mengilustrasikan *multiprogramming* dengan partisi memori yang tetap.

Gambar 2.3. Proses *Partisi Memori Tetap*



Ketika sebuah proses datang, ia akan diletakkan ke dalam *input queue* (antrian proses pada disk yang menunggu dibawa ke memori untuk dieksekusi) sesuai dengan ukuran terkecil partisi yang mampu menampungnya. Kerugian dari mengurutkan proses ke dalam antrian yang berbeda berdasarkan ukurannya muncul ketika partisi yang besar akan menjadi kosong karena tidak ada proses dengan ukuran sesuai yang diletakkan di partisi tersebut. Namun di lain sisi, antrian untuk partisi dengan ukuran kecil sangat padat karena banyaknya proses dengan ukuran yang sesuai. Cara alternatif yang dapat dilakukan adalah dengan membuat sebuah antrian tunggal seperti terlihat pada gambar diatas. Ketika sebuah partisi bebas, proses dengan ukuran sesuai partisi tersebut yang terletak di depan antrian dapat dimasukkan lalu dieksekusi. Namun metode ini memiliki kelemahan, yaitu bagaimana jika proses yang memasuki partisi yang cukup besar ternyata ukurannya jauh lebih kecil dari partisi itu sendiri? Masalah ini dapat diatasi dengan mencari proses terbesar ke dalam seluruh antrian yang dapat ditampung oleh sebuah partisi pada saat itu. Namun algoritma ini mendiskriminasikan proses yang kecil karena proses yang diambil adalah proses terbesar yang dapat dimuat ke dalam partisi yang sedang bebas saat itu.

Dalam partisi tetap ini, sistem operasi menggunakan sebuah tabel untuk mengindikasikan bagian memori mana yang kosong dan mana yang terisi. Pada awalnya semua partisi kosong dan dianggap sebagai sebuah blok besar yang tersedia (*hole*). Ketika sebuah proses datang dan membutuhkan memori, ia akan dicarikan lubang yang cukup besar yang mampu menampungnya. Setelah menemukannya, memori yang dialokasikan untuknya hanyalah sebesar memori yang dibutuhkannya sehingga menyisakan tempat untuk memenuhi kebutuhan proses lain. Sistem operasi mencatat kebutuhan memori masing-masing proses yang berada dalam antrian serta jumlah memori yang masih tersedia untuk menentukan proses mana yang harus dimasukkan. Sistem akan memiliki sebuah daftar yang berisi ukuran blok yang masih tersedia serta antrian masukan proses. Sistem operasi dapat mengurutkan antrian input tersebut berdasarkan algoritma penjadwalan. Memori dialokasikan pada proses yang ukurannya sesuai hingga akhirnya kebutuhan memori untuk proses berikutnya tidak dapat dipenuhi karena tidak ada lagi blok yang cukup untuknya. Sistem operasi akan menunggu hingga blok yang cukup besar untuk menampung proses tersebut tersedia atau sistem operasi dapat juga melewati proses tersebut dan mencari jikalau ada proses dengan kebutuhan memori yang dapat ditampung oleh blok memori yang tersedia. Pada kenyatannya, partisi tetap kurang mengoptimalkan memori sebagai sumber daya yang penting karena seringkali terjadi, partisi yang cukup besar dialokasikan untuk proses dengan ukuran yang lebih kecil sehingga sisa dari partisi tersebut tidak digunakan.

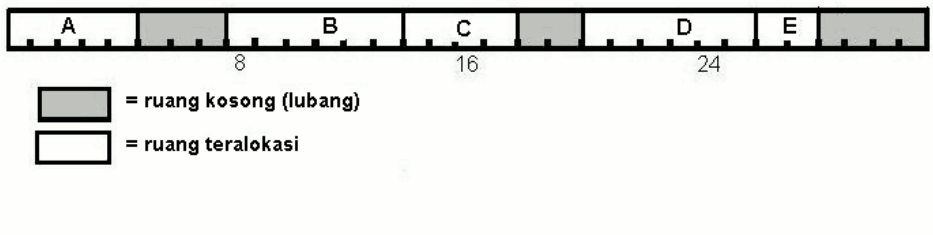
Pada alokasi penyimpanan dinamis, kumpulan lubang-lubang (ruang memori kosong) dalam berbagai ukuran tersebar di seluruh memori sepanjang waktu. Apabila ada proses yang datang, sistem operasi

akan mencari lubang yang cukup besar untuk menampung memori tersebut. Apabila lubang yang tersedia terlalu besar, maka ia akan dipecah menjadi 2. Satu bagian digunakan untuk menampung proses tersebut sedangkan bagian lain akan digunakan untuk bersiap-siap menampung proses lain. Setelah proses tersebut selesai menggunakan alokasi memorinya, ia akan melepaskan ruang memori tersebut dan mengembalikannya sebagai lubang-lubang kembali. Apabila ada 2 lubang yang berdekatan, keduanya akan bergabung untuk membentuk lubang yang lebih besar. Pada saat itu, sistem harus memeriksa apakah ada proses dalam antrian yang dapat dimasukkan ke dalam ruang memori yang baru terbentuk tersebut. Isu utama dari alokasi penyimpanan dinamis adalah bagaimana memenuhi permintaan proses berukuran n dengan kumpulan lubang-lubang yang tersedia. Ada beberapa solusi untuk masalah ini:

1. **First Fit.** *Memory manager* akan mencari sepanjang daftar yang berisi besarnya ukuran memori yang dibutuhkan oleh proses dalam antrian beserta ukuran memori yang tersedia pada saat itu. Setelah menemukan lubang yang cukup besar (ruang memori dengan ukuran lebih besar dari ukuran yang dibutuhkan oleh proses bersangkutan), lubang itu lalu dipecah menjadi 2 bagian. Satu bagian untuk proses tersebut dan bagian lain digunakan untuk memori yang tak terpakai, kecuali tentu saja jika memang ukuran ruang memori tersebut sama besar dengan yang dibutuhkan oleh proses. *First fit* ini merupakan algoritma yang bekerja dengan cepat karena proses pencariannya dilakukan sesedikit mungkin

2. **Next Fit.** Algoritma ini hampir sama persis dengan *first fit*, kecuali *next fit* meneruskan proses pencarian terhadap lubang yang cukup besar untuk sebuah proses mulai dari lubang sebelumnya yang telah sesuai dengan proses sebelumnya. Pendek kata, algoritma ini tidak memulai pencarian dari awal. Gambar di bawah ini mengilustrasikan sebuah contoh yang membedakan antara *first fit* dan *next fit*. Jika blok berukuran 2 dibutuhkan maka *first fit* akan memilih lubang pada alamat 5, namun *next fit* akan memilih lubang pada 18.

Gambar 2.4. Bagian Memori dengan 5 Proses dan 3 Lubang



3. **Best Fit.** *Best fit* mencari dari keseluruhan daftar (kecuali jika daftar tersebut telah terurut berdasarkan ukuran), dan memilih lubang terkecil yang cukup untuk menampung proses yang bersangkutan. Daripada harus memecah sebuah lubang besar, yang mungkin saja dapat lebih bermanfaat nantinya, *best fit* mencari lubang dengan ukuran yang hampir sama dengan yang dibutuhkan oleh proses. Strategi ini menghasilkan sisa lubang terkecil. Kekurangan *best fit* jika dibandingkan dengan *first fit* adalah lebih lambat karena harus mencari ke seluruh tabel tiap kali dipanggil. Berdasarkan gambar diatas jika blok berukuran 2 dibutuhkan maka berdasarkan *best fit* akan memilih lubang pada alamat 18 yaitu lubang terkecil yang cukup menampung permintaan proses tersebut.

4. **Worst Fit.** *Worst fit* akan mencari lubang terbesar. Sebagaimana *best fit* kita harus mencari dari keseluruhan daftar kecuali jika daftar tersebut telah terurut berdasarkan ukuran. Strategi ini menghasilkan sisa lubang terbesar. Berdasarkan gambar diatas jika blok berukuran 2 dibutuhkan maka berdasarkan *worst fit* akan memilih lubang pada alamat 28 yaitu lubang terbesar yang cukup menampung permintaan proses tersebut.

2.5. Fragmentasi

Fragmentasi merupakan fenomena munculnya lubang-lubang (ruang memori kosong) yang tidak cukup besar untuk menampung permintaan alokasi memori dari proses. Fragmentasi terdiri dari dua jenis:

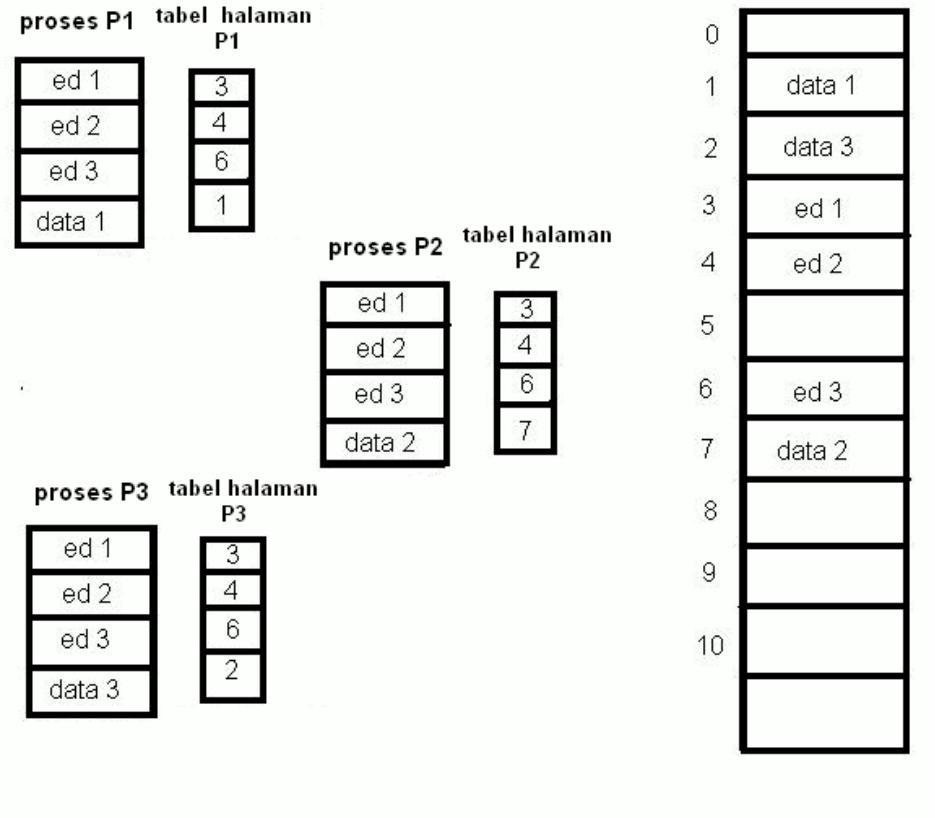
1. **Fragmentasi Eksternal.** Dalam kasus *first fit* dan juga *best fit* sebagaimana yang telah dijelaskan di atas, pada saat proses dimasukkan atau dipindahkan dari memori, ruang memori yang tidak terpakai akan dipecah menjadi bagian yang kecil (sisa dari alokasi sebuah proses pada sebuah ruang memori). Eksternal fragmentasi terjadi apabila jumlah keseluruhan memori bebas yang tersedia cukup untuk menampung permintaan ruang memori dari sebuah proses, namun dari ruang memori kosong tersebut terpisah-pisah sehingga proses tidak dapat menggunakannya. Hal ini sering terjadi pada alokasi penyimpanan yang dinamis. Sebagai contoh kita lihat contoh berikut ini: Sebuah proses meminta ruang memori sebesar 9 KB namun memori telah dipartisi menjadi blok-blok dengan ukuran masing-masing 4 KB. Maka proses tersebut akan mendapatkan bagiannya berupa 2 buah blok dengan kapasitas masing-masing 4 KB dan kapasitas tambahan sebesar 1 KB dari sebuah blok lain. Oleh karena masing-masing blok memiliki ukuran 4 KB dan ada sebuah blok yang hanya digunakan sebesar 1 KB maka blok ini masih akan memiliki sisa kapasitas sebesar 3 KB. Sisa tersebut dapat digunakan untuk menampung proses lain yang membutuhkannya atau jika ia terletak berurutan dengan sebuah blok kosong lain maka ia dapat digabungkan membentuk blok bebas yang lebih besar. Analisis statistik terhadap *first fit* menyatakan bahwa walaupun dengan optimisasi, sejumlah N blok yang dialokasikan maka setengahnya akan terbuang atau tidak berguna karena fragmentasi yang menyebabkan lebih dari setengah memori tidak dapat digunakan. Peristiwa ini disebut dengan *50-percent rule* (aturan 50 persen). Masalah fragmentasi eksternal ini dapat diatasi dengan melakukan penghalaman, segmentasi (2 hal ini akan dijelaskan secara detail pada bab lain) serta *compaction* (pemadatan). Tujuan dari pemadatan adalah untuk mengatur ruang memori yang kosong agar terletak di posisi yang berurutan sehingga dapat membentuk sebuah ruang memori kosong yang besar. Ruang kosong itu pada akhirnya diharapkan dapat menampung proses lain yang membutuhkan alokasi memori.

2. **Fragmentasi Internal.** Fragmentasi internal terjadi ketika kapasitas memori yang diberikan ke sebuah proses melebihi besarnya permintaan yang diajukan oleh proses. Selisih antara besarnya memori yang dialokasikan dengan besarnya permintaan proses disebut fragmentasi internal (memori yang ada di dalam sebuah partisi namun tidak digunakan). Hal ini sering terjadi pada partisi tetap karena besar lubang yang disediakan akan selalu tetap, berbeda halnya dengan sistem partisi dinamis yang memungkinkan suatu proses untuk diberikan alokasi memori sebesar yang ia butuhkan. Contoh solusi atas kasus diatas dengan fragmentasi internal adalah proses tersebut akan dialokasikan 3 buah blok yang masing-masing berukuran 4 KB sehingga ia akan mendapatkan jatah sebesar 12 KB, sisa 3 KB yang ada akan tetap menjadi miliknya walaupun ia tidak menggunakannya.

2.6. Berbagi Memori

Berbagi halaman atau berbagi memori merupakan salah satu teknik yang dapat digunakan untuk menghemat pengalokasian memori. Keuntungan yang dapat diperoleh dari teknik berbagi halaman ini adalah suatu kode dapat digunakan secara bersama-sama. Hal ini sangatlah penting dalam kondisi berbagi waktu (*time-sharing environment*). Bayangkan jika sebuah sistem harus menangani 40 pengguna. Masing-masing dari pengguna tersebut menggunakan sebuah *text editor*. Jika *text editor* tersebut terdiri dari 150 KB kode dan 50 KB data maka ruang memori yang dibutuhkan adalah 8000 KB. Jika *text editor* tersebut adalah kode *reentrant* (*programming routine* yang dapat digunakan oleh banyak program secara simultan) maka ia dapat digunakan secara bersama-sama oleh beberapa program (dapat dibagi). Ilustrasi berbagi halaman ini dapat dilihat pada gambar berikut ini:

Gambar 2.5. Contoh Berbagi Halaman



Kode *reentrant* dapat dieksekusi oleh 2 atau lebih proses dalam waktu yang bersamaan. Tiap-tiap proses tersebut memiliki salinan dari register dan tempat penyimpanan data untuk memperoleh data proses yang akan dieksekusi. Oleh karena itu 2 proses berbeda akan memiliki data yang berbeda pula.

Dalam berbagi halaman, hanya satu salinan dari *editor* yang akan disimpan dalam memori. Tiap halaman tabel pengguna akan memetakan *editor*nya masing-masing ke alamat fisik yang sama namun halaman data mereka akan dipetakan ke alamat fisik yang berbeda-beda. Sehingga untuk kasus 40 pengguna diatas, kita hanya butuh satu buah salinan dari *editor* (150 KB) serta 40 salinan masing-masing sebesar 50 KB. Maka jumlah ruang memori yang dibutuhkan adalah 2.150 KB yang jauh lebih sedikit dibandingkan dengan 8.000 KB jika tidak menggunakan teknik berbagi memori.

Program-program lain yang dapat dilakukan pembagian memori contohnya kompilator, *window systems*, *run-time libraries*, sistem basis data dan lain-lain.

2.7. Rangkuman

Memori merupakan salah satu sumber daya yang penting dalam pengeksekusian sebuah proses. Agar suatu proses dapat dieksekusi, ia harus terletak di dalam memori sebelum CPU mengambil instruksi-instruksi pada alamat yang ditunjuk oleh *program counter*.

Swapping menukarkan sebuah proses keluar dari memori untuk sementara waktu ke sebuah penyimpanan sementara (biasanya berupa sebuah *fast disk* dengan kapasitas yang dapat menampung semua salinan dari semua gambaran memori serta menyediakan akses langsung ke gambaran tersebut) dengan sebuah proses lain yang sedang membutuhkan sejumlah alokasi memori untuk dieksekusi. *Swapping roll out, roll in* menggunakan algoritma berbasis prioritas dimana ketika proses dengan prioritas lebih tinggi tiba maka *memory manager* akan mengeluarkan proses dengan prioritas yang lebih rendah serta memasukkan proses dengan prioritas yang lebih tinggi tersebut.

Pemetaan memori memetakan alamat logis yang dihasilkan CPU ke alamat fisik yang nantinya akan dibawa ke memori pada saat akan dieksekusi.Pada pemetaan memori ini terdapat *limit register* yang

terdiri dari rentang nilai alamat logis (*range of logical address*). Dengan adanya *limit register* dan register relokasi, tiap alamat logis haruslah lebih kecil dari *limit register*. Proses pemetaan dilakukan oleh MMU (*Memory Management Unit*) dengan menjumlahkan nilai register relokasi ke alamat logis

Partisi memori ada dua jenis yaitu statis (tetap) dan dinamis (berubah-ubah). Pada partisi dinamis ada beberapa teknik untuk memenuhi permintaan berukuran n dengan lubang-lubang yang tersedia yaitu *first fit* (menemukan lubang pertama yang cukup besar), *next fit* (sama seperti first fit namun pencarian tidak dari awal), *best fit* (lubang terkecil yang cukup), dan *worst fit* (lubang terbesar yang ada dalam daftar).

Fragmentasi merupakan peristiwa munculnya lubang-lubang kecil yang tidak cukup menampung permintaan proses. Terdiri dari dua jenis, yaitu eksternal (biasanya pada partisi dinamis dengan total kapasitas lubang-lubang cukup menampung sebuah proses namun letaknya terpisah-pisah, solusinya dengan penghalaman, segmentasi dan pemadatan) serta internal (biasanya pada partisi tetap dengan adanya kapasitas sisa sebuah lubang yang tidak dapat digunakan karena menjadi milik proses yang dialokasikan lubang tersebut).

Berbagi halaman memungkinkan beberapa proses untuk mengakses kode yang sama namun dengan data yang berbeda-beda. Hal ini jelas akan mengurangi jumlah ruang memori yang dibutuhkan untuk memenuhi permintaan beberapa proses tersebut.

Rujukan

[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.

[WEBPCMag] The Computer Language Company. 2007. *PCMag http://www.pcmag.com/ encyclopedia_term/ 0,2542,t=reentrant+code&i=50332,00.asp* . Diakses 29 Maret 2007.

Bab 3. Pemberian Halaman

3.1. Pendahuluan

Pada bab-bab sebelumnya telah dijelaskan bahwa memori harus digunakan dengan baik sehingga dapat memuat proses dalam satu waktu. Dalam implementasinya telah dijelaskan bahwa terdapat dua macam alamat memori yaitu alamat logika dan alamat fisik. Alamat logika *(logical address)* adalah alamat yang dihasilkan oleh CPU atau sering disebut *virtual address.* Sedangkan alamat fisik *(physical address)* adalah alamat yang terdapat di memori. Salah satu cara pengalokasian memori adalah dengan *contiguous memory allocation* di mana alamat yang diberikan kepada proses berurutan dari kecil ke besar. Selanjutnya sangat mungkin terjadi fragmentasi, yaitu munculnya lubang-lubang yang tidak cukup besar untuk menampung permintaan dari proses. Fragmentasi terdiri dari 2 macam, yaitu fragmentasi intern dan fragmentasi ekstern. Fragmentasi ekstern terjadi apabila jumlah seluruh 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. Sedangkan fragmentasi intern terjadi apabila jumlah memori yang diberikan oleh penjadualan CPU lebih besar daripada yang diminta proses dan fragmentasi ini tidak dapat dihindari.

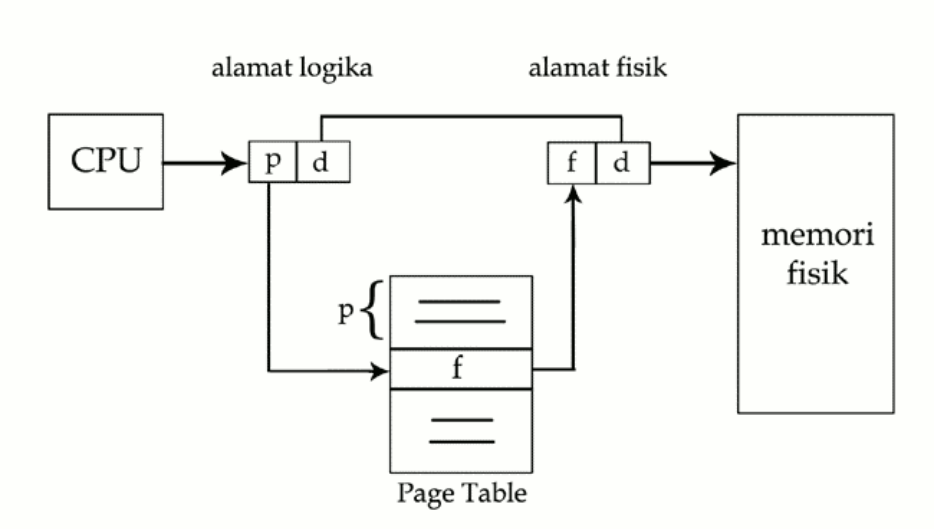
Salah satu solusi untuk mencegah fragmentasi ekstern adalah dengan *paging*. *Paging* adalah suatu metode yang memungkinkan suatu alamat fisik memori yang tersedia dapat tidak berurutan *(non contiguous).*

3.2. Metode Dasar

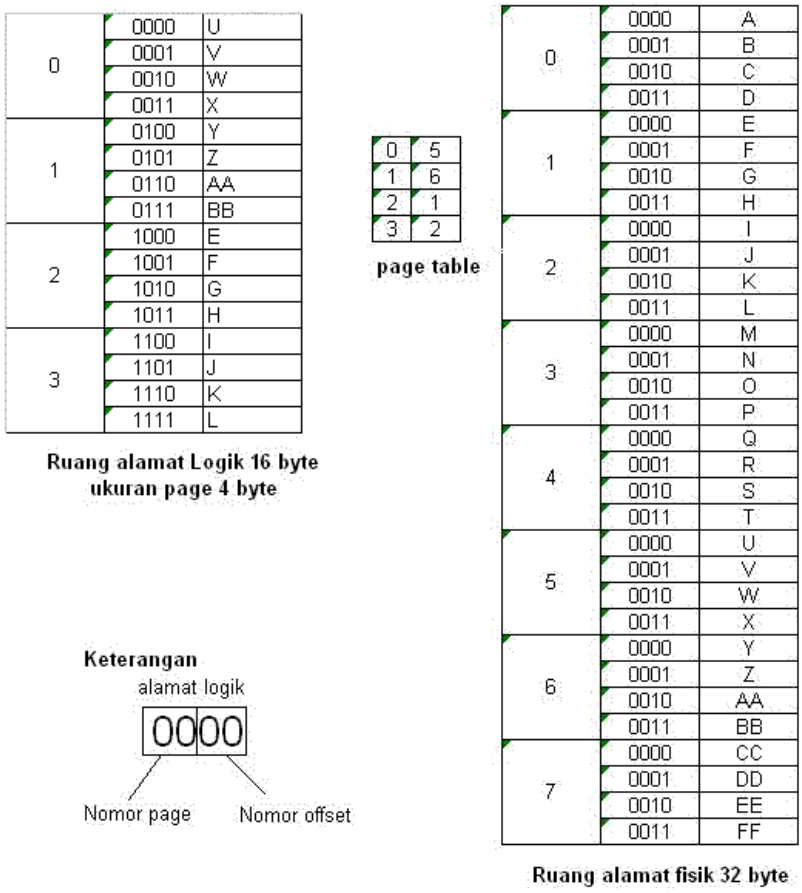
Metode dasar dari *paging* adalah dengan memecah memori fisik menjadi blok-blok yang berukuran tertentu yang disebut dengan *frame* dan memecah memori logika menjadi blok-blok yang berukuran sama dengan *frame* yang disebut dengan *page*. Selanjutnya sebuah *page table* akan menerjemahkan alamat logika ke alamat fisik.

Alamat logika terdiri dari 2 bagian yaitu nomor *page* dan nomor *offset*. Bila digunakan ruang alamat logika 2 pangkat m dan ukuran *page* 2 pangkat n *bytes*, maka m-n bit paling kiri menunjukkan nomor *page* dan n bit paling kanan menunjukkan *offset.*

Gambar 3.1. Translasi Alamat Pada Sistem *Paging*



Gambar 3.2. Contoh Translasi Alamat Pada Sistem *Paging*



Lihat gambar di atas. Bila kita mempunyai ruang alamat logika 16 byte (2 pangkat 4), dan ukuran *page*-nya 4 byte (2 pangkat 2), maka 2 bit (4-2 dari m-n) paling kiri menunjukkan nomor *page* dan 2 (didapat dari n) bit paling kanan menunjukkan nomor *offset*. Contohnya alamat logika 0000, maka bit 00 sebelah kiri menunjukkan bahwa nomor *page*nya adalah 0 , sedangkan 00 sebelah kanan menunjukkan bahwa nomor *offset*nya adalah 0000 (bilangan biner). Dilihat di *page table* bahwa *page* 0 dipetakan ke *frame* 5, berarti alamat logika 0000 dipetakan ke *frame* 5 *offset* 0000. Dan alamat logika 0000 menyimpan data dari *frame* 5 *offset* 0 yaitu 'U'. Begitu pula alamat logika 0110 berarti nomor *page*nya adalah 01 atau 1, dan nomor *offset*nya 0010. Sehingga dipetakan ke *frame* 6 *offset* 0010 dan menyimpan data AA.

Fragmentasi intern masih mungkin terjadi pada sistem *paging*. Contohnya adalah bila *page* berukuran 2KB (2048 byte), maka proses berukuran 20500 byte membutuhkan 10 *page* dan tambahan 20 byte, berarti diperlukan 11 *frame* sehingga terjadi fragmentasi intern sebesar 2028 byte (2048-20) dan *worst case* yang terjadi adalah fragmentasi intern sebesar ukuran *page* dikurang 1 byte.

3.3. Dukungan Perangkat Keras

Bila *page table* berukuran kecil (misal 256 entri), maka *page table* dapat diimplementasikan dengan menggunakan register. Namun pada masa sekarang ini *page table* memiliki ukuran yang terlalu besar (misal 1 juta entri) untuk diterapkan di register sehingga *page table* harus diletakkan di memori. Untuk itu digunakan *Page Table Base Register* (PTBR) dan *Page Table Length Register* (PTLR) yang menunjukkan ukuran *page table*. Masalah untuk implementasi ini adalah dibutuhkan dua kali akses ke memori pada program, yaitu untuk *page table* dan untuk data/instruksi. Untuk mempercepat

waktu akses ini, digunakan *cache* yang terdiri dari sekumpulan register asosiatif atau disebut TLB *(Translation Look-aside Buffer)* yang merupakan memori berkecepatan tinggi.

Register asosiatif

Setiap register asosiatif menyimpan pasangan nomor *page* dan nomor *frame* (alamat awal). Input untuk register asosiatif akan dibandingkan dengan data pada register asosiatif. Bila input ditemukan maka nomor *frame* yang sesuai akan dihasilkan, kejadian ini disebut TLB *hit*. Sedangkan TLB *miss* terjadi bila input tidak ditemukan, maka input tersebut akan dicari pada memori (yang lebih lambat dari *cache*).

Waktu Akses Memori Efektif / *Effective Access Time* (EAT)

Peluang bahwa nomor *page* akan ditemukan dalam register disebut *Hit Ratio.* *Hit ratio* 80% berarti menemukan nomor *page* yang ingin kita cari dalam TLB sebesar 80%. Jika untuk mencari TLB memakan waktu 20 nanodetik dan 100 nanodetik untuk akses ke memori, maka total waktunya adalah 120 nanodetik bila nomor *page* ada di TLB. Jika kita gagal mendapat nomor *page* di TLB (20 nanodetik), maka kita harus mengakses memori untuk *page table* dan nomor *frame* (100 nanodetik) dan mengakses byte yang diharapkan di memori (100 nanodetik), sehingga totalnya 220 nanodetik. Untuk mendapatkan waktu akses memori efektif, kita mengalikan tiap kasus dengan peluangnya:

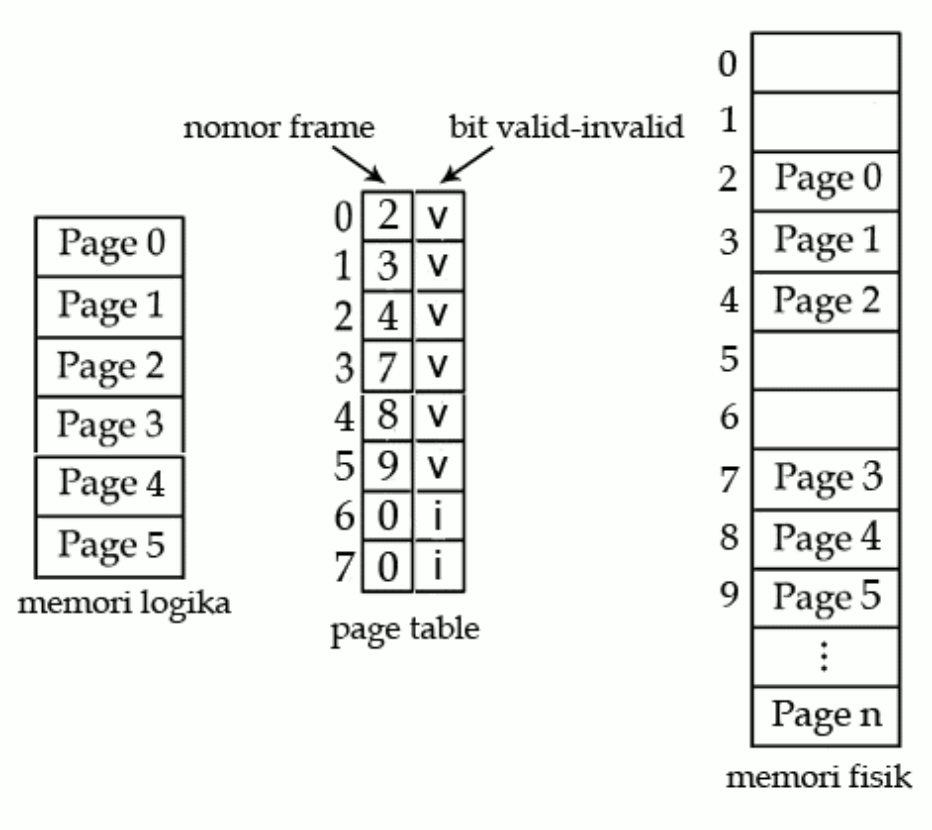
waktu akses efektif = (0,80 x 120) + (0,20 x 220) = 140 nanodetik

3.4. Proteksi

Proteksi memori dapat diterapkan pada sistem *paging* dengan meletakkan bit proteksi pada setiap *frame*. Bit proteksi umumnya disimpan pada *page table*. Sebuah bit proteksi dapat mendefinisikan apakah *page* bersifat *read-only* atau *read-write*. Untuk memberikan proteksi yang lebih baik, dapat pula ditambahkan bit lainnya, misalnya untuk sifat *execute-only.*

Bit lainnya yang umumnya terdapat di setiap entri pada *page table* adalah bit *valid-invalid* . Bit *valid* menyatakan bahwa *page* terletak di dalam ruang alamat logika proses. Bit *invalid* menyatakan bahwa *page* terletak di luar ruang alamat logika proses (dapat dilihat contohnya pada Gambar 3 bahwa *page* 6 dan 7 berada di luar ruang alamat logika sehingga diberikan bit *invalid*). Pelanggaran terhadap bit proteksi menyebabkan *trap* ke sistem operasi.

Gambar 3.3. Bit*Valid* (v) dan *Invalid* (i) pada *Page Table*

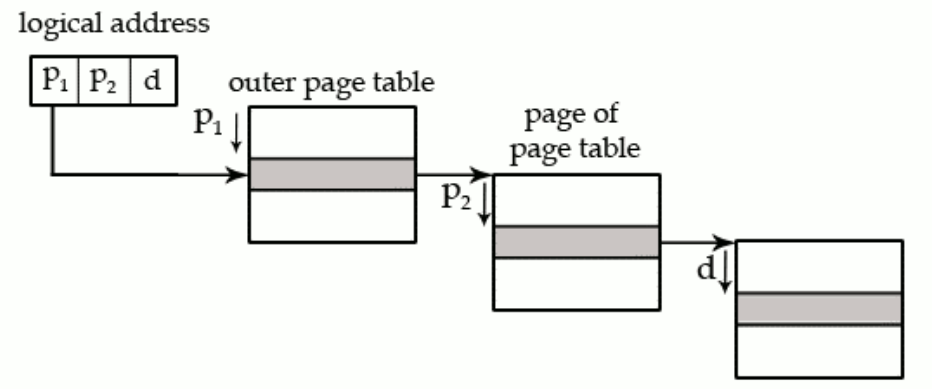


3.5. Tabel Halaman Bertingkat

Hierarchical paging atau pemberian halaman secara bertingkat merupakan sebuah metode pemberian halaman dengan cara membagi sebuah *page table* menjadi beberapa *page table* yang berukuran lebih kecil. Metode ini merupakan solusi dari permasalahan alokasi *page table* berukuran sangat besar pada *main memory* yang umumnya dihadapi pada sistem komputer modern yang memiliki ruang alamat logika yang besar sekali (mencapai 2 pangkat 32 sampai 2 pangkat 64).

Konsep dasar metode ini yaitu menggunakan pembagian tingkat setiap segmen alamat logika. Setiap segmen menunjukkan indeks dari sebuah *page table*, kecuali segmen terakhir yang menunjuk langsung ke *frame* pada memori fisik. Segmen terakhir ini disebut *offset* (d). Dapat disimpulkan bahwa segmen yang terdapat pada alamat logika menentukan berapa level *paging* yang digunakan yaitu banyak segmen dikurang 1.

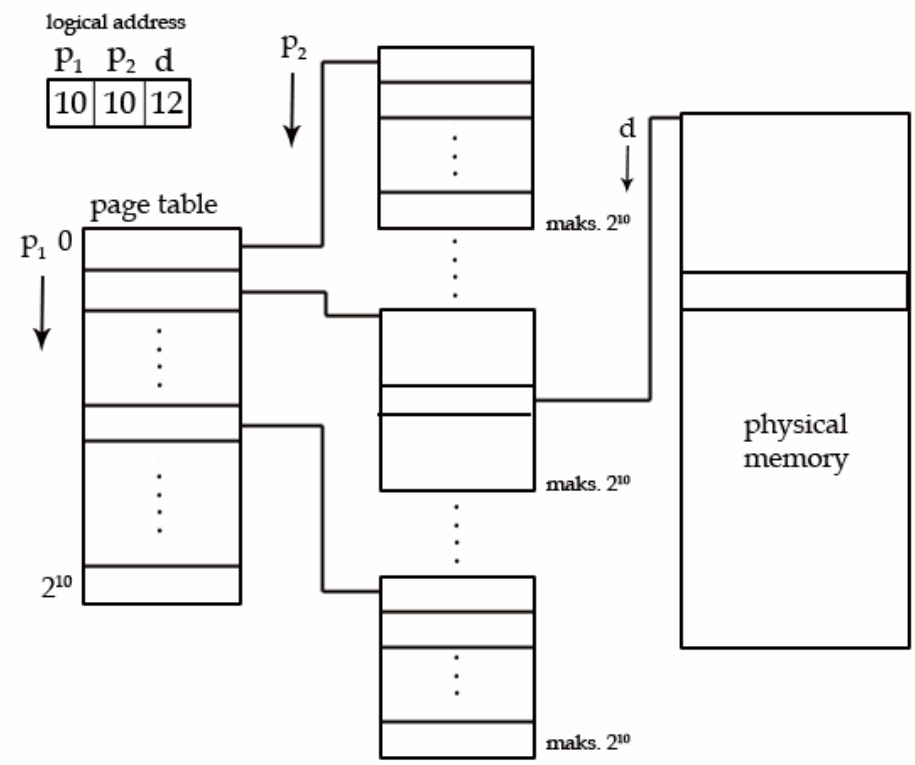
Gambar 3.4. Translasi Alamat pada *Two-Level Paging*



Dengan metode ini, isi pada indeks *page table* pertama akan menunjuk pada *page table* kedua yang indeksnya bersesuaian dengan isi dari *page table* pertama tersebut. Sedangkan isi dari *page table* kedua menunjukkan tempat dimana *page table* ketiga bermula, sedang segmen alamat logika kedua adalah indeks ke-n setelah *starting point page table* ketiga dan seterusnya sampai dengan segmen terakhir.

Sebagai contoh, pada suatu sistem komputer 32 bit dengan ukuran *page* 4 KB, alamat logika dibagi ke dalam nomor *page* yang terdiri dari 20 bit dan *page offset*-nya 12 bit. Karena *page table*-nya di-*paging*-kan lagi, maka nomor *page*-nya dibagi lagi menjadi 10 bit nomor *page* dan 10 bit *page offset*, sehingga dapat digambarkan sebagai berikut:

Gambar 3.5. Contoh *Two-level paging*



Metode *Hierarchical paging* ini memang dapat menghemat ruang memori yang dibutuhkan dalam pembuatan *page table*-nya sendiri, namun waktu akses yang diperlukan menjadi lebih besar karena harus melakukan akses berkali-kali untuk mendapatkan alamat fisik yang sebenarnya.

3.6. Tabel Halaman Dengan *Hash*

Metode ini umumnya digunakan untuk menangani masalah ruang alamat logika yang besarnya mencapai 64 bit karena struktur *page table* pada metode ini bisa menghemat ruang memori dalam jumlah yang cukup besar. *Hashed page table* menggunakan tabel *hash* sebagai *page table*-nya dengan ukuran yang terbatas untuk menghemat ruang memori dan sebuah *hash function* untuk mengalokasikan alamat virtual pada *page table* tersebut. Setiap entri/blok pada *page table* berisi *linked list* yang menghubungkan elemen-elemen yang di- *hash* ke lokasi yang sama. Tiap elemen tersebut terdiri dari 3 field, yaitu *virtual page number*, nomor *frame* dimana alamat virtual tersebut dipetakan, dan *pointer* yang menunjukkan elemen berikutnya dalam *linked list*. Fungsi linked list disini adalah untuk mengatasi *collision* yang terjadi pada saat pengalokasian alamat virtual ke *hash table* yang ukurannya sangat terbatas.

Mekanisme *paging* pada metode ini yaitu:

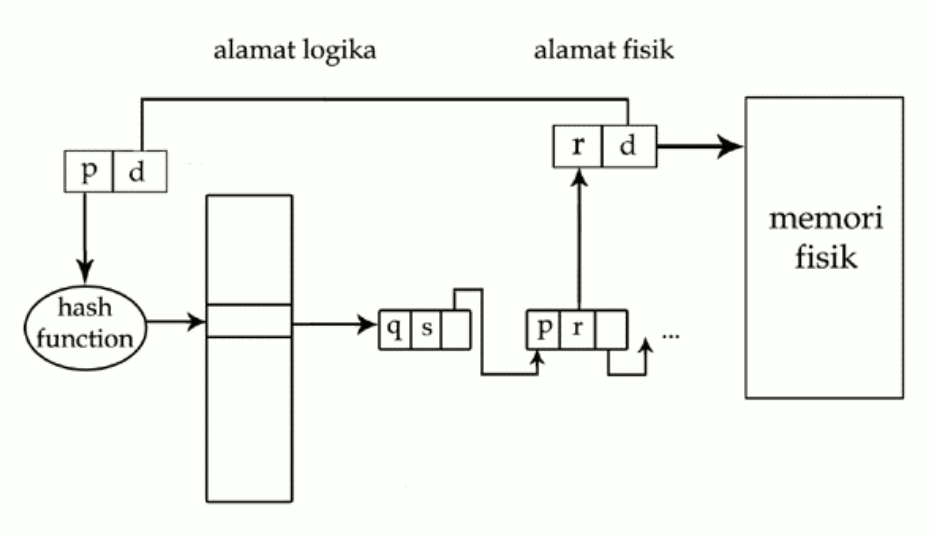
1. **Alamat logika dipetakan ke suatu lokasi/entri di *page table* dengan menggunakan *hash function*.**

2. ***Page number* tersebut kemudian di simpan sebagai *field* pertama pada sebuah elemen dalam entri yang teralokasikan.**

3. ***Page number* tersebut lalu dipasangkan dengan *frame number* yang *available* yang disimpan pada *field* kedua di elemen yang sama .**

4. **Untuk mendapatkan lokasi yang sebenarnya pada memori fisik, *frame number* pada field kedua di-*concate* dengan *offset* .**

Gambar 3.6. *Hashed Page Table*



3.7. Rangkuman

Paging adalah suatu metode yang mengizinkan alamat logika proses untuk dipetakan ke alamat fisik memori yang tidak berurutan, yaitu sebagai solusi dari masalah fragmentasi ekstern. Metode dasar dari *paging* adalah dengan memecah memori fisik menjadi blok-blok yang berukuran tertentu *(frame)* dan memecah memori logika menjadi blok-blok yang berukuran sama *(page)*. Penerjemahan alamat virtual ke alamat fisik dilakukan oleh *page table* melalui perantara *Memory Management Unit* (MMU). *Paging* menjamin keamanan data di memori saat suatu proses sedang berjalan. Proteksi memori dapat diterapkan pada sistem *paging* dengan meletakkan bit proteksi pada setiap *frame*.

Setiap sistem operasi mengimplementasikan *paging* dengan caranya masing-masing. *Hierarchical paging* dan *hashed page table* merupakan metode yang umum digunakan karena bisa menghemat ruang memori yang dibutuhkan.

Rujukan

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

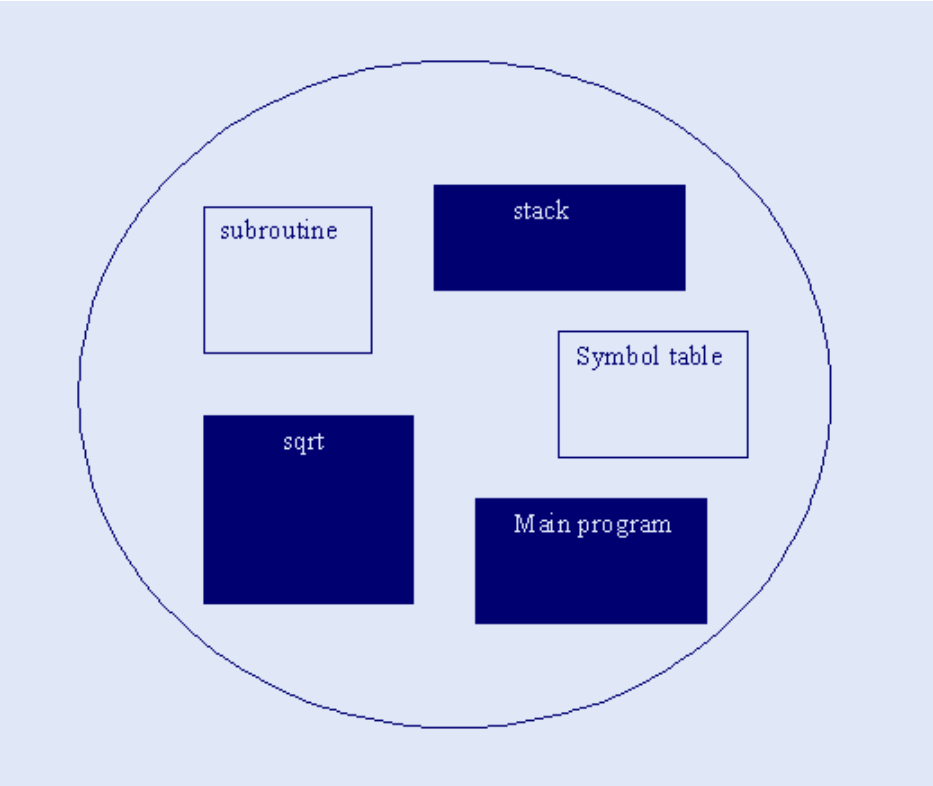
[FitriSari2005] Riri Fitri Sari dan Yansen Darmaputra. 2005. *Sistem Operasi Modern*. Penerbit Andi.

Bab 4. Arsitektur Intel Pentium

4.1. Pendahuluan

Aspek penting dari memori manajemen yang menjadi tak terhindarkan dengan *paging* adalah memori dari sudut pandang pengguna dan memori fisik 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 fisik dengan memori lojik. Orang-orang lebih suka memandang sebuah memori sebagai sekumpulan variabel-variabel yang berada dalam segmen-segmen dalam ukuran tertentu.

Gambar 4.1. Alamat Lojik



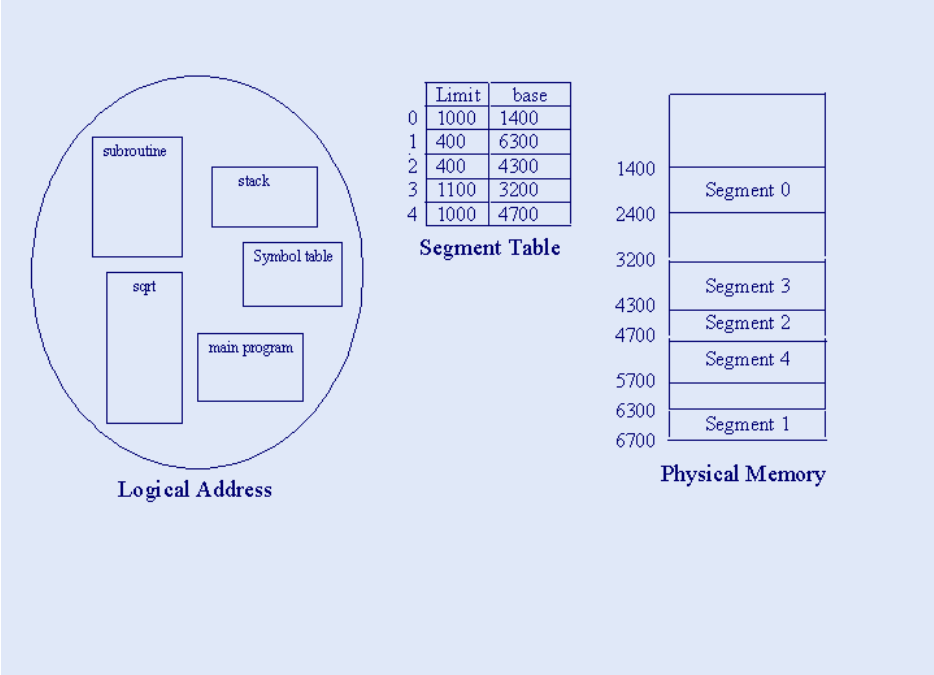
4.2. Segmentasi

Segmentasi merupakan skema manajemen memori yang mendukung cara pandang seorang programmer terhadap memori. Ruang alamat lojik merupakan sekumpulan dari segmen-segmen. Masing-masing segmen 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 segmen. Sehingga, alamat lojik terdiri dari dua tuple: [segmen-number, offset].

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 kesatu dimensi alamat fisik. Pemetaan ini disebut sebagai sebuah segmen table. Masing-masing masukan mempunyai segmen *base* dan segmen limit. Segmen *base* merupakan alamat fisik dan segmen limit merupakan panjang dari segmen.

Sebagai contoh, 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 = 4353. Suatu referensi ke segmen 3, *byte* 852, dipetakan ke 3200 + 852 = 4052. referensi ke *byte* 1222 dari segmen 0 akan menghasilkan suatu *trap* ke sistem operasi, karena segmen ini hanya mempunyai panjang 1000 *byte.* Lihat gambar 2. Segmentasi.

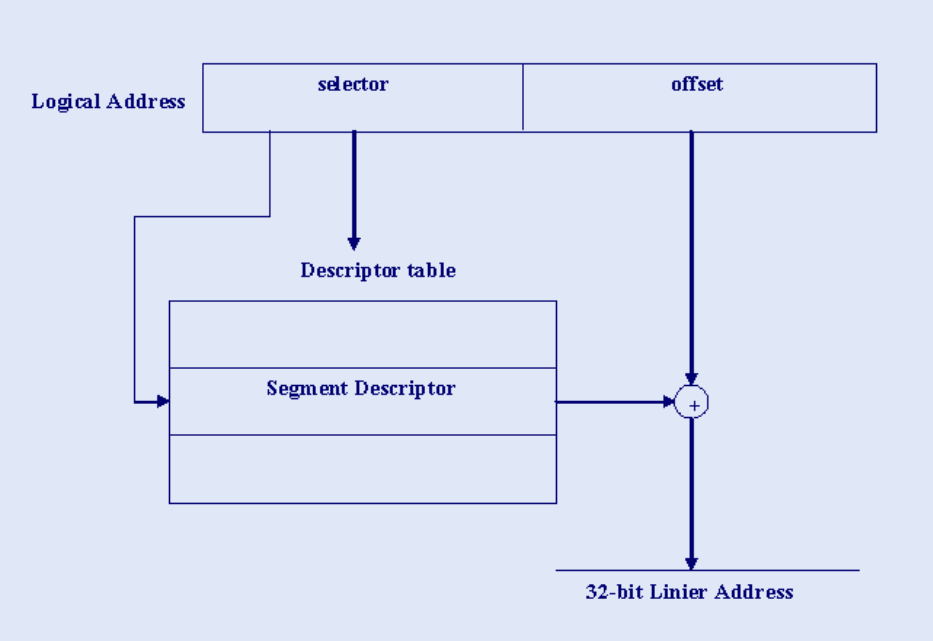
Gambar 4.2. Segmentasi



4.3. Segmentasi Pentium

Alamat linear pada pentium panjangnya 32 bit dan prosesnya adalah register segmen menunjuk pada *entry* yang sesuai. Informasi *base* dan *limit* tentang segmen pentium digunakan untuk menghasilkan alamat linier. Pertama, limit digunakan untuk memeriksa validitas suatu alamat. Jika alamat tidak valid, maka kesalahan memori akan terjadi yang menimbulkan *trap* pada sistem operasi. Tetapi jika alamat valid maka nilai *offset* dijumlahkan dengan nilai *base,* yang menghasilkan alamat linier 32 bit. Hal ini ditunjukkan seperti pada gambar berikut.

Gambar 4.3. Segmentasi



4.4. Penghalaman

Penghalaman adalah suatu metode yang memungkinkan suatu alamat fisik memori yang tersedia dapat tidak berurutan. Penghalaman dapat menjadi solusi untuk pemecahan masalah luar. Penghalaman dapat mencegah masalah penting dari pengepasan besar ukuran memori yang bervariasi kedalam penyimpanan cadangan. Biasanya bagian yang mendukung untuk penghalaman telah ditangani oleh perangkat keras.

Jadi metode yang digunakan adalah dengan memecah memori fisik menjadi blok-blok berukuran tetap yang akan disebut sebagai *frame.* Selanjutnya memori logis akan dipecah juga menjadi ukuran-ukuran tertentu berupa blok-blok yang sama disebut sebagai halaman. Selanjutnya kita akan membuat sebuah tabel halaman yang akan menerjemahkan 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.

4.5. Penghalaman Linux

Pada pentium, Linux hanya menggunakan 6 segmen:

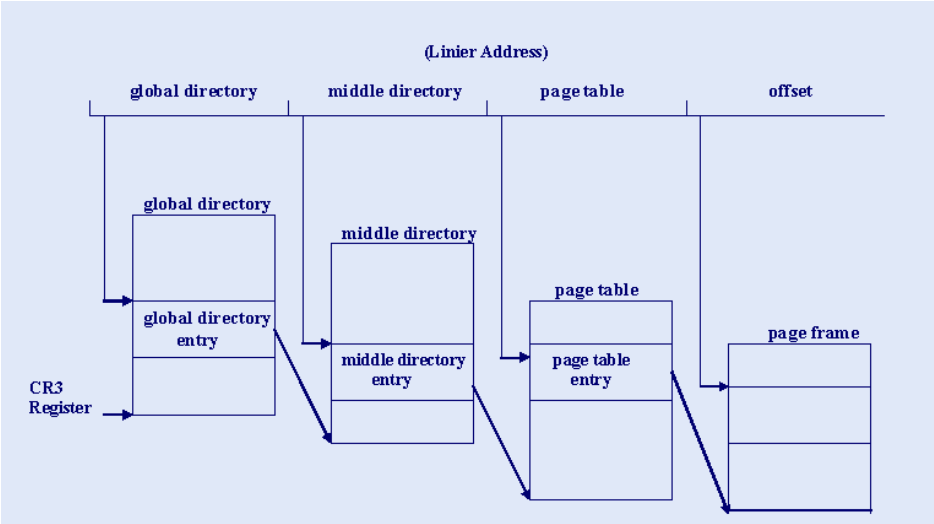
1. Segmen untuk kode kernel
2. Segmen untuk data kernel
3. Segmen untuk kode pengguna
4. Segmen untuk data pengguna
5. Segmen Task-State
6. Segment *default* untuk LDT

Segmen untuk kode pengguna dan data pengguna terbagi dengan semua proses yang *running* pada pengguna mode, karena semua proses menggunakan ruang alamat logis yang sama dan semua *descriptor* segmen terletak di GDT. TSS (*Task-State Segment*) digunakan untuk menyimpan *context hardware* dari setiap proses selama *context switch.* Tiap proses mempunyai TSS sendiri, dimana *descriptor* -nya terletak di GDT. Segment *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.

Tiap *selector* segmen mempunyai 2 bit proteksi. Mak, Pentium Mengizinkan proteksi 4 level. Dari 4 level ini, Linux hanya mengenal 2 level, yaitu pengguna mode dan kernel mode.

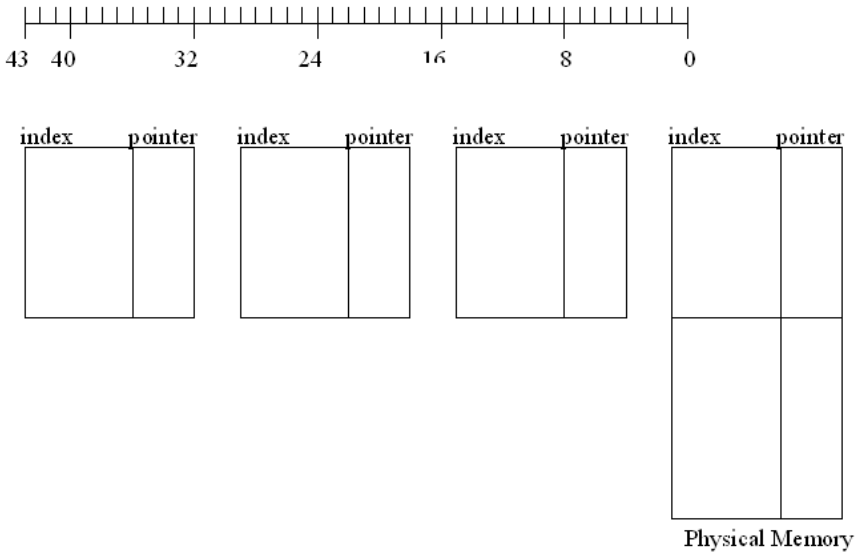
Berikut ini merupakan tiga level penghalaman dalam Linux

Gambar 4.4. Memori Virtual



Berikut ini merupakan contoh soal dari memori virtual linux:

Gambar 4.5. memori Virtual



004 0200 8004(HEX), merupakan alamat virtual memori linux yang sah (43 bit), dengan tiga tingkatan tabel halaman (*three level page tables*): *Global Directory* (10 bit), *Page Middle Directory* (10 bit), dan *Page table* (10 bit).

1. Uraikan alamat virtual tersebut di atas dari basis 16 (Hex) ke basis 2
2. Lengkapi gambar di atas seperti nama tabel-tabel, indeks tabel dalam basis heksadesimal(Hex), pointer (cukup dengan panah), alamat memori fisik (physical memory), dalam basis heksadesimal(Hex), isi memori fisik(bebas), serta silahkan menggunakan titik-titik "...." untuk menandakan "dan seterusnya".
3. Berapa ukuran bingkai memori (memori *frame*) ?

4.6. Rangkuman

Segmentasi merupakan skema manajemen memori yang mendukung cara pandang seorang programmer terhadap memori. Masing-masing segmen mempunyai panjang dan nama yang dapat mewakili sebagai suatu alamat. Maksimal pada Pentium hanya mengizinkan proteksi 4 level. Dan dari 4 level ini, linux hanya mengenal 2 level, yaitu pengguna mode dan kernal mode.

Rujukan

[Hariyanto1997] Bambang Hariyanto. 1997. *Sistem Operasi*. Buku Teks Ilmu Komputer. Edisi Kedua. Informatika. Bandung.

[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 5. Memori Virtual

5.1. Pendahuluan

Selama bertahun-tahun, pelaksanaan manajemen memori pada intinya adalah dengan menempatkan semua bagian proses yang akan dijalankan ke dalam memori sebelum proses dapat mulai dieksekusi. Dengan demikian semua bagian proses tersebut harus memiliki alokasi sendiri di dalam memori fisik.

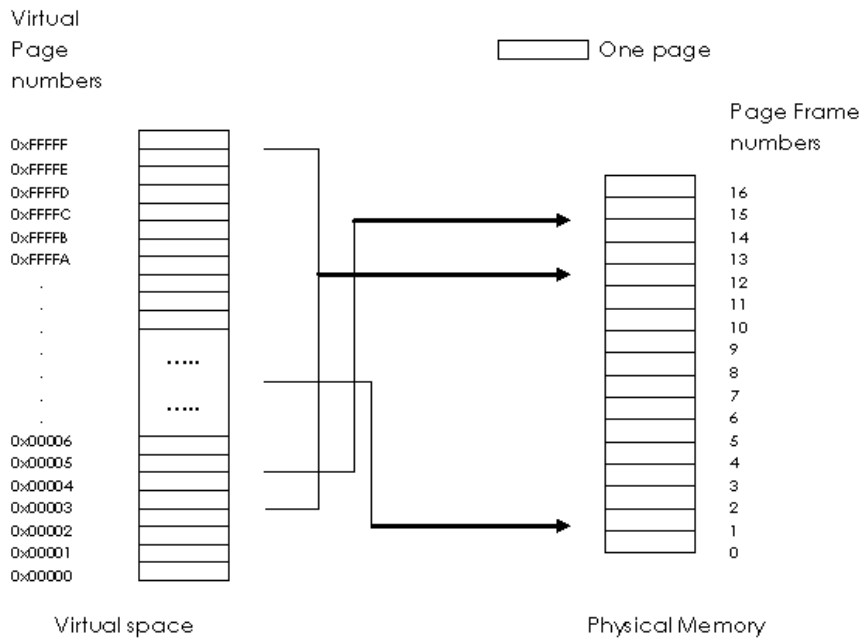
Pada kenyataannya tidak semua bagian dari program tersebut akan diproses, misalnya:
- Ada pernyataan-pernyataan atau pilihan yang hanya akan dieksekusi jika kondisi tertentu dipenuhi
- Terdapat fungsi-fungsi yang jarang digunakan
- Pengalokasian memori yang lebih besar dari yang sebenarnya dibutuhkan.

Pada memori berkapasitas besar, hal-hal ini tidak akan menjadi masalah. Namun pada memori dengan kapasitas yang sangat terbatas, hal ini akan menurunkan optimalisasi utilitas dari ruang memori fisik (memori utama).

Setiap program yang dijalankan harus berada di memori. Memori merupakan suatu tempat penyimpanan utama (*primary storage*) yang bersifat sementara (*volatile*). Ukuran memori yang terbatas dapat menimbulkan masalah bagaimana menempatkan program yang berukuran yang lebih besar dari ukuran memori fisik (memori utama) dan masalah penerapan *multiprogramming* yang membutuhkan tempat yang lebih besar di memori.

Memori virtual adalah suatu teknik yang memisahkan antara memori logis dan memori fisiknya. Memori logis merupakan kumpulan keseluruhan halaman dari suatu program. Tanpa memori virtual, memori logis akan langsung dibawa ke memori fisik (memori utama). Disinilah memori virtual melakukan pemisahan dengan menaruh memori logis ke *secondary storage* (disk sekunder) dan hanya membawa halaman yang diperlukan ke memori utama (memori fisik). Teknik ini menempatkan keseluruhan program di disk sekunder dan membawa halaman-halaman yang diperlukan ke memori fisik sehingga memori utama hanya akan menyimpan sebagian alamat proses yang sering digunakan dan sebagian lainnya akan disimpan dalam disk sekunder dan dapat diambil sesuai dengan kebutuhan. 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 ke disk sekunder.

Gambar 5.1. Memori Virtual



Pada gambar diatas ditunjukkan ruang sebuah memori virtual yang dibagi menjadi bagian-bagian yang sama dan diidentifikasikan dengan nomor *virtual pages*. Memori fisik dibagi menjadi *page frames* yang berukuran sama dan diidentifikasikan dengan nomor *page frames*. Bingkai (*frame*) menyimpan data dari halaman. Atau memori virtual memetakan nomor *virtual pages* ke nomor *page frames*. *Mapping* (pemetaan) menyebabkan halaman virtual hanya dapat mempunyai satu lokasi alamat fisik.

Dalam sistem *paging*, jika sebuah ruang diperlukan untuk proses dan halaman yang bersangkutan tidak sedang digunakan, maka halaman dari proses akan mengalami *paged out* (disimpan ke dalam disk) atau *swap out*, memori akan kosong untuk halaman aktif yang lain. Halaman yang dipindah dari disk ke memori ketika diperlukan dinamakan *paged in* (dikembalikan ke memori) atau *swap in*. Ketika sebuah item dapat mengalami *paging*, maka item tersebut termasuk dalam item yang menempati ruang virtual, yang diakses dengan alamat virtual dan ruangan yang ada dialokasikan untuk informasi pemetaan. Sistem operasi mengalokasikan alamat dari item tersebut hanya ketika item tersebut mengalami *paging in*.

Keuntungan yang diperoleh dari penyimpanan hanya sebagian program saja pada memori fisik adalah:
- Berkurangnya proses M/K yang dibutuhkan (lalu lintas M/K menjadi rendah)
- Ruang menjadi lebih leluasa karena berkurangnya memori fisik yang digunakan
- 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.

Teknik memori virtual akan memudahkan pekerjaan seorang programmer ketika besar data dan programnya melampaui kapasitas memori utama. Sebuah *multiprogramming* dapat mengimplementasikan teknik memori virtual sehingga sistem *multiprogramming* menjadi lebih efisien. Contohnya: 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.

Prinsip dari memori virtual adalah bahwa "Kecepatan maksimum ekseskusi proses di memori virtual dapat sama, tetapi tidak akan pernah melampaui kecepatan eksekusi proses yang sama di sistem yang tidak menggunakan memori virtual".

Memori virtual dapat diimplementasikan dengan dua cara:
1. *Demand Paging* yaitu dengan menerapkan konsep pemberian halaman pada proses
2. *Demand segmentation*, lebih kompleks diterapkan ukuran segmen yang bervariasi.

5.2. *Demand Paging*

Demand Paging atau permintaan pemberian halaman adalah salah satu implementasi dari memori virtual yang paling umum digunakan. Sistem *Demand Paging* pada prinsipnya hampir sama dengan sistem permintaan halaman yang menggunakan *swapping*, hanya saja pada sistem *demand paging*, halaman tidak akan dibawa ke dalam memori fisik sampai ia benar-benar diperlukan. Oleh sebab itu dibutuhkan bantuan perangkat keras untuk mengetahui lokasi dari halaman saat ia diperlukan. Daripada melakukan *swapping*, keseluruhan proses ke dalam memori utama, digunakanlah yang disebut *lazy swapper* yaitu tidak pernah menukar sebuah halaman ke dalam memori utama kecuali halaman tersebut diperlukan. Keuntungan yang diperoleh dengan menggunakan *demand paging* sama dengan keuntungan pada memori virtual di atas.

Saat melakukan pengecekan pada halaman yang dibutuhkan oleh suatu proses, terdapat tiga kemungkinan kasus yang dapat terjadi, yaitu:
- Halaman ada dan sudah langsung berada di memori utama - statusnya adalah valid ("v" atau "1")
- Halaman ada tetapi belum berada di memori utama atau dengan kata lain halaman masih berada di disk sekunder - statusnya adalah tidak valid/*invalid* ("i" atau "0")
- Halaman benar - benar tidak ada, baik di memori utama maupun di disk sekunder (*invalid reference*) - statusnya adalah tidak valid/*invalid* ("i" atau "0")

Ketika kasus kedua dan ketiga terjadi, maka proses dinyatakan mengalami kesalahan halaman (*page fault*). Selanjutnya proses tersebut akan dijebak ke dalam sistem operasi oleh perangkat keras.

Skema Bit Valid - Tidak Valid

Dalam menentukan halaman mana yang ada di dalam memori utama dan halaman mana yang tidak ada di dalam memori utama, diperlukan suatu konsep, yaitu skema bit valid - tidak valid. Kondisi valid berarti bahwa halaman yang dibutuhkan itu legal dan berada di dalam memori utama (kasus pertama). Sementara tidak valid/*invalid* adalah kondisi dimana halaman tidak ada di memori utama namun ada di disk sekunder (kasus kedua) atau halaman memang benar-benar tidak ada baik di memori utama maupun disk sekunder (kasus ketiga).

Pengaturan bit dilakukan sebagai berikut:

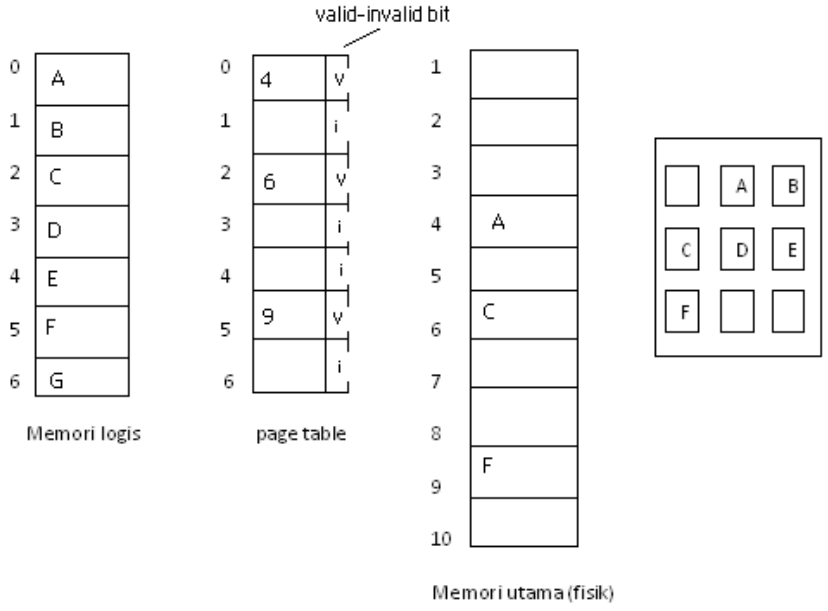
- Bit = 1 berarti halaman berada di memori utama
- Bit = 0 berarti halaman tidak berada di memori utama

Apabila ternyata hasil dari mengartikan alamat melalui *page table* menghasilkan bit halaman yang bernilai 0, maka akan menyebabkan terjadinya *page fault* .

Page fault adalah interupsi yang terjadi ketika halaman yang diminta/dibutuhkan oleh suatu proses tidak berada di memori utama. Proses yang sedang berjalan akan mengakses *page table* (tabel halaman) untuk mendapatkan referensi halaman yang diinginkan. *Page fault* dapat diketahui/dideteksi dari penggunaan skema bit valid-tidak valid ini. Bagian inilah yang menandakan terjadinya suatu permintaan pemberian halaman .

Jika suatu proses mencoba untuk mengakses suatu halaman dengan bit yang di-set tidak valid maka *page fault* akan terjadi. Proses akan dihentikan sementara halaman yang diminta/dibutuhkan dicari didalam disk.

Gambar 5.2. Tabel Halaman dengan Skema Bit Valid - Tidak valid



5.3. Penanganan *Page Fault*

Prosedur untuk menangani *page fault* adalah sebagai berikut:

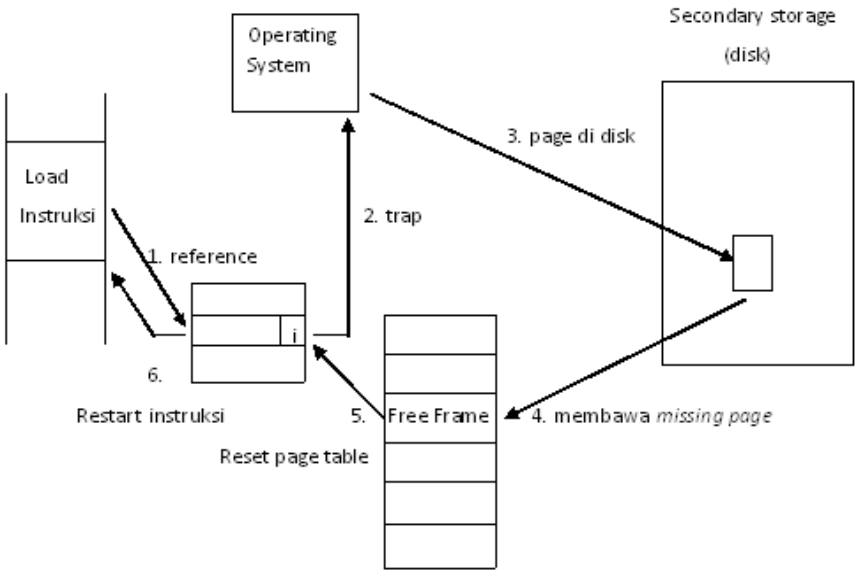
- CPU mengambil (*load*) instruksi dari memori untuk dijalankan. Pengambilan instruksi dilakukan dari halaman pada memori dengan mengakses tabel halaman. Ternyata pada tabel halaman bit ter-set tidak valid atau *invalid* (i).

- Interupsi *page fault* terjadi sehingga interupsi tersebut menyebabkan perangkat keras melakukan *trap* yaitu menjebak proses tersebut ke dalam sistem operasi.
- Jika referensi alamat yang diberikan ke sistem operasi ilegal atau dengan kata lain halaman yang ingin diakses tidak ada (tidak berada di disk), maka proses akan dihentikan. Namun jika referensi alamatnya adalah legal maka halaman yang diinginkan akan diambil dari disk.
- Halaman yang diinginkan akan dibawa dari disk ke memori utama (memori fisik).
- Tabel halaman akan diatur ulang lagi sesuai dengan kondisi yang baru. Jika tidak terdapat ruang kosong (*free frame*) di memori utama (fisik) untuk menaruh halaman yang baru maka dilakukan penggantian halaman dengan memilih salah satu halaman pada memori utama untuk digantikan dengan halaman yang baru tersebut. Penggantian halaman dilakukan dengan menggunakan algoritma tertentu. 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 utama (fisik) maka proses dapat diulang kembali. Dengan demikian proses sudah bisa mengakses halaman karena halaman telah diletakkan ke memori utama (fisik).

Perlu diingat bahwa status (register, *condition code*, counter insruksi) dari proses yang diinterupsi ketika terjadi *page fault* akan disimpan sehingga proses dapat diulang kembali di tempat dan status yang sama, kecuali jika halaman yang diinginkan sekarang telah berada di memori dan dapat diakses.

Pada berbagai kasus yang terjadi, ada tiga komponen yang akan dihadapi pada saat melayani *page fault*:
- Melayani interupsi *page fault*
- Membaca halaman
- Mengulang kembali proses

Gambar 5.3. Langkah-Langkah dalam Menangani Page Fault



5.4. Kinerja

Dalam proses *demand paging*, jika terjadi *page fault* maka diperlukan waktu yang lebih lambat untuk mengakses memori daripada jika tidak terjadi *page fault*. Hal ini dikarenakan perlu adanya penanganan *page fault* itu sendiri. Kinerja *demand paging* ini dapat dihitung dengan menggunakan *effective access time* yang dirumuskan sebagai berikut:

effective access time = (1 - *p*) x *ma* + *p* x *page fault time*

ma adalah *memory access time*, yang pada umumnya berkisar antara 10 hingga 200 nanosecond. *p* adalah probabilitas terjadinya *page fault*, yang berkisar antara 0 hingga 1. Jika *p* sama dengan 0 yang berarti bahwa tidak pernah terjadi *page fault*, maka *effective access time* akan sama dengan *memory access time*, dan itulah yang diharapkan. Sedangkan jika *p* sama dengan 1, yang berarti bahwa semua halaman mengalami *page fault*, maka *effective access time*-nya akan semaikin meningkat.

Untuk mendapatkan *effective access time*, kita harus mengetahui waktu yang diperlukan untuk menangani *page fault*. Komponen-komponen dalam penanganan *page fault* terdiri dari tiga kelompok besar, yaitu melayani interupsi dari *page fault*, membaca halaman, dan mengulang kembali proses.

Penggunaan *effective access time* dapat ditunjukkan dalam contoh berikut.

Contoh 5.1. Contoh penggunaan *effective address*

Diketahui waktu pengaksesan memori (ma) sebesar 100 ns. Waktu page fault sebesar 20 ms. Maka *effective access time* = (1 - p) x ma + p x *page fault time* = (1 - p) x 100 + p x 20000000 = 100 - 100p + 20000000p = 100 + 19.999.900p nanosecond

Pada *demand paging*, diusahakan agar kemungkinan terjadinya *page fault* rendah, karena bila *effective access time*-nya meningkat, maka proses akan berjalan lebih lambat.

5.5. *Copy-on-Write*

Pada pembahasan sebelumnya dijelaskan bahwa memori virtual memungkinkan proses untuk saling berbagi pakai memori. Proses ini adalah proses untuk berbagi pakai halaman (*page sharing*) memori virtual. Karena setiap proses membutuhkan halaman tersendiri, maka dibutuhkan teknik untuk mengaturnya. Teknik yang digunakan untuk mengoptimasi pembuatan dan penggunaan halaman adalah teknik *copy-on-write*, atau yang biasa disingkat dengan COW.

Pembuatan proses baru dengan menggunakan sistem call fork() menciptakan proses anak sebagai duplikat dari proses induknya. Setelah berhasil menciptakan proses anak, kemudian proses anak tersebut langsung memanggil sistem call exec(), yang berarti bahwa proses anak juga menduplikasi ruang alamat yang dimiliki proses induknya, beserta halaman yang diaksesnya. Padahal, hasil kopian dari halaman tersebut belum tentu berguna, yaitu jika tidak ada proses modifikasi pada halaman tersebut. Akan tetapi, dengan menggunakan teknik *copy-on-write* maka proses anak dan induk dapat bersama-sama menggunakan (mengakses) halaman yang sama.

Suatu halaman yang diakses secara bersama-sama (*shared*) oleh beberapa proses ditandai dengan COW (*copy-on-write*) jika suatu proses ingin memodifikasi (menulis) suatu halaman. Dan apabila hal tersebut terjadi, maka akan dibuat salinan dari halaman yang di-*shared* tersebut.

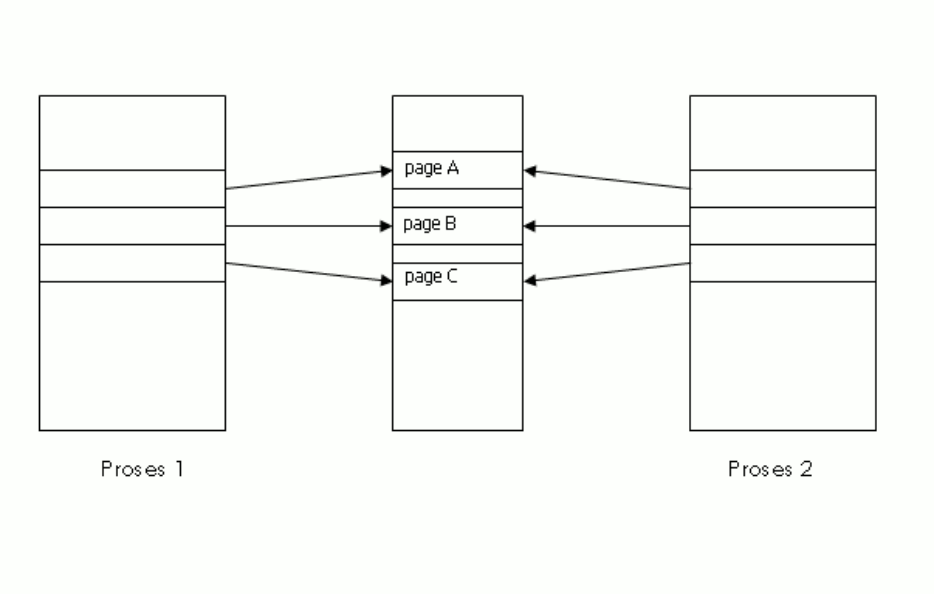
Sebagai contoh, sebuah proses anak akan memodifikasi suatu halaman yang terdiri dari sebagian dari *stack*. Sistem operasi akan mengenali halaman ini sebagai halaman *copy-on-write*. Sistem operasi kemudian akan membuat salinan dari halaman ini dan memetakannya kepada ruang alamat yang dimiliki proses anak. Proses anak kemudian memodifikasi halaman salinan yang telah berada di ruang alamat proses anak tersebut. Pada saat teknik *copy-on-write* ini digunakan, hanya halaman yang bisa dimodifikasi (oleh proses anak atau proses induk) saja yang disalin, sedangkan halaman yang tidak dimodifikasi dapat dibagi (di-*share*) untuk proses induk dan proses anak. Sebagai catatan, bahwa hanya halaman yang dapat dimodifikasi saja yang ditandai sebagai *copy-on-write*, sedangkan halaman yang tidak dapat dimodifikasi (misalnya halaman yang terdiri dari kode-kode yang bisa dieksekusi) tidak perlu ditandai karena tidak akan terjadi modifikasi pada halaman tersebut.

Pada banyak sistem operasi, disediakan sebuah *pool* yang terdiri dari halaman-halaman yang kosong untuk meletakkan halaman hasil duplikasi dengan teknik *copy-on-write*. Selain untuk meletakkan

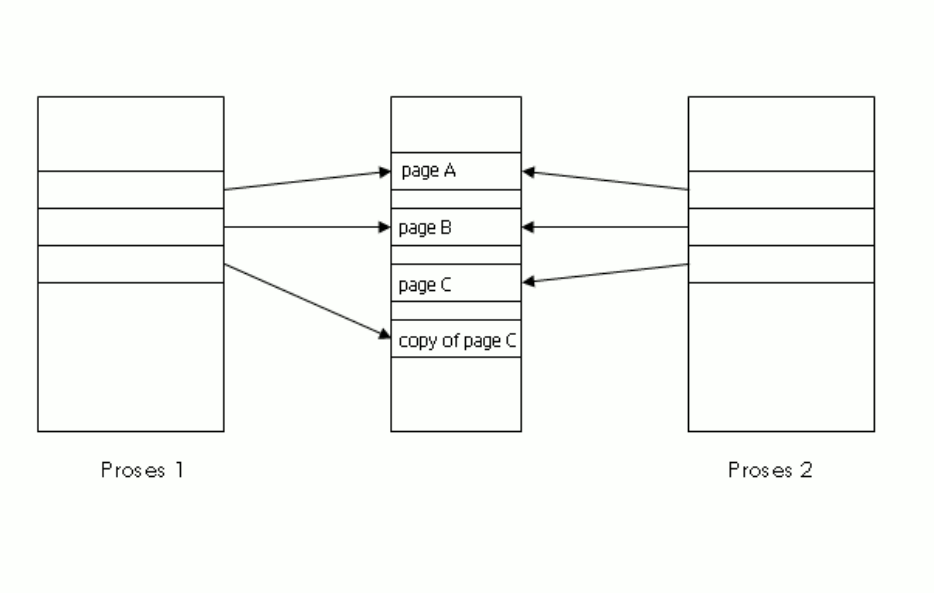
halaman hasil duplikasi tersebut, *pool* ini juga digunakan pada saat sebuah proses mengalami penambahan *stack* atau *heap*. Teknik yang digunakan sistem operasi untuk menyediakan halaman kosong tersebut dikenal dengan *zero-fill-on-demand*. Teknik ini dilakukan dengan mengosongkan halaman-halaman sebelum digunakan oleh proses yang baru.

Copy-on-write dapat diilustrasikan pada gambar 4 dan 5.

Gambar 5.4. Sebelum modifikasi pada page C



Gambar 5.5. Setelah modifikasi pada page C



5.6. Dasar Penggantian Halaman

Pada pembahasan mengenai masalah *page-fault*, diasumsikan bahwa setiap halaman minimal mengalami satu kali *page fault*, yaitu pada saat diakses pertama kali. Akan tetapi, tidak semua halaman tersebut akan digunakan oleh suatu proses. Jika terdapat sebuah proses yang memiliki sepuluh halaman, dan hanya menggunakan setengah di antaranya, yaitu lima halaman, maka *demand paging* menyimpan kelima proses yang tidak dibutuhkan tersebut agar tidak diakses oleh M/K. Dengan begitu,

kita dapat meningkatkan *degree of multiprogramming*, yaitu dengan menjalankan proses dua kali lebih banyak. Jika kita memiliki empat puluh bingkai, kita dapat menjalankan delapan proses. Bandingkan dengan jika kesepuluh halaman tersebut dipanggil, maka hanya dapat dijalankan maksimum empat proses.

Jika kita meningkatkan *degree of multiprogramming*, yaitu dengan menjalankan proses lebih banyak, maka dapat terjadi *over-allocating memory*. Misalnya kita menjalankan enam proses yang masing-masing memiliki sepuluh halaman dan seluruhnya dipanggil (di-*load*) ke memori, maka akan dibutuhkan 60 bingkai, padahal yang tersedia hanya empat puluh bingkai. *Over-allocating memory* juga dapat terjadi jika terdapat *page fault*, yaitu pada saat sistem operasi mendapatkan halaman yang dicari pada disk kemudian membawanya ke memori fisik tetapi tidak terdapat bingkai yang kosong pada memori fisik tersebut.

Sistem operasi memiliki dua cara untuk menangani masalah ini. Yang pertama dengan men-terminasi proses yang sedang mengakses halaman tersebut. Akan tetapi, cara ini tidak dapat dilakukan karena *demand paging* merupakan usaha sistem operasi untuk meningkatkan utilisasi komputer dan *throughput*-nya.

Cara yang kedua yaitu dengan penggantian halaman (*page replacement*). Sistem operasi dapat memindahkan suatu proses dari memori fisik, lalu menghapus semua bingkai yang semula digunakannya, dan mengurangi *level of multiprogramming* (dengan mengurangi jumlah proses yang berjalan).

Prinsip kerja penggantian halaman adalah sebagai berikut. "Jika tidak ada bingkai yang kosong, maka dicari (dengan suatu algoritma ganti halaman) salah satu bingkai yang sedang tidak digunakan dan kemudian dikosongkar. Suatu bingkai dapat dikosongkan dengan memindahkan isinya ke dalam ruang pemindahan kemudian mengubah semua tabel halaman hingga mengindikasikan bahwa halaman yang dipindah tersebut sudah tidak berada di memori fisik. Lalu bingkai yang telah kosong tersebut dapat digunakan oleh halaman yang akan ditempatkan di memori fisik".

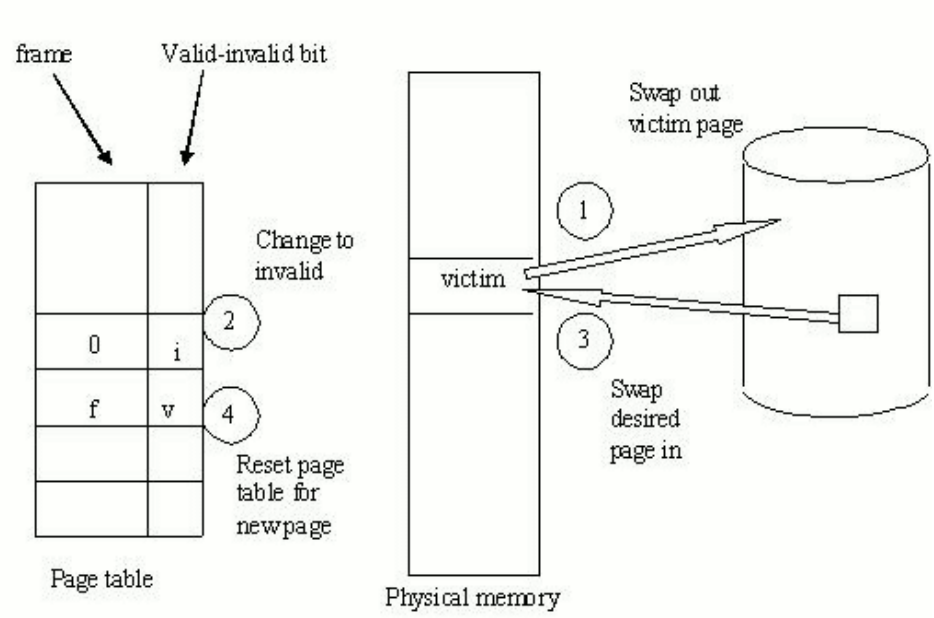
Dengan memodifikasi urutan penanganan *page fault*, maka dapat dijabarkan urutan proses *page replacement* sebagai berikut.

1. Mencari lokasi dari halaman yang dicari di disk.
2. Mencari bingkai yang kosong di memori fisik:
   a. Jika ada bingkai yang kosong, maka gunakan bingkai tersebut.
   b. Jika tidak ada bingkai yang kosong, gunakan algoritma ganti halaman untuk memilih bingkai "korban"
   c. Pindahkan bingkai "korban" tersebut ke disk dan sesuaikan tabel halaman.
3. Masukkan halaman yang berasal dari disk tersebut ke dalam bingkai yang baru dikosongkan tersebut. Sesuaikan tabel halaman.
4. Lanjutkan proses yang telah diinterupsi.

Dari penjelasan di atas, maka dapat disimpulkan bahwa jika tidak terdapat bingkai yang kosong maka terdapat dua transfer halaman (yang keluar dan masuk memori fisik). Hal ini tentu saja menambah waktu dalam penanganan *page fault* dan sceara otomatis menambah *effective access time*.

Hal tersebut dapat diselesaikan dengan menggunakan bit modifikasi (*modify bit*/*dirty bit*). Setiap halaman atau bingkai memiliki bit modifikasi yang sesuai pada perangkat keras. Bit modifikasi untuk sebuah halaman diatur oleh perangkat keras pada saat suatu byte atau word dituliskan ke halaman tersebut, yang menunjukan bahwa halaman tersebut telah dimodifikasi. Waktu suatu halaman dipilih untuk dipindahkan dari memori fisik ke disk, diperiksa terlebih dahulu bit modifikasinya di disk. Jika bit modifikasinya ada, maka halaman tersebut harus ditulis ke disk. Namun, apabila bit modifikasinya belum ada di disk, maka halaman tersebut belum dimodifikasi karena halaman tersebut masih berada di memori utama. Oleh karena itu, jika salinan dari halaman tersebut masih terdapat di disk (belum ditimpa oleh halaman lain) maka penulisan halaman dari memori utama ke disk tidak diperlukan. Hal ini juga berlaku pada halaman *read-only*, yaitu halaman yang tidak dapat dimodifikasi. Sehingga waktu yang diperlukan untuk penanganan *page fault* dapat berkurang dengan cukup signifikan karena berkurangnya waktu M/K dari dan ke disk.

Gambar 5.6. Page Replacement



5.7. Rangkuman

Memori virtual adalah teknik yang memisahkan antara alamat memori logis dengan alamat memori fisik. Hal tersebut berguna agar pengguna (*programmer*) tidak perlu menentukan alamat fisik dari program yang dijalankan. Memori vitual memungkinkan beberapa proses berjalan dengan alamat memori fisik yang terbatas. Teknik permintaan halaman (*demand paging*) digunakan untuk mengimplementasikan konsep memori virtual. Jika halaman yang diminta tidak terdapat pada memori utama, maka akan terjadi *page fault*. *Page fault* ini dapat ditangani dengan beberapa tahapan. Dengan adanya *page fault* ini, maka kinerja *demand paging* dapat dihitung berdasarkan *memory access time* dan *page fault time* (waktu yang dibutuhkan dalam penanganan *page fault*). Kinerja *demand paging* ini biasa disebut dengan *effective access time*.

Pada pembuatan suatu proses baru (proses anak), maka baik proses induk maupun proses anak dapat mengakses suatu halaman yang sama tanda perlu membuat salinannya terlebih dahulu, yaitu dengan teknik *copy-on-write*. Jika proses anak hendak memodifikasi halaman tersebut, maka baru akan dibuatkan salinan dari halaman tersebut untuk kemudian dimodifikasi oleh proses anak. Halaman yang disalin tersebut dinamakan halaman *copy-on-write* .

Jika ada suatu halaman diminta/dibutuhkan oleh suatu proses dan ternyata halaman tersebut terdapat di disk, maka halaman tersebut akan dipindahkan ke memori utama. Namun, jika di memori utama tidak lagi terdapat bingkai yang kosong (*free frame*) untuk ditempati oleh halaman tersebut, maka akan terjadi penggantian halaman (*page replacement*) dengan memilih suatu bingkai pada memori dan menggantikan isinya dengan halaman tersebut. Pada pemilihan suatu bingkai ini, dibutuhkan suatu algoritma penggantian halaman.

Rujukan

[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.

[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.

[WEBKUSJOKO2004] Kuspriyanto dan Putut Joko Wibowo. 2004. *Desain Memori Virtual Pada Mikroarsitektur PowerPC, MIPS, Dan X86 http://www.geocities.com/transmisi_eeundip/kuspriyanto.pdf* . Diakses 28 Maret 2007.

Bab 6. Algoritma Ganti Halaman

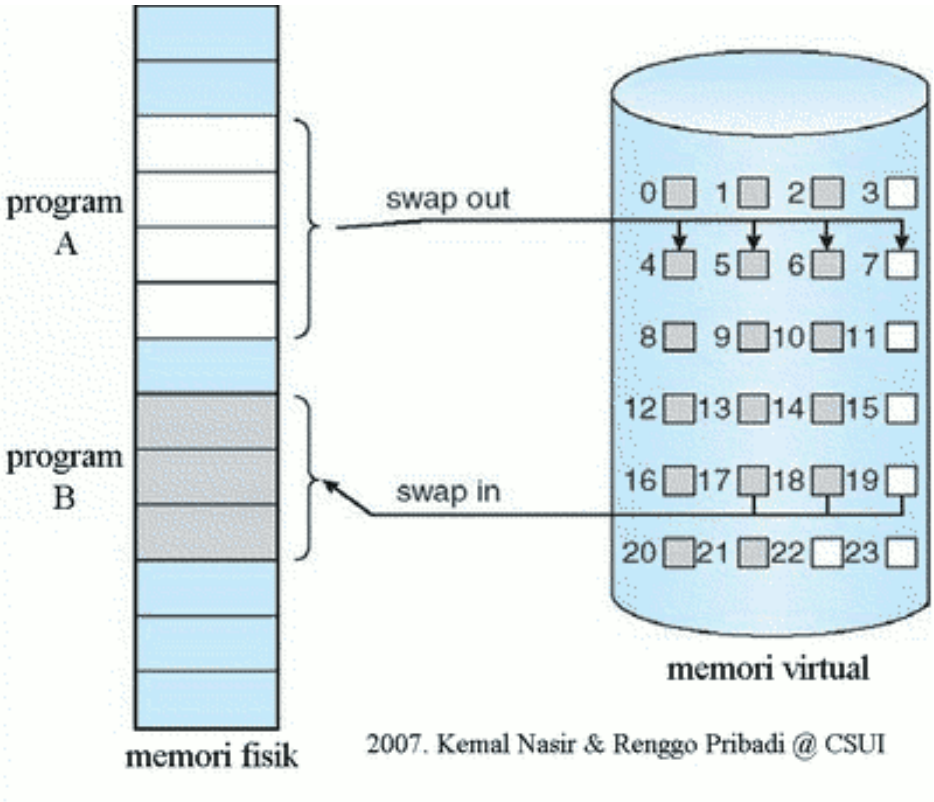
6.1. Pendahuluan

Ganti halaman dilakukan apabila terjadi *page fault*. *Page fault* bukan suatu jenis *error* yang fatal, *page fault* terjadi apabila ada halaman yang ingin diakses tetapi halaman tersebut tidak terdapat di dalam memori utama. *Page fault* pasti terjadi minimal satu kali saat pertama kali halaman itu ingin diakses.

Prinsip ganti halaman adalah sebagai berikut:

a. Proses meminta halaman tertentu.
b. Jika halaman berada di memori, tidak dilakukan ganti halaman.
c. Jika halaman tidak berada di memori, maka:
    a. Jika ada frame kosong, maka halaman itu di-*load* ke dalam frame yang kosong tersebut.
    b. Jika tidak ada frame yang kosong, maka pilih halaman yang akan di-*swap* dengan menggunakan algoritma ganti halaman.
d. *Update* tabel halaman dan table memori.
e. *Restart* proses.

Gambar 6.1. Ilustrasi *Swapping*



Semakin banyak dilakukan *swap*, semakin sibuk pula CPU mengurus hal ini. Bila berkelanjutan, maka akan terjadi *thrashing*. *Thrashing* adalah keadaan di mana banyak terjadi *page fault*, sehingga mengakibatkan utilisasi CPU menurun drastis karena lebih sibuk mengurusi pergantian halaman daripada mengurusi proses.

Untuk menghindari hal ini, diperlukan pemilihan algoritma ganti halaman yang baik. Kriteria algoritma yang baik adalah:

- Menyebabkan *page fault rate* yang rendah.
- Tidak menyebabkan *thrashing* .
- Tidak terlalu sulit untuk diimplementasikan.

Pada umumnya, semakin besar memori, semakin banyak pula jumlah *frame*-nya. Semakin banyak *frame*, semakin banyak pula jumlah halaman yang bisa masuk di memori, sehingga *page fault rate* menurun.

6.2. *Reference String*

Reference string adalah *string* yang merepresentasikan halaman-halaman yang ingin digunakan/ di-*load* . Kegunaannya adalah untuk menyederhanakan alamat dan mempermudah melihat *page fault rate* yang terjadi serta mensimulasikan algoritma ganti halaman. Biasanya *reference string* berisi kumpulan alamat-alamat halaman yang dikelompokkan berdasarkan aturan reduksi *reference string*. Bila pereduksian alamat sebanyak 1000 bytes, maka alamat-alamat yang berurutan sebanyak 1000 *bytes* diwakili oleh satu buah *reference string*. Misalnya 0003, 0563, 0094 diwakili oleh *reference string* 0. Demikian juga 1745, 1003, 1999 diwakili oleh *reference string* 1 dan seterusnya.

Contoh:

Urutan alamat yang digunakan oleh sebuah proses adalah 0301, 0213, 0312, 0321, 0341, 0421, 0431, 0132, 0431, 0152. Maka, *reference string*-nya dengan reduksi 100 *bytes* adalah 3, 2, 3, 4, 1, 4, 1.

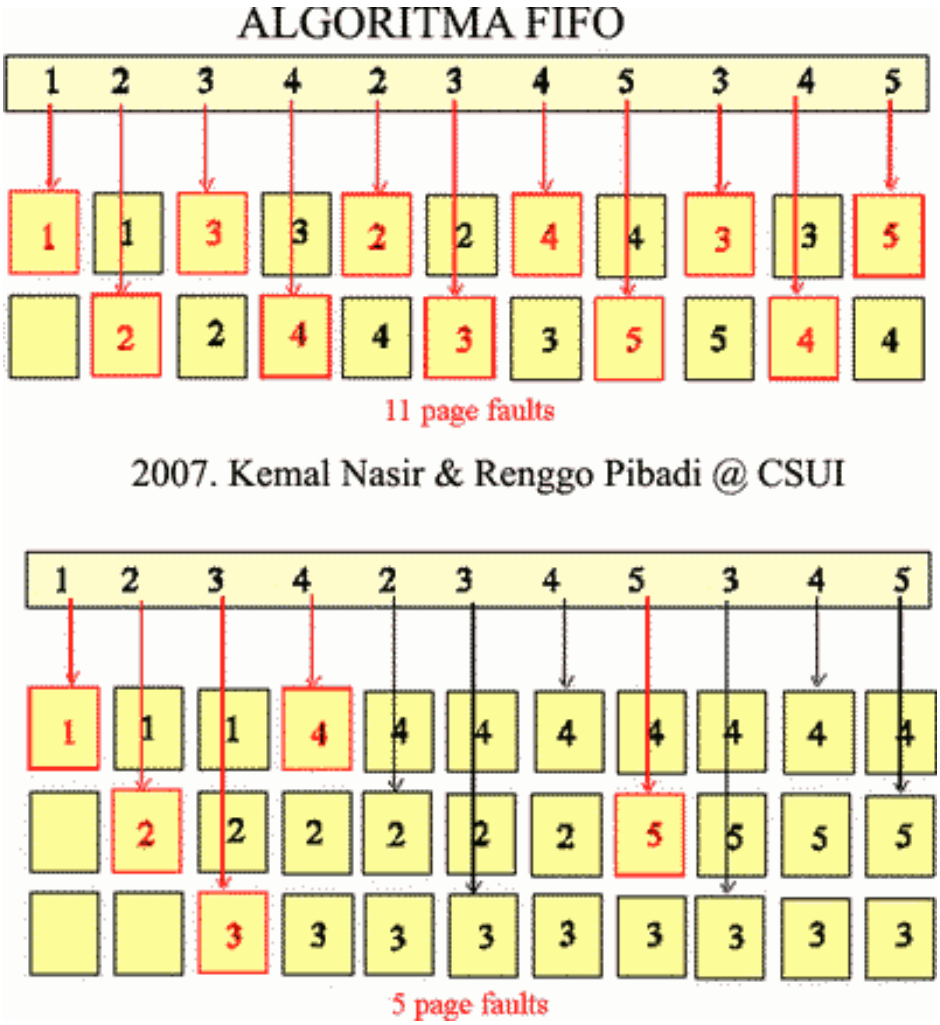
Bagaimana cara men-*generate* sebuah *reference string* dari urutan alamat? *Reference string* dihasilkan dari bit tertentu dari sebuah alamat (biasanya bit kedua dari kiri, yang berarti direduksi 100 *bytes*), maka alamat 0431 menjadi 4, 0241 menjadi 2, dan 0252 menjadi 2.

Apabila terdapat urutan alamat yang string acuannya sama berturut-turut, misalnya 0431 dan 0452, maka tetap ditulis sebagai 4, karena tidak me-*load* halaman yang baru.

6.3. Algoritma FIFO *(First In First Out)*

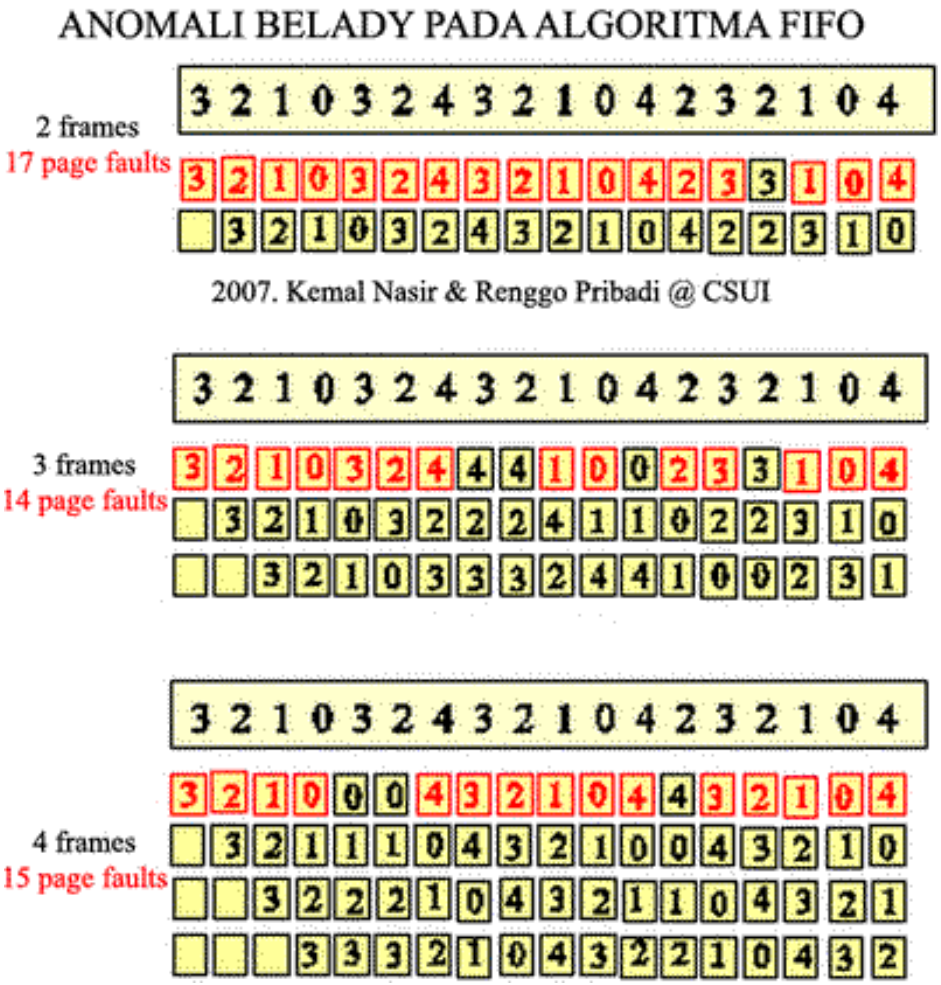
Algoritma ini adalah algoritma yang paling sederhana. Prinsip dari algoritma ini adalah seperti prinsip antrian (antrian tak berprioritas), halaman yang masuk lebih dulu maka akan keluar lebih dulu juga. Algoritma ini menggunakan struktur data *stack*. Apabila tidak ada *frame* kosong saat terjadi *page fault*, maka korban yang dipilih adalah *frame* yang berada di *stack* paling bawah, yaitu halaman yang berada paling lama berada di memori.

Gambar 6.2. Algoritma FIFO



Pada awalnya, algoritma ini dianggap cukup mengatasi masalah tentang pergantian halaman, sampai pada tahun 70-an, Belady menemukan keanehan pada algoritma ini yang dikenal kemudian dengan anomali Belady. Anomali Belady adalah keadaan di mana *page fault rate* meningkat seiring dengan pertambahan jumlah *frame* , seperti yang bisa dilihat pada contoh di bawah ini.

Gambar 6.3. Anomali Algoritma FIFO

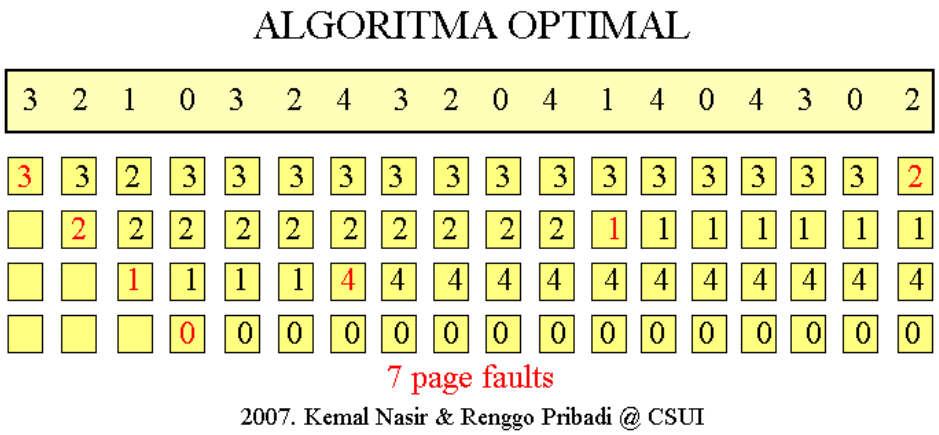


Ketika jumlah *frame* ditambah dari 3 *frame* menjadi 4 *frame*, jumlah *page fault* yang terjadi malah bertambah (dari 14 *page fault* menjadi 15 *page fault*). Hal ini biasanya terjadi pada kasus yang menginginkan halaman yang baru saja di-*swap-out* sebelumnya. Oleh karena itu, dicarilah algoritma lain yang mampu lebih baik dalam penanganan pergantian halaman seperti yang akan dibahas berikut ini.

6.4. Algoritma Optimal

Algoritma ini adalah algoritma yang paling optimal sesuai namanya. Prinsip dari algoritma ini adalah mengganti halaman yang tidak akan terpakai lagi dalam waktu lama, sehingga efisiensi pergantian halaman meningkat (*page fault* yang terjadi berkurang) dan terbebas dari anomali Belady. Algoritma ini memiliki *page fault rate* paling rendah di antara semua algoritma di semua kasus. Akan tetapi, optimal belum berarti sempurna karena algoritma ini ternyata sangat sulit untuk diterapkan. Sistem tidak dapat mengetahui halaman-halaman mana saja yang akan digunakan berikutnya.

Gambar 6.4. Algoritma Optimal

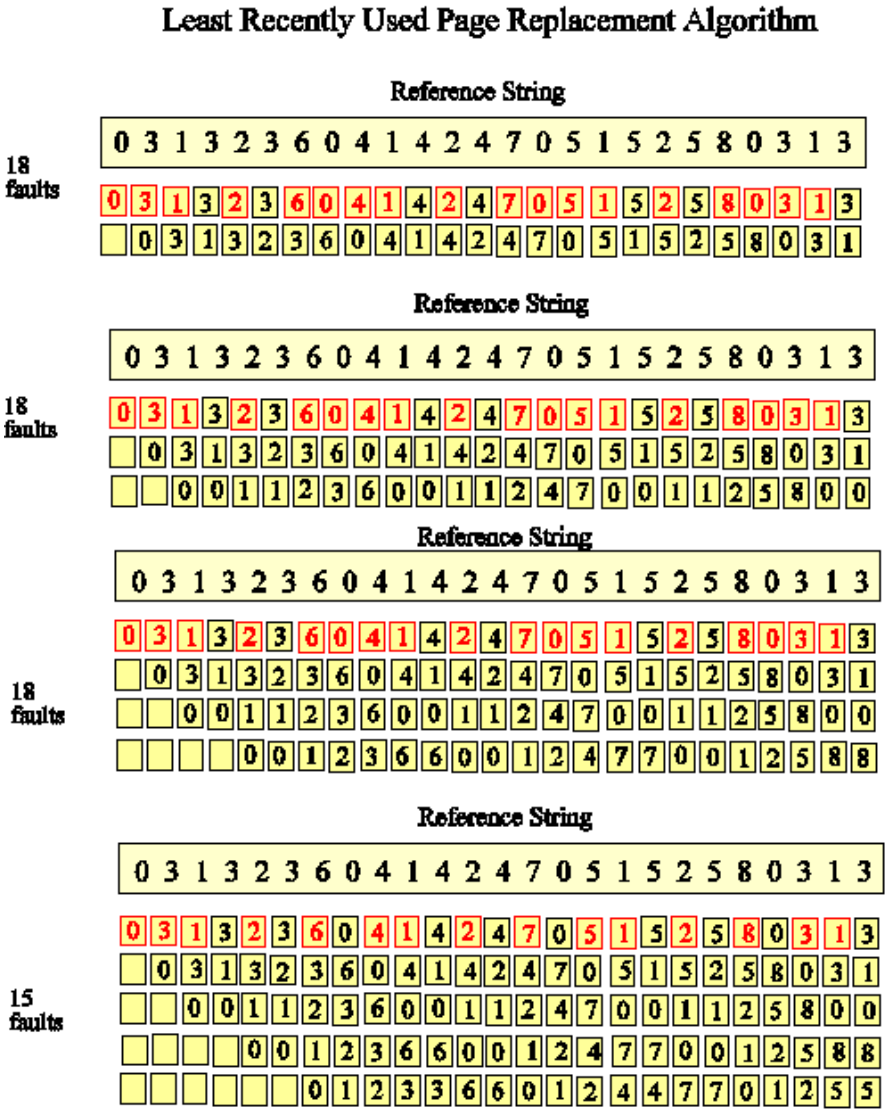


6.5. Algoritma LRU *(Least Recently Used)*

Dikarenakan algoritma optimal sangat sulit dalam pengimplementasiannya, maka dibuatlah algoritma lain yang *performance*-nya mendekati algoritma optimal dengan sedikit *cost* yang lebih besar. Algoritma ini mengganti halaman yang paling lama tidak dibutuhkan. Asumsinya, halaman yang sudah lama tidak digunakan sudah tidak dibutuhkan lagi dan kemungkinan besar, halaman yang baru di-*load* akan digunakan kembali.

Sama seperti algoritma optimal, algoritma LRU tidak mengalami anomali Belady. Algoritma ini memakai *linked list* untuk mendata halaman mana yang paling lama tidak terpakai. *Linked list* inilah yang membuat *cost* membesar, karena harus meng-*update linked list* tiap saat ada halaman yang di akses. Halaman yang berada di *linked list* paling depan adalah halaman yang baru saja digunakan. Semakin lama tidak dipakai, halaman akan berada semakin belakang dan di posisi terakhir adalah halaman yang paling lama tidak digunakan dan siap untuk di-*swap*.

Gambar 6.5. Algoritma LRU



6.6. Implementasi LRU

Ada beberapa cara untuk mengimplementasikan algoritma LRU. Tetapi, yang cukup terkenal ada 2, yaitu *counter* dan *stack*. Contoh algoritma di atas menggunakan *stack*.

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. Semakin sering halaman itu diakses, semakin besar pula nilai *counter*-nya dan sebaliknya. Untuk melakukan hal itu dibutuhkan *extra write* ke memori. Selain berisi halaman-halaman yang sedang di-*load*, memori juga berisi tentang *counter* masing-masing halaman. Halaman yang diganti adalah halaman yang memiliki nilai *clock* terkecil, yaitu halaman yang paling jarang diakses. Kekurangan dari cara ini adalah memerlukan dukungan tambahan *counter* pada *hardware*.

***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 seluruh isi *stack* harus di-*update* setiap kali mengakses halaman, sedangkan dengan *counter*, yang dirubah hanya *counter* halaman yang sedang diakses, tidak perlu mengubah *counter* dari semua halaman yang ada.

Gambar 6.6. Algoritma LRU dengan Stack

6.7. Algoritma Lainnya

Sebenarnya masih banyak algoritma ganti halaman yang lain selain 3 algoritma utama yang telah dibahas sebelumnya (utama bukan berarti paling sering dipakai). Berikut ini adalah 2 contoh algoritma lain yang juga cukup popular dan mudah diimplementasikan.

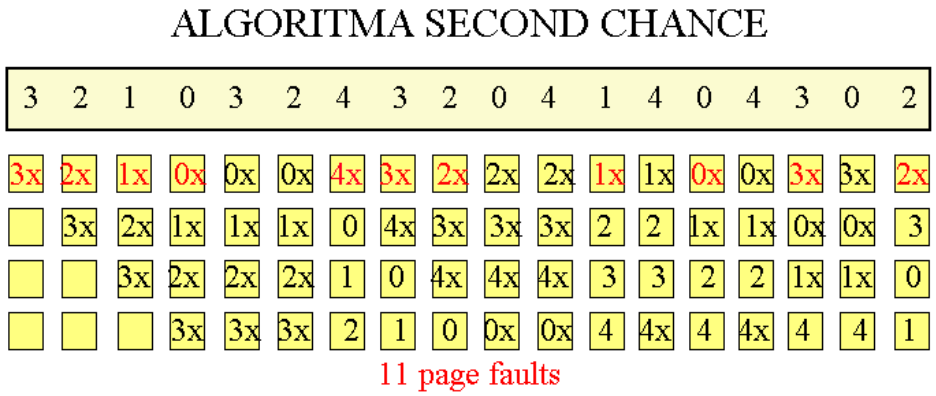
Algoritma yang pertama adalah algoritma *second chance*. Algoritma *second chance* berdasarkan pada algoritma FIFO yang disempurnakan. Algoritma ini menggunakan tambahan berupa *reference* bit yang nilainya 0 atau 1. Jika dalam FIFO menggunakan *stack* , maka *second chance* menggunakan *circular queue* . Halaman yang baru di-*load* atau baru digunakan akan diberikan nilai 1 pada *reference* bit-nya. Halaman yang *reference* bit-nya bernilai 1 tidak akan langsung diganti walaupun dia berada di antrian paling bawah (berbeda dengan FIFO).

Urutan langkah kerja algoritma *second chance* adalah sebagai berikut:

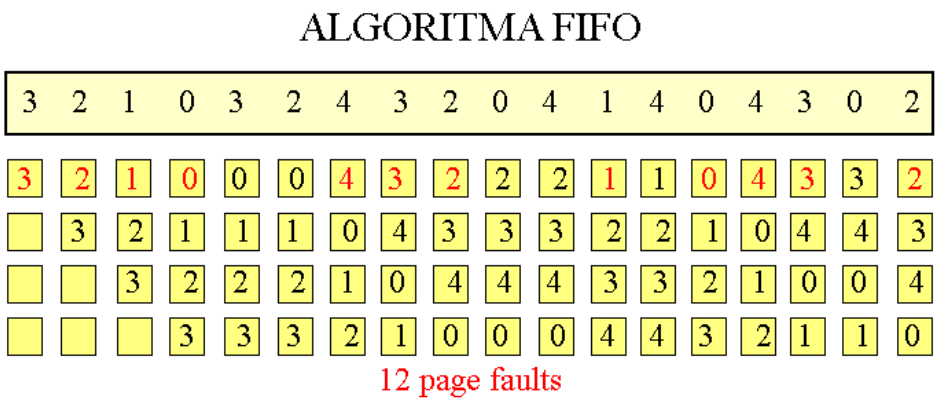
- Apabila terjadi *page fault* dan tidak ada *frame* yang kosong, maka akan dilakukan razia (pencarian korban) halaman yang *reference* bit-nya bernilai 0 dimulai dari bawah antrian (seperti FIFO).
- Setiap halaman yang tidak di- *swap* (karena *reference* bit-nya bernilai 1), setiap dilewati saat razia *reference* bit-nya akan diset menjadi 0.
- Apabila ditemukan halaman yang *reference* bit-nya bernilai 0, maka halaman itu yang di-*swap*.
- Apabila sampai di ujung antrian tidak ditemukan halaman yang *reference* bit-nya bernilai 0, maka razia dilakukan lagi dari awal.

Pada gambar di bawah ini, akan diilustrasikan algoritma *second chance* dan algoritma FIFO sebagai pembanding.

Gambar 6.7. Algoritma *Second Chance*



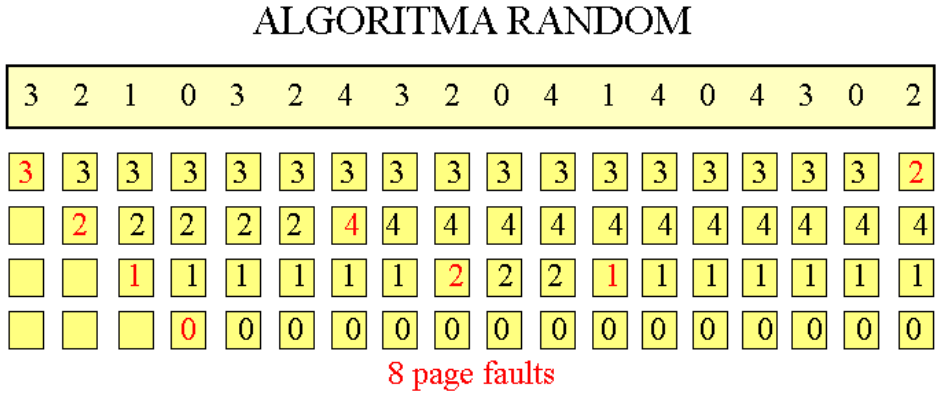
Gambar 6.8. Algoritma *FIFO*



Algoritma kedua adalah algoritma *random*. Algoritma *random* adalah algoritma yang cukup sederhana juga selain algoritma FIFO. Dalam algoritma ini, halaman yang dipilih menjadi korban dipilih secara acak.

Meskipun terdengar asal, tetapi algoritma ini relatif *low cost*, karena tidak memerlukan *stack*, *queue* atau *counter*. Dibandingkan dengan FIFO, rata-rata kasus menunjukkan *page fault rate* algoritma *random* lebih rendah daripada algoritma FIFO. Sedangkan dibandingkan dengan LRU, algorima *random* ini lebih unggul dalam hal *memory looping reference* , karena algoritma *random* sama sekali tidak memerlukan *looping*.

Gambar 6.9. Algoritma *Random*



6.8. Rangkuman

Page fault terjadi apabila terdapat halaman yang ingin diakses tetapi halaman tersebut tidak terdapat di dalam memori utama.

Jika terjadi *page fault* dan tidak ada *frame* yang kosong, maka dipilih *frame* tumbal yang akan di-*swap*.

Pemilihan halaman dilakukan dengan algoritma ganti halaman. Algoritma dipilih yang paling rendah *page fault rate*-nya dan tidak sulit untuk diimplementasikan.

Contoh algoritma ganti halaman:
- Algoritma FIFO
- Algoritma Optimal
- Algoritma LRU
- Algoritma *Second Chance*
- Algoritma *Random*

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.

[WEBWiki2007] Wikipedia. 2007. *Page Replacement Algortihm http://en.wikipedia.org/wiki/ Page_replacement_algorithm* . Diakses 4 April 2007.

Bab 7. Strategi Alokasi Bingkai

7.1. Pendahuluan

Setiap proses perlu mendapat alokasi memori agar proses tersebut dapat dieksekusi dengan baik. Masalah selanjutnya adalah bagaimana caranya untuk mengalokasikan memori bagi setiap proses yang ada. Saat proses akan dieksekusi, terjadi *page fault* sehingga sistem akan menggantinya dengan halaman di memori. Untuk melakukan penggantian ini diperlukan bingkai yang terdapat di sistem. Proses dapat menggunakan setiap bingkai yang sedang bebas di sistem. Hal ini mengakibatkan perlu adanya pengaturan lebih lanjut agar tiap proses bisa mendapatkan bingkai yang cukup untuk melakukan penggantian ini.

7.2. Jumlah Bingkai

Hal yang perlu diperhatikan dalam strategi alokasi bingkai adalah berapa jumlah bingkai yang harus dialokasikan pada proses tersebut. Jumlah bingkai yang dialokasikan tidak boleh melebihi jumlah bingkai yang tersedia. Hal lain yang perlu diperhatikan adalah jumlah bingkai minimum yang harus dialokasikan agar instruksi dapat dijalankan, karena jika terjadi kesalahan halaman sebelum eksekusi selesai, maka instruksi tersebut harus diulang. Sehingga jumlah bingkai yang cukup harus tersedia untuk menampung semua halaman yang dibutuhkan oleh sebuah instruksi.

7.3. Strategi Alokasi Bingkai

Ada dua jenis algoritma yang biasa digunakan untuk pengalokasian bingkai, yaitu:

1. **Algoritma *Fixed Allocation* .** Algoritma *fixed allocation* dibedakan menjadi dua macam yaitu *equal allocation* dan *proportional allocation*. Pada algoritma *equal allocation* jumlah bingkai yang diberikan pada setiap proses jumlahnya sama (m/n bingkai, m = jumlah bingkai, n = jumlah proses), misalnya: ada 5 buah proses dan 100 bingkai tersisa, maka tiap proses akan mendapatkan 20 bingkai. Algoritma ini kurang baik digunakan jika proses-proses yang ada besarnya berbeda-beda (proses yang besar diberikan bingkai yang sama dengan proses yang kecil), misalnya: ada 2 buah proses sebesar 10 K dan 127 K, ada 64 bingkai bebas. Jika kita memberikan bingkai yang sama yaitu sebesar 32 untuk tiap proses maka misalnya saja proses satu ternyata hanya memerlukan 10 bingkai, dan alhasil 22 bingkai pada proses pertama akan terbuang percuma. Untuk mengatasi masalah tersebut algoritma *proportional allocation*-lah yang cocok digunakan, yaitu pengalokasian bingkai disesuaikan dengan besarnya suatu proses, contoh:

   Si = besarnya proses Pi

   S = Σ Si

   m = jumlah total bingkai

   ai = alokasi bingkai untuk Pi ((Si/S) x m)

   m = 64

   S1 = 10

   S2 = 127

   a1 = (10/137) x 64 = 5 bingkai

   a2 = (127/137) x 64 = 59 bingkai

2. **Algoritma *Priority Allocation* .** Algoritma *priority allocation* merupakan algoritma pengalokasian dengan memberikan jumlah bingkai sesuai dengan prioritas proses tersebut.

Pendekatannya mirip dengan *proportional allocation*, perbandingan *frame*-nya tidak tergantung ukuran relatif dari proses, melainkan lebih pada prioritas proses atau kombinasi ukuran dan prioritas. Jika suatu proses mengalami *page fault* maka proses tersebut akan menggantinya dengan salah satu *frame* yang dimiliki proses tersebut atau menggantinya dengan *frame* dari proses yang prioritasnya lebih rendah. Dengan kedua algoritma di atas, tetap saja alokasi untuk tiap proses bisa bervariasi berdasarkan derajat *multiprogramming*nya. Jika *multiprogramming*nya meningkat maka setiap proses akan kehilangan beberapa frame yang akan digunakan untuk menyediakan memori untuk proses lain. Sedangkan jika derajat *multiprogramming*-nya menurun, *frame* yang sudah dialokasikan bisa disebar ke proses-proses lainnya.

7.4. Alokasi Global dan Lokal

Dalam pengalokasian bingkai, salah satu hal yang penting adalah penggantian halaman. Kita dapat mengklasifikasikan algoritma penggantian halaman ke dalam dua kategori:

1. **Penggantian Global.** Penggantian secara global memperbolehkan suatu proses mencari bingkai pengganti dari semua bingkai yang ada, meskipun bingkai tersebut sedang dialokasikan untuk proses lain. Hal ini memang efisien, tetapi ada kemungkinan proses lain tidak mendapatkan bingkai karena bingkainya terambil oleh proses lain.
2. **Penggantian Lokal.** Penggantian lokal hanya mengijinkan proses untuk mencari bingkai pengganti dari bingkai-bingkai yang memang dialokasikan untuk proses tersebut.

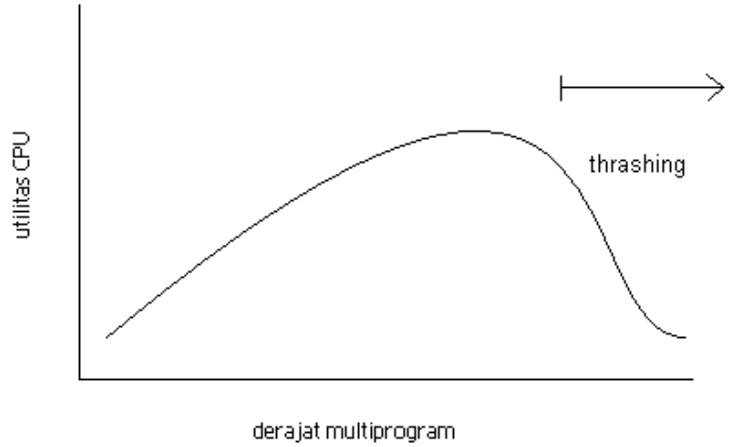
Pada algoritma penggantian lokal, jumlah bingkai yang dialokasikan pada suatu proses tidak akan berubah. Sedangkan pada algoritma penggantian global jumlah bingkai pada proses tersebut mungkin akan bertambah dengan asumsi proses lain tidak mengambil bingkai proses ini sebagai pengganti dari bingkai proses tersebut.

Masalah pada algoritma penggantian global adalah proses tidak dapat mengontrol *page fault rate* proses itu sendiri. Keunggulan algoritma ini adalah menghasilkan *system throughput* yang lebih bagus, oleh karena itu algoritma ini lebih sering dipakai.

7.5. *Thrashing*

Pada saat suatu proses tidak memiliki cukup bingkai untuk mendukung halaman yang akan digunakan maka akan sering terjadi *page fault* sehingga harus dilakukan penggantian halaman. *Thrashing* adalah keadaan dimana proses sibuk untuk mengganti halaman yang dibutuhkan secara terus menerus, seperti ilustrasi di bawah ini.

Gambar 7.1. *Thrashing*



Pada gambar terlihat *CPU utilization* meningkat seiring meningkatnya derajat *multiprogramming*, sampai pada suatu titik *CPU utilization* menurun drastis, di titik ini *thrashing* dapat dihentikan dengan menurunkan derajat *multiprograming*.

Pada saat *CPU utilization* terlalu rendah, maka sistem operasi akan meningkatkan derajat *multiprogramming* dengan cara menghasilkan proses-proses baru, dalam keadaan ini algoritma penggantian global akan digunakan. Ketika proses membutuhkan bingkai yang lebih, maka akan terjadi *page fault* yang menyebabkan *CPU utilization* semakin menurun. Ketika sistem operasi mendeteksi hal ini, derajat *multiprogramming* makin ditingkatkan, yang menyebabkan *CPU utilization* kembali menurun drastis, hal ini yang menyebabkan *thrashing*.

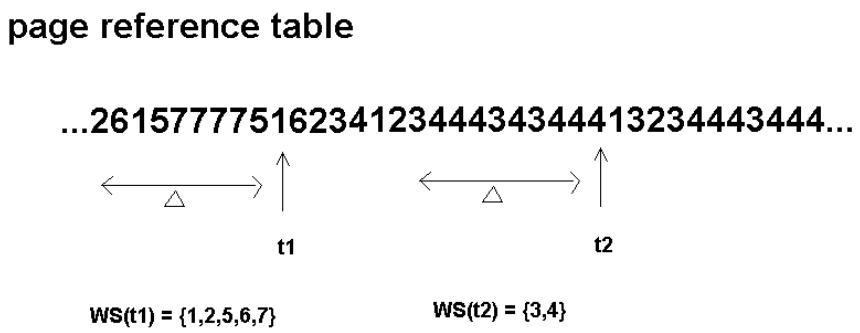
Untuk membatasi efek *thrashing* dapat menggunakan algoritma penggantian lokal. Dengan algoritma penggantian lokal, jika terjadi *thrashing*, proses tersebut dapat menggambil bingkai dari proses lain dan menyebabkan proses tersebut tidak mengalami *thrashing*. Salah satu cara untuk menghindari *thrashing* adalah dengan cara menyediakan jumlah bingkai yang pas sesuai dengan kebutuhan proses tersebut. Salah satu cara untuk mengetahui jumlah bingkai yang diperlukan pada suatu proses adalah dengan strategi *working set*.

7.6. *Working Set Model*

Salah satu cara menghindari *thrashing* adalah dengan menyediakan sebanyak mungkin bingkai sesuai dengan kebutuhan proses. Untuk mengetahui berapa bingkai yang dibutuhkan adalah dengan strategi *working set*. Strategi ini dimulai dengan melihat berapa banyak bingkai yang digunakan oleh suatu proses. *Working set* model mengatakan bahwa sistem hanya akan berjalan secara efisien jika proses diberikan bingkai yang cukup, jika bingkai tidak cukup untuk menampung semua proses maka suatu proses akan ditunda, dan memberikan halamannya untuk proses yang lain.

Working set model merupakan model lokalitas dari eksekusi proses. Model ini menggunakan parameter Δ (delta) untuk definisi *working set window*. Kumpulan dari halaman dengan halaman yang dituju yang paling sering muncul disebut *working set*.

Gambar 7.2. *Working Set Model*



Pada contoh gambar, terlihat bahwa dengan Δ = 10 *memory references*, maka *working set* pada t1 adalah {1,2,5,6,7} dan *working set* pada t2 adalah {3,4}.

Keakuratan *Working set* tergantung pada pemilihan Δ:

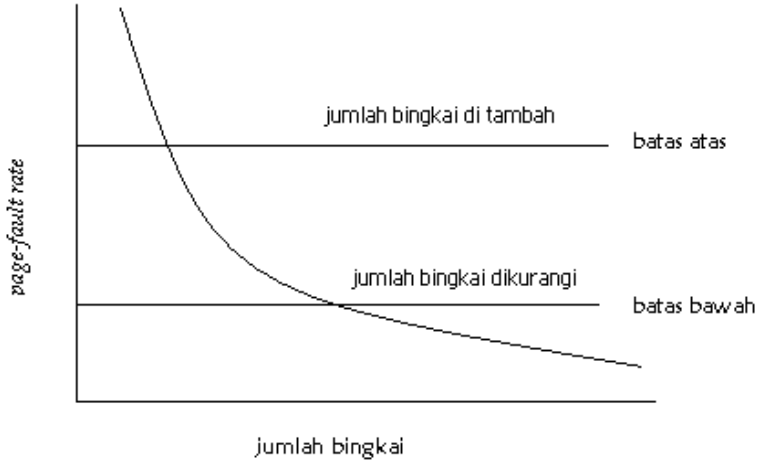
1. jika Δ terlalu kecil tidak akan mewakilkan seluruh lokalitas.
2. jika Δ terlalu besar menyebabkan *overlap.*
3. jika Δ tidak terbatas *working set* adalah kumpulan halaman sepanjang eksekusi program.

Jika total permintaan > total bingkai, maka akan terjadi *thrashing*. Jika ini terjadi maka proses yang sedang berjalan akan diblok.

7.7. *Page Fault*

Untuk mencegah *thrashing* maka kita harus mengatur tingkat *page fault* yang terjadi. Jika page fault sering terjadi maka dibutuhkan bingkai yang lebih banyak, jika *page fault* jarang terjadi maka bingkai yang ada terlalu banyak, maka diperlukan batasan untuk menentukan batas atas dan batas bawah dari frekuensi *page fault*. Jika melewati batas atas maka proses mendapat alokasi bingkai baru, jika melewati batas bawah maka bingkai akan didealokasi dari proses.

Gambar 7.3. *Page-Fault*



Dalam *working set* strategi jika melewati batas atas maka harus ada proses yang ditunda dan bingkai yang bebas akan distribusikan untuk proses dengan frekuensi *page fault* yang tinggi.

7.8. *Memory Mapped Files*

Mengakses file pada disk secara *sequential* menggunakan *system call* `open()`, `read()`, `write()`. Cara lain untuk mengakses *file* pada disk adalah dengan menggunakan memori virtual. Cara ini diberi nama *memory mapping* yang memperbolehkan sebagian memori virtual dihubungkan kepada *file*.

Memory-mapped file dapat dilakukan dengan memetakan blok dari disk ke halaman di memori. Proses membaca dan menulis *file* dapat dilakukan dengan akses ke memori sehingga lebih mudah dibandingkan dengan menggunakan *system call*.

Memodifikasi *file* yang dipetakan pada memori tidak harus langsung meng-*update* hasil modifikasi tersebut pada *file* di *disk*. Beberapa *system* meng-*update file* fisik jika sistem operasi menemukan halaman pada memori telah diubah. Hal ini dilakukan secara periodik oleh sistem operasi. Ketika *file* ditutup maka semua data pada memori ditulis ke *disk* dan dibuang dari memori virtual.

Pada beberapa sistem operasi pemetaan memori menggunakan *system call* yang khusus sedangkan untuk menjalankan proses M/K *file* menggunakan *standard system call*. Akan tetapi, beberapa sistem operasi justru tidak membedakan apakah *file* yang akan dimodifikasi tersebut ditujukan untuk *memory-mapped* atau tidak, contohnya adalah Solaris yang menganggap semua *file* yang akan dimodifikasi adalah *file* yang akan dipetakan ke memori.

Banyak proses diperbolehkan untuk memetakan *file* yang akan dipergunakan secara bersama-sama. Data yang dimodifikasi oleh sebuah proses dapat terlhat oleh proses lain yang dipetakan ke bagian yang sama. Memori virtual memetakan setiap proses pada halaman yang sama di memori fisik yang mengandung salinan *file* di *disk*. *System call memory-mapped* juga mendukung *copy-on-write*.

Proses untuk berbagi *memory-mapped file* tidak sama dengan proses berbagi memori di beberapa sistem operasi. Pada sistem UNIX dan Linux *memory-mapped* dilakukan oleh *system call* `mmap()`, sedangkan untuk berbagi memori digunakan *system call* `shmget()` dan `shmat()`. Pada Windows NT, 2000, dan XP berbagi memori dilakukan dengan *memory-mapped file*. Pada cara ini setiap proses

dapat berbagi memori dengan proses yang memetakan file yang sama ke memori. *Memory -mapped file* berlaku sebagai bagian memori yang digunakan bersama-sama oleh beberapa proses.

Berbagi memori menggunakan *memory-mapped file* pada Win32 API awalnya dengan memetakan *file* ke memori dan membuat *view* untuk *file* tersebut di setiap memori virtual milik proses. Proses lain dapat membuat *view* pada *file* tersebut. *File* yang dipetakan itu mewakili objek dari *shared-memory* yang memungkinkan proses untuk berkomunikasi.

Pada hal yang berhubungan dengan M/K, register berperan dalam mengendalikan perintah dan data yang akan di transfer. Untuk kenyamanan yang lebih maka digunakan M/K *memory-mapped*. Pada cara ini alamat memori dipetakan pada *device register*. Modifikasi pada alamat memori menyebabkan data ditransfer dari/ke *device register*. Cara ini digunakan untuk alat dengan *response time* yang cepat.

Cara ini juga digunakan untuk *serial port* dan *pararel port* yang menghubungkan *modem* dan *printer*. CPU mentransfer data melalui M/K *port*, ketika mengirim *string* panjang, CPU akan akan menulis data pada register dan mengeset bit yang menandakan bahwa data telah tersedia. Ketika data tersebut dibaca oleh alat misalkan *modem* maka bit akan di set kembali yang menandakan modem siap untuk data selanjutnya, lalu CPU mengirim data lagi. Jika CPU menggunakan *polling* untuk mengecek *bit control*, program ini disebut *programmed I/O (PIO)*. Jika CPU tidak menggunakan *polling* tetapi menerima interupsi ketika alat telah siap menerima data maka transfer ini disebut *interrupt driven*.

7.9. Rangkuman

Strategi untuk alokasi bingkai dapat dilakukan dengan cara *equal allocation* dan *proportional allocation*. *Equal allocation* dilakukan dengan membagi jumlah bingkai untuk setiap proses dengan jumlah yang sama. *Proportional allocation* membagi bingkai untuk setiap proses sesuai dengan besar ukuran proses. *Priority allocation* membagi jumlah bingkai sesuai prioritas masing-masing proses (alokasi besar untuk prioritas lebih tinggi).

Penggantian halaman dapat dilakukan dengan dua cara yaitu penggantian global dan lokal. Pada penggantian global setiap proses dapat mengganti halaman dari bingkai-bingkai yang tersedia, sedangkan pada penggantian lokal proses mengganti halaman dengan bingkai yang sudah dialokasi sebelumnya.

Thrashing adalah keadaan suatu proses sibuk melakukan *swapping* karena banyak terjadi *page fault*. *Thrashing* dapat menurunkan utilitas CPU karena setiap proses tidak dapat berjalan secara efisien. *Thrashing* dapat diatasi dengan menyediakan bingkai yang sesuai dengan proses. Ini dilakukan dengan strategi *working set*.

Memory-mapped file memetakan blok di disk ke halaman di memori. Proses dapat melakukan akses terhadap *file* melalui memori tanpa menggunakan *system call*. *Memory-mapped file* juga dapat digunakan untuk berbagi memori antar proses.

Rujukan

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

[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.

[FlynnMcHoes2006] Ida M. Flynn dan Ann Mclver McHoes. 2006. *Understanding Operating Systems*. Fourth Edition. Thomson Course Technology.

Bab 8. Seputar Alokasi Bingkai

8.1. Pendahuluan

Ketika sebuah proses yang berjalan dalam *user-mode* meminta tambahan memori, halaman akan dialokasikan dari daftar *frame* halaman bebas yang diatur dari kernel. Daftar tersebut diperoleh dengan menggunakan algoritma penggantian halaman yang telah dibahas pada bab-bab sebelumnya. Memori kernel sering dialokasikan dari sebuah daftar kumpulan memori bebas yang berbeda dari daftar yang digunakan untuk memenuhi permintaaan dalam *user-mode* proses. Terdapat dua alasan untuk hal ini:

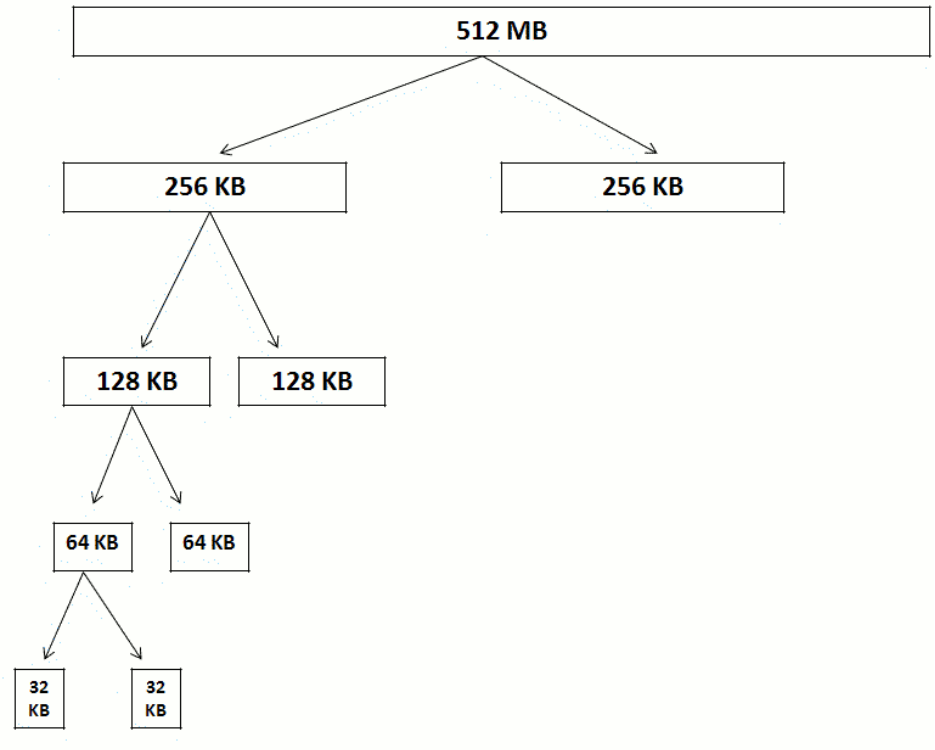
1. Kernel meminta memori untuk struktur data dengan berbagai ukuran, ada beberapa yang lebih kecil dari ukuran halaman. Jadi, kernel harus bisa meminimalisasi memori yang terbuang karena terjadinya fragmentasi
2. Halaman-halaman yang dialokasikan untuk proses-proses saat *user-mode* tidak harus dalam halaman yang saling berdekatan. Bagaimanapun juga *hardware devices* tertentu berinteraksi langsung dengan memori fisik. Hal itu mengakibatkan adanya kebutuhan memori sisa dalam halaman-halaman yang saling berdekatan. Ada dua strategi untuk me- *manage* memori bebas yang diserahkan untuk proses-proses kernel, yaitu: sistem *buddy* dan alokasi *slab*.

Dua hal utama yang harus kita pertimbangkan dalam membuat sistem *paging* adalah dari sisi pemilihan algoritma penggantian halaman dan aturan pengalokasian memori. Namun, ternyata ada beberapa hal juga harus kita pertimbangkan dalam membuat sistem *paging*, seperti: *prepaging*, TLB *reach*, ukuran halaman (*page size*), struktur program , penguncian M/K dll.

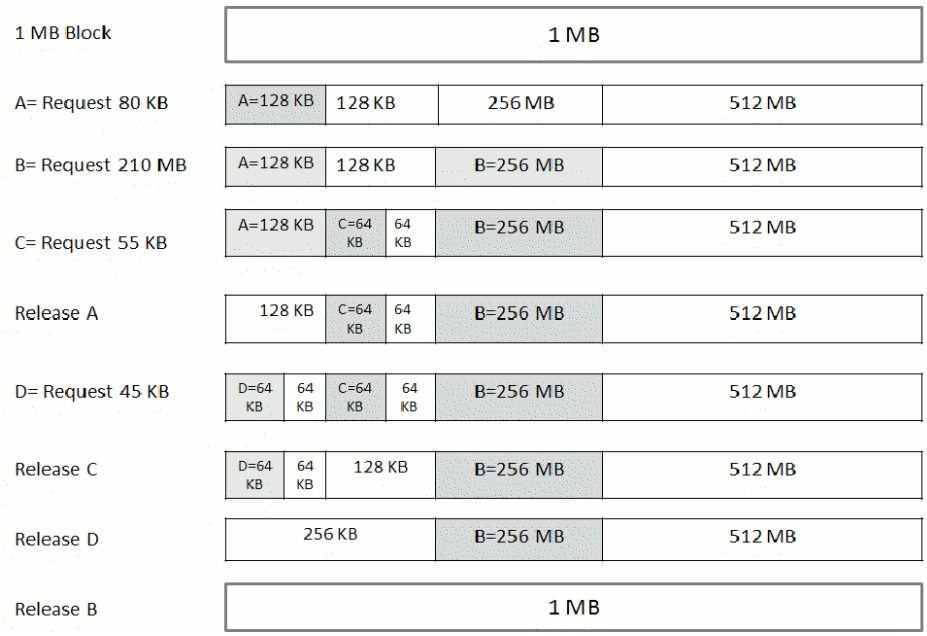
8.2. Sistem *Buddy*

Sistem buddy merupakan algoritma pengelolaan alokasi memori dimana pengalokasian memori untuk suatu proses dilakukan dengan memecah satu blok memori bebas menjadi dua bagian yang sama besar. Pemecahan dilakukan secara rekursif sehingga didapat blok yang besarnya sesuai kebutuhan.

Gambar 8.1. Ilustrasi alokasi memori dengan sistem buddy



Gambar 8.2. Contoh skema alokasi memori dengan sistem buddy



Mekanisme alokasi memori dengan sistem Buddy (lihat gambar):

1. Pada awalnya terdapat satu blok memori bebas berukuran 1 MB
2. Proses A dengan ukuran 80 KB memasuki memori tersebut.
3. Karena tidak tersedia blok berukuran 80 KB, maka blok 1MB dipecah menjadi 2 blok 512 KB. Blok-blok pecahan ini disebut buddies. Blok pertama beralamat mulai dari 0 dan blok lainnya mulai alamat 512. Kemudian Blok 512 KB pertama dipecah lagi menjadi dua blok buddies 256 KB. Blok 256 KB pertama dipecah lagi menjadi dua blok buddies 128 KB. Jika blok 128 dipecah lagi menjadi 2 blok buddies 64 KB, maka blok tersebut tidak bisa memenuhi request proses tersebut yang besarnya 80 KB. Oleh karena itu blok yang dialokasikan untuk proses 80 KB tersebut adalah blok pertama yang berukuran sebesar 128 KB.
4. Proses B dengan ukuran 210 KB memasuki memori tersebut. Karena blok pertama sudah dialokasikan untuk proses A, maka dicarilah blok berikutnya yang masih dalam keadaan bebas. Namun karena blok selanjutnya hanya berukuran 128 KB, maka proses tersebut dialokasikan ke blok berikutnya yang berukuran 256 KB.
5. Proses C dengan ukuran 55 KB memasuki memori tersebut. Sama seperti sebelumnya, karena blok pertama sudah dialokasikan, maka dicarilah blok berikutnya yang masih dalam keadaan bebas. Karena blok kedua belum dialokasikan dan masih berukuran 128 KB, maka blok tersebut dipecah lagi menjadi dua blok buddies berukuran 64 KB. Proses C dialokasikan pada blok 64 KB pertama.
6. Kemudian, proses A dibebaskan.
7. Proses D dengan ukuran sebesar 45 KB datang memasuki memori. Karena blok pertama sudah bebas, maka blok pertama dapat dialokasikan. Blok tersebut dipecah lagi menjadi dua blok buddies berukuran 64 KB. Proses D dengan ukuran 45 KB mendapat alokasi memori sebesar 64 KB.
8. Proses C dibebaskan. Dengan sistem buddy, kernel akan melakukan penggabungan dari pasangan blok buddy yang bebas dengan ukuran k ke dalam blok tunggal dengan ukuran 2k. Maka ketika proses C dibebaskan, blok tersebut akan digabung dengan blok bebas di sebelahnya menjadi blok tunggal bebas sebesar 128 KB. Sama juga halnya ketika proses D dan B di-*release*.

Dilihat dari mekanisme pengelolaan alokasi memorinya sistem buddy mempunyai keunggulan dalam dealokasi memori. Dibandingkan dengan algoritma-algoritma yang mengurutkan blok berdasarkan ukuran, sistem buddy mempunyai keunggulan ketika misalnya ada blok berukuran 2^k dibebaskan, maka manajer memori hanya mencari blok 2^k untuk diperiksa apakah dapat dilakukan penggabungan. Pada algoritma lain yang memungkinkan blok-blok memori dipecah dalam sembarang ukuran, keseluruhan blok harus dicari.

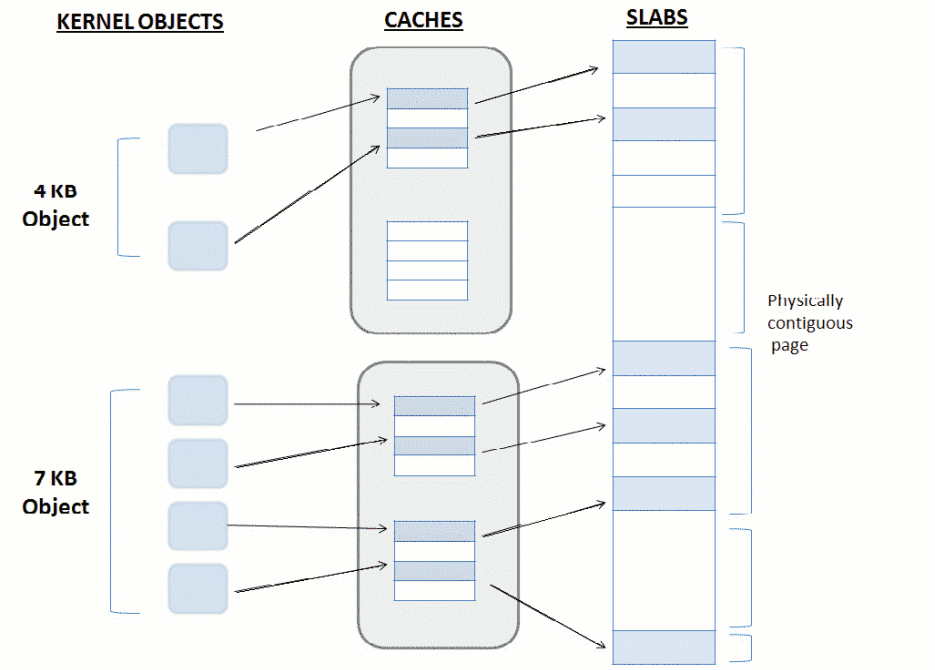
Tapi walaupun proses dealokasi mudah dilakukan. Sistem buddy mempunyai kelemahan, yaitu utilisasi memori sangat tidak efisien. Masalah ini muncul karena semua permintaan dibulatkan ke

dalam 2^k terdekat yang dapat memuat. Misal jika ada proses berukuran 35 KB, pengalokasian dilakukan di 64 KB. Maka terdapat 29 KB yang disiakan. Ini disebut fragmentasi internal karena memori yang disiakan adalah internal terhadap segmen-segmen yang dialokasikan.

8.3. Alokasi *Slab*

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 dan digunakan terutama untuk kegiatan pengalokasian memori fisik. Sebuah *cache* terdiri dari satu atau lebih *slab* dan diisi oleh beberapa *object*. *Object* itu sendiri merupakan bentuk instansiasi dari struktur data kernel yang direpresentasikan oleh *cache* yang bersangkutan.

Gambar 8.3. Hubungan antara *caches*, slab, dan kernel *objects*



Algoritmanya adalah sebagai berikut:

```
if (there is an object in the cache)
 take it (no construction required);
else {
 allocate a new slab and assign it to a cache;
 construct the object;}
```

Jika *object* yang diperlukan sudah terdapat dalam caches, maka ambil saja *object* tersebut tanpa harus dibuat ulang. Namun jika *object* yang dibutuhkan belum ada, alokasikanlah sebuah *slab* baru, berikan ke dalam *caches*, lalu buatlah *object* tersebut.

Keuntungan algoritma *slab*:

- Tidak terdapatnya fragmentasi pada memori fisik. Dengan alokasi *slab*, tidak ada lagi masalah fragmentasi karena ketika kernel me-*request* memori untuk sebuah *object*, memori yang diberikan adalah tepat sebesar ukuran *object* tersebut.

- Kebutuhan memori dapat terpenuhi dengan cepat. Proses pengalokasian dan pembebasan memori bisa memakan banyak waktu. Tapi dengan alokasi *slab*, ketika *object* pertama kali dibuat, *object* tersebut langsung dialokasikan ke dalam *caches*, lalu setelah *object* tersebut selesai digunakan, *object* tersebut diset statusnya menjadi *free* dan langsung dikembalikan ke dalam *caches*, sehingga ketika *object* tersebut dibutuhkan lagi, tidak perlu ada penginisialisasian ulang *object*. Hal ini membuat *object* tersedia setiap saat kernel membutuhkannya.

8.4. *Prepaging*

Prepaging merupakan suatu cara untuk mengurangi *page fault* pada saat proses dimulai. *Page fault* terjadi ketika halaman yang dibutuhkan tidak berada dalam memori utama, oleh karena itu strateginya adalah dengan membawa seluruh halaman yang akan dibutuhkan pada satu waktu ke memori.

Prepaging tidak selalu berguna. *Prepaging* akan berguna bila biaya yang digunakan *prepaging* lebih sedikit dari biaya menangani kesalahan halaman yang terjadi, yaitu ketika seluruh halaman yang dibawa terpakai sebagian besar. Namun *prepaging* juga bisa merugikan, yaitu saat biaya *prepaging* lebih besar dari biaya menangani kesalahan halaman atau dengan kata lain dari keseluruhan halaman yang dibawa yang terpakai hanya sebagian kecil saja.

8.5. Ukuran Halaman

Pada dasarnya tidak ada ukuran halaman yang paling baik, karena terdapat beberapa faktor yang mempengaruhinya. Salah satu faktornya adalah ukuran *page table*. Setiap proses yang aktif harus memiliki salinan dari *page table*-nya. Jadi, alangkah baiknya jika ukuran *page table* itu kecil. Untuk memperoleh ukuran *page table* yang kecil, jumlah halaman jangan terlalu banyak. Oleh karena, itu ukuran halaman sebaiknya diperbesar agar jumlah halaman tidak terlalu banyak. Misalnya 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.

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 halamannya. 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.

Pertimbangan lainnya adalah masalah lokalitas. Dengan ukuran halaman yang kecil, total M/K harus dikurangi, sehingga lokalitas akan lebih baik. Misalnya, jika ada proses berukuran 200 KB dimana hanya setengahnya saja yang dipakai (100 KB) dalam pengeksekusian. Jika kita mempunyai ukuran halaman yang besar, misalnya berukuran 200 KB, maka keseluruhan proses tersebut akan ditransfer dan dialokasikan, entah itu dibutuhkan atau tidak. Tapi dengan ukuran halaman yang kecil, misalnya 1 byte, maka kita hanya membawa 100 KB yang diperlukan saja.

Tapi, untuk memperkecil terjadinya *page fault* sebaiknya ukuran halaman diperbesar. Sebagai contoh, jika ukuran halaman adalah 1 byte dan ada sebuah proses sebesar 200 KB, dimana hanya setengahnya

yang menggunakan memori, akan menghasilkan 102.400 *page fault* . Sedangkan bila ukuran halaman sebesar 200 KB maka hanya akan terjadi 1 kali *page fault* . Jadi untuk mengurangi *page fault* , dibutuhkan ukuran halaman yang besar.

Masih ada faktor lain yang harus dipertimbangkan (misalnya hubungan antara ukuran halaman dengan ukuran sektor pada peranti pemberian halaman). Sampai saat ini belum 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.

8.6. TLB *Reach*

TLB *reach* atau jangkauan TLB adalah jumlah memori yang dapat diakses dari TLB *(Translation Lookaside buffers)*. Jumlah tersebut merupakan perkalian dari jumlah masukan dengan ukuran halaman.

Jangkauan memori = (jumlah masukan *TLB*) x (ukuran halaman)

Jika jumlah masukan dari TLB dilipatgandakan, maka jangkauan TLB juga akan bertambah menjadi dua kali lipat. 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. Tetapi untuk beberapa aplikasi hal ini masih belum cukup untuk menyimpan *working set*.

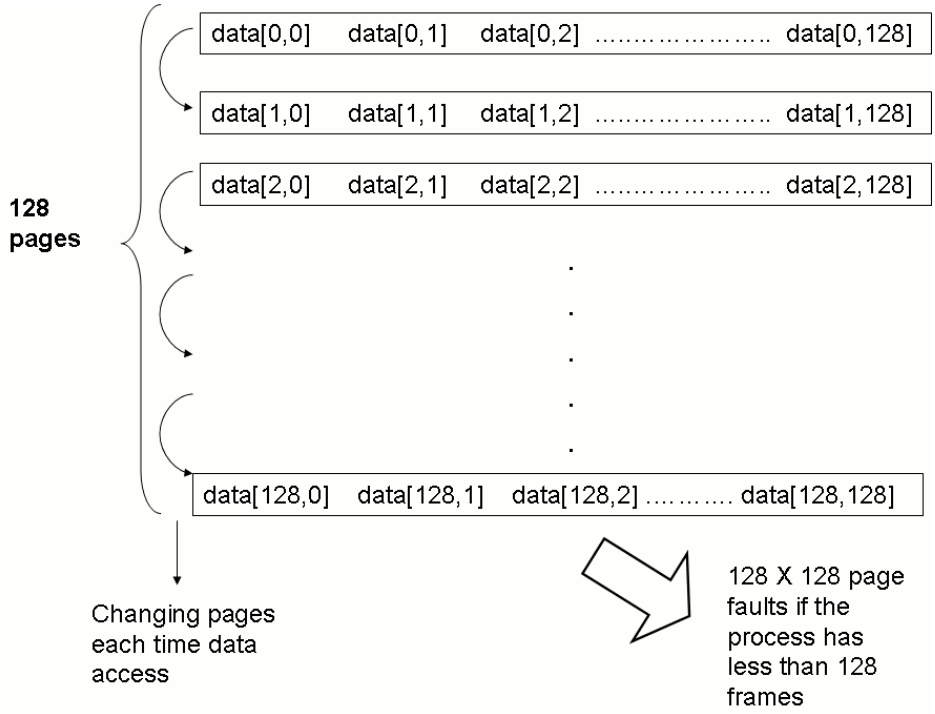
Cara lain untuk meningkatkan jangkauan TLB adalah dengan menambah ukuran halaman. Jika ukuran halaman dijadikan dua kali lipatnya, maka jangkauan TLB juga akan menjadi dua kali lipatnya. Namun hal ini akan meningkatkan fragmentasi untuk aplikasi-aplikasi yang tidak membutuhkan ukuran halaman sebesar itu. Sebagai alternatif, Sistem Operasi 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.

8.7. Struktur Program

Ketika program berjalan, maka ia akan menjadi suatu proses yang pasti membutuhkan memori. Oleh karena itu, implementasi dari suatu program akan sangat berpengaruh pada bagaimana cara proses tersebut menggunakan memori. Selain itu, pemilihan struktur data dan struktur pemrograman secara cermat juga dapat meningkatkan *locality* sehingga dapat pula menurunkan tingkat kesalahan halaman dan jumlah halaman di *working set*. Contoh kasus:

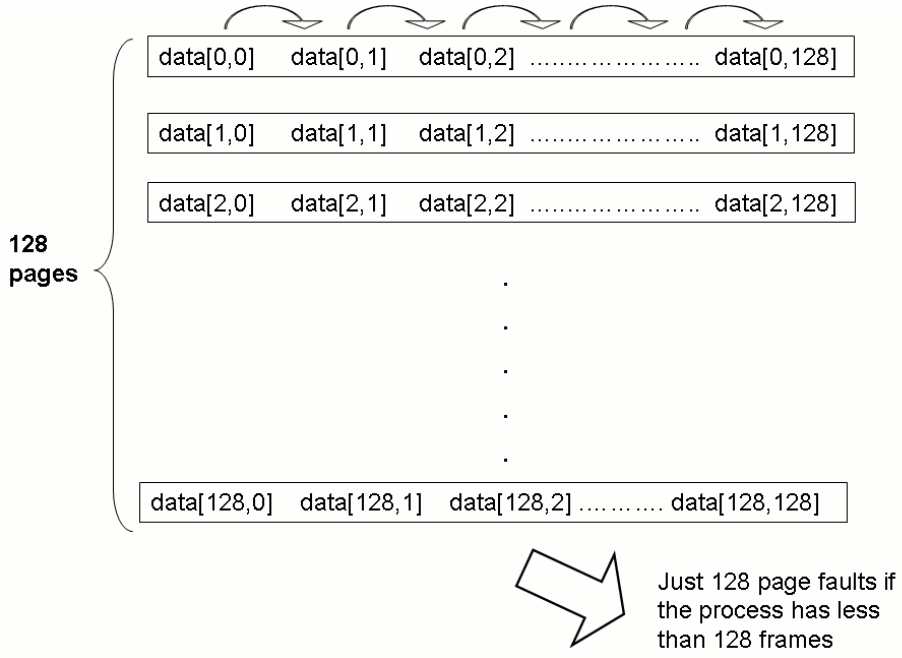
```
int i,j;
int[128][128] data;      /*Assume that pages are 128 words in size*/
for(j=0; j< 128; j++)
     for(i = 0; i< 128; i++)     /*Program to initialize to 0
   data[i][j] = 0;       each element of 128x128 array*/
```

Gambar 8.4. Ilustrasi Program 1




```
int i,j;
int[128][128] data;             /*Assume that pages are 128 words
in size*/
for(i=0; i< 128; i++)
     for(j = 0; j< 128;j++)     /*Program to initialize to 0
   data[i][j] = 0;       each element of 128x128 array*/
```

Gambar 8.5. Ilustrasi Program 2

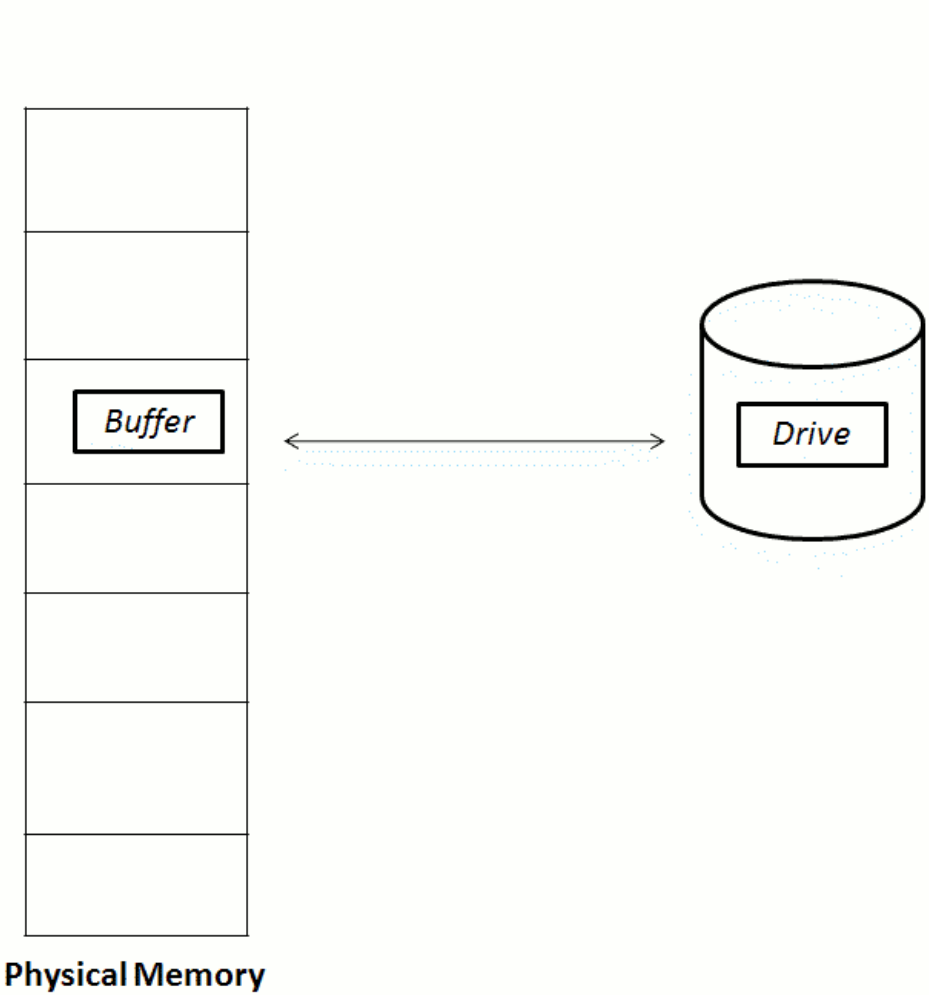


Pada program diatas, penggantian halaman dilakukan tiap kali pengaksesan data. Oleh karena itu, jika sistem operasi mengalokasikan kurang dari 128 frames maka program diatas mempunyai potensi 128 x 128 = 16.384 *page faults*. Jika program di atas diubah menjadi Program 2, maka potensi terjadinya *page fault* bisa berkurang menjadi 128 *page faults*. Hal ini disebabkan pergantian halaman tidak dilakukan setiap kali pengaksesan data, tapi setelah selesai pengaksesan 128 data, barulah terjadi pergantian halaman.

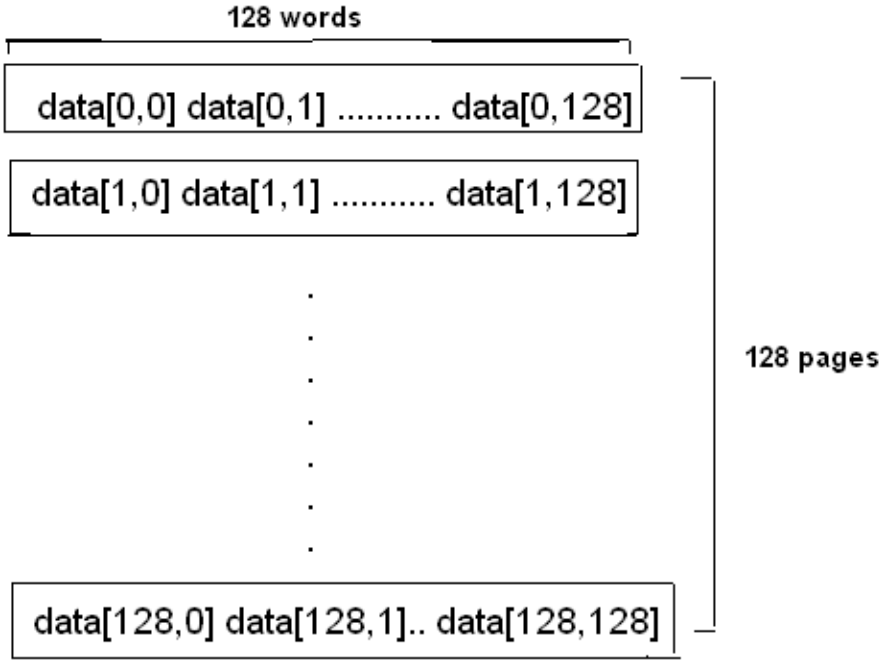
8.8. Penguncian M/K

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.

Gambar 8.6. Why we need *I/O Interlock*



Gambar 8.7. Blok Struktur



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. 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. Harus diperhatikan agar urutan dari kejadian-kejadian di atas tidak muncul.

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 agar tidak terjadi *page out* akibat suatu proses mengalami *page fault*.

8.9. *Windows XP*

Windows XP mengimplementasikan memori virtual dengan menggunakan permintaan halaman melalui *clustering*. *Clustering* menangani kesalahan halaman dengan menambahkan tidak hanya halaman yang terkena kesalahan, tetapi juga halaman-halaman yang berada disekitarnya ke dalam 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 tersedia mencukupix, 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.

8.10. Rangkuman

Proses kernel menyaratkan memori yang akan dialokasikan menggunakan halaman-halaman yang saling berdekatan. Sistem *buddy* mengalokasikan memori untuk proses kernel sebesar 2^k (2, 4, 8,16,...), yang mengakibatkan timbulnya fragmentasi. Alternatif lainnya adalah dengan alokasi *slab* yang mengalokasikan memori sebesar ukuran *object* yang dibutuhkan, sehingga tidak ada memori yang terbuang sia-sia karena tidak adanya fragmentasi.

Mengenai sistem *paging*, ternyata ada beberapa hal lain yang harus kita pertimbangkan dalam pembuatan sistem *paging* selain mempertimbangkan algoritma penggantian halaman dan aturan pengalokasian memori. Hal lain yang harus kita pertimbangkan juga adalah dalam memutuskan ukuran halaman, penguncian M/K, *prepaging*, pembuatan proses, struktur program, dll.

Rujukan

[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.

[WEBPRI2003] Romadhony,Ade. Erik Evanny . 2003. *Institut Teknologi Bandung– http://kur2003.if.itb.ac.id/ file/FMKL-K1-08.doc* . Diakses 7 april 2007.

[WEBTEK2005] Tei-Wei Kuo.. 2005. *National Taiwan University,– http://csie.ntu.edu.tw/~ktw/uos/uos-ch9.pdf* . Diakses 7 april 2007.

Bab 9. Memori Linux

9.1. Pendahuluan

Alokasi memori pada Linux menggunakan dua buah alokasi yang utama, yaitu algoritma *buddy* dan *slab*. Untuk algoritma *buddy*, setiap *routine* pelaksanaan alokasi ini dipanggil, maka blok memori berikutnya akan diperiksa. Jika ditemukan dia dialokasikan, namun 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 dibawahnya. Sedangkan algoritma *slab* menggunakan *slab* yang dibentuk dari halaman-halaman memori fisik yang berdekatan dan digunakan terutama untuk kegiatan pengalokasian memori fisik.

Linux juga menggunakan variasi dari algoritma *clock*. *Thread* dari kernel Linux akan dijalankan secara periodik. 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.

9.2. Memori Fisik

Manajemen memori pada Linux mengandung dua komponen utama yang berkaitan dengan:

1. Pembebasan dan pengalokasian halaman/blok pada main memori.
2. Penanganan memori virtual.

Berdasarkan arsitektur Intel x86, Linux memisahkan memori fisik ke dalam tiga zona berbeda, dimana tiap zona mengindentifikasikan blok (*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 M/K, 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.

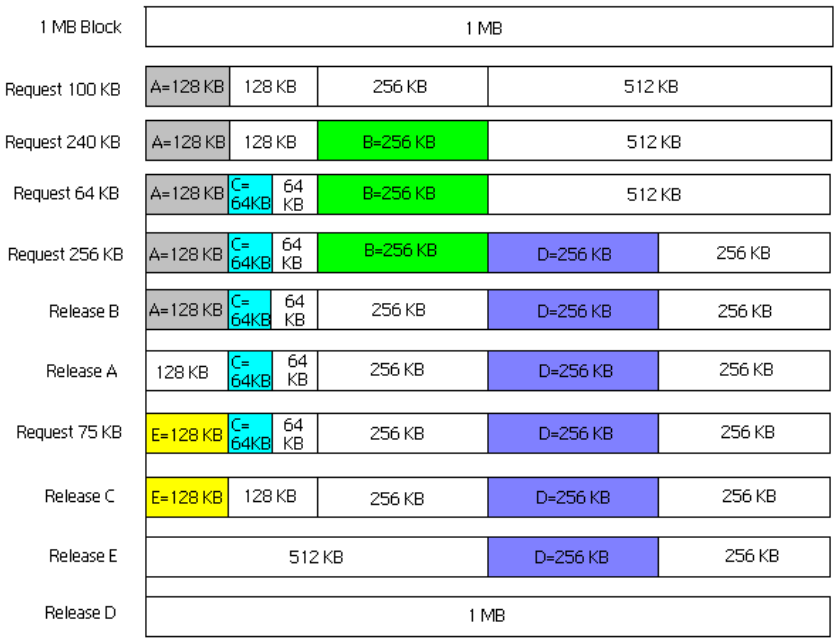
Tabel 9.1. Pembagian Zona Pada Arsitektur Intel x86

Zone	***Physical Memory***
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 - 896 MB
ZONE_HIGHMEM	> 896 MB

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 teknik alokasi, yaitu alokasi halaman yang ditangani oleh *page 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, blok di memori yang disediakan oleh kernel kepada suatu proses akan dibagi menjadi dua blok yang berukuran sama besar. Kejadian ini akan terus berlanjut hingga didapat blok yang sesuai dengan ukuran data yang diperlukan oleh proses tersebut. Dalam hal ini *page allocator* akan memanggil *system call* `kmalloc()` yang kemudian akan memerintahkan kernel untuk melakukan kegiatan pembagian blok tersebut.

Gambar 9.1. Contoh Alokasi Memori dengan Algoritma Buddy



Akan tetapi, algoritma *buddy* memiliki kelemahan , yaitu kurang efisien. Sebagai contoh, misalnya ada 1 MB memori. Jika ada permintaan 258 KB, maka yang akan digunakan sebesar 512 KB. Tentu hal ini kurang efisien karena yang dibutuhkan hanya 258 KB saja.

9.3. *Slab*

Alokasi *slab* bertujuan untuk mengalokasikan struktur data (obyek) 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* 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.

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.

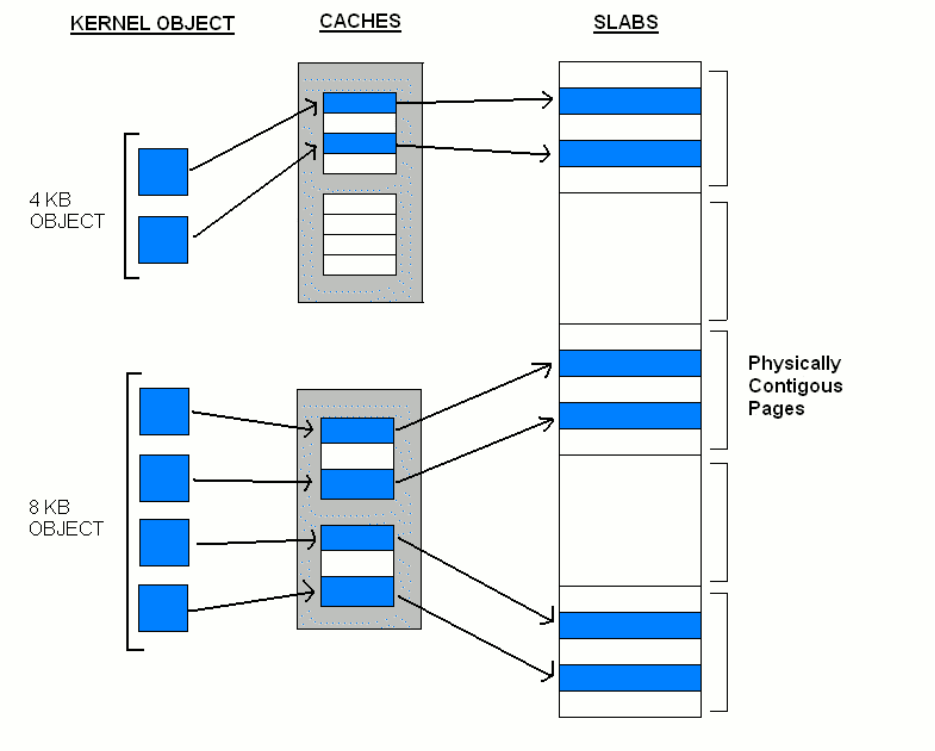
Sebuah *slab* dapat berstatus:

1. ***Full.*** Semua obyek di dalam *slab* tersebut adalah *used*.
2. ***Empty.*** Semua obyek di dalam *slab* tersebut adalah *free*.
3. ***Partial.*** 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. Permintaan oleh memori cepat terpenuhi dengan mendayagunakan kerja dari *cache* yang dibentuk pada disk.

Gambar 9.2. Contoh Alokasi Slab



9.4. Memori Virtual

Manajemen memori melakukan tugas penting dan kompleks berkaitan dengan:

1. Memori utama sebagai sumber daya yang harus dialokasikan dan dipakai bersama diantara sejumlah proses yang aktif. Agar dapat memanfaatkan prosesor dan fasilitas M/K secara efisien, maka diinginkan memori yang dapat menampung sebanyak mungkin proses.
2. Upaya agar *programmer* atau proses tidak dibatasi kapasitas memori fisik di sistem komputer.

Linux memanfaatkan memori virtual untuk mendukung kinerja sistem. Sebagai sistem operasi *multiprogramming*, memori virtual dapat meningkatkan efisiensi sistem. Sementara proses menunggu bagiannya di- *swap in* ke memori, menunggu selesainya operasi M/K dan proses di-*block*, jatah waktu prosesor dapat diberikan ke proses-proses lain.

Sistem memori virtual Linux berperan dalam mengatur beberapa hal:

1. Mengatur ruang alamat supaya dapat dilihat oleh tiap proses.
2. Membentuk halaman-halaman yang dibutuhkan.
3. Mengatur lokasi halaman-halaman tersebut dari disk ke memori fisik atau sebaliknya, yang biasa disebut *swapping*.

Sistem memori virtual Linux juga mengatur dua *view* berkaitan dengan ruang alamat:

1. ***Logical View.*** Mendeskripsikan instruksi-instruksi yang diterima oleh sistem memori virtual mengenai susunan ruang alamat.
2. ***Physical View.*** Berupa entri-entri tabel halaman, dimana entri-entrinya akan menentukan apakah halaman itu berada di memori fisik yang sedang dipakai untuk proses atau masih berada di disk yang berarti belum dipakai.

Blok Memori Virtual

Berkaitan dengan blok memori virtual, maka memori virtual dalam Linux memiliki karakteristik:

1. ***Backing Store* untuk blok.** *Backing store* mendeskripsikan tempat asal halaman pada disk. Kebanyakan blok dalam memori virtual berasal dari suatu berkas pada disk atau kosong (*nothing*). Blok dengan *backing store* yang kosong biasa disebut *"demand zero memory"* yang merupakan tipe paling sederhana dari memori virtual.
2. **Reaksi blok dalam melakukan *write*.** Pemetaan dari suatu blok ke dalam ruang alamat proses dapat bersifat *private* atau *shared*. Jika ada proses yang akan menulis blok yang bersifat *private*, maka akan dilakukan mekanisme *Copy-On-Write* atau dengan menulis salinannya.

9.5. Umur Memori Virtual

Kernel berperan penting dalam manajemen memori virtual, dimana kernel akan membentuk ruang alamat yang baru di memori virtual dalam dua kondisi:
1. Proses menjalankan suatu program dengan *system call* `exec()`. Ketika *system call* `exec()` dipanggil oleh proses untuk menjalankan suatu program, maka proses akan diberikan ruang alamat virtual yang masih kosong. Kemudian *routine-routine* akan bekerja me-*load* program dan mengisi ruang alamat ini.
2. Pembentukan proses baru dengan *system call* `fork()`. Intinya menyalin secara keseluruhan ruang alamat virtual dari proses yang ada. Langkah-langkahnya adalah sebagai berikut:
   a. kernel menyalin *descriptor* `vm_area_struct` dari proses induk,
   b. kernel membentuk tabel halaman untuk proses anak,
   c. kernel menyalin isi tabel halaman proses induk ke proses anak,
   d. setelah `fork()`, maka induk dan anak akan berbagi halaman fisik yang sama.

Di samping itu, ada kasus khusus yang harus diperhatikan, yaitu ketika proses penyalinan dilakukan terhadap blok di memori virtual yang bersifat *private*, dimana blok tersebut dipakai lebih dari satu proses selain proses induk dan anak yang memang berbagi halaman yang sama dan ada proses yang hendak menulis blok tersebut. Jika ini terjadi maka akan dilakukan mekanisme *Copy-On-Write*, yang berarti mengubah dan memakai salinannya.

9.6. *Swap*

Keterbatasan memori fisik mengharuskan Linux mengatur halaman-halaman mana saja yang harus diletakkan di dalam memori fisik atau *swap-in* dan juga halaman-halaman yang harus dikeluarkan dari memori fisik atau *swap-out*.

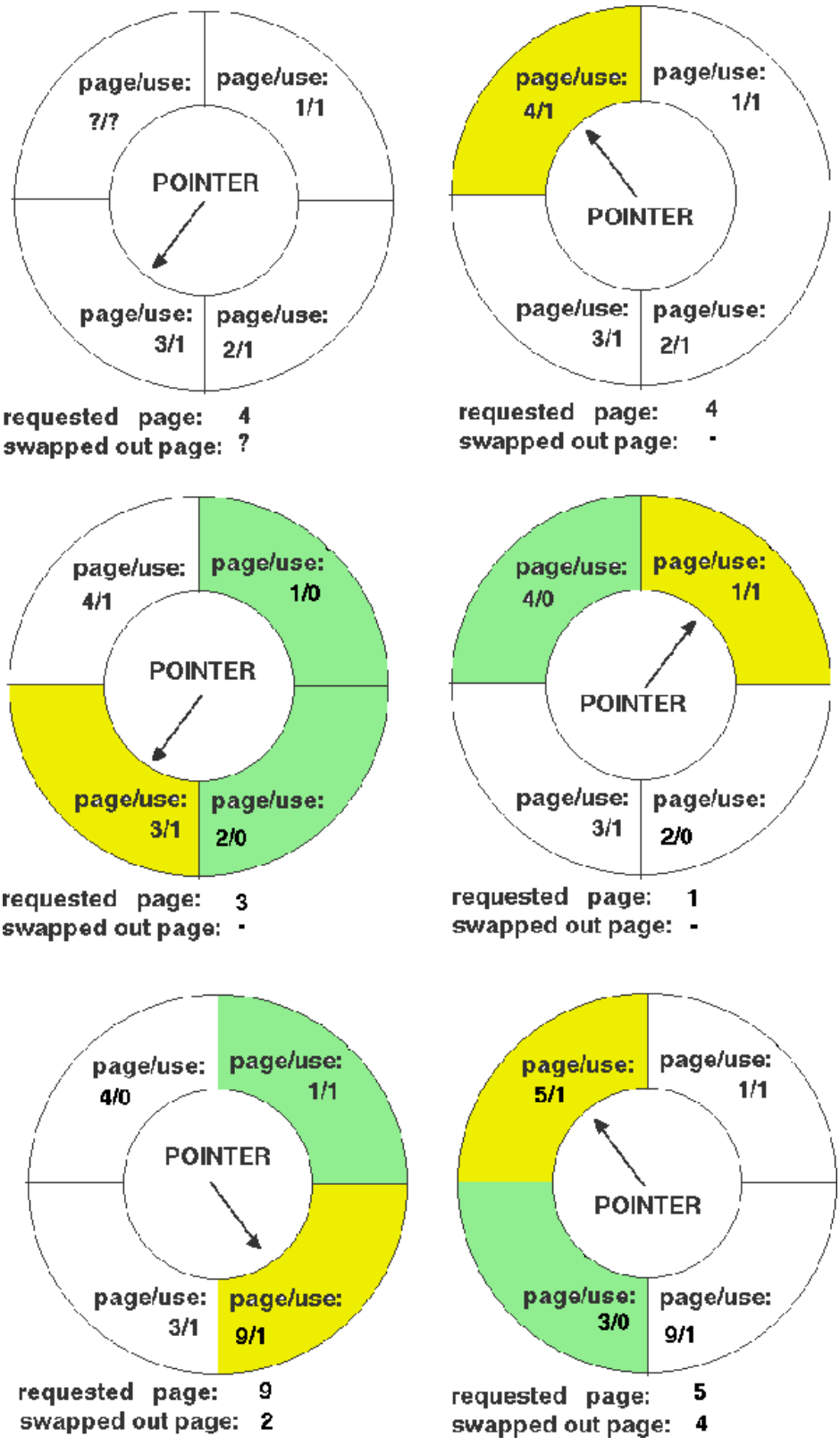
Paging system dari memori virtual dapat dibagi menjadi dua:
1. ***The pageout-policy algorithm*.** Menentukan halaman-halaman mana saja yang di *swap-out* dari memori fisik. *Pageout-policy algorithm* menggunakan algoritma *clock* dalam menentukan halaman mana yang harus di *swap-out*. Dalam Linux, *multipass clock* digunakan, setiap satu kali *pass* dari *clock*, *age* dari suatu halaman akan disesuaikan. Makin sering suatu halaman di akses, makin tinggi *age*-nya, tapi *age* dari suatu halaman berkurang setiap satu kali *pass*.
2. ***The paging mechanism*.** Menentukan halaman-halaman mana saja yang harus dibawa kembali ke dalam memori. Halaman-halaman ini pernah berada dalam memori sebelumnya.

Berikut adalah ilustrasi untuk algoritma *clock*. Di dalam memori virtual terdapat *page* 1, 2 dan 3 dengan *pointer last-used* di *page* 3. *Flag use* akan bernilai 1 jika *page* tersebut digunakan, sedangkan *use* akan bernilai 0 jika *page* tersebut dilewati *pointer* namun tidak digunakan.

Ketika ada permintaan *page* 4, sedangkan tidak ada *page* 4 dalam memori virtual sehingga terjadi *page fault*, maka *page* 4 akan dimasukkan ke tempat yang masih kosong, *pointer* akan menunjuk ke *page* 4, dan *use* diubah menjadi 1. Saat datang permintaan *page* 3, *pointer* akan mencari *page* tersebut, sekaligus mengubah *flag use* menjadi 0 jika *page* tersebut hanya dilewati, tetapi tidak digunakan.

Ketika ada permintaan untuk *page* 9, maka terjadi *page fault* karena *page* 9 tidak ada dalam memori virtual. Kemudian *pointer* akan mencari *page* yang nilai *use*-nya = 0, yaitu *page* 2. Hal yang serupa terulang ketika ada permintaan untuk *page* 5. Sehingga page 4 di *swapped-out*, dan nilai *use* dari *page* 3 diubah menjadi 0.

Gambar 9.3. Algoritma Clock



9.7. Pemetaan Memori Program

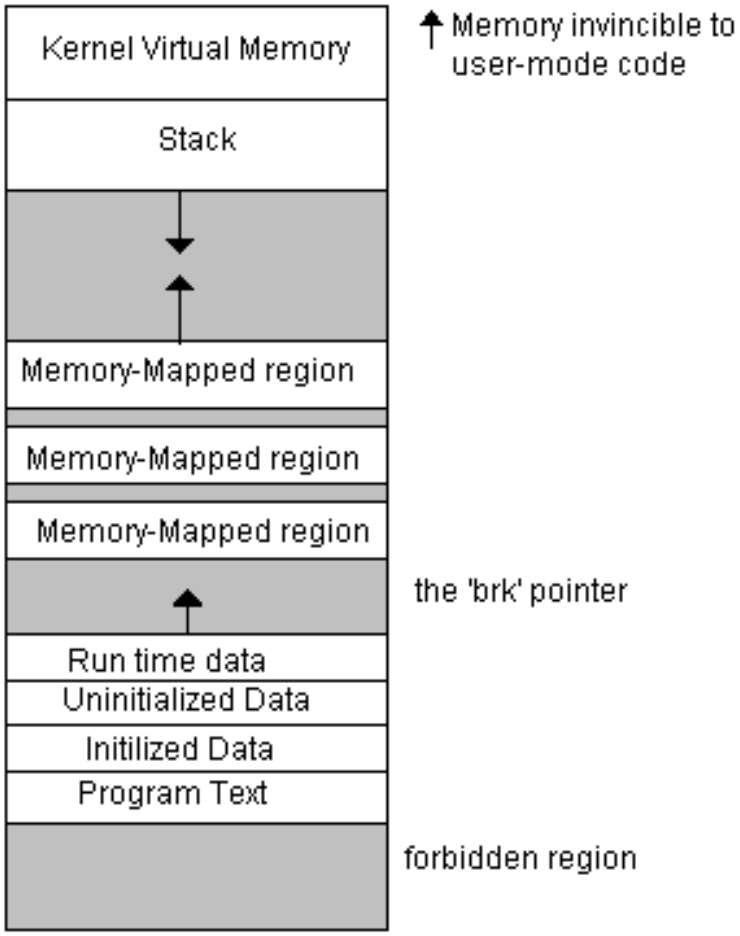
Pada Linux, *binary loader* tidak perlu me-*load* berkas biner ke memori fisik, melainkan dengan cara memetakan halaman dari *binary file* ke *region* dari memori virtual. Sehingga hanya ketika

program mengakses halaman tertentu akan menyebabkan *page fault* yang mengakibatkan halaman yang dibutuhkan di-*load* ke memori fisik.

Dalam pemetaan program ke memori juga terjadi proses *load* dan eksekusi. Eksekusi dari kernel Linux dilakukan oleh panggilan terhadap *system call* `exec()`. *System call* `exec()` memerintahkan kernel untuk menjalankan program baru di dalam proses yang sedang berlangsung atau *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.

Gambar 9.4. Executable and Linking Format



Linux menggunakan tabel *loader* untuk *loading* program baru. Dengan menggunakan tabel tersebut, Linux memberikan kesempatan bagi setiap fungsi untuk me-*load* program ketika *system 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, contohnya dengan menambahkan informasi *debugging*, tanpa menyebabkan *loader routine* menjadi bingung. Saat ini Linux mendukung pemakaian baik format *binary* ELF dan `a.out` pada *single running system* karena menggunakan registrasi dari *multiple loader routine*.

9.8. Link Statis dan Dinamis

Ketika program di-*load* dan sudah mulai dieksekusi, semua berkas biner yang dibutuhkan telah di-*load* ke ruang alamat virtual. Meski pun demikian, sebagian besar program juga butuh menjalankan fungsi yang terdapat di sistem pustaka seperti algoritma *sorting*, fungsi-fungsi aritmatika, dan lain-lain. Untuk itulah fungsi pustaka perlu untuk di-*load* juga. Untuk mendapatkan fungsi-fungsi yang terdapat di sistem pustaka, ada dua cara, yaitu:

1. ***Link* Statis.** Aplikasi dikatakan dikompilasi statis apabila pustaka-pustaka yang dibutuhkan dikompilasi ke dalam *binary apllication*. Dengan demikian, aplikasi tidak lagi membutuhkan pustaka tambahan. Fungsi pustaka yang dibutuhkan diload langsung ke berkas biner yang dapat dijalankan (*executable*) program. Kerugian *link* statis adalah setiap program yang dibuat harus menggandakan fungsi-fungsi dari sistem pustaka, sehingga tidak efisien dalam penggunaan memori fisik dan pemakaian ruang disk.
2. ***Link* Dinamis.** Pada dasarnya *link* dinamis merupakan suatu metode penghubungan antara program dengan suatu sistem pustaka secara dinamis dengan cara menghubungkan *routine-routine* yang ada ke dalam sistem pustaka. Hal ini sangat berguna pada program yang membutuhkan suatu pustaka. Bayangkan saja jika di dalam suatu sistem operasi tidak mempunyai metode penghubungan seperti ini. Setiap sistem program harus mempunyai salinan dari pustakanya agar program tersebut dapat berjalan dengan baik. Hal ini tentu saja akan membuang *disk space* dan memori utama untuk hal yang kurang perlu.

9.9. Rangkuman

Alokasi memori pada Linux menggunakan dua buah alokasi utama, yaitu algoritma *buddy* dan *slab*. Manajemen memori pada Linux mengandung dua komponen utama yang berkaitan dengan:

1. Pembebasan dan pengalokasian halaman/blok pada memori utama.
2. Penanganan memori virtual.

Linux memisahkan memori fisik ke dalam tiga zona berbeda, dimana tiap zona mengindentifikasikan blok-blok yang berbeda pada memori fisik. Ketiga zona tersebut adalah:

1. Zona DMA
2. Zona NORMAL
3. Zona HIGHMEM

Pada Linux, sebuah *slab* dapat berstatus:

1. *Full*
2. *Empty*
3. *Partial*

Sistem memori virtual Linux berperan dalam mengatur beberapa hal:

1. Mengatur ruang alamat supaya dapat dilihat oleh tiap proses.
2. Membentuk halaman-halaman yang dibutuhkan.
3. Mengatur lokasi halaman-halaman tersebut dari disk ke memori fisik atau sebaliknya, yang biasa disebut *swapping*

Sistem memori virtual Linux juga mengatur dua *view* berkaitan dengan ruang alamat:

1. *Logical View*
2. *Physical View*

Eksekusi dari Kernel Linux dilakukan oleh panggilan terhadap *system call* `exec()`. *System call* `exec()` memerintahkan kernel untuk menjalankan program baru di dalam proses yang sedang berlangsung atau *current process*, dengan cara meng- *overwrite current execution* dengan *initial context* dari program baru yang akan dijalankan.

Terdapat dua cara untuk mendapatkan fungsi-fungsi yang terdapat di sistem pustaka, yaitu:

1. ***Link* Statis.** Kerugian *link* statis adalah setiap program yang dibuat harus meng-*copy* fungsi-fungsi dari sistem pustaka, sehingga tidak efisien dalam penggunaan memori fisik dan pemakaian ruang disk.

2. ***Link* Dinamis.** *Link* dinamis menggunakan *single loading*, sehingga lebih efisien dalam penggunaan memori fisik dan pemakaian ruang disk.

Rujukan

[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.

[WEBWIKI2007] Wikipedia. 2007. *Slab Allocator - Wikipedia, the free encyclopedia http://en.wikipedia.org/ wiki/slab_allocator* . Diakses 11 April 2007.

[WEBilmukomputer2007] M Zainal Arifin. 2007. *Manajemen Proses dan Memori di Linux. http:// ilmukomputer.com/ 2006/08/28/ manajemen-proses-dan-memori-di-linux/* . Diakses 11 April 2007.

Bagian VI. Masukan/Keluaran

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 10. Sistem M/K

10.1. Pendahuluan

Pada dasarnya, tugas utama komputer adalah *processing* dan M/K. Bahkan, sebagian besar waktunya digunakan untuk mengolah M/K sedangkan *processing* hanya bersifat insidental. Jadi, pada konteks M/K, peranan sistem operasi adalah mengatur dan mengontrol perangkat M/K dan operasi M/K.

Perangkat M/K sangat bervariasi. Oleh karena itu, bagaimana cara mengontrol perangkat-perangkat tersebut mendapat perhatian besar dalam organisasi komputer. Bayangkan, perangkat M/K yang sangat banyak jumlahnya dan setiap perangkat memiliki fungsi dan kecepatan sendiri-sendiri, tentunya memerlukan metode yang berbeda pula. Oleh karena itu, dikenal klasifikasi perangkat M/K menjadi perangkat blok dan perangkat karakter, walaupun ada perangkat yang tidak termasuk ke dalam satupun dari kedua golongan ini.

Perangkat terhubung ke komputer melalui *port*, diatur oleh *device controller* dan berkomunikasi dengan prosesor dan perangkat lain melalui bus. Perangkat berkomunikasi dengan prosesor melalui dua pendekatan yaitu *memory mapped* dan instruksi M/K langsung.

Bila prosesor ingin mengakses suatu perangkat, dia akan terus mengecek perangkat untuk mengetahui statusnya, apakah mengizinkan untuk diakses. Cara ini dilakukan berulang-ulang yang disebut dengan *polling*. Sedangkan bila perangkat ingin memberitahu prosesor ketika siap diakses, maka perangkat akan menggunakan interupsi. Kedua cara ini mempunyai kelebihan dan kelemahan masing-masing. Adanya *Direct Memory Access* (DMA) dapat mengurangi beban CPU karena terjadinya transfer data antara perangkat dan memori tanpa melalui CPU.

Perbedaan detil untuk setiap alat akan dienkapsulasi pada modul kernel yang disebut *device driver*. Sedangkan untuk mengetahui waktu dan lama suatu proses digunakan *clock* dan *timer*.

10.2. Perangkat Keras M/K

Kategori Perangkat M/K

Pada saat sekarang ini, terdapat berbagai macam perangkat M/K seperti perangkat penyimpanan (*disk, tape*), perangkat transmisi (*network card, modem*), dan perangkat antar muka dengan pengguna (layar, *keyboard*, *mouse*). Secara umum, perangkat M/K dapat dibagi menjadi dua kategori yaitu:

1. **Perangkat blok.**
   Perangkat yang menyimpan informasi dalam blok-blok berukuran tertentu (umumnya 512 sampai 32.768 byte) dan setiap blok memiliki alamat masing-masing. Setiap blok pada perangkat ini bisa diakses dan ditulis secara independen. Contoh perangkat blok adalah *disk*.
2. **Perangkat karakter.**
   Perangkat yang mengirim dan menerima sebarisan karakter tanpa menghiraukan struktur blok. Contoh perangkat karakter adalah *printer* , *network interface* dan perangkat yang bukan *disk*.

Namun, pembagian ini tidak sepenuhnya benar karena ada perangkat yang tidak memenuhi kedua kriteria tersebut yaitu *clock*. *Clock* merupakan perangkat yang tidak memiliki blok beralamat, tidak mengirim dan menerima barisan karakter melainkan hanya menginterupsi dalam jangka waktu tertentu.

Komponen M/K

Unit M/K terdiri dari dua komponen yaitu:

1. **Komponen Mekanis.** Komponen Mekanis adalah perangkat M/K itu sendiri seperti *mouse*, *monitor*, dll.
2. **Komponen Elektronis.** Komponen Elektronis disebut juga dengan *controller* perangkat. Perangkat tidak berhubungan langsung dengan prosesor, *controller* -lah yang berhubungan dengan prosesor.

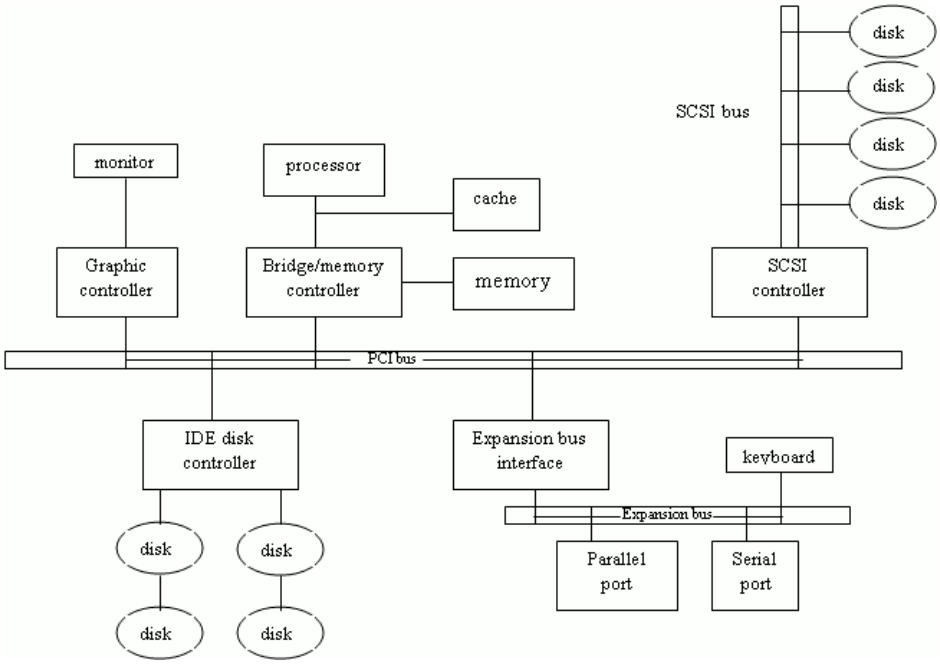
Cara Perangkat Terhubung Ke Komputer

Perangkat M/K berkomunikasi dengan sistem komputer melalui sinyal yang dikirimkan melalui kabel maupun udara (*wireless*). Perangkat M/K berhubungan dengan mesin melalui suatu titik yang bernama *port*. *Port* M/K terdiri dari 4 *register*, yaitu:

1. ***Data-in register .*** *Register* ini yang akan dibaca CPU untuk mendapatkan input.
2. ***Data-out register .*** CPU menulis bit disini sebagai *output* data.
3. **Status.** CPU akan membaca *register* ini untuk mengetahui status perangkat. Status perangkat bisa menandakan apakah tersedia input di *data-in register*, perangkat selesai melaksanakan tugasnya dengan baik atau terjadi *error* di perangkat.
4. ***Control.*** *Register* ini ditulis oleh CPU untuk memulai perintah atau untuk mengganti modus perangkat. Salah satu contoh penggantian modus perangkat adalah terdapat bit di *register control* di *serial port* yang berfungsi untuk memilih kecepatan transfer yang didukung oleh *serial port* tersebut.

Bus adalah kumpulan kabel dan protokol yang menetapkan sekumpulan pesan yang bisa dikirim melalui kabel. Beberapa perangkat bisa terhubung ke bus yang sama. Bila perangkat A terhubung ke perangkat B, perangkat B terhubung ke perangkat C dan seterusnya sampai perangkat yang terakhir terhubung ke komputer, rangkaian perangkat ini disebut *daisy chain*. *Daisy chain* juga berfungsi sebagai bus.

Gambar 10.1. Struktur bus pada PC



PCI (Peripheral Component Interconnect) bus adalah bus berkecepatan tinggi yang menghubungkan subsistem memori-prosesor ke perangkat berkecepatan tinggi dan ke *Expansion bus* yang berhubungan dengan perangkat yang lebih lambat seperti *keyboard*, *serial port* dan *paralell port*. SCSI atau *Small Computer System Interface* (baca: skazi) adalah bus yang menghubungkan beberapa *disk* ke *SCSI controller*. Dengan SCSI, kita bisa mendapatkan hingga tujuh perangkat terhubung ke komputer tetapi ini akan memperlambat komputer pada saat *start-up*.

Kita semua tentu sudah mengenal perangkat penyimpanan seperti *floopy drive*, *hard drive* dan *CD-ROM drive*. Biasanya perangkat-perangkat ini terhubung ke komputer melalui *IDE (Integrated Drive Electronics)* . Antarmuka ini menyatukan *controller* ke *drive*, sehingga dengan instruksi yang lebih sederhana dan rute yang lebih dekat antara *drive* dan *controller*, membuatnya lebih cepat dan mudah untuk digunakan.

Bus, *port* dan perangkat bisa dioperasikan oleh *controller* yang merupakan sekumpulan perangkat elektronik. *Serial-port controller* adalah salah satu *controller* perangkat yang sederhana karena hanya sebuah chip yang mengontrol sinyal dari kabel di *port*. Di sisi lain, juga ada *controller* perangkat yang kompleks yaitu *SCSI controller* yang sedemikian rumitnya sehingga harus diimplementasikan secara khusus sebagai papan sirkuit tersendiri di dalam komputer. Ini disebut dengan *host adapter*. *SCSI controller* berisi prosesor, *microcode*, dan beberapa memori sendiri.

Komunikasi Perangkat Dengan Prosesor

Kita sudah membahas tentang perangkat dan bagaimana mereka tersambung ke komputer, sekarang fokus selanjutnya adalah bagaimana prosesor berkomunikasi dengan perangkat. Ada dua pendekatan:

1. **Instruksi M/K langsung.**
   Setiap perangkat diberi nomor *port* M/K sepanjang 8/16 bit yang unik. Pada transfer data antara *register* perangkat dan *register* CPU digunakan instruksi M/K khusus. Instruksi M/K ini berbeda dari sekedar instruksi memori biasa karena alamat *port* M/K tidak menggunakan lokasi yang sama dengan alamat memori. Oleh sebab itu, 2 instruksi berikut: –in R3, 0x200, 4,– dan –mov R3, 0x200--- memiliki dua arti yang berbeda. Instruksi pertama merupakan instruksi M/K khusus yang meminta CPU untuk membaca nilai dari *register* nomor 4 dari alat M/K pada nomor *port* 0x200 kemudian meletakkannya pada *register* nomor 3 di CPU. Instruksi kedua merupakan instruksi memori biasa yang hanya menyalin isi alamat memori 0x200 ke *register* tiga di CPU.
2. ***Memory mapped* .**
   Pendekatan ini menggunakan pemetaan alamat M/K ke memori. *Register* data dan *buffer* data dipetakan ke ruang alamat yang digunakan CPU.

Keunggulan *memory mapped* adalah:

1. Prosesor akan memiliki jumlah instruksi yang lebih sedikit karena prosesor tidak perlu menyediakan instruksi M/K khusus.
2. Akses ke memori dilakukan dengan instruksi memori biasa, sehingga *driver* untuk peralatan dapat ditulis dalam bahasa C / C++ (untuk instruksi memori biasa) daripada bahasa *assembly* (untuk instruksi M/K khusus).
3. Sistem operasi dapat mengontrol akses ke perangkat M/K, yaitu dengan tidak meletakkan ruang alamat perangkat pada ruang alamat virtual proses.

Namun, ada masalah yang cukup signifikan pada pendekatan *memory mapped* yaitu masalah *caching*. Pada *caching*, proses menyimpan isi dari lokasi memori yang baru direferensikan sehingga bila ada instruksi yang mereferensikan ke alamat yang sama tidak perlu mengambil ke memori lagi, dengan demikian *caching* dapat meningkatkan kinerja sistem.

Terjadi masalah pada suatu sistem yang berulang-ulang membaca *register* status pada perangkat untuk melihat apakah perangkat tersebut siap diakses. Pembacaan yang pertama, isi dari *register* status akan disimpan di *cache*. Namun, perulangan selanjutnya akan mengambil nilai dari *cache* bukan pada *register* status pada peralatan. Akibatnya, tidak akan diketahui kapan perangkat siap digunakan karena nilai yang terus dibaca adalah nilai yang disimpan pertama kali di *cache*, bukan status ter-*update* dari perangkat itu sendiri. Untuk mengatasi masalah ini, proses *caching* tetap dilakukan kecuali untuk lokasi memori dimana register alat M/K dipetakan.

Beberapa sistem menggunakan kedua teknik ini. Contohnya pada penggunaan *graphics controller*. *Graphics controller* mempunyai alamat *port* M/K (untuk pendekatan instruksi M/K langsung), namun dia juga mempunyai wilayah *memory-mapped* yang besar untuk menampung tampilan layar. Proses mengubah tampilan dengan menulisi wilayah *memory-mapped* ini dan controller akan menyesuaikan tampilan layar berdasarkan informasi dari *memory mapped* tersebut. Jauh lebih cepat menulis jutaan Bytes ke memory grafik daripada memuat jutaan instruksi. Namun, kelemahannya, adalah saat *pointer* menunjuk ke wilayah memori yang salah dan menulisinya.

10.3. *Polling*

Bila prosesor ingin mengakses perangkat, salah satu pendekatannya adalah dengan membiarkan prosesor melakukan semua pekerjaan. Prosesor berinteraksi dengan *controller* melalui protokol yang rumit tetapi dasar *handshaking*-nya cukup sederhana yaitu:

1. CPU terus menerus membaca bit status sampai bit tersebut menandakan perangkat siap menerima perintah CPU.
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 *controller*.
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* menghapus *command ready bit*, bit *error* di status dan bit kerja.

Langkah 1 disebut *polling* atau *busy waiting*. Prosesor terus-menerus membaca bit status, berharap suatu saat bit tersebut menandakan perangkat siap menerima perintahnya. Pada dasarnya *polling* dapat dikatakan efisien bila kinerja perangkat dan *controller*-nya cepat. Kelemahan dari cara ini adalah bila waktu tunggu lama, maka lebih baik prosesor mengerjakan tugas yang lain. Sedangkan untuk mengetahui apakah perangkatnya sudah siap menerima perintah lagi atau belum, digunakanlah interupsi.

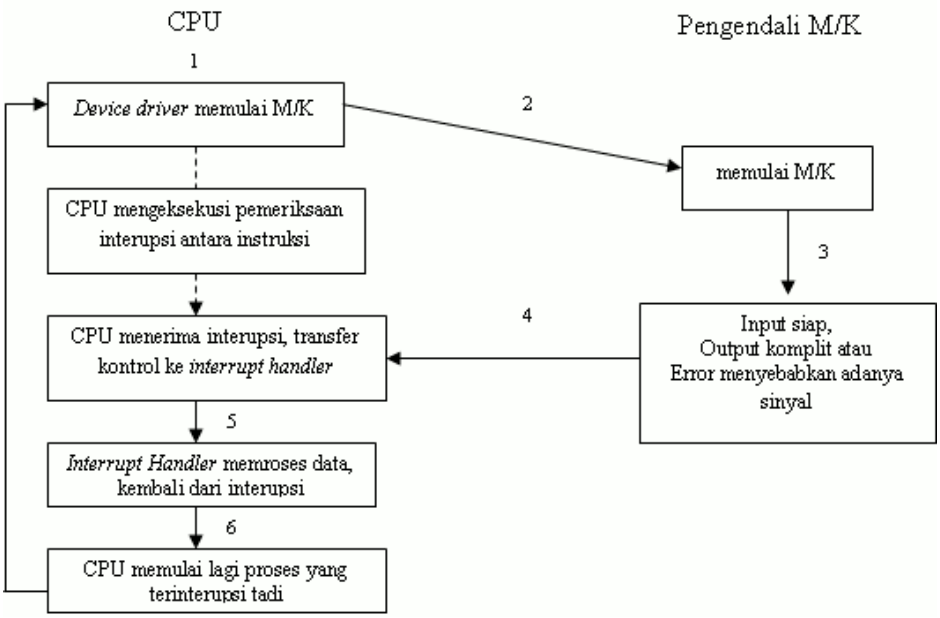
10.4. Interupsi

Interupsi terjadi bila suatu perangkat M/K ingin memberitahu prosesor bahwa ia siap menerima perintah, *output* sudah dihasilkan, atau terjadi *error*.

Penanganan Interupsi

Ada beberapa tahapan dalam penanganan interupsi:
1. *Controller* mengirimkan sinyal interupsi melalui *interrupt-request-line*
2. Sinyal dideteksi oleh prosesor
3. Prosesor akan terlebih dahulu menyimpan informasi tentang keadaan *state*-nya (informasi tentang proses yang sedang dikerjakan)
4. Prosesor mengidentifikasi penyebab interupsi dan mengakses tabel vektor interupsi untuk menentukan *interrupt handler*
5. Transfer kontrol ke *interrupt handler*
6. Setelah interupsi berhasil diatasi, prosesor akan kembali ke keadaan seperti sebelum terjadinya interupsi dan melanjutkan pekerjaan yang tadi sempat tertunda.

Gambar 10.2. Siklus penanganan interupsi



Interrupt Request Line

Pada kebanyakan CPU, ada dua *interrupt request line*. Pertama, interupsi *nonmaskable*, interupsi ini biasanya berasal dari perangkat keras dan harus segera dilaksanakan, seperti terjadinya *error* pada memori.

Kedua, interupsi *maskable*, jenis interupsi ini bisa dilayani oleh prosesor atau bisa tidak dilayani. Kalaupun dilayani, harus dilihat keadaan prosesor saat itu. Ada kemungkinan prosesor langsung menangani bila saat itu, prosesor *preemptive*, bila *nonpreemptive*, maka harus menunggu proses yang sedang dikerjakan selesai.

Vektor Interupsi dan *Vector Chaining*

Bila ada sebuah sinyal interupsi pada *interrupt request line*, bagaimana sebuah *interrupt handler* mengetahui sumber dari interupsi itu? Apakah harus menelusuri semua sumber interupsi satu-persatu? Tidak perlu, karena setiap *interrupt handler* mempunyai alamat memori masing-masing. Alamat ini adalah *offset* pada sebuah tabel yang disebut dengan vektor interupsi.

Tabel 10.1. Tabel Vector-Even pada Intel Pentium

Vector number	Description
0	Divide error
1	Debug Exception
2	Null Interrupt
3	Breakpoint
4	INTO-detected overflow
5	Bound range exception
6	Invalid opcode
7	Device not available
8	Double fault
9	Compressor segment overrun (reserved)
10	Invalid task state segment
11	Segment not present
12	Stack fault
13	General protection
14	Page fault
15	(Intel reserved, do not use)
16	Floating point error
17	Alignment check
18	Machine check
19-31	(Intel reserved, do not use)
32-255	Maskable interrupt

Sesuai dengan perkembangan zaman, komputer mempunyai lebih banyak perangkat (dan lebih banyak *interrupt handlers*) daripada elemen alamat di vektor. Hal ini bisa diatasi dengan teknik *vector chaining*. Setiap elemen di vektor interupsi menunjuk ke kepala dari sebuah daftar *interrupt handlers*, sehingga bila ada interupsi, *handler* yang terdapat pada daftar yang ditunjuk akan dipanggil satu persatu sampai didapatkan *handler* yang bisa menangani interupsi yang bersangkutan.

Prioritas Interupsi

Mekanisme interupsi juga menerapkan sistem level prioritas interupsi. Sistem ini memungkinkan interupsi berprioritas tinggi menyela eksekusi interupsi berprioritas rendah. Sistem ini juga memungkinkan perangkat M/K yang membutuhkan pelayanan secepatnya didahulukan daripada perangkat lainnya yang prioritasnya lebih rendah. Pengaturan prioritas dan penanganan perangkat berdasarkan prioritasnya diatur oleh prosesor dan *controller* interupsi.

Penyebab Interupsi

Mekanisme interupsi tidak hanya digunakan untuk menangani operasi yang berhubungan dengan perangkat M/K. Sistem operasi menggunakan mekanisme interupsi untuk beberapa hal, di antaranya:

1. Menangani *exception*
   Exception adalah suatu kondisi dimana terjadi sesuatu, atau dari sebuah operasi didapatkan hasil tertentu yang dianggap khusus sehingga harus mendapat perhatian lebih, contohnya, pembagian dengan nol, pengaksesan alamat memori yang *restricted* atau tidak valid, dll.
2. Mengatur *virtual memory paging*.
3. Menangani perangkat lunak interupsi.
4. Menangani alur kontrol kernel.

Jika interupsi yang terjadi merupakan permintaan untuk transfer data yang besar, maka penggunaan interupsi menjadi tidak efisien, untuk mengatasinya digunakanlah DMA.

10.5. DMA

Seperti yang telah 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 *context switch*, padahal *context switch* akan menimbulkan *overhead*. Oleh karena itu kelemahan mekanisme interupsi untuk menangani transfer data yang besar disebabkan oleh *context switch*. Untuk menangani kelemahan tersebut, digunakan suatu unit kontrol khusus yang disediakan untuk mentrasfer data langsung antar perangkat eksternal dan memori utama tanpa intervensi terus menerus dari prosesor. Unit kontrol khusus tersebut adalah DMA.

Sistem modern dapat mengurangi beban CPU untuk melakukan operasi M/K, yaitu dengan menggunakan pengendali DMA. Dengan demikian CPU dapat melakukan tugas lain sementara operasi M/K dilakukan. Setiap pengendali peralatan dapat saja memiliki perangkat keras DMA tersendiri. Alternatif lain adalah dengan memiliki sebuah pengendali DMA pada *motherboard* yang mengatur transfer ke berbagai peralatan.

Untuk memulai transfer data secara DMA, *driver* peralatan akan menulis blok perintah DMA memori yang menunjuk sumber data, tujuan, dan jumlah byte yang akan ditransfer. CPU kemudian akan mengirimkan alamat blok perintah ini pada pengendali DMA. pengendali DMA akan memproses informasi ini untuk kemudian mengoperasikan bus memori.

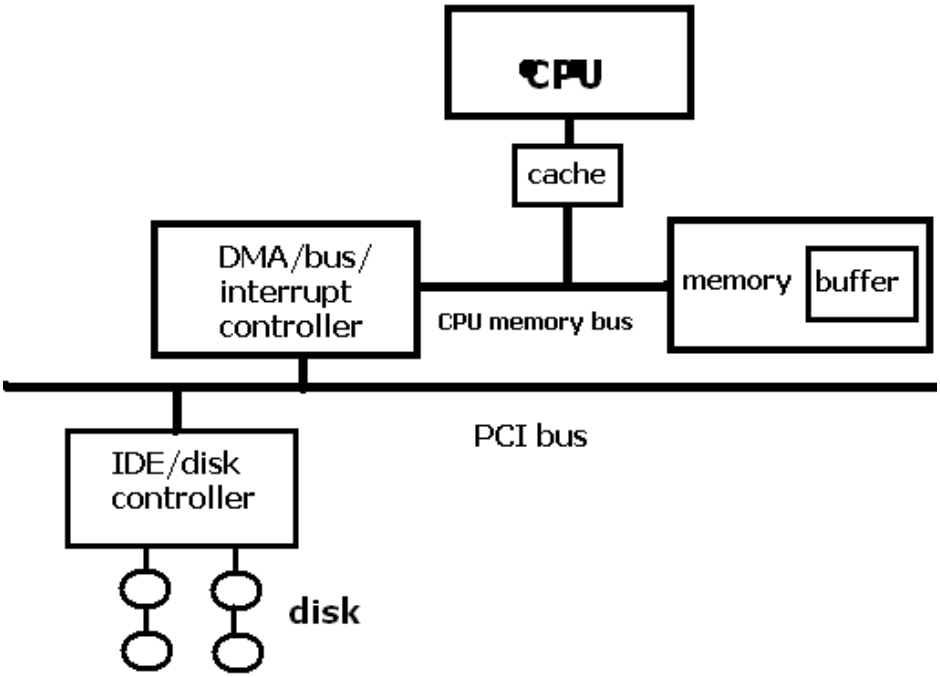
Transfer sebanyak 1 byte/word per satuan waktu oleh pengendali DMA disebut sebagai *cycle stealing* karena pengendali menggunakan *bus cycle* milik CPU. Dengan *cycle stealing* penggunaan bus oleh CPU akan tertunda beberapa waktu karena bus digunakan untuk proses DMA.

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) samapai seluruh blok sudah ditransfer.

3. Pengendali DMA menginterupsi prosesor, dimana selanjutnya akan ditentukan tindakan berikutnya.

Gambar 10.3. DMA



10.6. Aplikasi Antarmuka M/K

Perbedaan dari alat-alat M/K dapat dipisahkan dengan mengelompokkan alat-alat yang serupa ke beberapa kelas generik. Untuk setiap kelas generik terdapat beberapa fungsi yang diberikan melalui antar muka standar yang diberikan. Perbedaan detil untuk setiap alat akan dienkapsulasi pada modul *kernel* yang disebut *device driver*. *Driver* ini dibuat oleh pembuat perangkat untuk memenuhi kebutuhan setiap peralatan dengan menggunakan salah satu antarmuka standar. Penggunaan layer untuk driver peralatan ini menyembunyikan perbedaan setiap pengendali peralatan dari subsistem M/K pada kernel, sama seperti bagaimana *system call* M/K menyembunyikan perbedaan perangkat keras dari aplikasi melalui abstraksi yang berisi kelas-kelas peralatan generik.

Peralatan dapat berupa:

1. ***Character stream* .**
   atau *block* Sebuah peralatan *character stream* (contoh: terminal) untuk mentransfer *byte* satu persatu sedangkan *block device* akan mentransfer sekumpulan byte sebagai unit contohnya adalah *disk*.
2. ***Sequensial* atau *random-access* .**
   Sebuah perangkat yang sekuensial memindahkan data yang sudah pasti seperti yang ditentukan oleh perangkat, contohnya modem, sedangkan pengguna akses *random* dapat meminta perangkat untuk mencari keseluruh lokasi penyimpanan data yang tersedia, contohnya CD-ROM.
3. ***Synchronous* atau *asyinchronous* .**
   Perangkat *synchronous* menampilkan data-data transfer dengan reaksi yang dapat diduga, contohnya *tape*, sedangkan perangkat *asyinchronous* menampilkan waktu reaksi yang tidak dapat diduga, contohnya *keyboard*.
4. ***Sharable* atau *dedicated* .**
   Perangkat yang dapat dibagi digunakan secara bersamaan oleh beberapa prosesor atau *sharable*, contohnya *keyboard*, sedangkan perangkat yang *dedicated* tidak dapat digunakan secara bersamaan oleh beberapa prosesor, contohnya *tape*.
5. ***Speed of operation* .**

Rentangan kecepatan perangkat dari beberapa bytes per detik sampai beberapa gigabytes per detik.

6. ***Read-write,read only, write only*.**
   Beberapa perangkat memungkinkan baik *input-output* dua arah, contohnya CD-ROM, tapi beberapa lainnya hanya menunjang data satu arah saja, contohnya *disk*.

10.7. *Clock* dan *Timer*

Adanya *clock* dan *timer* pada perangkat keras komputer, setidaknya memiliki tiga fungsi yang sering digunakan oleh sistem operasi yaitu:

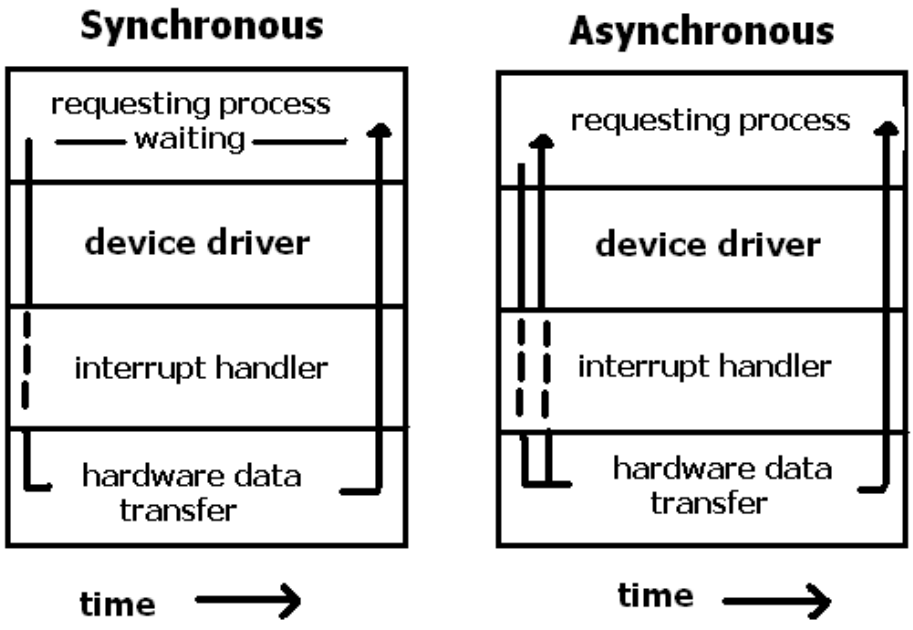
1. Memberi informasi waktu saat ini.
2. Memberi informasi waktu lamanya sebuah proses.
3. Sebagai *trigger* untuk suatu operasi pada suatu waktu.

Perangkat keras yang mengukur waktu dan melakukan operasi *trigger* dinamakan *programmable interval timer*. Perangkat keras ini dapat diatur untuk menunggu waktu tertentu dan kemudian melakukan interupsi. Contoh penerapan adalah pada *scheduler* dimana dia melakukan interupsi yang akan memberhentikan suatu proses pada akhir dari bagian waktunya.

10.8. *Blocking/Nonblocking*

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 pengeksekusian dilanjutkan. Banyak sistem operasi yang bersifat *blocking* karena lebih mudah dimengerti daripada *nonblocking*. Sedangkan *nonblocking* M/K saat aplikasi tersebut meminta data dari M/K maka pada saat menunggu diterimanya data dari M/K akan dikerjakan proses lain.

Gambar 10.4. metode *blocking* dan *nonblocking*



10.9. Rangkuman

Port adalah titik koneksi antara perangkat M/K dengan komputer. Data mengalir keluar masuk peralatan melalui *bus*. Komputer berkomunikasi dengan peralatan melalui *device controller*. Dua

pendekatan dasar komunikasi perangkat dengan komputer: instruksi M/K langsung dan *memory-mapped*. Status suatu perangkat terlihat dari bit *memory-mapped*-nya, 1 berarti sibuk, 0 berarti siap. CPU akan memeriksa keadaan bit ini berulang kali untuk melihat apakah perangkat siap, hal ini dinamakan *polling*. Interupsi bisa terjadi karena input di perangkat telah siap, *output* komplit atau terjadi *error*. Sinyal Interupsi akan disampaikan melalui *interrupt request line* (IRQ) yang akan diterima CPU dan mengalihkannya ke *interrupt handler*. Dua jenis interupsi yaitu *maskable* (bisa ditunda/diabaikan) dan *nonmaskable interrupt*. *Interrupt vector* berisi alamat awal dari *interrupt handler*.

DMA adalah suatu unit kontrol khusus yang disediakan untuk mentransfer data langsung antar perangkat eksternal dan memori utama, tanpa intervensi terus menerus dari prosesor. Perbedaan detil untuk setiap alat akan dienkapsulasi pada modul *kernel* yang disebut *device driver*. *Driver* ini dibuat oleh pembuat perangkat untuk memenuhi kebutuhan setiap peralatan dengan menggunakan salah satu antarmuka standar (aplikasi M/K).

Adanya *clock* dan *timer* pada perangkat keras komputer, setidaknya memiliki tiga fungsi, yaitu: memberi informasi waktu saat ini, memberi informasi waktu lamanya sebuah proses, sebagai *trigger* untuk suatu operasi pada suatu waktu. Suatu aplikasi dapat menggunakan *blocking* dan *nonblocking* I/O.

Rujukan

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

[Djoko2004] Djoko Hartomo. 2005. *Sistem Operasi*. First Edition. Graha Ilmu.

[Riri2005] Riri Fitri Sari dan Yansen Darmaputra. 2005. *Sistem Operasi Modern*. First Edition. Andi Yogyakarta.

Bab 11. Subsistem M/K Kernel

11.1. Pendahuluan

Kernel menyediakan banyak layanan yang berhubungan dengan M/K. Kinerja suatu sistem dilihat dari bagus atau tidaknya M/K, oleh karena itu sistem menjadwalkan permintaan dari beberapa aplikasi seefisien mungkin dengan metode penjadwalan M/K. Kita tahu bahwa setiap perangkat mempunyai sifat yang berbeda. Perbedaan ini membuat M/K menjadi tidak efisien. Oleh karena itu, sejumlah teknik dapat kita gunakan untuk mengatasi masalah tersebut antara lain *cache, buffer*, dan *spooling*. Permintaan ilegal dapat mempengaruhi kinerja sistem sehingga sistem melindungi M/K dari pengguna. Sistem juga mempunyai struktur data yang berisi status komponen M/K, dan cara sistem mengubah status permintaan M/K menjadi operasi perangkat keras. Banyak pilihan yang dapat kita gunakan untuk meningkatkan kinerja M/K di berbagai tingkatan.

11.2. Penjadwalan M/K

Menjadwalkan sekumpulan permintaan M/K berarti menentukan urutan yang benar untuk mengeksekusi permintaan tersebut. Urutan aplikasi memanggil *system call* adalah pilihan yang paling baik. Beberapa kelebihan dari penjadwalan diantaranya:

1. **Dapat meningkatkan kinerja sistem .**
2. **Dapat berbagi perangkat secara adil diantara banyak proses yang ingin mengakses perangkat tersebut. .**
3. **Dapat mengurangi waktu tunggu rata-rata (*average waiting time*) dalam menyelesaikan operasi M/K. .**

Berikut adalah contoh sederhana untuk menggambarkan penjadwalan tersebut. Jika sebuah lengan disk (*disk arm*) terletak di dekat permulaan disk dan ada 3 aplikasi yang memblokir panggilan yang mau membaca disk tersebut. Aplikasi pertama meminta sebuah blok di dekat bagian terakhir disk, aplikasi kedua meminta sebuah blok di dekat bagian permulaan disk, dan aplikasi ketiga meminta sebuah blok di tengah-tengah disk. Sistem operasi akan mengurangi jarak yang harus ditempuh untuk memenuhi ketiga permintaan tersebut dengan mengurutkan aplikasi yang lebih dulu dilayani oleh disk dengan urutan 2-3-1. Pengurutan urutan layanan inilah yang menjadi inti dari penjadwalan M/K.

Pengembang sistem operasi menerapkan penjadwalan dengan cara mengatur antrian permintaan untuk masing-masing perangkat. Ketika sebuah aplikasi meminta sebuah *blocking* sistem M/K, permintaan tersebut diletakkan dalam daftar antrian untuk perangkat yang berkaitan dengan M/K dari aplikasi tersebut. Penjadwalan M/K mengurutkan urutan antrian untuk meningkatkan efisiensi sistem secara keseluruhan dan waktu tunggu rata-rata (*average waiting time*) dari sebuah aplikasi. Sistem operasi juga mencoba untuk menjadi adil sehingga tidak satupun aplikasi yang akan mendapatkan layanan yang lebih sedikit, atau sistem operasi memberikan prioritas layanan antara banyak permintaan aplikasi. Sebagai contoh, permintaan dari subsistem memori virtual akan mendapatkan prioritas yang lebih tinggi daripada permintaan aplikasi.

Salah satu cara subsistem M/K meningkatkan efisiensi sebuah komputer adalah dengan menjadwalkan operasi M/K. Cara lain adalah menggunakan ruang penyimpanan pada memori utama atau pada disk melalui teknik yang disebut *buffering* , *caching*, dan *spooling*.

11.3. *Cache, Buffer, Spool*

Cache

Cache adalah sebuah daerah memori cepat yang berisi salinan data. Akses ke sebuah salinan yang di-*cache* lebih efisien daripada akses ke data yang asli. Sebagai contoh, instruksi-instruksi yang baru saja

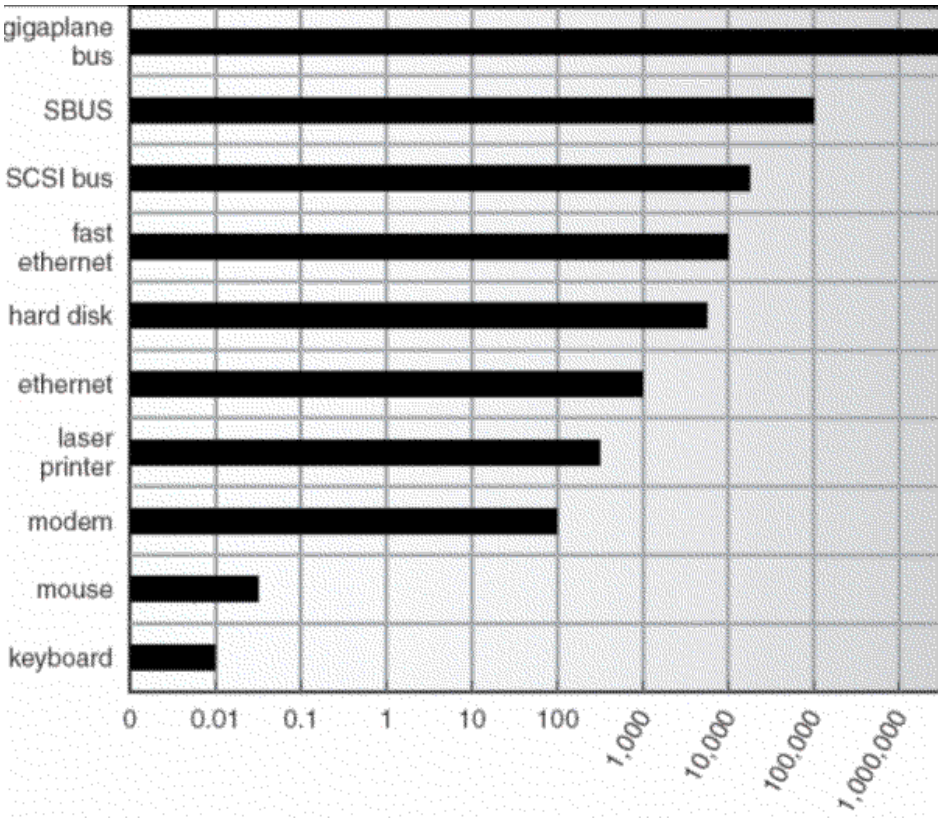
menjalankan proses-proses yang disimpan dalam disk,proses tersebut di- *cache* ke memori fisikal, dan disalin lagi ke *cache* primer dan sekunder dari CPU. Perbedaan sebuah *buffer* dan *cache* adalah sebuah *buffer* berisi salinan informasi data yang sudah ada, sedangkan sebuah *cache* berisi sebuah salinan data pada ruang penyimpanan yang dapat diakses dengan cepat informasi data ada di ruang lainnya.

Caching dan *buffering* adalah dua fungsi yang berbeda, tetapi kadang-kadang sebuah daerah memori dapat menggunakan kedua fungsi tersebut. Sebagai contoh, untuk menghemat *copy semantic* dan untuk membuat penjadwalan M/K menjadi efisien, sistem operasi menggunakan memori utama untuk menyimpan data yang ada di dalam disk. *Buffer-buffer* ini juga digunakan sebagai sebuah *cache* untuk meningkatkan efisiensi M/K pada berkas yang digunakan secara bersama-sama oleh beberapa aplikasi, atau sedang ditulis atau dibaca berulang-ulang. Ketika kernel menerima permintaan sebuah berkas, pertama-tama kernel mengakses *buffer cache* untuk melihat apakah daerah berkas tersebut sudah tersedia dalam memori utama. Jika daerah itu ada, disk fisikal M/K dapat dihindari atau tidak dipakai. Penulisan disk juga diakumulasikan ke *buffer cache* dalam beberapa detik sehingga transfer data yang besar dikumpulkan untuk mengefisienkan penjadwalan penulisan. Strategi penundaan penulisan ini untuk meningkatkan efisiensi M/K akan dibahas pada bagian *remote file access*.

Buffer

Buffer adalah sebuah daerah memori yang menyimpan data ketika data tersebut ditransfer antara dua perangkat atau antara sebuah perangkat dan sebuah aplikasi. *Buffering* digunakan karena tiga alasan, antara lain:

Gambar 11.1. Ukuran Transfer Data berbagai Perangkat



- **Untuk mengatasi perbedaan kecepataan antara produsen dan konsumen dari sebuah aliran data .** Sebagai contoh, sebuah berkas diterima melalui sebuah modem dan disimpan ke *harddisk*. Kita tahu bahwa modem itu ribuan kali lebih lambat daripada *harddisk*. Sehingga sebuah *buffer* dibuat pada memori utama untuk menampung jumlah byte yang diterima dari modem. Ketika semua data sudah sampai di *buffer*, *buffer* dapat ditulis ke disk dengan operasi tunggal. Karena penulisan ke

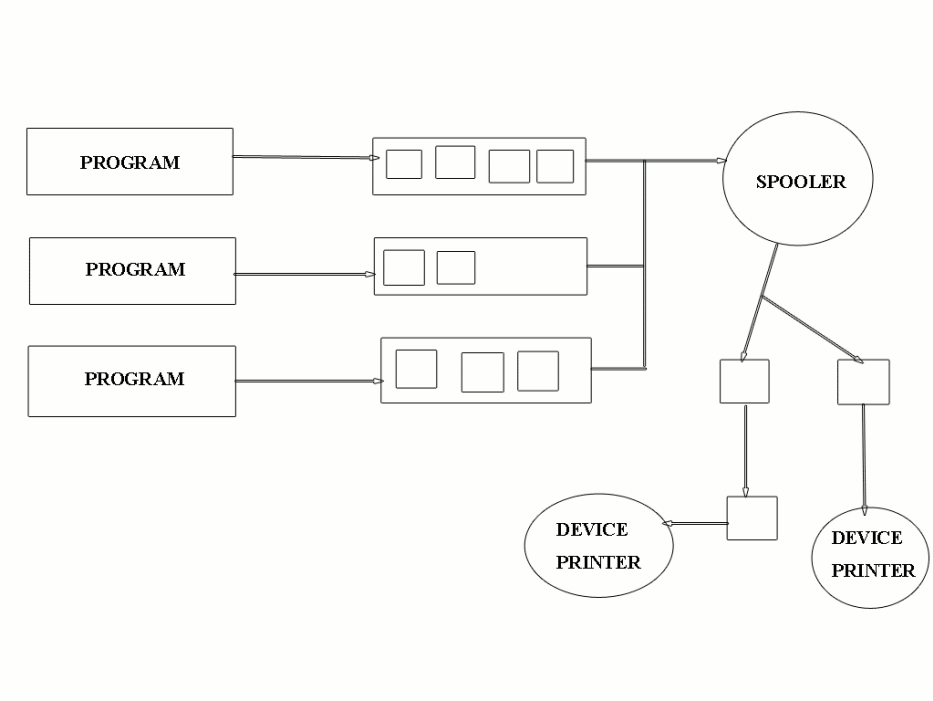
disk tidak terjadi seketika dan modem masih membutuhkan ruang untuk menyimpan data tambahan yang masuk maka digunakanlah dua *buffer*. Setelah modem mengisi *buffer* pertama, penulisan ke disk dilakukan. Modem kemudian mulai mengisi *buffer* kedua sementara *buffer* pertama ditulis ke disk. Pada saat modem sudah mengisi *buffer* kedua, penulisan disk dari *buffer* pertama seharusnya sudah selesai sehingga modem dapat menggunakan kembali *buffer* pertama ketika *buffer* kedua melakukan penulisan ke disk. Metode ini disebut *double buffering*.

- **Untuk menyesuaikan antara perangkat-perangkat yang mempunyai perbedaan ukuran transfer data .** Perbedaan ukuran transfer data ini sangat umum terjadi pada jaringan komputer dimana *buffer* digunakan secara luas untuk fragmentasi dan pengaturan kembali pesan-pesan. Pada bagian pengiriman, pesan yang ukurannya besar akan dipecah-pecah menjadi paket-paket kecil (fragmentasi). Paket-paket ini dikirim melalui jaringan, di ruang penerimaan, paket-paket kecil tadi diletakkan dalam *buffer* untuk disatukan kembali.
- **Untuk mendukung copy semantic pada aplikasi M/K .** Sebuah contoh akan menjelaskan arti dari *copy semantic*. Misalkan sebuah aplikasi mempunyai *buffer* data yang ingin dituliskan ke disk. Aplikasi tersebut akan memanggil *system call write*, lalu menyediakan sebuah pointer ke *buffer* dan sebuah bilangan bulat (integer) yang menspesifikasikan jumlah byte yang ditulis. Setelah *system call* tersebut selesai, apa yang terjadi jika aplikasi mengubah isi *buffer*? Dengan *copy semantic*, versi data yang ditulis ke disk sama dengan versi data pada saat aplikasi memanggil *system call write* , tidak tergantung dengan perubahan apapun yang ada pada *buffer*. Cara sederhana sistem operasi dapat menjamin *copy semantic* adalah untuk *system call write* dengan menyalin data aplikasi ke *buffer kernel* sebelum mengembalikan kontrol ke aplikasi. Penulisan ke disk dilakukan dari *buffer kernel* sehingga perubahan yang terjadi pada *buffer aplikasi* tidak mempunyai efek apapun. Menyalin data antara *buffer kernel* dan *buffer aplikasi* adalah hal yang umum dalam sistem operasi, kecuali *overhead* yang ada pada *clean semantic*. Efek yang sama dapat diperoleh dengan hasil yang lebih efisien dengan penggunaan yang cermat pada pemetaan memori virtual dan perlindungan halaman *copy-on-write*.

Spool

Spool adalah sebuah *buffer* yang berisi keluaran untuk sebuah perangkat, seperti sebuah printer dimana aliran data tidak dapat mengalir bersamaan. Walaupun printer hanya dapat melayani satu pekerjaan pada satu waktu, beberapa aplikasi mungkin ingin mencetak keluaran masing-masing secara bersamaan tanpa harus tercampur. Sistem operasi menyelesaikan masalah ini dengan cara meng-*intercept* semua keluaran tersebut ke printer. Masing-masing keluaran aplikasi tadi di-*spool* ke disk berkas yang terpisah. Ketika sebuah aplikasi selesai mencetak keluarannya, sistem *spooling* akan melanjutkan ke antrian berikutnya. Pada beberapa sistem operasi, *spooling* ditangani oleh sebuah sistem proses *daemon* yaitu suatu sistem yang terus mengawasi apakah aliran data berjalan lancar. Pada sistem operasi lainnya, *spooling* ditangani oleh sebuah *thread in- kernel*. Pada kedua macam penanganan *spooling* tersebut, sistem operasi menyediakan kontrol antarmuka yang membolehkan user dan sistem administrator untuk membentuk antrian, untuk membuang job yang tidak diinginkan sebelum job tersebut dicetak, untuk menunda pencetakan ketika printer diperbaiki, dan sebagainya.

Gambar 11.2. Spooling



Beberapa perangkat, seperti *tape drives* dan printer tidak dapat mengumpulkan permintaan M/K dari banyak aplikasi secara bersamaan. Cara lain adalah dengan menggunakan akses perangkat secara bersamaan dengan menyediakan fasilitas langsung dengan cara koordinasi. Beberapa sistem operasi (termasuk Virtual Machine System) menyediakan dukungan akses perangkat secara ekslusif dengan mengalokasikan sebuah proses ke perangkat yang menganggur atau *idle* dan membuang perangkat tersebut jika sudah tidak diperlukan lagi. Sistem operasi lainnya memaksakan sebuah batasan dari penanganan sebuah berkas yang dibuka ke perangkat tersebut. Kebanyakan sistem operasi menyediakan fungsi yang membuat proses-proses untuk menangani koordinat akses eksklusif diantara mereka sendiri. Sebagai contoh, Windows NT menyediakan *system call* untuk menunggu objek perangkat sampai statusnya tersedia. Dia juga mempunyai sebuah parameter untuk *system call open* yang mendeklarasikan tipe-tipe akses yang diperbolehkan untuk *thread*\ lainnya secara bersamaan. Pada sistem ini, penghindaran *deadlock* diserahkan kepada aplikasi.

11.4. Proteksi M/K

Sebuah sistem operasi yang menggunakan *protected memory* dapat menjaga banyak kemungkinan error yang terjadi pada aplikasi maupun perangkat keras sehingga sebuah *failure* sistem yang sulit tidak biasanya terjadi pada kesalahan mekanik yang kecil. Perangkat dan transfer M/K dapat gagal dalam berbagai cara karena alasan *transient* seperti jaringan yang *overloaded* atau karena alasan permanen seperti pengontrol disk yang rusak. Sistem operasi biasanya dapat mengganti kerugian secara efektif untuk kegagalan *transient*. Sebagai contoh, sebuah kegagalan pembacaan disk berakibat pengulangan pembacaan disk itu lagi dan jaringan mengirimkan hasil yang salah dalam pengiriman ulang jika protokol diketahui. Jika sebuah kegagalan permanen terjadi pada sebuah komponen yang penting, sistem operasi tidak dapat memulihkan error yang terjadi.

Ada aturan umum yang menyebutkan sebuah *system call* M/K akan mengembalikan 1 bit informasi tentang status pemanggilan yang akan menandakan apakah pemanggilan tersebut sukses atau gagal. Pada sistem operasi UNIX, variable integer tambahan bernama *errno* digunakan untuk mengembalikan sebuah kode kesalahan dalam rentang 1 dari 100 nilai yang menandakan jenis dari kesalahan tersebut, (sebagai contoh: argumen yang keluar dari rentang batas yang disediakan, *bad pointer* atau berkas yang tidak dapat dibuka).

Sebaliknya, beberapa perangkat keras dapat menyediakan informasi yang lebih lengkap tentang kegagalan tersebut walaupun banyak sistem operasi tidak dibuat untuk menyampaikan informasi ini kepada aplikasi. Sebagai contoh:

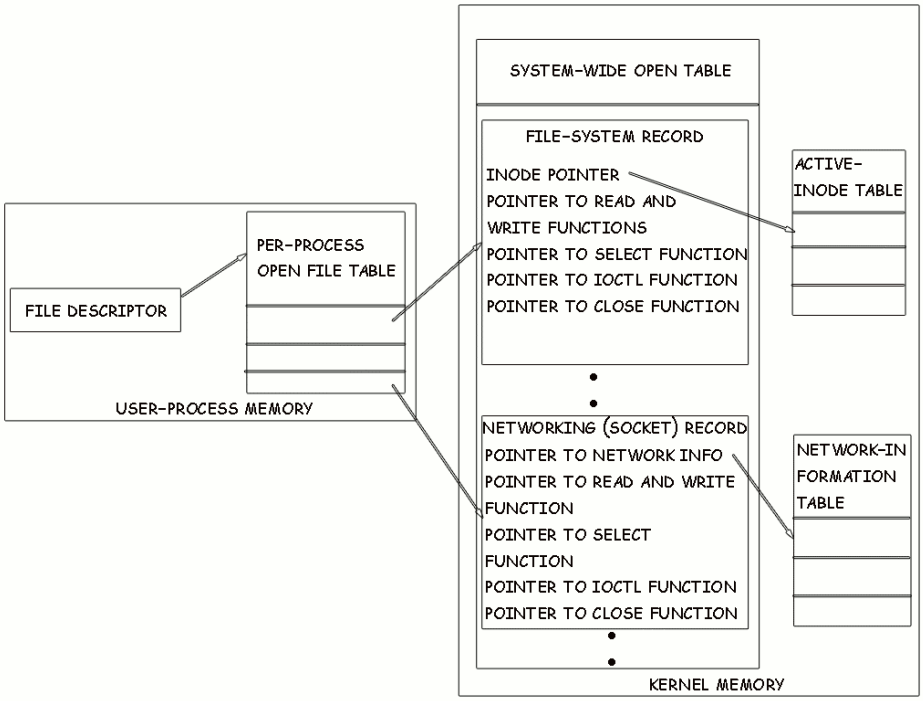
1. kegagalan pada perangkat SCSI dilaporkan oleh protokol SCSI dalam bentuk *sense key* yang memberitahukan jenis kegagalan seperti error pada perangkat keras atau permintaan yang ilegal,
2. ada juga yang disebut *additional sense-code* yang menyatakan kategori dari kegagalan seperti parameter perintah yang tidak sesuai atau kesalahan *self-test* seperti pada *booting*, *load program*, dan *bootstrapping*.
3. sebuah *additional sense-code qualifier* yang memberikan informasi yang lebih detil seperti parameter perintah yang error atau subsistem perangkat keras yang gagal dalam *self-test*.

11.5. Struktur Data

Kernel menyimpan informasi keadaan penggunaan komponen M/K, maka kernel melakukan hal tersebut pada bermacam-macam struktur data kernel seperti struktur *table open-file* . UNIX menyediakan akses sistem berkas untuk berbagai macam entitas seperti berkas pengguna, *raw device*, dan ruang alamat dari proses-proses.Walaupun masing-masing entitas ini mendukung operasi baca, tetapi semantik untuk masing-masing entiti berbeda. Sebagai contoh, untuk membaca berkas pengguna, kernel perlu memeriksa *buffer cache* sebelum memutuskan untuk melakukan M/K disk. Untuk membaca *raw disk*, kernel perlu meyakinkan bahwa ukuran permintaan adalah kelipatan dari ukuran disk dan masih terdapat dalam sektor berkas. Untuk memproses citra, cuma perlu menyalin data dari memori. UNIX menyembunyikan perbedaan-perbedaan dalam struktur yang seragam dengan menggunakan teknik berorientasi objek atau *object-oriented*. *Open-file record* pada gambar adalah sebuah tabel *dispatch* yang berisi pointer ke *routine* yang bersesuaian tergantung pada tipe berkas.

Beberapa sistem operasi menggunakan metode *object-oriented* secara lebih ekstensif. Sebagai contoh, Windows NT menerapkan metode *message-passing* untuk M/K. Sebuah permintaan M/K diubah menjadi sebuah pesan yang dikirim melalui kernel kepada manajer M/K dan kemudian kepada *device driver* ,yang masing-masing mereka dapat mengubah isi pesan. Untuk keluaran pesan tersebut, digunakan *buffer* untuk menerima data. Pendekatan *message-passing* ini bisa menambah biaya, perbandingan dengan teknik yang menggunakan strukutur data yang dibagi-bagi menyederhanakan struktur dan perancangan sistem M/K serta menambah fleksibilitas.

Gambar 11.3. Struktur Kernel M/K pada UNIX



Kesimpulannya, subsistem M/K mengkoordinir kumpulan layanan yang ekstensif yang tersedia untuk aplikasi dan bagian lainnya dari kernel. Subsistem M/K mengawasi:

1. Manajemen nama berkas dan perangkat
2. Kontrol akses untuk berkas dan perangkat
3. Operasi kontrol, contoh: modem yang tidak dapat dikenali
4. Alokasi ruang sistem berkas
5. Alokasi perangkat
6. *Buffering*, *caching*, dan *spooling*
7. Penjadwalan M/K
8. Pengawasan status perangkat, penanganan *error* dan pemulihan kegagalan
9. Konfigurasi dan inisialisasi *device driver*

11.6. Operasi Perangkat Keras

Pada bagian sebelumnya, kita menjelaskan *handshaking* antara sebuah *device driver* dan sebuah *device controller*, tapi kita tidak menjelaskan bagaimana sistem operasi menghubungkan permintaan aplikasi ke dalam kumpulan kabel jaringan atau ke dalam sektor disk yang spesifik.

Aplikasi menunjuk data dengan menunjuk sebuah nama berkas. Di dalam sebuah disk, hal ini adalah pekerjaan dari sistem berkas untuk memetakan dari nama berkas melalui direktori sistem berkas untuk memperoleh ruang pengalokasian berkas. Sebagai contoh, dalam MS-DOS, nama dipetakan ke sebuah nomor yang mengindikasikan sebuah entri dalam tabel akses berkas, dan entri tabel tersebut mengatakan blok disk dialokasikan ke berkas. Pada UNIX, nama dipetakan ke sebuah nomor cabang dan nomor cabang yang bersesuaian berisi informasi tempat pengalokasian.

Sekarang kita lihat MS-DOS, sistem operasi yang relatif sederhana.

1. Bagian pertama dari sebuah berkas MS-DOS diikuti tanda titik dua, adalah sebuah string yang menandakan sebuah perangkat keras yang spesifik. Sebagai contoh, c:\ adalah bagian pertama dari setiap nama berkas pada hard disk utama.
2. Fakta bahwa c: mewakili hard disk utama yang dibangun ke dalam sistem operasi, c:\ dipetakan ke alamat spesifik melalui *device table*.
3. Karena pemisah tanda titik dua tadi, tempat nama perangkat dipisahkan dari ruang nama sistem berkas di dalam masing-masing perangkat. Pemisahan ini memudahkan sistem operasi untuk menghubungkan fungsi tambahan untuk masing-masing perangkat. Sebagai contoh, hal yang mudah untuk melakukan *spooling* pada banyak berkas yang akan dicetak ke printer.

Jika ruang nama perangkat disertakan dalam ruang nama sistem berkas seperti pada UNIX, layanan nama sistem berkas yang normal disediakan secara otomatis. Jika sistem berkas menyediakan kepemilikan dan kontrol akses untuk semua nama berkas, maka perangkat mempunyai pemilik dan kontrol akses. Karena berkas disimpan dalam perangkat, sebuah antarmuka penguhubung menyediakan sistem M/K pada dua tingkatan. Nama dapat digunakan perangkat untuk mengakses dirinya sendiri atau untuk mengakses berkas yang disimpan pada perangkat tersebut.

UNIX menghadirkan nama perangkat dalam ruang nama sistem berkas reguler. Tidak seperti sebuah nama berkas MS-DOS yang mempunyai tanda titik dua, alur nama pada UNIX tidak mempunyai pemisahan yang jelas dari bagian nama perangkat. Faktanya, tidak ada bagian dari nama alur adalah nama perangkat.

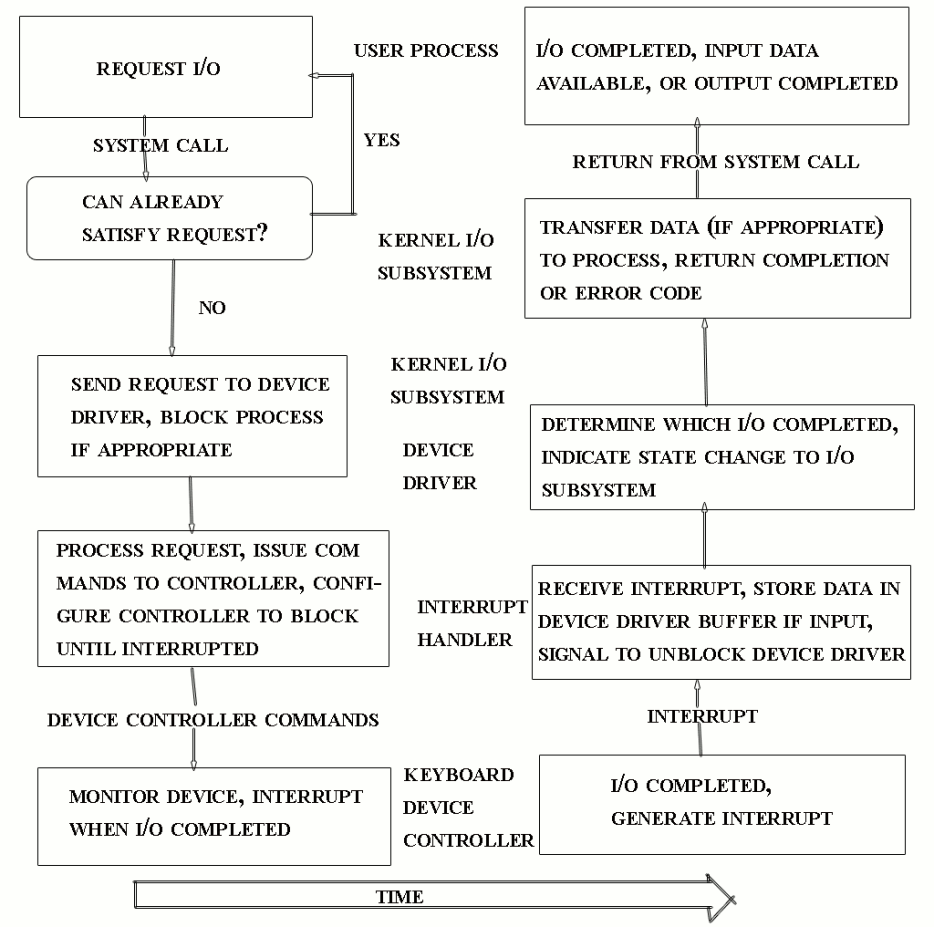
1. UNIX mempunyai sebuah *mount table* yang menghubungkan awalan dari nama alur dengan nama perangkat yang sesuai.
2. Untuk memecahkan masalah alur nama ini, UNIX mencari nama ini di dalam *mount table* untuk mencari awalan(prefix) yang paling cocok. entry dalam *mount table* yang bersesuaian adalah nama perangkat.
3. Nama perangkat ini juga mempunyai bentuk dari sebuah nama dari ruang nama sistem berkas.
4. Ketika UNIX mencari nama ini di dalam strukutur direktori sistem berkas, daripada mencari nomor cabang, UNIX mencari sebuah nomor perangkat *major* dan *minor*.

5. Nomor perangkat *major* menandakan sebuah *device driver* yang harus dipanggil untuk menangani M/K untuk perangkat ini.
6. Nomor perangkat *minor* dikirim ke *device driver* untuk diindekskan ke tabel perangkat.
7. Entri tabel perangkat yang bersesuaian memberikan alamat atau alamat pemetaan memori dari pengendali perangkat.

Sistem operasi modern mendapatkan fleksibilitas yang sangat penting dari tahapan-tahapan mencari tabel dalam jalur antara permintaan dan pengendali perangkat fisikal. Mekanisme yang melewatkan permintaan antara aplikasi dan *driver* adalah hal yang umum. Oleh sebab itu, kita dapat menambahkan perangkat dan *driver* baru ke dalam komputer tanpa mengkompilasi ulang kernel. Faktanya, beberapa sistem operasi mempunyai kemampuan untuk menambahkan *device driver* yang diinginkan. Pada waktu *boot*, sistem pertama-tama memeriksa bus perangkat keras untuk menentukan perangkat apa yang tersedia dan kemudian sistem menambahkan atau *load driver* yang diperlukan secara langsung atau ketika ada permintaan M/K yang membutuhkannya.

Berikut dideskripsikan sebuah *stream* yang unik dari sebuah permintaan blokir membaca.

Gambar 11.4. Lifecycle of I/O request



Penjelasan gambar di atas sebagai berikut:

1. Sebuah proses mengeluarkan sebuah blocking untuk *system call* baca untuk sebuah deskriptor berkas dari berkas itu yang sudah pernah dibuka sebelumnya.
2. Kode *system call* dalam kernel memeriksa kebenaran parameter. Pada masukan, jika data tersedia dalam *buffer cache*, data dikembalikan ke proses dan permintaan M/K selesai.
3. Jika data tadi tidak tersedia di *buffer cache*, M/K fisikal perlu dilakukan sehingga proses akan dikeluarkan dari antrian yang sedang berjalan dan ditempatkan pada antrian *wait* untuk perangkat tersebut, dan permintaan M/K pun dijadwalkan. Secepatnya, subsistem M/K mengirimkan

permintaan ke *device driver*. Bergantung pada sistem operasi, permintaan dikirim melalui pemanggilan *subroutine* atau melalui pesan in-kernel.

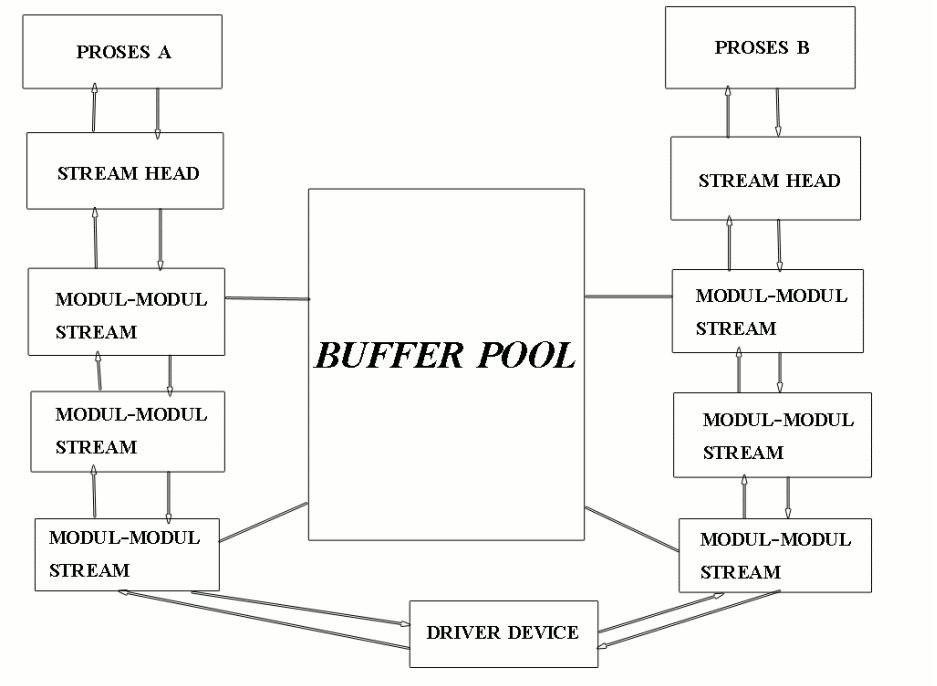
4. *Device driver* mengalokasikan ruang *kernel buffer* untuk menerima data dan menjadwalkan M/K. Secepatnya, *driver* mengirim perintah untuk pengendali perangkat dengan menulis ke dalam register yang mengontrol perangkat tersebut.
5. Pengendali perangkat mengoperasikan perangkat keras untuk melaksanakan transfer data
6. Driver dapat menerima status dan data atau dapat menyiapkan sebuah transfer DMA ke dalam memori kernel. Kita asumsikan bahwa transfer diatur oleh sebuah pengendali DMA yang akan menghasilkan interupsi ketika transfer data selesai.
7. *Interrupt handler* yang sesuai menerima interupsi melalui tabel vektor interupsi atau *interrupt-vector table*, menyimpan data yang diperlukan, memberi sinyal kepada *device driver*, kembali dari interupsi.
8. *Device driver* menerima sinyal, menentukan permintaan M/K yang mana yang telah selesai, menentukan status permintaan, dan memberi sinyal kepada subsistem M/K kernel yang permintaannya sudah selesai.
9. Kernel mentransfer data atau mengembalikan kode untuk ruang alamat dari proses yang diminta, dan memindahkan proses dari antrian *wait* ke antrian *ready*.
10. Memindahkan proses ke antrian *ready* tidak memblok proses tersebut. Ketika penjadwal atau *scheduler* menaruh proses ke CPU, proses itu akan melanjutkan eksekusi pada penyelesaian *system call*.

11.7. *STREAMS*

Sistem V UNIX mempunyai mekanisme yang menarik yang disebut *stream* yang membuat aplikasi dapat memakai *pipeline* dari *driver code* secara dinamis. Sebuah *stream* adalah sebuah koneksi *full duplex* antara sebuah *device driver* dan proses di tingkat pengguna. *Stream* terdiri dari sebuah:

1. *stream head* yang terhubung dengan proses pengguna.
2. sebuah *driver end* yang mengendalikan perangkat.
3. banyak *stream modules* antara *stream head* dan *driver end*.

Gambar 11.5. STREAMS



Module dapat dimasukkan ke dalam sebuah arus atau *stream* untuk menambah fungsionalitas di sebuah model berlapis atau *layer*. Sebagai contoh, sebuah proses dapat membuka alat port serial melalui

sebuah *stream* dan dapat memasukkan sebuah *module* untuk menangani modifikasi pada masukan. Streams dapat digunakan untuk komunikasi antara proses dan jaringan. Sebagai fakta, pada sistem V, mekanisme soket diterapkan oleh *stream*.

11.8. Kinerja

M/K adalah faktor terpenting dalam kinerja sistem M/K. M/K menempatkan permintaan yang besar pada CPU untuk menjalankan kode *device driver* dan menjadwalkan proses dengan adil dan efisien ketika proses tersebut diblok atau tidak diblok. Hasil *context switch* menekan CPU dan cache perangkat kerasnya. M/K juga memberitahukan ketidakefisienan mekanisme penanganan interupsi dalam kernel, dan M/K mengisi bus memori ketika data disalin antara pengendali dan memori fisikal, dan lagi-lagi ketika data disalin antara *buffer kernel* dan ruang data aplikasi. Menyalin dengan semua perintah-perintah ini adalah hal yang sangat diperhatikan dalam arsitektur komputer.

Walaupun komputer modern dapat menangani ratusan interupsi tiap detik, penanganan interupsi adalah tugas yang mahal. setiap instruksi menyebabkan sistem melakukan perubahan status untuk menjalankan interrupt handler dan kemudian untuk mengembalikan status seperti semula. M/K terprogram dapat lebih efisien daripada M/K yang dikendalikan oleh interupsi, jika jumlah cycle yang dihabiskan pada saat sibuk menunggu tidak berlebihan. M/K yang sudah selesai biasanya meng-*unblock* sebuah proses membawanya menjadi sebuah *context-switch* sepenuhnya.

Lalu lintas jaringan juga dapat menyebabkan tingkat *context-switch* yang tinggi. Sebagai contoh, login jarak jauh diantaranya:

1. Setiap karakter yang diketik pada mesin lokal harus dipindahkan ke mesin jarak jauh atau *remote machine* . Pada mesin lokal, karakter diketik; interupsi keyboard dihasilkan; dan karakter melewati *interrupt handler* ke *device driver*, menuju kernel, dan kemudian ke proses pengguna.
2. Proses pengguna mengeluarkan sebuah *system call* M/K jaringan untuk mengirimkan karakter menuju mesin jarak jauh. Karakter kemudian mengalir ke dalam kernel lokal melalui lapisan jaringan yang membangun sebuah paket jaringan dan menuju *device driver* jaringan.
3. *Device driver* jaringan mentransfer paket itu kepada pengendali jaringan yang mengirimkan karakter dan menghasilkan sebuah interupsi. Interupsi dilewatkan melalui kernel supaya *system call* M/K jaringan selesai.
4. Sekarang, perangkat keras jaringan sistem jarak jauh menerima paket dan sebuah interupsi dihasilkan. Karakter dibuka paketnya dari protokol jaringan dan diberikan kepada jaringan daemon yang sesuai. Jaringan daemon mengidentifikasi *login* jarak jauh mana yang terlibat dan mengirim paket kepada subdaemon yang sesuai untuk sesi tersebut.
5. Keseluruhan aliran pengiriman dan penerimaan pesan ini ada konsep *context-switch* dan *state-switch* ada pada gambar Komunikasi antar komputer. Biasanya, penerima mengirimkan kembali karakter kembali ke pengirim; pendekatan ini menggandakan pekerjaan

Pengembang Solaris menerapkan kembali telnet daemon menggunakan *thread in-kernel* untuk menghapus context switch yang terlibat dalam pemindahan karakter antara daemon dan kernel. Sun memperkirakan bahwa peningkatan ini meningkatkan jumlah maksimum *login* jaringan dari cuma beberapa ratus menjadi beberapa ribu pada server yang lebih besar.

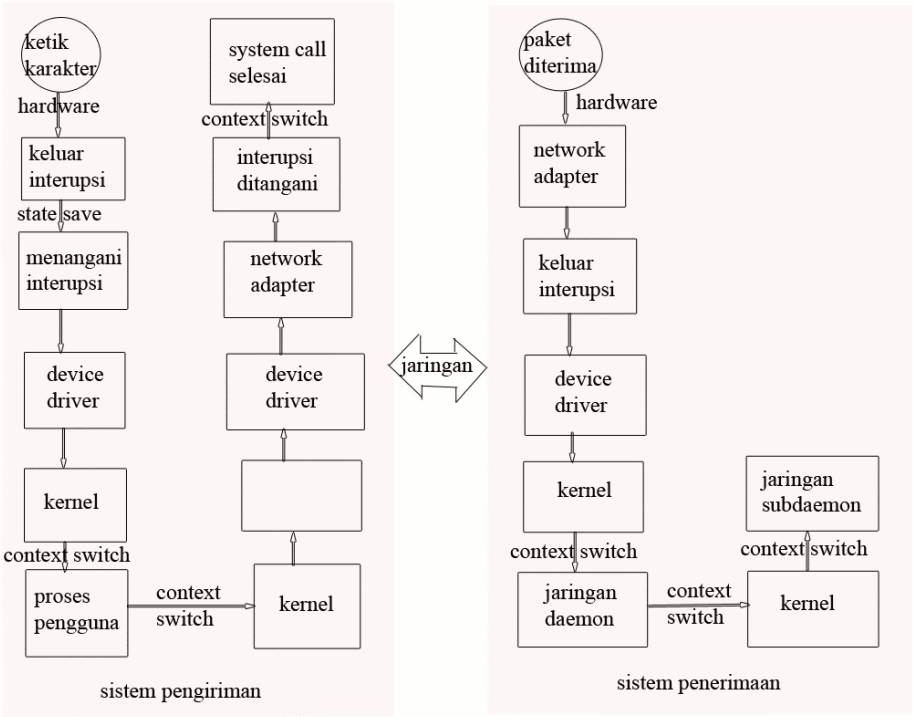
Sistem lain menggunakan *front-end processor* terpisah untuk terminal M/K untuk mengurangi beban interupsi pada CPU utama. Sebagai contoh, terminal *concentrator* dapat mengirim sinyal secara bersamaan dari ratusan terminal jarak jauh ke dalam satu port pada sebuah komputer yang besar. Sebuah *channel* M/K bersifat *dedicated* artinya CPU ang mempunyai tujuan khusus yang ditemukan pada bingkai utama dan sistem high-end lainnya. Pekerjaan dari sebuah channel adalah untuk mengambil pekerjaan M/K dari CPU utama. Prinsipnya adalah channel menjaga lalu lintas data lancar ketika CPU utama dapat bebas memproses data. Seperti *device controller* dan *DMA controller* yang ditemukan pada komputer berukuran kecil, sebuah channel dapat memproses program yang lebih umum dan canggih sehingga channel dapat digunakan untuk *workload* tertentu.

Kita dapat menggunakan beberapa prinsip untuk meningkatkan efisiensi M/K:

1. mengurangi jumlah *context-switch*.

2. mengurangi jumlah waktu untuk data harus disalin ke dalam memori ketika dikirim antara perangkat dan aplikasi.
3. mengurangi frekuensi interupsi dengan menggunakan transfer besar-besaran, *smart controller* dan *polling* jika *busy waiting* dapat diminimalkan.
4. meningkatkan konkurensi dengan menggunakan DMA *controller* ang sudah diketahui atau channel untuk mengambil salinan data sederhana dari CPU.
5. memindahkan pemrosesan primitif ke perangkat keras supaya operasi pada *device controller* dapat berjalan bersamaan dengan CPU dan operasi bus.
6. seimbangkan antara CPU, subsistem memori, bus dan kinerja M/K karena sebuah *overloaded* pada salah satunya akan menyebabkan *idle* pada yang lain.

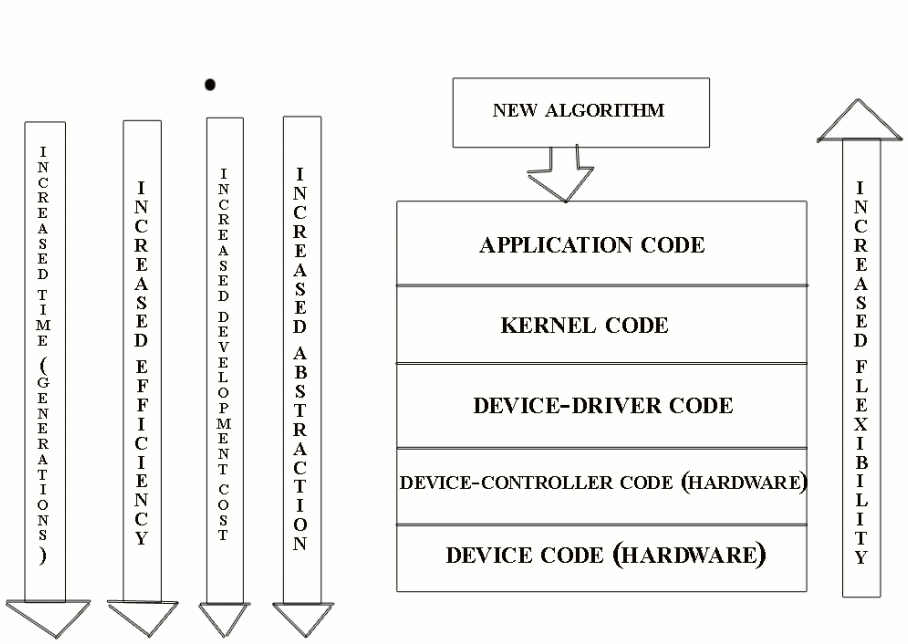
Gambar 11.6. Komunikasi antar komputer



Perangkat berbeda-beda dalam kompleksitas, sebagai contoh sebuah mouse yang sederhana. Pergerakan mouse dan button click diubah menjadi nilai numerik yang akan dikirim dari perangkat keras melalui *mouse device driver* lalu menuju aplikasi. Sebaliknya, fungsionalitas yang disediakan disk windows NT device driver sangatlah kompleks. Tidak hanya mengatur disk tunggal, tetapi juga menerapkan RAID array. Untuk melakukan hal tersebut, NT device driver mengubah permintaan baca dan tulis aplikasi menjadi sebuah kumpulan operasi disk M/K. Lebih dari itu, NT device driver juga menerapkan penanganan error yang canggih dan algoritma pemulihan data, dan mengambil banyak langkah untuk mengoptimalkan kinerja disk karena pentingnya kinerja penyimpanan sekunder untuk kinerja sistem secara keseluruhan.

Gambar di bawah ini menjelaskan pergerakan fungsionalitas perangkat dalam meningkatkan kinerja M/K.

Gambar 11.7. Peningkatan Fungsionalitas Perangkat



1. Mula-mula, kita terapkan eksperimen algoritma M/K pada tingkat aplikasi karena kode aplikasi bersifat fleksibel, dan bug pada aplikasi tidak menyebabkan system crash. Lebih jauh lagi, dengan mengembangkan kode pada tingkat aplikasi, kita menghindari kebutuhan untuk me-*reboot* atau me- *reload device driver* setelah setiap perubahan pada kode.

   Sebuah penerapan pada tingkatan aplikasi tidak dapat efisien. Bagaimana pun juga, karena biaya *context-switch*, dan karena aplikasi tidak dapat menerima kemudahan dari struktur data kernel internal dan fungsionalitas kernel seperti internal *kernel messaging*, *threading*, dan *locking*.
2. Ketika algoritma tingkat aplikasi sudah memperlihatkan kegunaannya, kita akan menerapkannya kembali pada kernel. Hal ini akan meningkatkan kinerja, tapi usaha pengembangan adalah hal yang lebih menantang karena sebuah sistem operasi kernel sangatlah besar dan sebuah sistem yang kompleks. Lebih dari itu, penerapan *in-kernel* harus di-*debug* secara hati-hati untuk menghindari *data corruption* dan *system crashes*.
3. Kinerja yang sangat tinggi mungkin bisa didapat dengan sebuah penerapan spesialisasi dalam perangkat keras, baik pada devicenya maupun pada controller. Ketidakuntungan penerapan perangkat keras ini meliputi kesukaran dan mahalnya pembuatan pengembangan lebih jauh atau untuk memperbaiki bug, waktu pengembangan yang bertambah, dan berkurangnya fleksibilitas. Sebagai contoh, sebuah pengendali RAID perangkat keras tidak menyediakan cara untuk kernel untuk mempengaruhi urutan atau lokasi dari pembacaan dan penulisan disk tunggal walaupun kernel mempunyai informasi khusus tentang *workload* yang akan memudahkan kernel untuk meningkatkan kinerja M/K.

11.9. Rangkuman

Subsistem M/K kernel menyediakan banyak layanan antara lain penjadwalan M/K, *buffering*, *caching*, *spooling*, dan penanganan error. Layanan lainnya adalah pengartian nama untuk membuat koneksi antara perangkat keras dan nama berkas yang digunakan oleh aplikasi. Hal ini melibatkan beberapa tingkatan pemetaan yang mengartikan dari nama karakter string untuk *device driver* yang spesifik dan alamat perangkat, dan kemudian untuk alamat fisikal dari port M/K atau *bus controller*. Pemetaan ini terjadi pada ruang sistem berkas di UNIX atau dipisahkan pada ruang alamat perangkat seperti pada MS-DOS. *System call* M/K mahal pada masalah konsumsi CPU karena banyaknya layer pada aplikasi antara perangkat fisikal dan aplikasi. Layer-layer ini menyiratkan biaya *context switching*

untuk melalui batas proteksi kernel dari sinyal dan penanganan interupsi untuk melayani perangkat M/K dan memasukkannya ke dalam CPU dan sistem memori untuk menyalin data antara *buffer kernel* dan ruang aplikasi.

Rujukan

[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.

[WEBWIKI2007] Wikipedia. 2007. *Self-test http://en.wikipedia.org/wiki/Power-on_self_test* . Diakses 29 Apr 2007.

Bab 12. M/K Linux

12.1. Pendahuluan

– FIX ME –

12.2. Perangkat Blok

– FIX ME – 21.8.1

12.3. Perangkat Karakter

– FIX ME – 21.8.2

12.4. Perangkat Jaringan

– FIX ME – Linux Kernel

12.5. Penjadwal Linux

– FIX ME – Linux Kernel

12.6. Elevator Linus

– FIX ME – Linux Kernel

12.7. Elevator Linus

– FIX ME – Linux Kernel

12.8. Antrian M/K

– FIX ME – Linux Kernel

12.9. Waktu Tengat M/K

– FIX ME – Linux Kernel

12.10. Rangkuman

– FIX ME –

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 VII. Penyimpanan Masal

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 13. Sistem Berkas

13.1. Pendahuluan

Semua aplikasi komputer membutuhkan penyimpanan dan pengambilan informasi. Ketika sebuah proses sedang berjalan, proses tersebut menyimpan sejumlah informasi yang dibatasi oleh ukuran alamat virtual. Untuk beberapa aplikasi, ukuran ini cukup, namun untuk yang lainnya terlalu kecil.

Masalah berikutnya adalah apabila proses tersebut berhenti, maka informasinya akan hilang. Padahal ada beberapa informasi yang penting dan harus bertahan beberapa waktu, bahkan selamanya. Adapun masalah ketiga adalah terkadang sangat perlu lebih dari satu proses untuk mengakses satu informasi secara bersamaan. Untuk memecahkan masalah ini, informasi ini harus dapat berdiri sendiri tanpa bergantung pada sebuah proses.

Pada akhirnya, kita memiliki masalah- masalah yang cukup signifikan dan penting untuk dicari solusinya yaitu :

1. Kita harus dapat menyimpan informasi dengan ukuran yang besar.
2. Informasi harus tetap ada, meskipun proses berhenti.
3. Informasi harus dapat diakses oleh lebih dari satu proses secara bersamaan.

Solusi dari ketiga masalah diatas adalah dengan kehadiran berkas.

13.2. Konsep Berkas

Berkas atau *file* adalah kumpulan dari informasi yang saling berhubungan yang disimpan di *secondary storage*. Berkas dapat dipandang sebagai bagian terkecil dari *secondary storage*, sehingga dapat dikatakan bahwa data tidak dapat disimpan di *secondary storage* kecuali jika data tadi ditulis dalam berkas.

Setiap berkas mempunyai struktur tersendiri tergantung dari apa tipe berkas tersebut, diantaranya:

- ***Text File .*** Barisan dari karakter-karakter yang berurutan yang disusun dalam baris-baris atau halaman-halaman
- ***Source File .*** Sekumpulan barisan subrutin dan fungsi yang disusun atas deklarasi disusul dengan pernyataan atau *statement* yang dapat dijalankan.
- ***Object File .*** Sebarisan byte yang diatur menjadi blok-blok yang dapat dimengerti oleh penghubung sistem (*linker system*).
- ***Executable File .*** Sekumpulan bagian kode yang dapat dibawa ke memori untuk dijalankan oleh *loader*.

13.3. Atribut Berkas

Setiap berkas diberi nama agar mempermudah kita untuk membedakannya. Nama berkas biasanya terdiri dari sederetan karakter. Dalam beberapa sistem, huruf besar dan huruf kecil dari nama berkas tersebut dianggap sama, sedangkan untuk sistem yang lain, hal itu dianggap berbeda. Setelah berkas diberi nama, nama dari berkas tersebut menjadi independen terhadap proses. Maksud dari independen disini adalah bahwa tindakan yang dilakukan user terhadap berkas, misalnya mengedit isinya, menyalin berkas tersebut ke disk, mengirimkannya lewat email dan tindakan-tindakan lainnya tidak akan mengubah nama dari berkas tersebut.

Berkas mempunyai bermacam-macam atribut, yang dapat berbeda-beda antar satu sistem operasi dengan sistem operasi lainnya. Tapi umumnya, sebuah berkas memiliki atribut sebagai berikut:

- **Nama.** Nama dari berkas yang dituliskan secara simbolik adalah satu-satunya informasi yang disimpan dalam bentuk yang dapat dibaca oleh kita.

- ***Identifier .*** *Tag* unik yang biasanya berupa angka, yang mengidentifikasikan berkas dalam sistem berkas. Identifier ini tidak dapat dibaca oleh manusia.
- **Jenis.** Informasi yang dibutuhkan sistem yang biasanya medukung bermacam-macam tipe berkas yang berbeda.
- **Lokasi.** Informasi yang berisi *pointer* ke *device* dan lokasi dari berkas dalam *device* tersebut.
- **Ukuran.** Ukuran dari berkas saat ini, dan mungkin ukuran maksimum berkas juga dimasukkan dalam atribut ini.
- **Waktu, tanggal dan identifikasi pengguna .** Informasi yang disimpan untuk pembuatan, modifikasi terakhir dan kapan berkas terakhir digunakan. Data-data ini dapat berguna dalam proteksi, keamanan dan monitoring penggunaan berkas.

13.4. Operasi Berkas

Fungsi dari berkas adalah sebagai penyimpanan data dan mengijinkan kita untuk membacanya lagi nantinya. Adapun operasi-operasi dasar yang dapat dilakukan berkas adalah sebagai berikut:

- **Membuat berkas (*Create File*).** Terdapat dua hal yang harus kita lakukan untuk membuat suatu berkas. Pertama, kita harus menemukan tempat dalam sistem berkas untuk berkas yang akan kita buat tadi. Kedua, adalah membuat *entry* untuk berkas tersebut. *Entry* ini mencatat nama dari berkas dan lokasinya dalam sistem.

- **Menulis sebuah berkas (*Write File*).** Untuk menulis berkas, kita membuat sebuah *system call* yang meyebutkan nama berkas dan informasi apa yang akan kita tulis dalam berkas tersebut. Setelah diberikan nama berkasnya, sistem akan mencari berkas yang akan kita tulis tadi dan meletakkan pointer di lokasi yang akan kita *write* berikutnya. *Pointer write* harus *diupdate* setiap kali *write* dilakukan.

- **Membaca sebuah berkas (*Read File*).** Untuk membaca berkas, kita menggunakan sebuah *system call* yang menspesifikasikan nama file dan di blok mana di memori berkas harus diletakkan. Lalu direktori kembali dicari hingga ditemukan *entry* yang bersesuaian. Sistem harus menjaga agar *pointer* berada di posisi dimana *read* berikutnya akan dilakukan. Setelah pembacaan berkas selesai, maka pointer akan di-*update*.

- **Memposisikan sebuah berkas (*Reposition*).** Direktori dicari untuk *entry* yang bersesuaian, lalu kemudian *current file position* dari berkas di set ke suatu nilai tertentu. Operasi berkas ini dikenal juga sebagai *file seek*.

- **Menghapus berkas (*Delete*).** Untuk menghapus sebuah berkas, kita mencari direktori dari berkas yang ingin kita hapus tersebut, dan setelah ditemukan, semua tempat yang dipakai berkas tadi kita lepaskan sehingga dapat digunakan oleh berkas lainnya. *Entry* dari direktori itu kemudian dihapus.

- **Menghapus sebagian isi berkas (*Truncate*).** *User* mungkin ingin menghapus isi dari sebuah berkas, tapi tetap ingin menjaga atribut-atributnya. *Truncating file* mengijinkan pendefinisian ulang panjang berkas menjadi nol tanpa mengubah atribut lainnya sehingga tempat yang digunakan oleh berkas dapat dilepaskan dan dipergunakan oleh berkas lain.

13.5. Membuka Berkas

Setelah berkas dibuat, berkas tersebut dapat digunakan sebagai M/K. Pertama-tama, berkas tersebut harus dibuka. Perintah `open()` memberikan nama berkas kepada sistem, lalu sistem mencari direktori untuk nama yang diberikan tadi. Perintah *open* mengembalikan sebuah *pointer* ke *entry* yang bersesuaian. *Pointer* inilah yang kemudian akan digunakan untuk operasi-operasi pada berkas.
Informasi-informasi yang berkaitan dengan membuka berkas adalah sebagai berikut:

1. ***File Pointer .*** Dalam sistem yang tidak memasukkan *file offset* sebagai bagian dari *system call read* dan *write*, sistem harus mengetahui lokasi terakhir dari *read* dan *write* sebagai *current-file-posistion pointer*. *Pointer* dari tiap proses pada *file* sifatnya unik.

2. ***File Open Count .*** Ketika sebuah berkas ditutup, sistem operasi harus bisa menggunakan lagi tabel *entry open-file* agar *entry* dalam tabel tersebut tidak habis. Karena suatu berkas dapat dibuka oleh berbagai proses, sistem harus mengunggu hingga seluruh proses yang menggunakan berkas tersebut selesai untuk bisa menghapus *entry* di tabel *open-file.*

3. ***Disc Location of the File .*** Kebanyakan operasi yang kita lakukan akan mengubah data di dalam berkas. Oleh karena itu, informasi yang diperlukan untuk menentukan lokasi berkas di dalam disk disimpan di memori untuk mecegah pembacaan berulang dari disk untuk setiap operasi.

4. ***Access Right .*** Tiap proses membuka berkas dalam *access mode* . Informasi ini disimpan pada tabel per-process sehingga sistem operasi dapat mengijinkan atau menolak permintaan M/K.

13.6. 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 mencari bentuk obyek biner sebuah program, yang tercetak biasanya tidak dapat dibaca, namun hal ini dapat dihindari apabila sistem operasi telah diberitahu akan adanya jenis berkas tersebut.

Cara yang paling umum untuk mengimplementasikan jenis bekas tersebut adalah dengan memasukkan jenis berkas tersebut ke dalam nama berkas. Nama berkas dibagi menjadi dua bagian. Bagian pertama adalah nama dari jenis berkas tersebut, dan yang kedua, atau biasa disebut *extension* adalah jenis dari berkas tersebut. Kedua nama ini biasanya dipisahkan dengan tanda '.', contoh: "berkas.txt".

Tabel 13.1. Jenis-jenis berkas

JENIS BERKAS	EXTENSION	FUNGSI
Executable	exe, com, bin, atau tidak ada	Siap menjalankan program bahasa mesin
Object	obj atau o	Dikompilasi, bahasa mesin, tidak terhubung (*link*)
Source code	c, cc, java, asm, pas	Kode-kode program dalam berbagai bahasa pemrograman
Batch	bat, sh	Memerintahkan ke *command intepreter*
Text	txt, doc	*Data text*, dokumen
Word processor	wp, tex, rtf, doc	Macam-macam format dari *text processor*
Library	lib, a, sol, dll	*Libraries* dan *routine* untuk *programmer*
Print/ view	jpg, pdf, ps	Berkas ASCII/*binary* dalam format untuk mencetak atau melihat
Archive	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

13.7. Struktur Berkas

Berkas-berkas tertentu harus mempunyai struktur yang dimengerti oleh sistem operasi. Contohnya, sistem operasi mungkin mensyaratkan bahwa sebuah berkas *executable* harus mempunyai struktur yang spesifik sehingga dapat ditentukan dimana berkas tersebut dapat di-*load* dari memori dan dimana lokasi dari instruksi pertama. Berkas dapat distruktur 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 apapun yang mereka mau dalam berkas, dan sistem operasi tidak membantu, namun tidak juga menghalangi.

Cara yang kedua adalah dengan *record sequence*. Dalam model ini semua berkas adalah sebuah urutan dari rekaman-rekaman yang telah ditentukan panjangnya, masing-masing dengan beberapa struktur internal. Artinya bahwa sebuah operasi *read* membalikkan sebuah rekaman dan operasi *write* menimpa atau menambahkan suatu rekaman.

Cara 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 diterapkan dalam rekaman tersebut. *Tree* ini di-*sort* dalam *field key* dan mengizinkan pencarian yang cepat untuk sebuah *key* tertentu.

13.8. Metode Akses Berkas

Berkas menyimpan informasi. Apabila sedang digunakan informasi ini harus diakses dan dibaca melalui memori komputer. Informasi dalam berkas dapat diakses dengan beberapa cara, yaitu:

1. **Akses sekuensial.** Akses ini merupakan yang paling sederhana dan paling umum digunakan. Informasi dalam berkas diproses secara berurutan. Sebagai contoh, editor dan kompilator biasanya mengakses berkas dengan cara ini.
2. **Akses langsung (*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.

13.9. Proteksi Berkas

Saat sebuah informasi disimpan di komputer, kita menginginkan agar informasi tersebut aman dari kerusakan fisik (ketahanan) dan akses yang tidak semestinya (proteksi).

Ketahanan biasanya disediakan dengan duplikasi dari berkas. Banyak komputer yang mempunyai program sistem yang secara otomatis menyalin berkas dari disk ke *tape* dalam interval tertentu (misalnya sekali dalam sehari, atau seminggu, atau sebulan) untuk menjaga *copy* -an berkas agar tidak rusak secara tidak disengaja. Sistem berkas dapat rusak karena masalah *hardware* (seperti *error* dalam membaca atau menulis), mati listrik, debu, suhu yang ekstrim, atau perusakan dengan sengaja. *Bug* dalam *software* sistem berkas juga dapat mengakibatkan isi dari dokumen hilang.

1. **Tipe-tipe akses.** Kebutuhan untuk mengamankan berkas berhubungan langsung dengan kemampuan untuk mengakses berkas. Kita bisa menyediakan proteksi secara menyeluruh dengan pelarangan akses. Kita juga dapat menyediakan akses bebas tanpa proteksi. Kedua pendekatan tersebut terlalu ekstrem untuk penggunaan umum, sehingga yang kita butuhkan adalah akses yang terkontrol.

   Mekanisme proteksi menyediakan akses yang terkontrol dengan membatasi tipe dari akses terhadap berkas yang dapat dibuat. Akses diizinkan atau tidak tergantung pada beberapa faktor, salah satunya adalah tipe dari akses yang diminta. Beberapa tipe operasi yang bisa dikontrol:
   - ***Read.*** membaca dari berkas.
   - ***Write.*** menulis atau menulis ulang berkas.
   - ***Execute.*** me-*load* berkas ke memori dan mengeksekusinya..
   - ***Append.*** menulis informasi baru di akhir berkas.
   - ***Delete.*** menghapus berkas dan mengosongkan spacenya untuk kemungkinan digunakan kembali.

- ***List.*** mendaftar nama dan atribut berkas.

2. **Kontrol akses.** Pendekatan paling umum dalam masalah proteksi adalah untuk membuat akses tergantung pada identitas pengguna. Pengguna yang bervariasi mungkin membutuhkan tipe akses yang berbeda atas suatu berkas atau direktori. Skema yang paling umum untuk mengimplementasikannya adalah dengan mengasosiasikan setiap berkas dan direktori pada sebuah list kontrol akses, yang menspesifikasikan *user name* dan tipe akses yang diperbolehkan untuk setiap *user*. Saat seorang pengguna meminta untuk mengakses suatu berkas, sistem operasi akan mengecek daftar akses yang berhubungan dengan berkas tersebut. Apabila pengguna tersebut ada di dalam daftar, maka akses tersebut diizinkan. Jika tidak, terjadi pelanggaran proteksi, dan pengguna tidak akan diizinkan untuk mengakses berkas tersebut.

   Masalah utama dengan pendekatan ini adalah panjang dari daftar yang harus dibuat. Tapi dapat dipecahkan dengan cara menggunakan daftar dalam versi yang di- *condense*. Untuk itu, pengguna dapat diklasifikasikan ke dalam tiga kelas:
   - ***Owner.*** pengguna yang membuat berkas tersebut.
   - ***Group.*** sekelompok pengguna yang berbagi berkas dan memiliki akses yang sama.
   - ***Universe.*** semua pengguna yang lain.

3. **Pendekatan lain.** Pendekatan lain dalam masalah proteksi adalah dengan memberikan kata kunci untuk setiap berkas.

13.10. 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, yaitu:
- Kebutuhan untuk menyimpan data dalam jumlah yang besar.
- Kebutuhan agar data tidak mudah hilang (*non-volatile*)
- Informasi harus tidak bergantung pada proses.

Pada sistem berkas ini, diatur segala hal yang berkaitan dengan sebuah berkas mulai dari atribut, tipe, operasi, struktur, sampai metode akses berkas.

Rujukan

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

Bab 14. Struktur Direktori

14.1. Pendahuluan

Direktori atau *folder* merupakan suatu entitas dalam sebuah sistem berkas yang mengandung berkas atau direktori lain. Pada hakikatnya, berkas atau direktori lain tersebut terdapat di dalam disk, sedangkan direktori hanya menyediakan *link* atau mengarahkan pada berkas yang ada. Direktori digunakan sebagai sarana untuk pengorganisasian berkas pada suatu sistem komputer. Dengan adanya direktori, setiap berkas dapat dikelompokkan. Sebuah direktori dapat berisi berkas maupun direktori lain, sehingga direktori dapat juga disebut sebagai berkas istimewa. Dalam pengorganisasian sebuah berkas, sistem operasi dapat memartisi disk menjadi beberapa direktori atau menjadikan dua disk menjadi sebuah direktori.

14.2. Atribut dan Struktur Direktori

Atribut Direktori

Atribut atau informasi dalam suatu direktori dapat berbeda-beda tergantung pada sistem operasi yang digunakan. Sebagai sebuah berkas, direktori bisa memiliki beberapa atribut, antara lain:

a. **Nama.** Merupakan nama dari direktori itu sendiri.
b. **Alamat.** Merupakan alamat dari direktori tersebut.
c. **Tanggal.** Berisi keterangan mengenai tanggal pembuatan direktori tersebut.
d. **Ukuran.** Merupakan besarnya ukuran suatu direktori, biasanya dalam satuan byte, kilobyte, megabyte, gigabyte. Batas maksimum dari suatu direktori bergantung pada sistem berkas yang digunakan.
e. **Proteksi.** Berguna untuk perlindungan. Hal ini mencakup siapa saja yang berhak mengakses, penyembunyian file, *read-only*, dan lain-lain. Dalam Unix, proteksi berguna untuk mengubah atribut berkas dengan menggunakan perintah "*chmod*".

Atribut-atribut pada direktori dirancang sewaktu pembuatan sistem operasi, sehingga atribut yang ada tergantung pada pembuat sistem operasi tersebut. Atribut-atribut di atas merupakan atribut yang umum dan sering digunakan.

Struktur Direktori

Mempelajari struktur direktori, memberikan kita pemahaman bagaimana menyusun sebuah direktori dalam suatu sistem berkas. Ada beberapa tujuan yang ingin dicapai dalam menyusun sebuah direktori dalam suatu sistem. Namun, terdapat beberapa kendala, seperti, penamaan berkas, pengelompokan berkas dan berbagi berkas (*file sharing*). Ada tiga struktur direktori yang dikenal, antara lain:

- Struktur Direktori Bertingkat, dimana direktori ini dibagi menjadi direktori satu tingkat (*Single Level Directory*) dan direktori dua tingkat (*Two Level Directory*).

- Direktori berstruktur pohon (*Tree-Structured Directory*).

- Direktori berstruktur graf, dimana direktori ini dibagi menjadi struktur graf asiklik (*Acyclic-structured Directory*) dan struktur graf sederhana (*General-graph Directory*).

Bentuk-bentuk direktori tersebut mempunyai nilai historis tersendiri. Misalnya direktori bertingkat satu, di masa-masa awal perkembangan komputer terdahulu, kapasitas dari *secondary storage* masih terbatas. Besarnya hanya berkisar beberapa megabyte saja. Oleh karena itu, struktur direktori bertingkat satu sudah mencukupi untuk kebutuhan penggunaan komputer sehari-hari. Namun, seiring berkembangnya zaman direktori satu tingkat tersebut dirasakan kurang mencukupi dikarenakan berbagai keterbatasan yang dimilikinya. Setelah itu, muncul direktori dua tingkat dan seterusnya. Hal-hal itulah yang akan kita lihat dalam beberapa pembahasan selanjutnya mengenai struktur direktori.

14.3. Operasi Direktori

Silberschatz, Galvin dan Gagne mengkategorikan operasi-operasi terhadap direktori sebagai berikut:

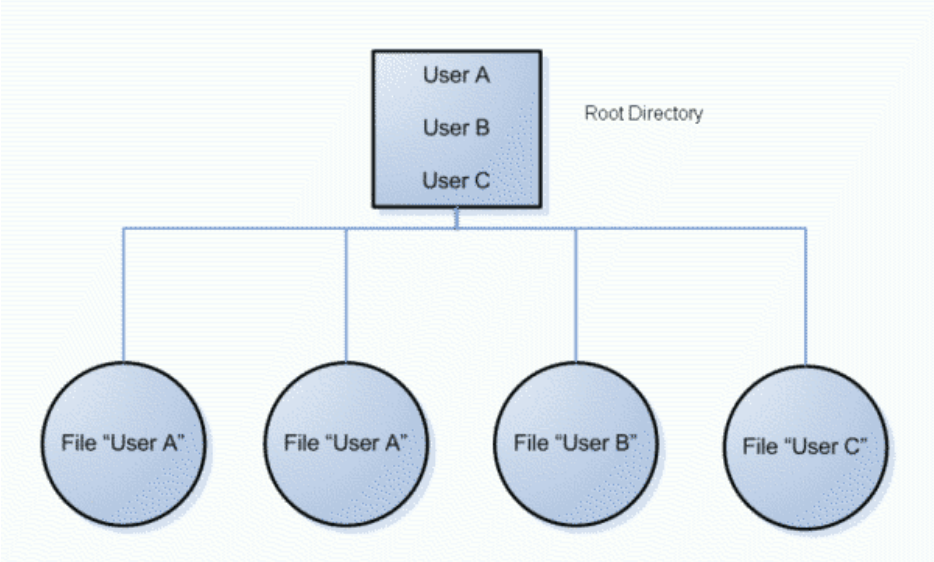
a. **Mencari berkas.** Bila pengguna atau sebuah aplikasi ingin melakukan suatu operasi terhadap suatu berkas, tentu berkas tersebut harus dibuka terlebih dahulu. Untuk itu, sebuah direktori harus mencari entri yang bersesuaian dengan file tersebut dengan menelusuri struktur dari direktori yang bersangkutan.
b. **Membuat berkas.** Saat sebuah berkas baru dibuat, maka sebuah entri akan ditambahkan ke direktori.
c. **Menghapus berkas.** Ketika suatu berkas tidak dibutuhkan lagi, maka berkas tersebut bisa dihapus dari direktori.
d. **Menampilkan isi direktori.** Menampilkan seluruh atau sebagian daftar berkas-berkas yang ada di direktori dan atribut dari berkas-berkas dalam direktori tersebut (misalnya, *information access control*, *type* dan *usage information*).
e. **Mengubah nama berkas.** Nama suatu berkas merepresentasikan isi berkas terhadap pengguna. Oleh karena itu, nama berkas harus bisa diubah ketika isi dan kegunaannya sudah tidak sesuai lagi. Mengubah nama suatu berkas memungkinkan berpindahnya posisi berkas di dalam struktur direktori.
f. **Akses sistem berkas.** Pengguna bisa mengakses setiap direktori dan setiap berkas yang berada dalam struktur direktori.
g. **Update direktori.** Karena sebagian atribut dari berkas disimpan dalam direktori, maka perubahan yang terjadi terhadap suatu berkas akan berpengaruh terhadap atribut dari berkas yang bersangkutan di direktori tersebut.

14.4. Direktori Bertingkat

Direktori Satu Tingkat (*Single—Level Directory*)

Direktori satu tingkat merupakan suatu struktur direktori yang paling sederhana karena semua berkas yang ada disimpan dalam direktori yang sama. Direktori satu tingkat ini memiliki keterbatasan, yaitu bila berkas bertambah banyak atau bila sistem memiliki lebih dari satu pengguna. Jumlah berkas yang terlalu banyak dalam sebuah direktori dapat menyebabkan ketidaknyamanan. Hal ini mungkin saja terjadi karena pengguna hanya dapat menyimpan berbagai berkas (misal: *games*, *video*, *email*) dalam sebuah direktori saja.

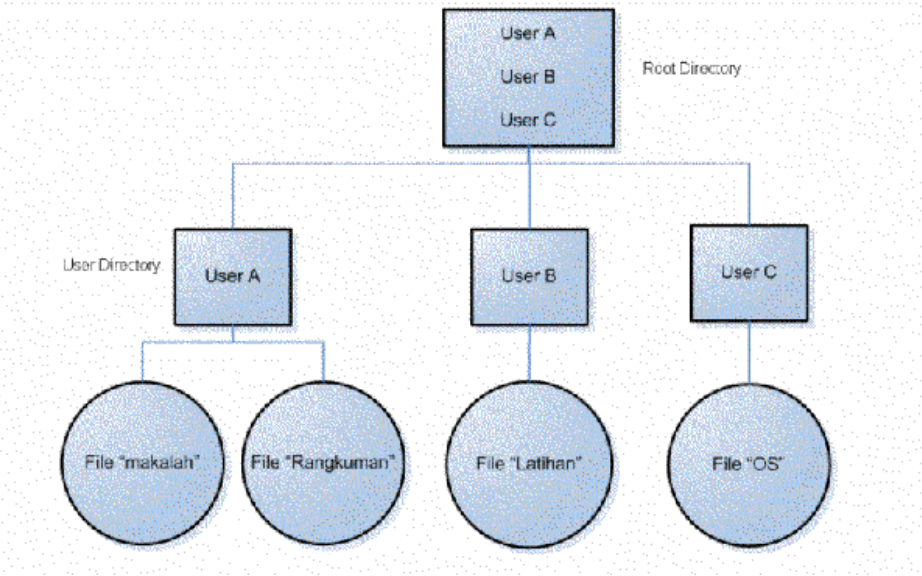
Gambar 14.1. Direktori Satu Tingkat



Direktori Dua Tingkat (*Two—Level Directory*)

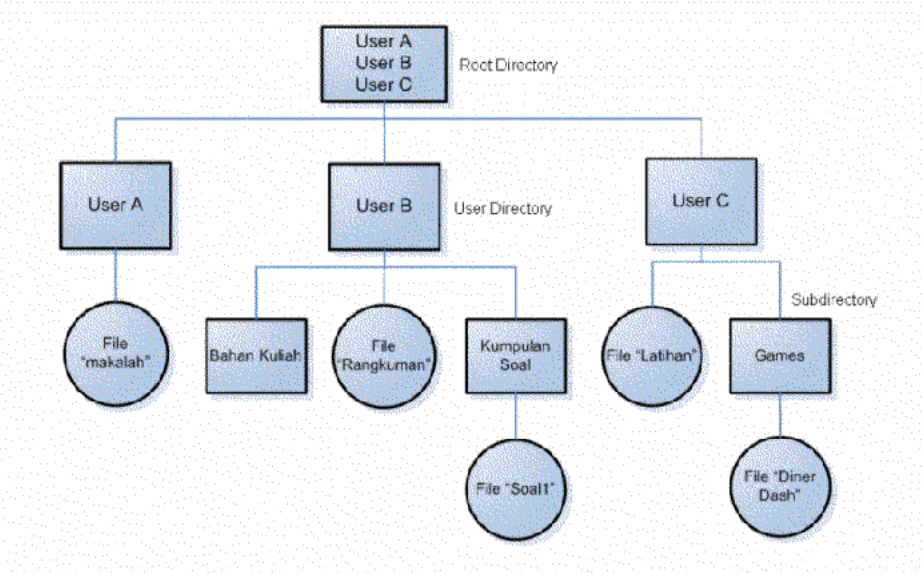
Pada direktori satu tingkat, sering terjadi kesulitan dalam menentukan nama file dari dua pengguna yang berbeda. Penyelesaian umumnya adalah dengan membuat direktori terpisah untuk tiap pengguna yang dikenal dengan *User File Directory* (UFD). Di struktur direktori dua tingkat, setiap pengguna mempunyai UFD masing-masing. Ketika pengguna melakukan *login*, maka *Master File Directory* (MFD) dipanggil. Indeks yang dimiliki oleh MFD didasarkan pada *username* atau *account number*, dan setiap entri menunjuk pada UFD pengguna tersebut. Sehingga, pengguna bisa mempunyai nama berkas yang sama dengan berkas lain.

Gambar 14.2. Direktori Dua Tingkat



14.5. Direktori Berstruktur Pohon

Gambar 14.3. *Tree-Structured Directories*

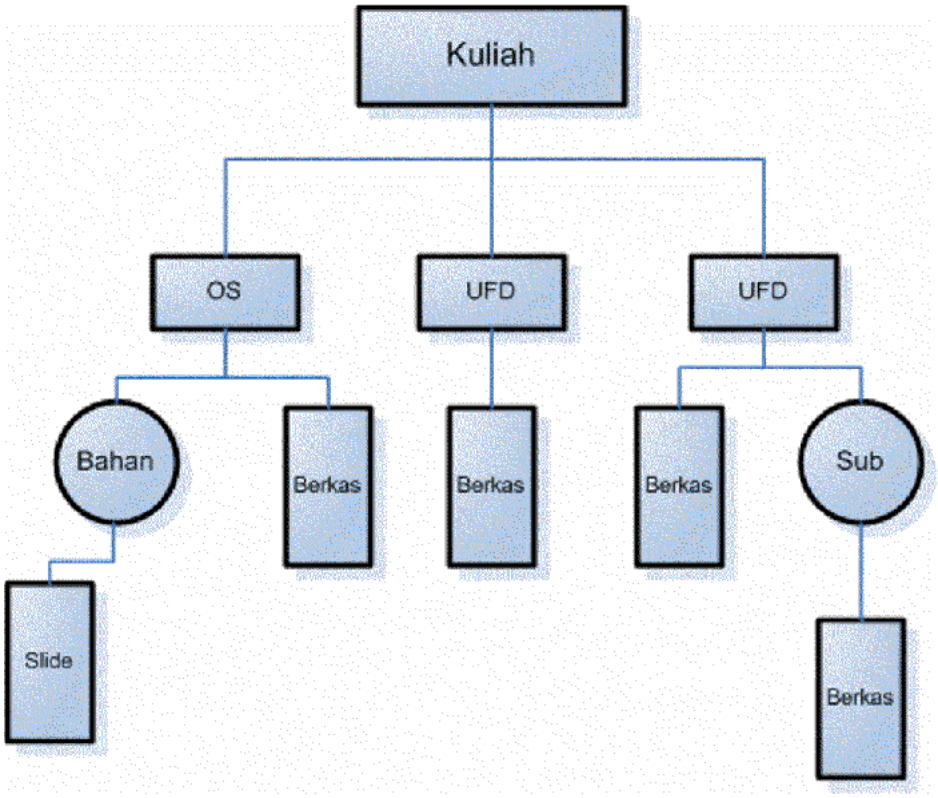


Pada *Tree-Structured Directories*, setiap pengguna dapat membuat sub-direktori sendiri dan mengorganisasikan berkas-berkas yang dimiliki. Dalam penggunaan yang normal, setiap pengguna

memiliki direktori saat ini (*current directory*). *Current directory* ini terdiri dari berkas-berkas yang baru-baru ini digunakan oleh pengguna. Nama lintasan (*path name*) bisa digolongkan menjadi dua jenis, yaitu:

1. **Lintasan mutlak (*absolute path*).** Merupakan lintasan yang dimulai dari *root directory*.
2. **Lintasan relatif (*relative path*).** Merupakan lintasan yang dimulai dari direktori saat ini (*current directory*).

Gambar 14.4. *Path*



Misalkan kita sedang berada pada direktori bahan, maka penulisan lintasan dari berkas *slide*:

1. ***Absolute path*.** Yaitu "/Kuliah/OS/bahan/slide".
2. ***relative path*.** Yaitu "../bahan/slide".

Dengan sistem *Tree-Structured Directories*, para pengguna dapat mengakses dan menambahkan berkas pengguna lain kedalam direktori mereka. Sebagai contoh, pengguna B dapat mengakses berkas-berkas pengguna A melalui spesifikasi nama lintasannya. Dengan alternatif lain, pengguna B dapat mengubah *current directory*-nya menjadi direktori yang dimiliki oleh pengguna A dan mengakses berkas-berkas tersebut melalui *file names*-nya. Sebuah lintasan ke sebuah berkas didalam *Tree-Structured Directories* bisa lebih panjang daripada lintasan di direktori dua tingkat.

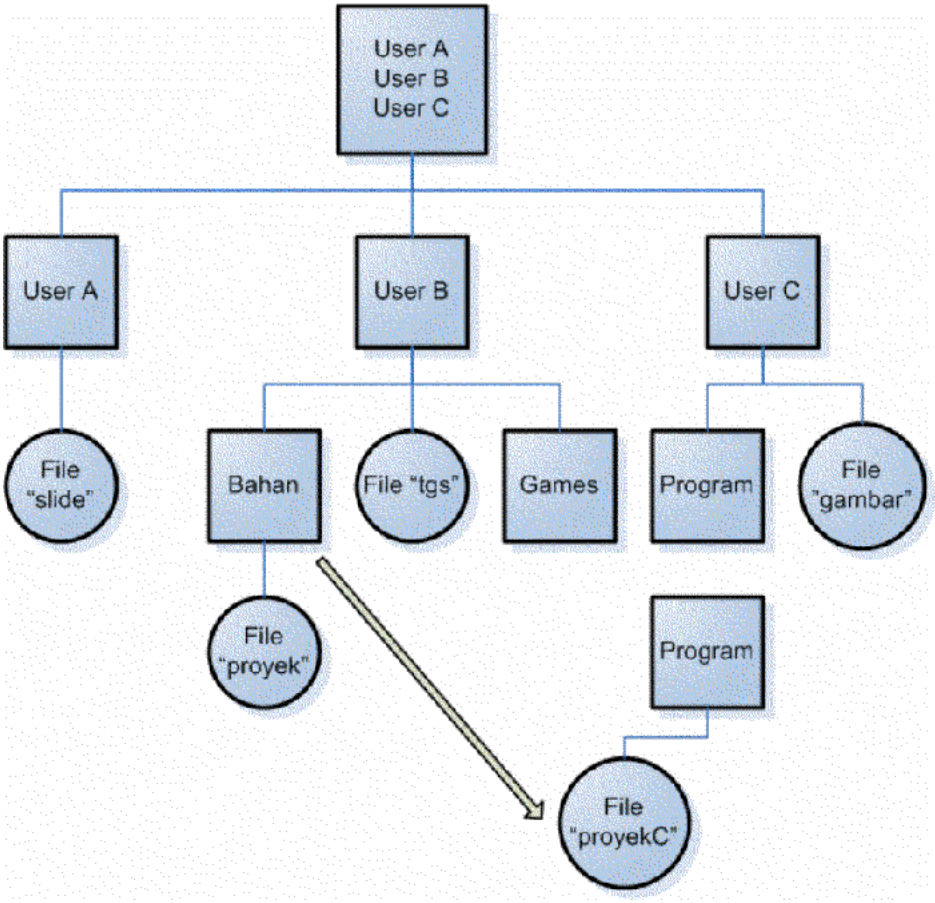
14.6. Direktori Berstruktur Graf

Direktori dengan struktur pohon (*Tree-Structured Directories*) tidak memperbolehkan adanya pembagian berkas/direktori. Sedangkan 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-structured Directory* bisa mengatasi permasalahan pada direktori dengan struktur pohon (*Tree-Structured Directories*).

Permasalahan yang timbul dalam penggunaan struktur graf asiklik adalah meyakinkan apakah tidak ada siklus. Bila kita mulai dengan struktur direktori dua tingkat dan memperbolehkan pengguna untuk

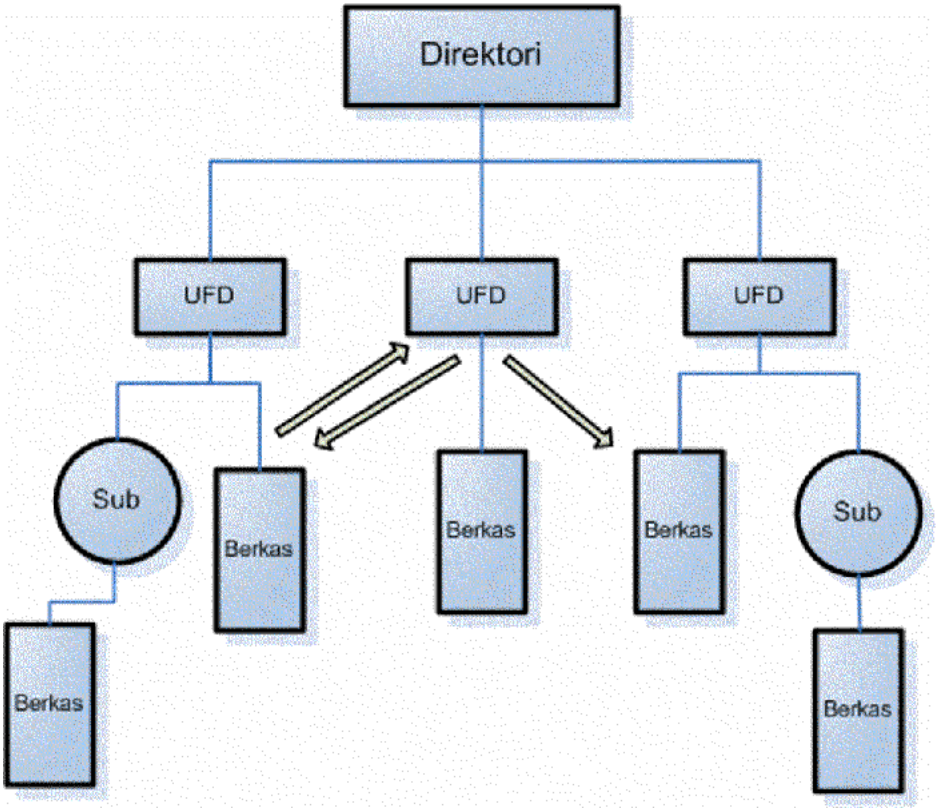
membuat subdirektori, maka kita akan mendapatkan struktur direktori pohon. Mempertahankan sifat pohon bukan suatu hal yang sulit, tetapi bila kita menambahkan sambungan pada direktori dengan struktur pohon, maka sifat pohon akan hilang dan menghasilkan struktur graf sederhana (*General-graph directory*).

Gambar 14.5. ***Acyclic-Structured Directory***



Bila siklus diperbolehkan dalam direktori, tentunya kita tidak ingin mencari sebuah berkas dua kali. Algoritma yang tidak baik akan menghasilkan *infinite loop* dan tidak akan pernah berakhir. Oleh karena itu, diperlukan skema pengumpulan sampah (*garbage-collection scheme*). Skema ini berhubungan dengan pemeriksaan seluruh sistem berkas dengan menandai tiap berkas yang dapat diakses. Kemudian mengumpulkan apapun yang tidak ditandai pada tempat yang kosong. Hal ini tentunya dapat menghabiskan banyak waktu.

Gambar 14.6. *General-graph Directory*



Pada direktori dengan struktur pohon, setiap pengguna dapat membuat direktori sendiri sehingga dalam UFD akan terdapat direktori yang dibuat oleh pengguna dan di dalam direktori tersebut dapat dibuat direktori lain (sub-direktori), begitu seterusnya. Hal ini tentu akan memudahkan pengguna dalam pengelompokan dan pengorganisasian berkas. Masalah yang muncul adalah ketika pengguna ingin menggunakan suatu berkas secara bersama-sama. Hal ini timbul dikarenakan sistem tidak mengizinkan seorang pengguna mengakses direktori pengguna lain.

Pada *general-graph directory*, sebuah direktori me-*link* pada direktori yang me-*link* nya. Dengan kata lain, jika direktori A berisi/ me-*link* direktori B maka ketika direktori B dibuka akan terdapat direktori A (ada siklus).

14.7. *Mounting*

Mounting adalah proses mengaitkan 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 di-*mount* akan diberikan *mount point* atau sebuah direktori dalam pohon direktori sistem yang sedang anda akses. *Mount point* adalah direktori tempat dimana akan meletakkan sistem berkas tesebut. Kalau kita ingin me-*mount* sistem berkas berupa direktori, maka *mount point*-nya harus berupa direktori. Sebaliknya, jika yang hendak kita *mount* adalah file, maka *mount point*-nya juga harus berupa file.

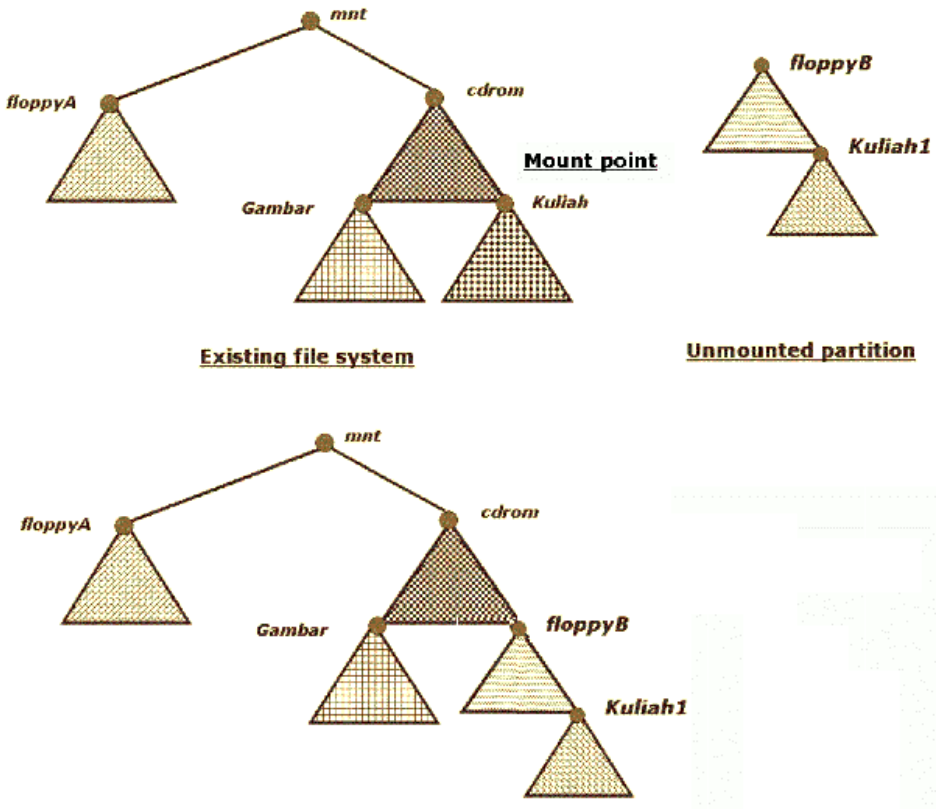
Daftar sistem berkas yang di-*mount* dapat dilihat kapan saja dengan menggunakan perintah *mount*. Karena izinnya hanya diatur *read-only* di berkas *fstab*, maka tidak perlu khawatir pengguna lain akan mencoba mengubah atau menulis *mount point* yang baru. Seperti biasa, saat ingin mengubah-ubah berkas 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-disc* dan mem-*backup* berkas *fstab* tersebut sebelum membukanya di editor teks untuk diubah-ubah.

Semua ruang kosong yang tersedia di disk diatur dalam sebuah pohon direktori tunggal. Dasar sistem ini adalah *root directory* 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-*umount*. Sistem berkas yang lain di-*mount* seperlunya, contohnya yang berisi *hard drive* berbeda dengan *floppy disc* atau *CD-ROM*.

Sebenarnya setiap akan memproses suatu sistem berkas (*read and write*) kita harus me-*mount* sistem berkas itu terlebih dahulu. 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 didalamnya. Untuk me-*mount* suatu sistem berkas, sistem operasi memerlukan data tentang *device* yang membawakan sistem berkas tersebut dan *mount point* tempat sistem berkas itu hendak diletakkan.

Gambar 14.7. *Existing File System*



14.8. Berbagi Berkas

Saat sebuah sistem memutuskan untuk menyediakan fasilitas berbagi berkas, maka tantangan yang muncul adalah memperluas *file-sharing* agar dapat diakses oleh berbagai sistem berkas. Hal lain yang menjadi perhatian adalah konflik yang mungkin muncul akibat berbagi berkas, misalnya beberapa pengguna melakukan operasi penulisan terhadap suatu berkas secara bersama-sama.

Multiple User

Ada tiga isu penting saat suatu sistem mengakomodasi banyak pengguna (*multiple users*), yaitu berbagi berkas, penamaan berkas, dan proteksi berkas. Dalam pengimplementasian berbagi berkas dan proteksi berkas di *multiple user system*, suatu sistem perlu untuk memberikan tambahan pada atribut

dari suatu berkas atau direktori. Pendekatan yang umum dilakukan adalah dengan konsep *owner* dan *group*. Dalam bahasa Indonesia, kata *owner* berarti pemilik. Istilah pemilik dalam suatu sistem yang menerapkan berbagi berkas dapat diartikan sebagai seorang pengguna yang mempunyai hak penuh atas suatu berkas atau subdirektori. *Owner* tersebut dapat melakukan apa saja terhadap berkas miliknya, termasuk memberikan hak akses tertentu kepada pengguna lain terhadap berkas tersebut.

Konsep dari *owner* ini diimplemetasikan oleh beberapa sistem dengan memanfaatkan daftar dari nama pemakai dan diasosiasikan dengan *user identifiers* atau *user IDs*. Tentu saja ID ini bersifat unik, tidak akan ada dua pengguna yang memiliki ID yang sama. Selanjutnya untuk setiap proses dan *thread* yang dijalankan oleh seorang pengguna, maka proses dan thread tersebut akan dikaitkan dengan *user ID* tadi.

Sekumpulan pengguna dapat membentuk suatu *group* yang mempunyai *group identifier* dan akan dikaitkan dengan setiap proses dan *thread* yang dijalankan oleh *group* tersebut.

Saat seorang pengguna melakukan operasi pada suatu berkas, maka *user ID* dari pengguna tersebut akan dicocokkan dengan atribut dari pemilik berkas tersebut. Proses tersebut dilakukan untuk mengetahui hak apa saja yang diberikan oleh pemilik berkas kepada pengguna lain. Hal itu juga berlaku pada *group*.

Remote File System

Seiring berkembangnya jaringan dan teknologi berkas, mekanisme berbagi berkas juga mengalami perubahan. Awalnya, cara yang digunakan dalam *file-sharing* adalah dengan aplikasi seperti *File Transfer Protocol* (FTP). Selanjutnya, berkembang apa yang disebut dengan *Distributed File Systems*, disingkat DFS. Dengan DFS, sebuah *remote-directories* dapat diakses dari *local-machine*. Cara lainnya adalah melalui *World Wide Web* (*www*), merupakan pengembangan dari metode FTP.

Mekanisme *file-sharing* memungkinkan seorang pengguna dapat mengakses sebuah sistem berkas yang ada di komputer lain yang terhubung ke jaringan atau biasa disebut *remote machine*. Ada dua kemungkinan saat seorang pengguna terhubung ke *remote machine*. Pertama, ia harus melakukan proses otentifikasi atau proses identifikasi bahwa ia telah terdaftar sebagai seorang pengguna yang mempunyai hak akses tertentu terhadap *remote machine* tersebut. Kedua, ia cukup dikenali sebagai *anonymous* pengguna yang bisa jadi mempunyai hak akses tidak seluas dibandingkan dengan pengguna yang telah terotentifikasi.

Absolute path. Yaitu "/Kuliah/OS/bahan/slide".

Client-Server Model. *Remote File System* mengijinkan suatu komputer untuk me-*mounting* beberapa sistem berkas dari satu atau lebih *remote machine*. Dalam kasus ini, komputer yang menyediakan berkas-berkas yang diakses oleh komputer-komputer lain disebut dengan *server* dan komputer yang mengakses berkas-berkas yang di-*share* disebut dengan *client*. Yang menjadi isu dalam model ini adalah masalah keamanan, pengaksesan suatu sistem oleh seseorang yang tidak mempunyai hak, atau disebut juga *unauthorized user*.

Failure Modes. Suatu *local system* dapat mengalami *failure* atau *crash* yang menyebabkan sistem tersebut tidak dapat berfungsi sebagaimana mestinya karena berbagai hal. Teknologi *Redundant Arrays of Inexpensive Disk* (RAID) cukup membantu hal ini. Namun demikian, penjelasan lebih detil tentang RAID akan dibahas dalam topik-topik lainnya, terutama yang berkaitan dengan *storage*.

Remote File System bukan tanpa gangguan. Kompleksitas dari sistem jaringan dan juga adanya interaksi antara *remote machine* memberi peluang lebih besar akan terjadinya kegagalan dalam sistem tersebut.

14.9. Rangkuman

Direktori atau folder merupakan suatu entitas dalam sebuah sistem berkas yang mengandung berkas atau direktori lain. Mempelajari struktur direktori memberikan kita pemahaman bagaimana menyusun

sebuah direktori dalam suatu sistem berkas. Ada tiga struktur direktori yang diketahui, yaitu direktori bertingkat, direktori berstruktur pohon dan direktori berstruktur graf.

Mounting adalah proses mengaitkan sebuah sistem berkas yang baru ditemukan pada sebuah piranti ke struktur direktori utama yang sedang dipakai. Piranti-piranti yang akan di-*mount* dapat berbentuk *CD-ROM*, disket atau sebuah *zip-drive*. Tiap-tiap sistem berkas yang di-*mount* akan diberikan *mount point* atau sebuah direktori dalam pohon direktori sistem yang sedang diakses. *Mount point* adalah direktori tempat dimana akan meletakkan sistem berkas tersebut. Kalau kita ingin me-*mount* sistem berkas berupa direktori, maka *mount point*-nya harus berbentuk direktori. Sebaliknya, jika yang ingin di-*mount* adalah file, maka *mount point*-nya harus berbentuk file.

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.

[Nana Langstedt] Nana Langstedt. 2005. *How to mount partitions and file-systems in Linux – http:// www.tuxfiles.org/linuxhelp/mounting.html* . Diakses 27 April 2007.

[Mayang Sarup] Mayang Sarup. 2001. *The Linux Filesystem explained – http://www.freeos.com/articles/3102/* . Diakses 27 April 2007.

Bab 15. FHS

15.1. Pendahuluan

FHS (*Filesystem Hierarchy Standard*) adalah sebuah aturan standar penempatan lokasi berkas dan direktori yang ada pada sistem operasi. Dengan adanya standar ini maka pengguna dan perangkat lunak dapat mengetahui dimana letak suatu berkas atau direktori yang tersimpan di suatu komputer.

Pada masa-masa awal pengembangan Linux, masing-masing distribusi Linux menggunakan skema buatan mereka sendiri untuk menentukan lokasi suatu berkas pada hirarki direktori. Sayangnya, hal ini menyebabkan banyak masalah. Struktur hirarki berkas dan direktori pada sistem operasi Linux menjadi tidak teratur. Tidak ada suatu acuan baku untuk semua distribusi Linux yang ada saat itu. Dilatarbelakangi oleh kekacauan ini, pada tahun 1993 dibuatlah sebuah berkas yang berisi tentang aturan standar penempatan lokasi hirarkis berkas dan direktori yang disebut FSSTND (*Filesystem Standard*). FSSTND telah membantu menstandarisasikan rancangan sistem berkas pada semua distribusi Linux. Seiring dengan berjalannya waktu, FSSTND ini semakin berkembang. Jika pada awalnya FSSTND hanya mengatur sistem berkas dari Linux. Pada tahapan selanjutnya FSSTND juga mulai mengatur semua sistem operasi berbasis UNIX. Sejalan dengan perluasan objek dari FSSTND ini maka namanya diganti menjadi FHS, yang merupakan kependekan dari *Filesystem Hierarchy Standard*.

FHS berisi sekumpulan syarat dan petunjuk penempatan berkas dan direktori pada sistem operasi berbasis UNIX. Petunjuk ini dimaksudkan untuk mendukung interoperabilitas dari suatu aplikasi, perangkat administrasi sistem, perangkat pengembangan dan *script* sehingga didapatkan keseragaman pada semua sistem yang berbasis UNIX.

FHS dibuat dengan cara:
- Menentukan petunjuk-petunjuk dasar untuk setiap area pada sistem berkas
- Menentukan berkas dan direktori minimum yang dibutuhkan
- Menandai setiap pengecualian (*exception*)
- Menandai setiap kasus spesifik yang pernah mengalami konflik

Berkas FHS tersebut akan bermanfaat bagi penyedia perangkat lunak yang ingin membuat perangkat lunak untuk sistem operasi yang menggunakan FHS. Sehingga penyedia perangkat lunak tersebut dapat membuat perangkat lunak yang dapat dijalankan pada sistem operasi tersebut. Berkas FHS juga dapat dipakai oleh pembuat sistem operasi. Contoh paling umum dari sistem operasi yang menggunakan model FHS ini adalah Linux, walaupun ada beberapa hal spesifik yang hanya terdapat di Linux dan tidak ada pada FHS. Selain itu, berkas FHS juga bermanfaat bagi pengguna yang ingin lebih mengerti sistem berkas dari sistem operasi yang mereka gunakan.

15.2. Sistem Berkas

FHS mengasumsikan bahwa sistem operasi yang menggunakan standarisasi FHS telah mengimplementasikan sistem berkas yang mendukung fitur-fitur keamanan dasar yang ada pada sebagian besar sistem berkas UNIX.

Untuk melakukan pengelompokan atau kategorisasi terhadap suatu berkas, FHS menggunakan dua parameter *independent* yang membedakan berkas satu dengan berkas yang lain. Parameter tersebut adalah:
- *shareable - unshareable*
- statis - variabel.

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". *Host* yang tidak memiliki direktori /usr bisa saja menggunakan direktori ini dengan cara me-*mount* direktori /usr yang dimiliki oleh suatu *host*. Bukan suatu hal yang mustahil jika dari sepuluh komputer yang terhubung, hanya satu komputer yang memiliki direktori "/

usr" karena direktori tersebut dapat diakses oleh semua komputer. Berkas *unshareable* adalah berkas-berkas yang tidak dapat di- *share*.

Berkas statik adalah berkas yang tidak dapat diubah tanpa intervensi administrator sistem. Contohnya adalah berkas binari (*binary file*), pustaka (*libraries*) dan berkas dokumentasi. Berkas variabel adalah berkas yang tidak statik, yaitu berkas yang dapat diubah oleh siapapun, baik administrator maupun pengguna. Berkas statik dan variabel sebaiknya diletakkan di direktori yang berbeda, karena berkas statik, tidak seperti berkas variabel, dapat ditaruh di media yang *read-only* dan tidak perlu di-*backup* secara terjadwal sebagaimana berkas variabel. Dengan demikian, tiap-tiap berkas dapat dikategorikan sesuai dengan karakteristiknya masing-masing.

15.3. Sistem Berkas ROOT

Sistem berkas *root* berisi hal-hal vital bagi kelangsungan sistem. Tanpa sistem berkas *root* maka tak ada sistem yang bisa berjalan. Secara kuantitas, peranan dari sistem berkas *root* ini hanya sedikit dibandingkan dengan peranan dari sistem berkas yang lain. Oleh karena itu, sistem berkas *root* ini cenderung jarang dipakai dibandingkan dengan sistem berkas lainnya. Walaupun sedikit peranan yang dimiliki olehnya, namun sistem berkas *root* sangat penting bagi sistem. Isi dari sistem berkas *root* harus bisa memadai keperluan untuk melakukan proses *boot, restore, repair* dan *recovery* sistem. Untuk melakukan operasi *boot* pada sistem, perlu dilakukan hal-hal untuk melakukan proses *mount* pada sistem berkas lain. Hal ini meliputi konfigurasi data, informasi *boot loader* dan keperluan-keperluan lain yang mengatur *start-up* data. Semua hal tersebut harus berada pada direktori *root*. Untuk melakukan *recovery* maupun *repair* dari sistem, maka hal-hal yang dibutuhkan untuk mendiagnosa dan memulihkan sistem yang rusak harus diletakkan dalam sistem berkas *root*. Kemudian, untuk me-*restore* suatu sistem maka hal-hal yang dibutuhkan untuk mem-*backup* sistem seperti *floppy, tape, disk* dll harus diletakkan di dalam direktori *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 seminimal mungkin. Lagipula, lokasi-lokasi lain dalam FHS menyediakan fleksibilitas yang lebih dari cukup untuk *package* aplikasi manapun.

Syarat

Terdapat beberapa direktori yang menjadi syarat atau harus ada pada sistem berkas *root*. Setiap direktori akan dibahas dalam subbagian di bawah. /usr dan /var akan dibahas lebih mendetail karena direktori tersebut sangat kompleks. Berikut daftar subdirektori yang harus ada pada direktori *root*:

Tabel 15.1. Direktori atau *link* yang harus ada pada /root

Direktori	Keterangan
bin	Perintah-perintah dasar pada sistem UNIX
boot	berkas statik dari *boot loader*
dev	berkas piranti
etc	sistem konfigurasi untuk *host*
lib	modul pustaka dan kernel esensial yang di-*share*
media	*mount-point* untuk *removeable* media
mnt	*mount-point* untuk melakukan operasi *mount* pada sistem berkas secara temporer
opt	paket tambahan dari suatu perangkat lunak
srv	data untuk layanan yang diberikan sistem
tmp	data sementara
usr	hirarki sekunder

Direktori	Keterangan
var	data variabel

Pilihan spesifik

Tabel 15.2. Direktori atau *link* yang harus diletakkan pada direktori /root, jika memang subsistemnya ter-*install*.

Direktori	Keterangan
home	direktori *home* dari pengguna
lib<qual>	pustaka esensial alternatif
root	direktori *home* untuk pengguna *root*

/bin : perintah biner esensial bagi pengguna

Direktori ini berisi perintah-perintah dasar esensial yang dapat dipakai baik oleh administrator sistem maupun oleh pengguna. Namun perintah-perintah biner ini hanya bisa dijalankan ketika tidak ada sistem berkas lain yang sedang di-*mount*. Direktori ini juga dapat berisi perintah yang digunakan secara tidak langsung oleh *script*. Pada direktori /bin tidak diperbolehkan adanya subdirektori.

Berikut beberapa perintah umum pada UNIX yang disimpan dalam direktori /bin:

Tabel 15.3. Perintah-perintah dan atau *link* simbolik yang harus ada pada /bin

Perintah	Keterangan
cat	perintah untuk menggabungkan berkas ke keluaran standar
chmod	perintah untuk mengubah *mode* akses dari suatu berkas
cp	perintah untuk menggandakan berkas atau direktori
date	perintah untuk menampilkan waktu dan tanggal pada sistem
echo	perintah untuk menampilkan satu baris teks
kill	perintah untuk mematikan sebuah proses
ls	perintah untuk menampilkan isi direktori
mkdir	perintah untuk membuat direktori
rm	perintah untuk menghapus berkas atau direktori
.....	dan lain-lain

Selain perintah-perintah dasar tersebut, direktori /bin juga dapat berisi perintah-perintah opsional yang memang harus diletakkan pada direktori ini, jika memang subsistemnya ter-*install*. Beberapa contohnya adalah:

- csh : *shell* khusus untuk bahasa pemrograman C
- tar : perintah untuk melakukan pengarsipan / kompresi berkas
- netstat : perintah untuk menampilkan statistik jaringan
- ping : perintah untuk menguji ada tidaknya jaringan

/boot : berkas statik untuk me- *load* proses *boot*

Direktori ini berisi segala sesuatu yang dibutuhkan untuk menjalankan *boot*. Salah satu contohnya adalah kernel sistem operasi yang nantinya akan di- *load* untuk mengaktifkan sistem. /boot menyimpan

data yang digunakan sebelum kernel mulai menjalankan program pada mode pengguna. Hal ini dapat meliputi *master boot sector* dan *map file sector*.

/dev : Berkas piranti

Direktori ini berisi berkas-berkas dari suatu piranti komputer. Jika berkas dari suatu piranti harus dibuat secara manual, maka direktori ini harus berisi perintah MAKEDEV. Perintah ini berfungsi membuat berkas piranti yang dibutuhkan. Direktori ini juga dapat berisi perintah MAKEDEV.local untuk membuat berkas piranti lokal.

/etc : Konfigurasi sistem yang spesifik untuk suatu *host*

Direktori /etc mernyimpan berkas-berkas konfigurasi bagi satu *host*. Berkas konfigurasi adalah berkas lokal yang digunakan untuk mengatur operasi dari sebuah program pada suatu *host*. Berkas ini harus statik dan bukan merupakan instruksi biner yang bisa dieksekusi.

Berikut adalah syarat-syarat subdirektori yang harus ada pada direktori /etc:

Tabel 15.4. Direktori atau link simbolik yang harus ada pada pada /etc

Direktori	Keterangan
opt	konfigurasi untuk direktori /opt. berisi berkas konfigurasi untuk paket tambahan dari suatu perangkat lunak
X11	konfigurasi untuk X Window System (opsional)
sgml	konfigurasi untuk sgml (opsional)
xml	konfigurasi untuk xml (opsional)

berikut adalah contoh beberapa berkas yang harus diletakkan pada /etc, jika memang subsistemnya ter-*install*. Perlu diketahui bahwa semua berkas yang disebutkan disini bersifat opsional :
- hosts : informasi statis tentang nama *host*
- issue : pesan sebelum *login* dan berkas identifikasi
- networks : informasi statis tentang nama jaringan
- passwd : berkas *password*
- dan lain lain.

/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. Perlu diingatkan bahwa keberadaan subdirektori /home pada direktori /root bersifat opsional.

/lib : Pustaka dasar bersama dan modul kernel

Direktori /lib meliputi pustaka bersama yang dibutuhkan untuk melakukan proses *boot* pada 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 instruksi biner, mungkin terdapat satu atau lebih varian dari direktori /lib. Jika direktori ini terdapat lebih dari satu, maka persyaratan dari isi tiap direktori adalah sama dengan direktori /lib normalnya, kecuali jika memang /lib<qual>/xxx tidak dibutuhkan.

/media: *Mount point* untuk *removeable* media

Direktori ini berisi subdirektori yang digunakan sebagai *mount point* untuk media-media *removeable* seperti *floppy disk*, CDROM, dll.

/mnt: *Mount point* temporer

Direktori ini disediakan agar administrator sistem dapat melakukan operasi *mount* secara temporer pada suatu sistem berkas yang dibutuhkan. Isi dari direktori ini bersifat lokal sehingga tidak mempengaruhi sifat-sifat dari program yang sedang dijalankan.

/opt: Aplikasi tambahan (*add-on*) untuk paket perangkat lunak

/opt disediakan untuk aplikasi tambahan paket perangkat lunak. Paket tambahan 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*

Direktori *home* untuk pengguna *root* sebenarnya dapat ditentukan secara manual baik oleh pengembang maupun pengguna, namun direktori /root disini adalah lokasi *default* yang direkomendasikan.

/sbin: Sistem Biner

Perintah-perintah yang digunakan untuk administrasi sistem disimpan di /sbin, /usr/sbin, dan /usr/local/sbin. /sbin berisi perintah biner dasar untuk melakukan operasi *boot* pada sistem, mengembalikan sistem dan memperbaiki sistem sebagai tambahan untuk perintah-perintah biner di direktori /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

Direktori ini berisi data-data *site-specific* yang disediakan oleh sistem.

/tmp: Berkas-berkas temporer

Direktori /tmp harus tersedia untuk program-program yang membutuhkan berkas temporer.

15.4. Sistem Berkas `/usr/`

Direktori /usr merupakan salah satu direktori terpenting pada suatu sistem berkas. Direktori ini bersifat *shareable*, itu berarti isi dari direktori /usr dapat digunakan oleh *host* lain yang menggunakan sistem operasi berbasis FHS. Isi dari direktori /usr tidak dapat ditulis (*read-only*). Paket perangkat lunak yang besar tidak boleh membuat subdirektori langsung pada hirarki /usr ini.

Tabel 15.5. Direktori atau link yang harus ada pada direktori /usr.

Direktori	Keterangan
bin	sebagian besar perintah pengguna

Direktori	Keterangan
include	berkas *header* yang di-*include* pada program C
lib	pustaka
local	hirarki lokal (kosong setelah instalasi utama)
sbin	sistem biner yang tidak vital
share	data yang *architecture-independent*

/usr/bin: Sebagian perintah pengguna

Direktori ini adalah direktori primer untuk perintah-perintah *executable* dalam sistem UNIX.

Beberapa contoh berkas yang harus diletakkan dalam direktori /usr/bin adalah *perl, phyton, tclsh, wish* dan *expect*

/usr/include: Direktori untuk *include-files* standar bahasa pemrograman C.

Direktori ini berisi berkas *include* oleh sistem yang bersifat umum, 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 oleh pengguna atau *shell script*. Aplikasi-aplikasi dapat menggunakan subdirektori tunggal di bawah /usr/lib. Jika aplikasi tersebut menggunakan subdirektori, semua data yang bergantung pada arsitektur mesin 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 kemanapun yang dituju oleh link simbolik/lib/X11.

/usr/lib<qual>: Format pustaka alternatif

/usr/lib<qual> memiliki 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 *synonimous*(biasanya ini berarti salah satunya harus merupakan *link* simbolik).

/usr/sbin: Sistem biner standar yang non-vital

Direktori ini berisi perintah-perintah 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. Contohnya 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.

15.5. Sistem Berkas `/var/`

Direktori /var merupakan direktori yang dikhususkan untuk berkas-berkas data variabel (berkas yang dapat diubah-ubah). Selain itu, direktori ini juga berisi berkas dan data variabel *spool* , data *administrative* dan *logging* , serta data tersier dan temporer. Hirarki "/var" dapat berisi berkas-berkas yang bisa di-*share* dan ada pula yang tidak dapat di-*share*. Contoh isi direktori "/var" yang dapat di-*share* adalah, "/var/mail", "/var/cache/man", "/var/cache/fonts" dan "/var/spool/news". Sedangkan isi direktori "/var" yang tidak dapat di-*share* yaitu, "/var/log", "/var/lock" dan "/var/run".

Direktori "/var" dibuat untuk memungkinkan operasi *mount* pada direktori "/usr" secara *read-only*. Semua berkas yang ada di direktori "/usr", yang ditulis selama sistem berjalan, harus diletakkan pada direktori "/var". Jika direktori "/var" tidak dapat dibuat pada partisi yang terpisah, biasanya hirarki "/var" dipindahkan ke luar dari partisi *root* dan dimasukkan ke dalam partisi "/usr". (Hal ini kadang dilakukan untuk mengurangi ukuran partisi *root* atau saat kapasitas dipartisi *root* mulai berkurang). Walaupun demikian, hirarki "/var" tidak boleh di-*link* ke "/usr", karena akan membuat pemisahan antara "/usr" dan hirarki "/var" semakin sulit dan bisa menciptakan konflik dalam penamaan dan begitu sebaliknya.

Berikut ini adalah direktori / *link* yang dibutuhkan dalam hirarki "/var"

Tabel 15.6. Contoh

Direktori	Keterangan
cache	Data *cache* aplikasi
lib	Informasi status variabel
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 *symbolic link* yang ada di bawah ini, harus diletakkan dalam Hirarki "/var", jika subsistem yang berhubungan dengan direktori tersebut memang di- *install*:

Tabel 15.7. Direktori yg harus diletakkan di /var

Direktori	Keterangan
account	*Log accounting* proses
crash	*System crash dumps*
games	Data variabel *game*

Direktori	Keterangan
mail	Berkas *mailbox* pengguna
yp	*Network Information Service* (NIS) berkas *database*

Berikut ini penjelasan masing-masing dari direktori diatas

/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 untuk mengurangi *time-consuming* M/K yang besar. 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 yaitu, "/var/cache/apt", "/var/cache/cups", "/var/cache/debconf"', dll.

/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 *games*

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 direktori lain, seperti"/usr/share/games".

/var/lib: Informasi status variabel

Direktori ini berisi informasi status suatu aplikasi dari sistem. 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* ataupun data *spool*.

Sebuah aplikasi harus menggunakan subdirektori "/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", dll.

/var/lock: *Lock* berkas

Berkas *lock* harus disimpan dalam struktur direktori /var/lock. Berkas *lock* untuk piranti 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 didalam"/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, ``/var/log/aptitude", ``/var/log/auth.log", ``/var/log/cups", dll.

/var/mail: Berkas *mailbox* pengguna

Mail spool harus dapat diakses melalui "/var/mail" dan berkas *mail spool* harus menggunakan format <nama_pengguna>. Sedangkan 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 harus dapat mengabaikan ekstra spasi, baris-baris tambahan, angka nol.

/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 berikutnya (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*, tetapi direkomendasikan bahwa penghapusan dilakukan tidak sesering penghapusan di "/tmp".

/var/yp: Berkas *database* NIS

Adapun yang terdapat dalam direktori ini yaitu data variabel dalam *Network Information Service* (NIS) atau biasa dikenal dengan *Sun YellowPages(YP)*.

15.6. Spesifik GNU/Linux

Sistem berkas yang ada pada Linux berbeda dengan sistem berkas pada sistem operasi lainnya. Pada Linux, sistem berkas harus mengikuti syarat-syarat dan anjuran tertentu. Walaupun demikian, syarat

dan anjuran tersebut tidak bertentangan dengan *standard* yang telah dijelaskan sebelumnya. Adapun syarat yang harus dipenuhi tersebut adalah sebagai berikut:

- "/": 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.
- "/bin". Pada"/bin", Sistem Linux membutuhkan berkas tambahan *set serial*
- "/dev: berkas piranti". berkas-berkas yang harus terdapat di direktori ini antara lain: "/dev/null", "/dev/zero", "/dev/tty".
- "/etc": Sistem Konfigurasi *host-specific*. Pada "/etc", sistem Linux membutuhkan berkas tambahan, yaitu lilo.cnof
- "/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.

15.7. Rangkuman

FHS (*Filesystem Hierarchy Standard*) adalah sebuah aturan standar penempatan lokasi *file* dan direktori yang ada pada sistem operasi. Sistem berkas diklasifikasikan berdasarkan dua parameter: *shareable* - *unshareable* dan *static* - variable. FHS terdiri dari tiga bagian besar:

- sistem berkas /root
- sistem berkas /usr
- sistem berkas /var.

Tujuannya adalah untuk memudahkan pengguna dan perangkat lunak dalam mengetahui letak suatu berkas atau direktori yang tersimpan pada suatu komputer.

Rujukan

[WEBRusQuYo2004] Rusty Russell, Daniel Quinlan, dan Christopher Yeoh. 2004. *Filesystem Hierarchy Standard http://www.pathname.com/fhs/* . Diakses 27 Juli 2006.

[FHS] Wikipedia. 1995. *FileSystem Hierarchy Standard http://en.wikipedia.org/ wiki/ Filesystem_Hierarchy_Standard* . Diakses 23 April 2007.

Bab 16. Implementasi Sistem Berkas

16.1. Pendahuluan

Salah satu bagian yang sangat tampak pada sebuah sistem operasi adalah sistem berkas. Banyak program yang membaca atau menulis setidaknya pada sebuah berkas. Bagi kebanyakan pengguna, *performance* dari sebuah sistem operasi sangat ditentukan oleh *interface*, *structure*, dan *reliability* dari sistem berkasnya. Hal ini disebabkan oleh kemampuan yang dimiliki sistem berkas dalam menyediakan mekanisme penyimpanan secara *online* dan pengaksesan isi berkas, yaitu data atau program. Sistem berkas terdapat pada *secondary storage* yang dirancang agar mampu menampung data dalam jumlah yang besar secara permanen.

Setiap berkas yang telah kita simpan, dalam implementasinya membutuhkan sebuah struktur direktori untuk mengaksesnya kembali. Sistem operasi harus mampu mengimplementasikan sistem berkas dengan berbagai tipe yang berbeda. Pada bab ini akan dibahas bagaimana penanganan yang dilakukan oleh sistem operasi. Bab ini juga membahas mengenai implementasi sistem berkas, implementasi direktori, serta beberapa lapisan yang ada di *file system* dan *virtual file system*.

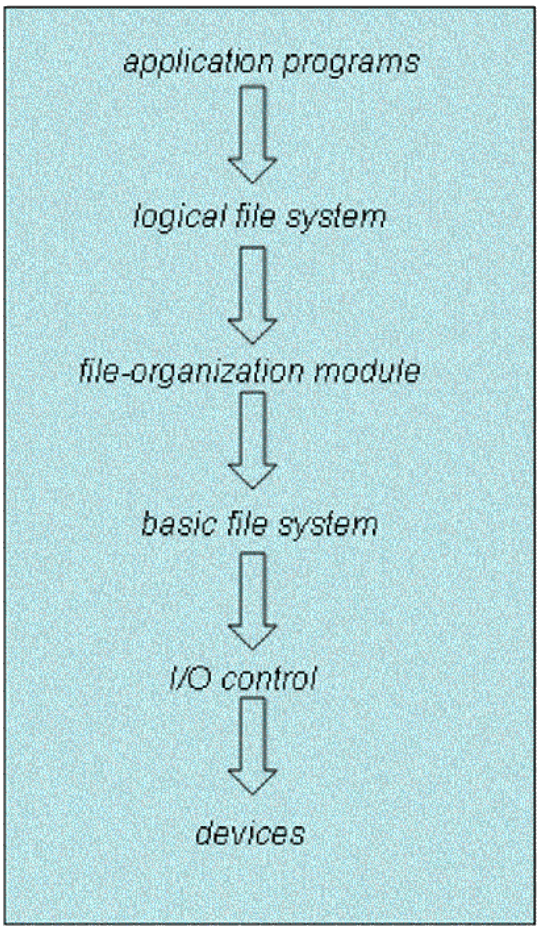
16.2. Struktur Sistem Berkas

Disk merupakan tempat penyimpanan sekunder dimana sistem berkas disimpan. Ada dua karakteristik dalam penyimpanan banyak berkas, yaitu:

1. Sebuah disk dapat ditulis kembali, memungkinkan untuk membaca, memodifikasi, dan menulis blok kembali pada tempat yang sama.
2. Sebuah blok yang menyimpan informasi yang kita cari, dapat diakses secara langsung. Dalam aplikasinya, kita dapat mengakses berkas secara *random* maupun *sequential*.

Sistem operasi menyediakan satu atau lebih sistem berkas untuk menyimpan dan mengambil data dengan mudah. Struktur berkas itu sendiri terdiri dari beberapa *layer*. Strukturnya dapat dilihat pada Gambar 16.1, “*Layered File System*”.

Gambar 16.1. ***Layered File System***



Setiap level menggunakan *feature-feature* dari lapisan di bawahnya untuk digunakan sebagai *feature* baru bagi lapisan di atasnya. Level terbawah adalah *I/O control* yang terdiri dari beberapa *device driver* dan penanganan interupsi untuk memindahkan informasi antara memori utama dan disk *system*. *Device driver* dapat dianggap sebagai penerjemah. Inputnya terdiri dari perintah-perintah *high level*, misalkan "ambil blok 123". Outputnya berupa instruksi-instruksi *hardware* yang lebih spesifik. Instruksi ini digunakan oleh pengendali *hardware* yang menghubungkan *I/O device* dengan sistem lainnya.

Basic file system bertugas dalam hal menyampaikan perintah-perintah *generic* ke *device driver* yang dituju untuk membaca dan menulis blok-blok fisik pada disk. Masing-masing blok fisik diidentifikasi dengan alamat disknya.

File organization modul adalah lapisan dalam sistem berkas yang mengetahui tentang berkas, blok-blok logis, dan blok-blok fisik. Dengan mengetahui tipe dan lokasi dari berkas yang digunakan, *file organization modul* dapat menerjemahkan alamat blok logis ke alamat blok fisik untuk selanjutnya diteruskan ke lapisan *basic file system*. *File organization modul* juga mengandung *free space manager* yang mencatat jejak blok-blok yang tidak dialokasikan dan menyediakannya ke *file organization modul* ketika dibutuhkan.

Lapisan yang terhubung dengan program aplikasi yaitu *logical file system* yang bertugas dalam mengatur informasi metadata. Metadata meliputi semua struktur dari sistem berkas, kecuali data sebenarnya (isi dari berkas). Lapisan ini juga mengatur struktur direktori untuk menyediakan informasi yang dibutuhkan *file organization modul*. Struktur dari sebuah berkas ditentukan oleh lapisan ini dengan adanya *file control block.*

16.3. *File Control Block*

Pada disk, sistem berkas bisa mengandung informasi mengenai bagaimana cara mem-*boot* sistem operasi, jumlah total blok, jumlah dan lokasi dari *free blocks*, struktur direktori dan berkas individu. Berikut ini penjelasan dari struktur-struktur tersebut.

- Sebuah *boot control block* (*per volume*)

  Mengandung informasi yang dibutuhkan oleh sistem untuk mem-*boot* sistem operasi dari *volume* yang dimilikinya. Jika disk tidak mempunyai sistem operasi, blok disknya bisa saja kosong. Pada UFS, ini disebut *boot block*, sedangkan pada NTFS, ini disebut *partition boot sector*.

- *Volume control block (per volume)*

  Mengandung informasi khusus mengenai partisi, misalnya jumlah blok yang dipartisi, ukuran blok, *free block count* dan *free block pointer*, *free FCB count*, dan *FCB pointer*. Pada UFS, disebut *superblock*, dan pada NTFS, disimpan di tabel *master file*.

- *Directory structure per file system*

  Digunakan untuk mengorganisasi berkas. Pada UFS, struktur direktori ini meliputi nama berkas serta jumlah *inode* yang terkait dan disimpan dalam tabel *master file*.

- *Per-file FCB*

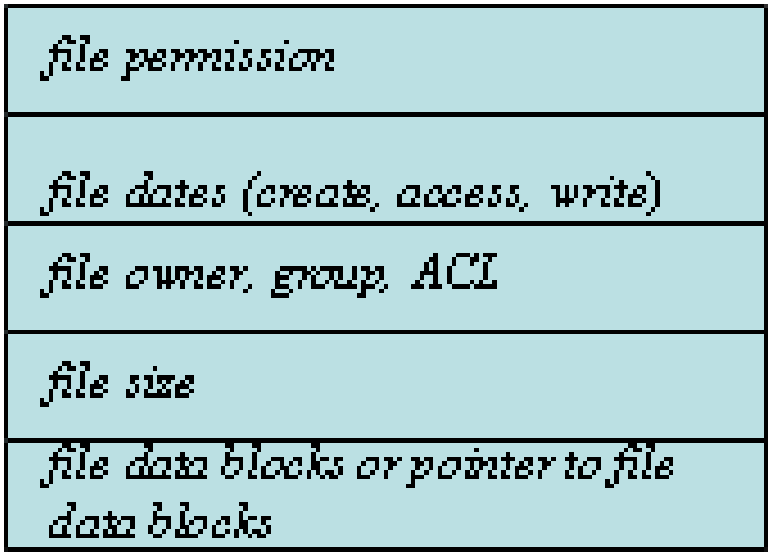
  Mengandung semua informasi tentang berkas yaitu meliputi *file permission*, kepemilikan, ukuran, dan lokasi dari blok data.

Informasi yang ada di dalam memori digunakan untuk pengaturan sistem berkas dan peningkatan *performance* dengan *caching*. Berikut ini adalah penjelasan mengenai strukturnya :

- *In-memory mount table* mengandung informasi setiap *volume* yang di-*mount*.
- *In-memory directory structure cache* menyimpan informasi direktori yang baru diakses.
- *System-wide open file table* menyimpan hasil *copy* FCB.

Untuk membuat berkas baru, program aplikasi memanggil *logical file system* karena *logical file system* mengetahui format dari struktur direktori. Kemudian, FCB yang baru dialokasikan dan sistem membaca direktori yang dituju ke memori, mengubahnya dengan nama berkas, dan FCB yang baru menulisnya ke dalam disk. Contoh struktur FCB, bisa dilihat pada Gambar 16.2, "*File Control Block*"

Gambar 16.2. *File Control Block*



Beberapa sistem operasi termasuk UNIX, memperlakukan berkas sama seperti direktori. Sistem operasi yang lain, seperti Windows NT, mengimplementasikan *system call* yang terpisah untuk berkas dan direktori, direktori diperlakukan sebagai objek yang berbeda dari berkas.

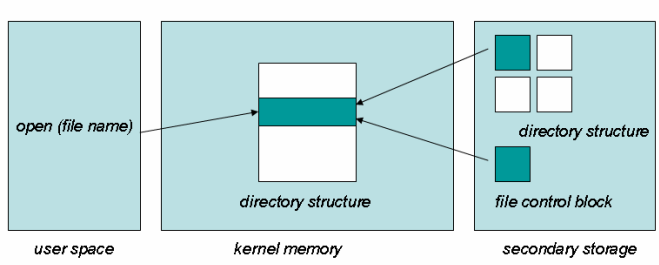
Pada tahap selanjutnya, *logical file system* memanggil *file organization module* untuk memetakan direktori M/K ke *disk-block number* yang dikirimkan ke sistem berkas dasar dan sistem kendali M/K.

Setelah berkas berhasil dibuat, berkas tersebut dapat digunakan untuk operasi M/K. Pertama, tentu saja berkas tersebut harus dibuka terlebih dahulu. Perintah `open()` mengirim nama berkas ke sistem berkas. *System call* `open()` mencari *system-wide open-file table* untuk melihat apakah berkas yang ingin dibuka sedang digunakan oleh proses lain. Jika benar, maka *entry* dari *per-process open-file table* menunjuk *system-wide open-file table* yang sedang eksis.

Ketika berkas sedang dibuka, nama berkas tersebut dicari dalam struktur direktori. Setelah nama berkas ditemukan, FCB dari berkas tersebut disalin ke dalam *system-wide open-file table* di memori. Pada tahapan selanjutnya, sebuah *entry* dibuat dalam *per-process open-file table* di mana *entry* tersebut menunjuk ke *system-wide open-file table*. Semua bentuk operasi yang melibatkan berkas dilakukan oleh pointer dari *entry* ini. Selama berkas tersebut belum ditutup, semua operasi dari berkas dilakukan pada *open-file table*.

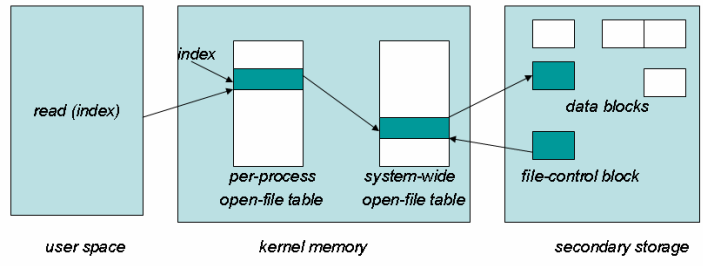
Ketika sebuah berkas ditutup, *entry* dari *per-process table* dihapus, dan jumlah *entry* dari *system-wide open-file table* dikurangi. Ketika semua pengguna yang menggunakan berkas menutup berkas tersebut, semua metadata yang telah diubah disalin kembali ke disk berdasarkan struktur direktori, dan yang terakhir, *entry* dari *system-wide open-file* dihilangkan atau dihapus. Gambar 16.3, "*Fungsi* `open` *Sebuah Berkas* " mengilustrasikan pembukaan sebuah berkas.

Gambar 16.3. *Fungsi* `open` *Sebuah Berkas*



Sedangkan pada Gambar 16.4, "*Reading a File*" mengilustrasikan pembacaan sebuah berkas.

Gambar 16.4. *Reading a File*



16.4. Partisi Sistem ROOT

Partisi adalah pembagian secara logis, namun secara fisik tidak terbagi. Dengan partisi *hard disk*, kita bisa menginstal lebih dari dua sistem operasi dalam sebuah komputer. Partisi dilakukan untuk

dapat memudahkan saat melakukan perbaikan. Misalkan ada salah satu partisi yang rusak, maka kita hanya perlu memperbaiki partisi tersebut, karena partisi yang lain tidak terpengaruhi. Partisi juga dapat mempercepat akses ke *hard disk*. Partisi merupakan hal yang esensial dalam Linux. Minimal, Linux hanya memerlukan dua partisi yaitu */swap* untuk menyimpan memori virtual dan */root*. Namun, ada yang menambahkan satu partisi lagi yang dianggap esensial juga yaitu */home* untuk menyimpan semua berkas yang kita buat. Berdasarkan ada atau tidak adanya sistem berkas yang terlibat dalam partisi, partisi ada dua macam, yaitu *raw partition* di mana partisi ini tidak memiliki sistem berkas di dalamnya, dan *cooked partition* yang memiliki sifat sebaliknya.

Root merupakan puncak dari sebuah hirarki direktori, biasanya dinotasikan dengan tanda *slash* . Di dalam *root*, misalnya di Linux, terdapat kernel Linux dan apapun yang kita *install* dengan Linux. *Root* tersebut ekuivalen dengan *drive* C:\ di Windows. *Root partition* di-*mount* saat *boot* . Sedangkan partisi yang lain bisa di-*mount* baik pada saat *boot* maupun di luar waktu *boot*, tergantung pada sistem operasinya. Untuk menandakan bahwa operasi *mount* telah berhasil, sistem operasi akan memverifikasi bahwa *device* telah memiliki sistem berkas yang valid. Cara yang dilakukan adalah sistem operasi akan meminta *device driver* untuk memeriksa apakah sistem berkas yang ada di dalamnya sesuai dengan format yang diinginkan atau tidak, kemudian *device driver* akan memverifikasi kembali pada sistem operasi. Jika tidak valid, maka perlu dilakukan perbaikan partisi. Sistem operasi menyimpan sistem berkas mana saja yang sudah di-*mount* dalam *in-memory mount table structure*.

16.5. *Virtual File System*

Sistem operasi modern harus mampu mengimplementasikan berbagai sistem berkas dengan tipe yang berbeda dalam waktu yang bersamaan. Salah satu teknik yang digunakan sebagai solusinya adalah dengan menggunakan *virtual file system* (VFS). VFS saat ini banyak digunakan oleh berbagai sistem operasi. Ide dari VFS adalah meletakkan informasi di kernel untuk merepresentasikan keseluruhan tipe sistem berkas, dan juga terdapat sebuah fungsi untuk setiap operasi yang dimiliki sistem berkas. Sehingga, untuk setiap *system call* seperti fungsi `read()`, `write()`, `open()`, dan lainnya, kernel akan mensubstitusikannya menjadi *actual function* yang dimiliki oleh setiap sitem berkas dengan berbagai tipe.

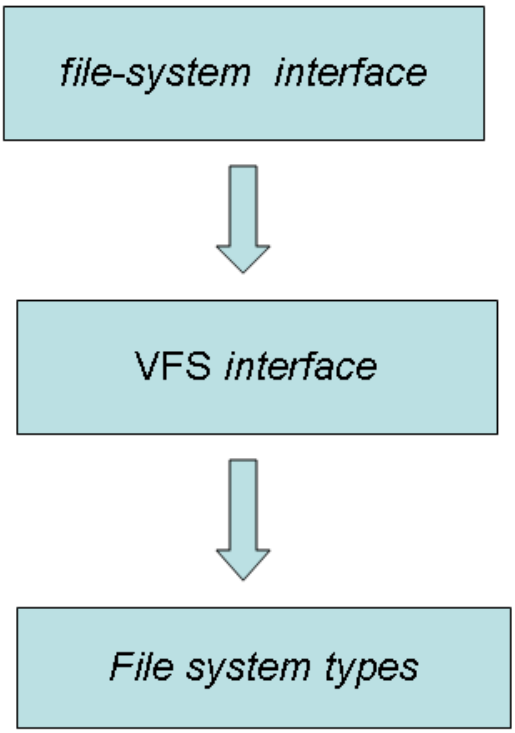
VFS menggunakan konsep *object oriented* dalam mengimplementasikan sistem berkas. Di dalam *VFS* terdapat sebuah berkas yang merepresentasikan seluruh tipe sistem berkas yang ada, berkas ini dinamakan *common file model*. Berkas inilah yang menggunakan konsep *object oriented*, yang di dalamnya terdapat struktur data dan *method* yang diimplementasikan.

Terdapat empat objek di dalam *common file model*, diantaranya :

1. ***Superblock object.*** Objek ini menyimpan informasi tentang *mounted file system* atau sistem berkas secara keseluruhan.
2. ***Inode object.*** Objek ini menyimpan informasi umum tentang file tertentu (*individual file*).
3. ***File object.*** Objek ini menyimpan informasi tentang file yang sedang dibuka.
4. ***Dentry object.*** Objek ini menyimpan informasi tentang *link-link* dari sebuah *entry directory file*.

Struktur data dan *method* yang diimplementasikan, digunakan untuk menyembunyikan implementasi detail dari *actual function* pada sistem berkas dengan *system call* yang mengaksesnya. Oleh karena itu, dalam mengimplementasikan sistem berkas, terdapat tiga *layer* utama, seperti pada Gambar 16.5, "*Virtual File System Layer*".

Gambar 16.5. *Virtual File System Layer*



Lapisan yang pertama adalah *file system interface*. Contohnya adalah beberapa *system call* seperti `read()`, `write()`, `open()` dan lainnya. *System call* ini tidak berhubungan langsung dengan sistem, namun terhubung melalui sebuah lapisan abstrak yaitu *virtual file system*.

Lapisan yang Kedua adalah *VFS Interface*. *Virtual file system* memiliki dua fungsi penting, yaitu:
1. Memisahkan operasi-operasi *file system generic* dari implementasi detailnya, dengan cara mendefinisikan *virtual file system interface*.
2. *Virtual file system interface* didasarkan pada struktur representasi berkas yang disebut *vnode*, yang memiliki *numerical designator* yang unik untuk setiap *network file*.

Lapisan yang ketiga adalah sistem berkas dengan berbagai tipe. Secara umum, terdapat tiga macam tipe sistem berkas, yaitu:
1. ***Disk-based file system*.** Sistem berkas ini mengatur ruang memori yang tersedia di dalam partisi disk lokal. Misalnya, *Ext2 (Second Extended file system)*, *Ext3 (Third Extended file system)*, dan *Reiser file system* yang tedapat di Linux.
2. ***Network file system*.** Sistem berkas ini terdapat di *network*, misalnya NFS.
3. ***Special file system*.** Sistem berkas ini tidak terdapat di *disk space*, baik lokal maupun *network*, misalnya */proc file system*.

16.6. Implementasi Direktori Linier

Pemilihan algoritma untuk pencarian sebuah direktori merupakan salah satu penentu tingkat efisiensi dan *performance* suatu sistem berkas. Salah satu algoritma itu adalah implementasi direktori linier. Algoritma ini merupakan algoritma yang paling sederhana dalam pembuatan program yang mengimplementasikan *linier list* dari nama-nama berkas yang memiliki pointer ke blok-blok data. Namun, algoritma ini tidak efisien apabila digunakan pada suatu direktori yang memiliki jumlah berkas yang sangat banyak karena proses eksekusi berkas membutuhkan waktu yang lama. Misalnya, untuk membuat berkas, kita harus memastikan bahwa dalam direktori tidak ada berkas yang

mempunyai nama yang sama. Kemudian, berkas yang baru tersebut ditambahkan pada akhir direktori tersebut. Untuk menghapus sebuah berkas, kita mencari terlebih dahulu nama berkas yang hendak dihapus dalam direktori, kemudian membebaskan *space* yang dialokasikan pada berkas tersebut. Apabila kita menginginkan untuk menggunakan kembali *entry* berkas tersebut, ada beberapa alternatif yang bisa kita gunakan, yaitu:

1. kita bisa menandai berkas tersebut misalnya melalui pemberian nama berkas yang khusus pada berkas tersebut.
2. Kedua, kita bisa menempatkan berkas tersebut pada sebuah *list of free directory entries*.
3. alternatif yang ketiga adalah dengan menyalin *entry* terakhir dalam direktori ke suatu *freed location*.

Salah satu kerugian yang ditimbulkan dalam penggunaan algoritma ini adalah pencarian berkas dilakukan secara *linier search*. Oleh karena itu, banyak sistem operasi yang mengimplementasikan sebuah *software cache* yang menyimpan informasi tentang direktori yang paling sering digunakan, sehingga pengaksesan ke disk bisa dikurangi. Berkas-berkas yang terurut dapat mengurangi rata-rata waktu pencarian karena dilakukan secara *binary search*. Namun, untuk menjaga agar berkas-berkas selalu dalam keadaan terurut, pembuatan maupun penghapusan berkas akan lebih rumit. Struktur data *tree* seperti *B-tree* bisa digunakan untuk mengatasi masalah ini.

16.7. Implementasi Direktori *Hash*

Pada implementasi ini, *linier list* tetap digunakan untuk menyimpan direktori, hanya saja ada tambahan berupa struktur data *hash*. Prosesnya yaitu *hash table* mengambil nilai yang dihitung dari nama berkas dan mengembalikan sebuah *pointer* ke nama berkas yang ada di *linier list*. Oleh karena itu, waktu pencarian berkas bisa dikurangi. Akan tetapi, ada suatu keadaan yang menyebabkan terjadinya peristiwa *collisions*, yaitu suatu kondisi di mana terdapat dua berkas yang memiliki nilai *hash* yang sama, sehingga menempati lokasi yang sama. Solusi yang dipakai untuk mengatasi hal tersebut yaitu dengan menggunakan *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. Efek samping dari penambahan *chained-overflow* tersebut adalah dapat memperlambat pencarian.

Ada beberapa kelemahan dari implementasi direktori *hash*, yaitu ukurannya yang tetap dan adanya ketergantungan fungsi *hash* dengan ukuran *hash table*. Sebagai contoh, misalnya kita membuat sebuah *linear-probing hash table* yang memiliki 32 *entry*. Sebuah fungsi *hash* dibutuhkan untuk mengubah nama berkas menjadi bilangan bulat dari 0 s.d. 31, misalnya dengan menggunakan fungsi modulo 32. Jika kita ingin untuk menambahkan sebuah berkas yang harus diletakkan pada lokasi yang ke-33, kita harus memperbesar ukuran dari *hash table* tersebut, sehingga diperlukan sebuah fungsi *hash* baru untuk disesuaikan dengan ukuran *hash table* tersebut. Oleh karena itu, berkas-berkas yang sebelumnya sudah ditempatkan di suatu lokasi pada *hash table* yang lama harus dicari tempat yang bersesuaian dengan menggunakan fungsi *hash* yang baru.

16.8. Rangkuman

Untuk memberikan akses yang efisien ke disk, sistem operasi mengizinkan satu atau lebih sistem berkas yang digunakan untuk menyimpan, meletakkan, dan mengambil data dengan mudah. Sistem berkas tersebut terdiri dari beberapa lapisan yang mana setiap lapisan akan menyediakan fitur-fitur untuk dipergunakan pada lapisan di atasnya. Lapisan tersebut (dari tingkat yang terbawah) antara lain *device driver, I/O control, basic file system, file-organization module, logical file system*, dan *application program*. Berbagai operasi yang dilakukan pada sistem berkas, seperti membuka maupun membaca berkas, sangat membutuhkan informasi tentang berkas yang hendak digunakan. Semua informasi tentang berkas tersebut didapatkan pada *file control block.*

Dalam pengimplementasiannya, sistem operasi harus mampu mendukung berbagai jenis sistem berkas. Teknik yang digunakan oleh sistem operasi untuk mendukung kemampuan ini disebut *Virtual File System (VFS)*. Konsep dalam mengimplementasikan VFS ini adalah konsep *object oriented* yaitu menggunakan sebuah berkas yang dapat merepresentasikan seluruh tipe berkas. Berkas ini disebut *common file model*. Oleh karena itu, digunakanlah tiga lapisan utama dalam mengimplementasikan sistem berkas, diantaranya *file system interface, VFS interface*, dan *file system types*.

Algoritma untuk mencari berkas dalam sebuah direktori juga digunakan sebagai parameter *performance* dari sebuah sistem operasi. Ada dua algoritma yang sering digunakan dalam pengimplementasian direktori, yaitu implementasi direktori linier dan *hash*. Masing-masing bentuk implementasi ini memiliki kelebihan dan kekurangan tersendiri. Untuk implementasi linier, kelebihannya adalah sederhana dalam pembuatan programnya dan kekurangannya adalah lamanya mengakses berkas apabila terdapat banyak sekali berkas dalam sebuah direktori. Sedangkan pada implementasi direktori *hash*, kecepatan pencarian berkas lebih cepat dibandingkan dengan pencarian dalam direktori linier dan kekurangannya adalah ukuran tabel yang statis, sehingga perlu dilakukan pengubahan fungsi *hash* apabila ingin mengubah ukuran tabel *hash*.

Rujukan

[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.

[Bovet2002] Daniel P. Bovet dan Marco Cesati. 2002. *Understanding the Linux Kernel*. Second Edition. O'Reilly.

[WEBITC2006] Infoteknologi.com. 2006. *http://infoteknologi.com/info/index.php?option* . Diakses 30 Mei 2007.

Bab 17. Metode Alokasi Blok

17.1. Pendahuluan

Ruang untuk menyimpan berkas pada tempat penyimpanan utama, dalam hal ini memori, tidak cukup besar untuk menampung berkas dalam jumlah besar karena ukurannya yang terbatas dan harganya yang mahal, memory hanya dapat menyimpan berkas saat komputer dalam keadaan terhubung dengan arus listrik.Karena hal-hal tersebut di atas, maka diperlukan tempat penyimpanan sekunder (disk) yang dapat mempertahankan berkas walaupun tidak ada arus listrik yang mengalir ke komputer. Dalam bab ini, akan dijelaskan bagaimana caranya mengalokasikan blok untuk sebuah berkas agar disk dapat digunakan secara optimal dan agar berkas dapat diakses secara cepat.

Ada beberapa mekanisme dalam mengalokasikan blok untuk menyimpan berkas ke dalam disk yang akan dibahas, diantaranya:

1. **Alokasi Berkesinambungan .**
2. **Alokasi *Link* .**
3. **Alokasi Berindeks.**

beserta cara mengatur ruang kosong pada disk dan cara mempertahankan kekonsistenan berkas serta mencegah kehilangan data.

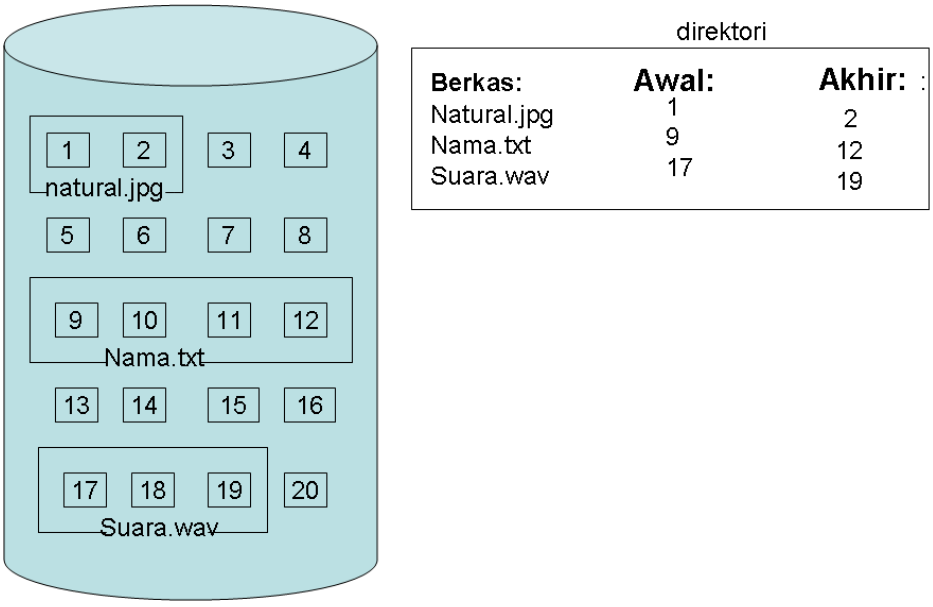
17.2. Alokasi Berkesinambungan

Jenis alokasi ini menempatkan berkas-berkas pada blok secara berkesinambungan atau berurutan dalam disk.

Alokasi berkesinambungan dari suatu berkas diketahui melalui alamat dan panjang disk dalam unit blok dari blok pertama. Sebagai contoh, jika ada berkas dengan panjang n blok dan mulai dari lokasi b maka berkas tersebut menempati blok b, b+1, b+2, #, b+n-1. Direktori untuk setiap berkas mengindikasikan alamat blok awal dan panjang area yang dialokasikan untuk berkas tersebut. Terdapat dua macam cara untuk mengakses berkas yang dialokasi dengan metode ini, yaitu:

- **Akses secara berurutan.** Sistem berkas mengetahui alamat blok terakhir dari disk dan membaca blok berikutnya jika diperlukan.
- **Akses secara langsung.** Untuk akses langsung ke blok i dari suatu berkas yang dimulai pada blok b, dapat langsung mengakses blok b+i.

Gambar 17.1. Gambar Alokasi Berkesinambungan



Kelebihan dari metode ini adalah:
1. Penerapannya mudah karena perpindahan *head* membutuhkan waktu yang sedikit untuk menyimpan suatu berkas karena letaknya berdekatan.Perpindahan *head* menjadi masalah hanya bila sektor terakhir dari suatu *track* a dan sektor awal track a+1.
2. Waktu pengaksesan suatu berkas lebih cepat karena *head* tidak berpindah terlalu jauh dalam pembacaan berkas.

Kekurangan metode ini adalah:
1. Pencarian blok kosong untuk menyimpan sebuah berkas baru, ukuran berkas harus diketahui terlebih dahulu untuk dapat meletakkan berkas pada blok dengan ukuran yang tepat. Dan jika peletakkan berkas pada blok menggunakan sebagian kecil blok tersebut, maka tidak ada berkas lain yang dapat menggunakan ruang kosong itu. Inilah yang disebut dengan fragmentasi internal.
2. Bila ada berkas yang dihapus, maka ruang disk yang dibebaskan kemungkinan tidak akan cukup untuk berkas baru. Inilah yang dinamakan fragmentasi eksternal.
3. Perlu blok khusus untuk menyimpan direktori yang berisi nama berkas, alamat awal sebuah berkas, dan panjang berkas.

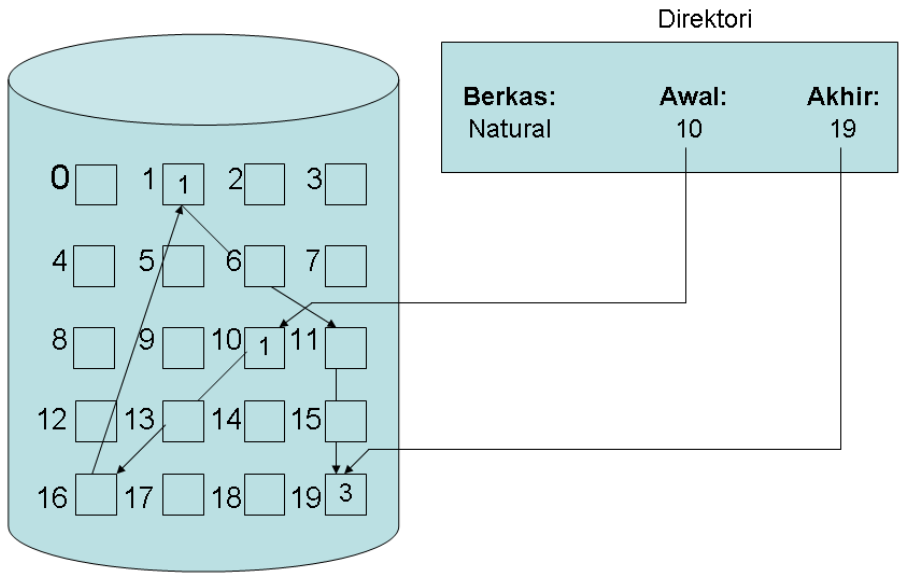
Penempatan suatu berkas baru pada disk, maka harus mencari ruang-ruang kosong yang ada pada disk. Pencarian ini dapat menggunakan metode *first fit* dan *best fit*. *First fit* adalah ruang kosong pertama yang ditemukan oleh *head* dalam pencariannya yang ukurannya mencukupi berkas tersebut. *Best fit* adalah memakai ruang kosong yang memiliki besar yang paling sesuai dengan ukuran berkas yang akan disimpan.

Terdapat cara pencegahan fragmentasi eksternal pada alokasi blok berkesinambungan ini, dengan cara, menyalin seluruh berkas yang ada pada disk ke suatu *floppy disk* atau *tape* magnetik yang kemudian akan disalin kembali ke disk secara berkesinambungan. Dengan demikian, ruang kosong terpisah dari ruang yang berisi berkas, sehingga ukuran ruang kosong yang lebih besar dan ruang kosong tersebut dapat dimanfaatkan oleh berkas lain. Untuk mengatasi kelemahan-kelemahan dari alokasi blok berkesinambungan, ada suatu modifikasi alokasi berkesinambungan, yaitu *extent-file system.*

17.3. Alokasi Link

Alokasi *link* dapat memecahkan semua masalah yang terdapat pada alokasi berkesinambungan. Dengan alokasi *link* , setiap berkas dipandang sebagai sebuah *linked-list* pada blok disk. Alokasi *link* juga memiliki direktori yang berisi nama berkas, alamat awal sebuah berkas, dan alamat akhir sebuah berkas. Direktori tersebut memiliki *pointer* ke alamat awal dari sebuah berkas. Untuk berkas yang kosong, *pointer*-nya diinisialisasi dengan nilai NULL (akhir dari nilai *pointer*).

Gambar 17.2. Gambar Alokasi Link

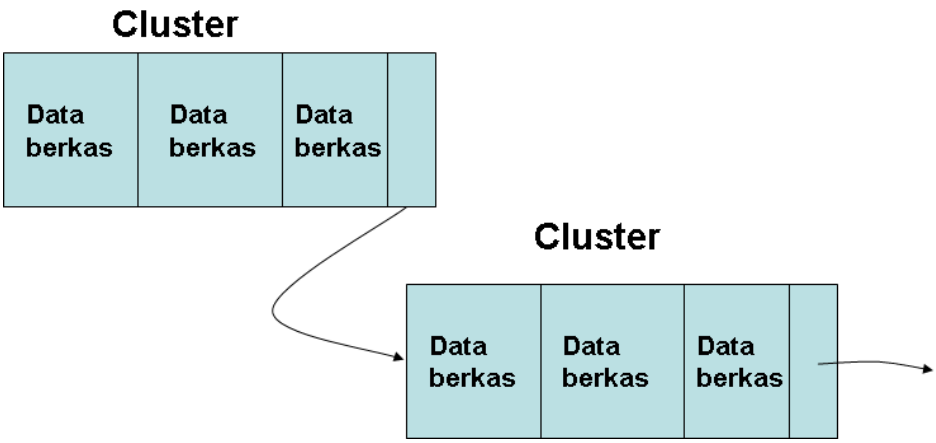


Pada alokasi *link*, sebuah berkas disimpan menyebar ke seluruh disk yang dihubungkan dengan *pointer*. *Pointer* pertama menunjuk ke berkas data kedua, *pointer* dari berkas kedua menunjuk ke berkas data ketiga, dst.

Proses membaca berkas dapat dilakukan dengan mengikuti *pointer* dari blok ke blok. Di sini tidak terjadi fragmentasi eksternal dan blok yang kosong dapat digunakan oleh berkas lain. Ukuran dari berkas tidak perlu didefinisikan pada saat berkas dibuat. Masalah yang terdapat pada alokasi *link* adalah ketika mencari sebuah blok berkas kita harus mencarinya dari awal dan mengikuti *pointer* sampai menemukan blok berkas yang dicari.

Hal lain yang jadi masalah adalah *pointer* memerlukan ruang tersendiri. Untuk mengatasi masalah ini maka menggunakan *cluster* yaitu menggabungkan berkas-berkas yang berdekatan menjadi satu, sehingga gabungan dari berkas- berkas itu hanya menggunakan satu *pointer* .Hal ini bisa menyebabkan fragmentasi internal karena banyak ruang yang tidak terpakai ketika suatu *cluster* hanya sebagian menggunakan blok.

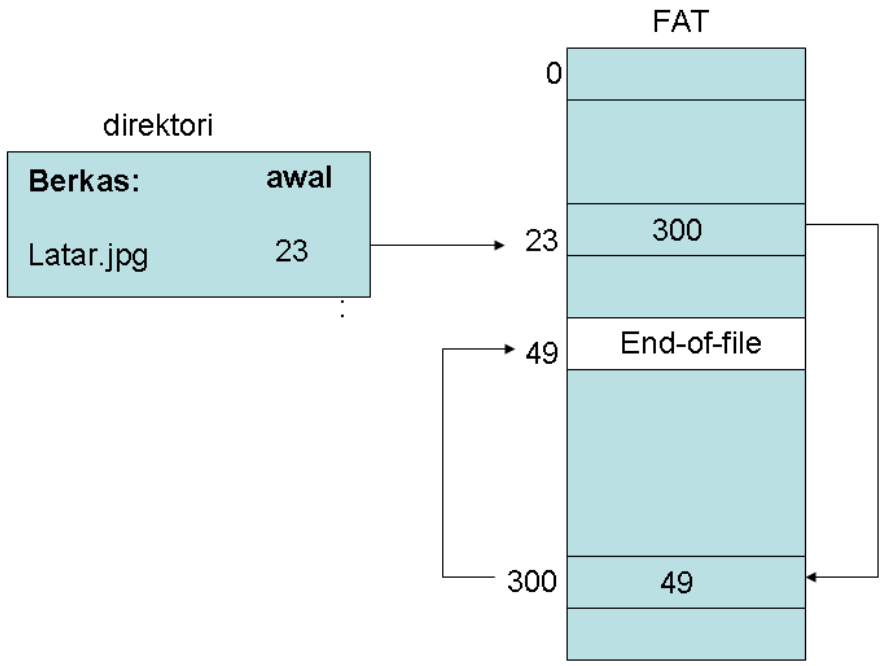
Gambar 17.3. Gambar Cluster



Masalah lainnya adalah kebenaran dari suatu berkas. Jika suatu berkas yang dihubungkan oleh *pointer* dan tersebar di seluruh disk mengalami kerusakan sehingga *pointer* -nya tidak menunjuk ke berkas yang tepat.

Suatu variasi penting dari alokasi *link* adalah dengan menggunakan *File Allocation Table* (FAT) yang direktorinya hanya berisi nama berkas dan alamat pertama suatu berkas. Dan penanda dari akhir sebuah berkas terdapat pada blok yang ditunjuk oleh *pointer* terakhir.

Gambar 17.4. Gambar FAT



Keunggulan dari sistem FAT adalah pengaksesan acak yang lebih mudah. Hal ini karena meski harus menelusuri rantai berkait untuk menemukan lokasi blok berkas, rantai blok seluruhnya di memori sehingga dapat dilakukan secara cepat tanpa membuat pengaksesan disk. Selain itu, direktori yang dimiliki sistem FAT juga cukup menyimpan bilangan bulat nomor blok awal saja. Blok awal ini digunakan untuk menemukan seluruh blok, tak peduli jumlah blok berkas itu. Direktori menunjuk blok pertama berkas dan FAT menunjukkan blok-blok berkas berikutnya.

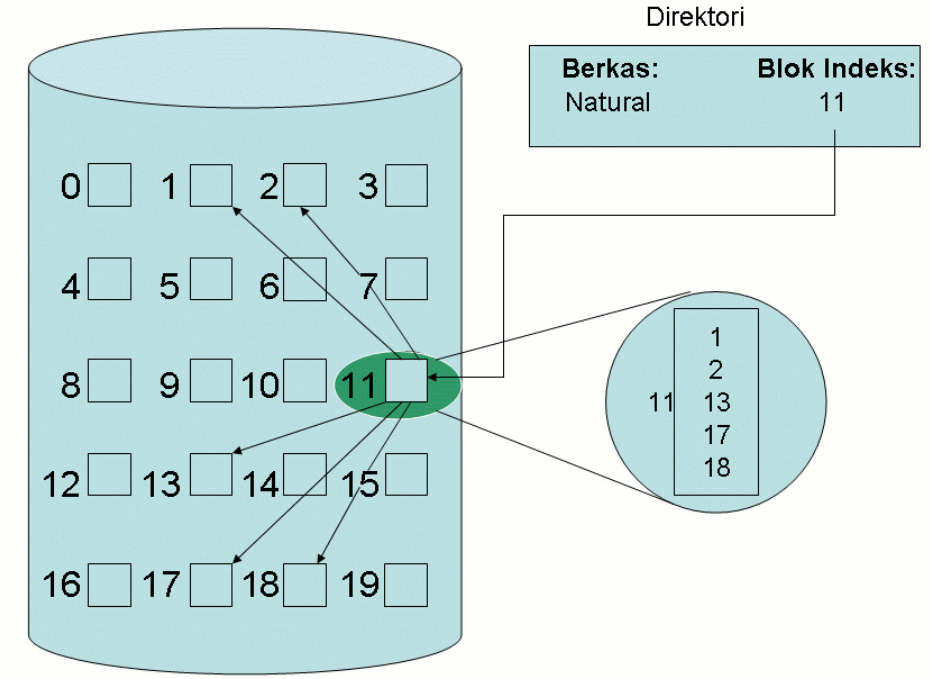
Disamping kelebihan, sistem FAT juga memiliki kekurangan yaitu jika penyimpanan berukuran besar mengakibatkan tabel berukuran besar dan harus ditaruh di memori utama meskipun hanya satu berkas yang dibuka

17.4. Alokasi Berindeks

Alokasi *link* memecahkan masalah fragmentasi eksternal dan masalah deklarasi ukuran berkas pada alokasi berkesinambungan tetapi pada FAT alokasi *link* tidak mendukung akses langsung dari *pointer* ke blok yang letaknya tersebar pada disk dan harus diurutkan. Alokasi berindeks memecahkan masalah ini dengan menyimpan semua *pointer* pada suatu lokasi khusus. Lokasi ini disebut blok indeks.

Setiap berkas memiliki satu blok indeks. Alokasi berindeks memiliki direktori yang berisi nama bekas dan blok indeks. Ketika sebuah berkas dibuat semua *pointer* pada blok indeks diset NULL. Ketika suatu blok baru pertama kali ditulis blok itu diatur oleh manajemen ruang kosong dan alamatnya dimasukkan ke blok indeks.

Gambar 17.5. Gambar Alokasi Berindeks



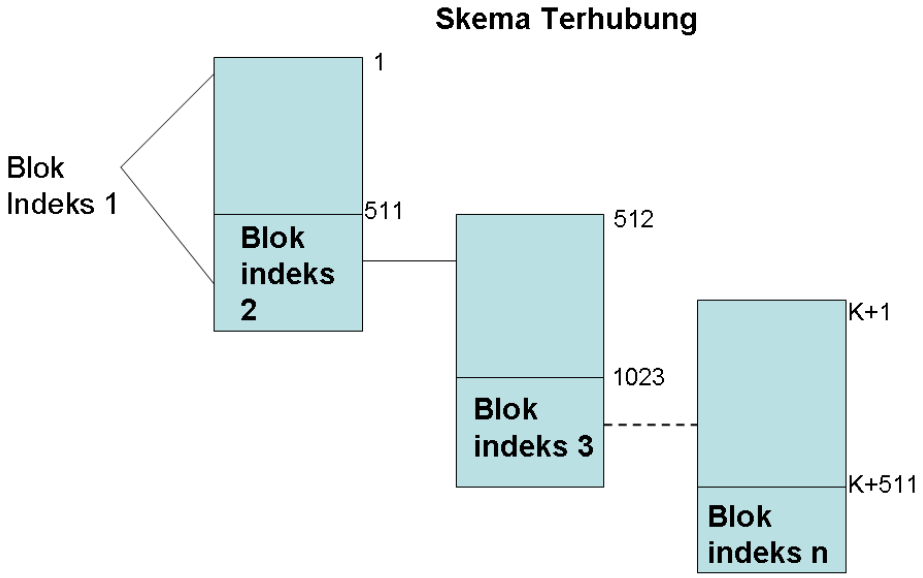
Alokasi berindeks menggunakan akses langsung tanpa mengalami fragmentasi eksternal karena blok kosong pada disk dapat digunakan untuk berkas lain. Alokasi berindeks memerlukakan suatu blok khusus yang berisi blok indeks. Hal ini sangat merugikan jika blok indeks lebih banyak daripada *pointer*.

Dengan alokasi *link* kita kehilangan banyak ruang untuk satu *pointer* per blok. Dengan alokasi berindeks semua blok indeks harus dialokasikan, walaupun hanya satu atau dua *pointer* yang tidak NULL.

Hal diatas menimbulkan pertanyaan seberapa besar seharusnya blok indeks yang tepat. Setiap berkas harus ada dalam blok indeks, maka harus menyediakan blok indeks seminimal mungkin. Jika blok indeks terlalu kecil, maka tidak bisa memenuhi kebutuhan berkas yang ukurannya besar. Ada tiga mekanisme yang bisa mengatasi masalah ini, yaitu sebagai berikut:

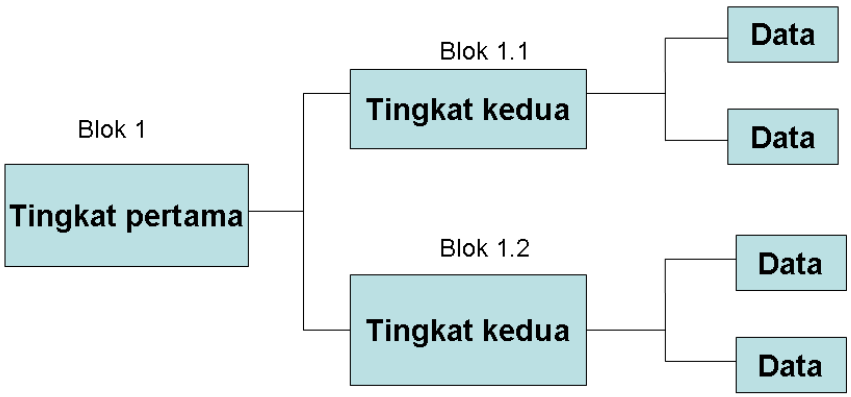
1. ***Linked Scheme .*** Untuk suatu berkas, blok indeks normalnya adalah satu blok. Untuk berkas yang besar, dapat menggabungkan beberapa blok indeks. Jadi, bila berkas kita masih berukuran kecil, maka isi dari tempat yang terakhir dari blok indeks berkas tersebut adalah NULL. Namun, bila berkas tersebut berkas besar, maka tempat terakhir itu berisikan alamat untuk ke blok indeks selanjutnya, dan begitu seterusnya.

Gambar 17.6. Gambar Linked Scheme



2. **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 ke tingkat ketiga dan keempat tergantung dengan ukuran berkas tersebut. Untuk blok indeks dua tingkat dengan ukuran blok 4.096 *byte* dan petunjuk yang berukuran 4 *byte* dapat mengalokasikan berkas hingga 4 GB, yaitu 1.048.576 blok berkas.

Gambar 17.7. Gambar Indeks Bertingkat



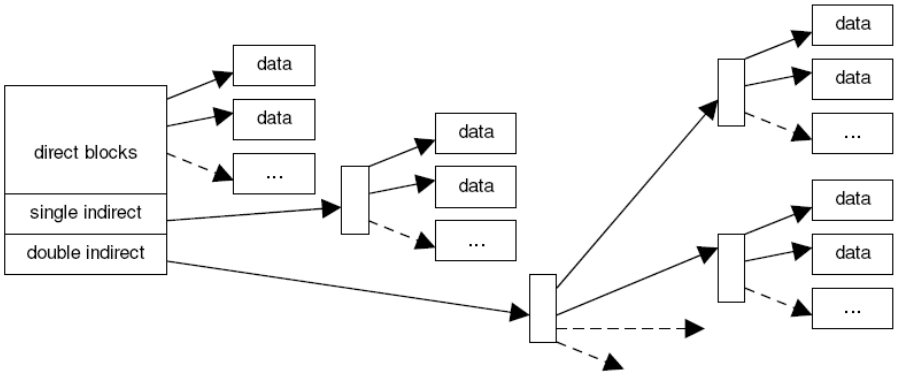
3. **Skema Kombinasi.** Alternatif lain adalah seperti yang dipakai oleh UNIX File System adalah kombinasi dari *direct blocks* dan *indirect blocks*. *Indirect blocks* terdiri dari *single, double,* dan *triple* .

17.5. Kombinasi Alokasi

Seperti yang telah di jelaskan sebelumnya, kombinasi alokasi menggunakan mekanisme menggabungkan *direct block* dan *indirect block*. *Direct block* akan langsung menunjukkan alamat dari blok berkas, dan pada *indirect block* , penunjuk (*pointer*) akan menunjukkan blok indeks seperti dalam mekanisme indeks bertingkat. Penunjuk pada sebuah *single indirect block* akan menunjuk ke blok indeks yang berisi alamat dari blok berkas, penunjuk *double indirect block* akan menunjuk suatu

blok yang bersifat sama dengan blok indeks 2 tingkat, dan *triple indirect block* akan menunjukkan blok indeks tiga tingkat.

Gambar 17.8. Gambar INode pada UNIX File System



17.6. Manajemen Ruang Bebas

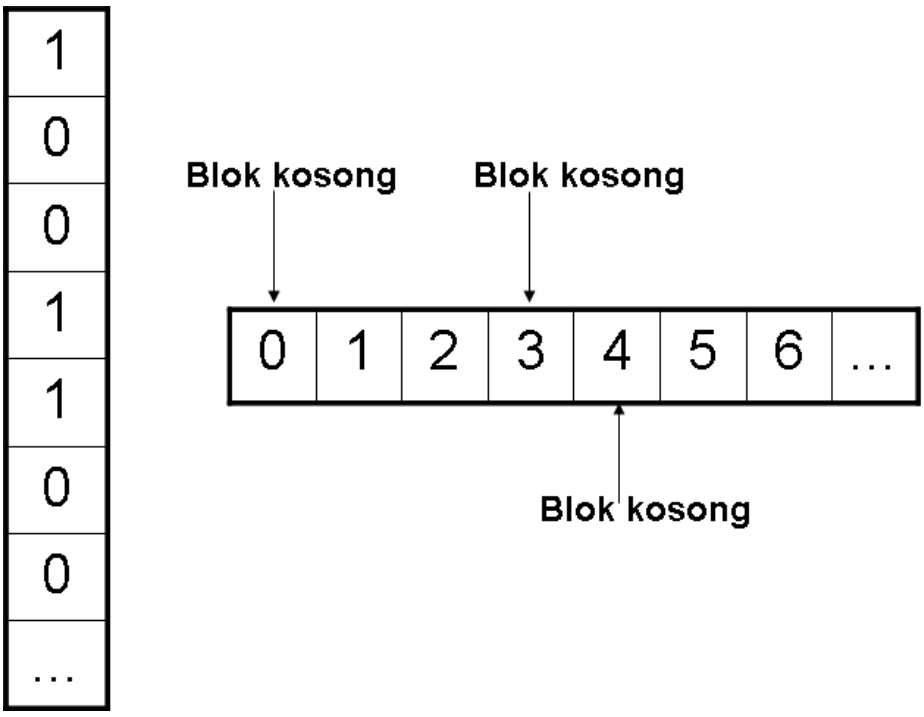
Penyimpanan yang tidak berkesinambungan dan adanya penghapusan data menyebabkan adanya ruang-ruang bebas di disk. oleh karena itu diperlukan manajemen ruang bebas. Caranya, dengan membuat daftar ruang-ruang kosong. Apabila ada berkas baru yang ingin disimpan, maka ruang bebas dicari pada daftar ini.

Ada empat jenis daftar ruang bebas:

Vektor Bit

Blok yang kosong ditandai dengan angka 1 dan blok yang ada isinya ditandai dengan angka 0. Contoh: 0100100, ini berarti blok yang kosong adalah blok ke 1 dan 4.

Gambar 17.9. Gambar Vektor Bit



Perhitungan nomor blok kosong pada vektor bit ini adalah:

(jumlah bit per word)*(jumlah nilai-0 word)+offset dari bit pertama.

Kelemahan dari cara ini adalah pemetaan bit-nya membutuhkan ruang tambahan (blok tersendiri).

Contoh:

ukuran blok = 2^12 byte,

ukuran disk = 2^30 byte (1 gigabyte),

ruang untuk vektor bit=2^30/2^12 bit (atau 32Kbyte).

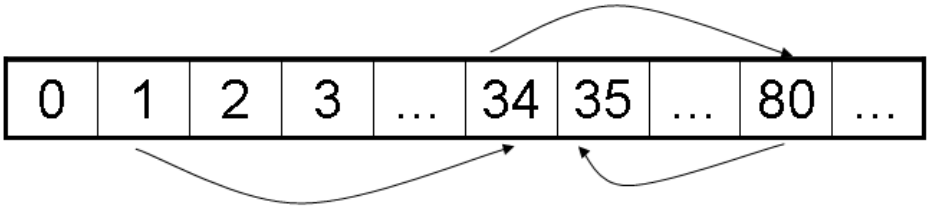
Dengan menggunakan vektor bit bisa terjadi kesalahan dimana bit[i]=1 pada memory dan bit[i]=0 pada disk. Untuk mencegah terjadinya perbedaan ini, maka pada saat pengalokasian suatu ruang kosong untuk suatu berkas dilakukan cara berikut:

- set bit[i]=0 pada disk
- alokasikan blok[i]
- set bit[i]=0 pada memory

Linked-List

Blok kosong pertama *pointer* ke blok kosong kedua, dan blok kosong kedua *pointer* ke blok ketiga yang kosong.

Gambar 17.10. Gambar Linked-List

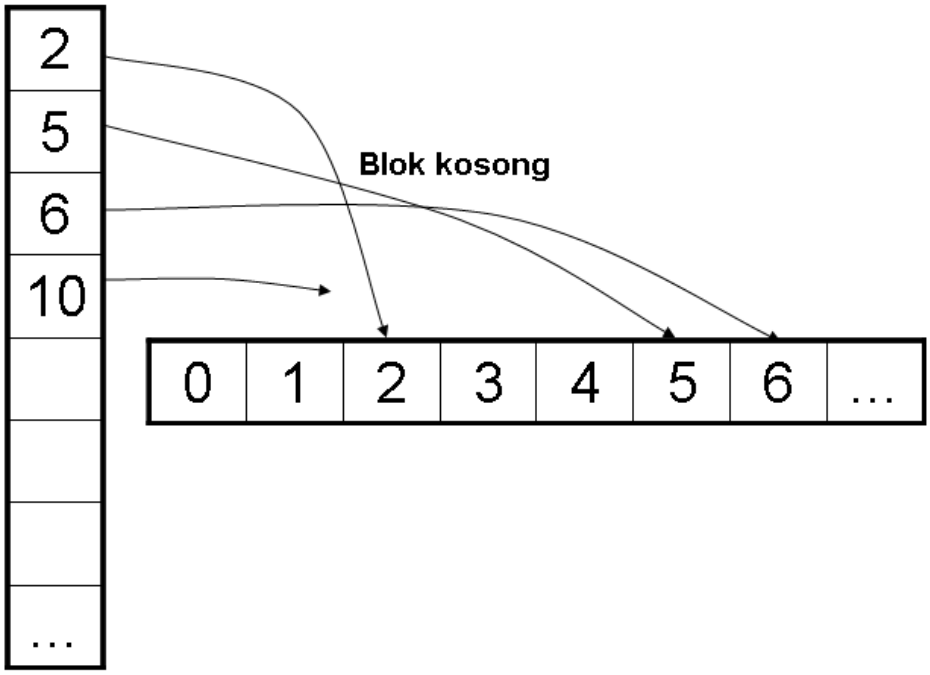


Keunggulan *linked-list* adalah *linked- list* tidak membutuhkan terlalu banyak ruang khusus untuk pointer seperti pada vektor bit yang membutuhkan banyak ruang kosong untuk menyimpan bit-bit yang menyatakan status dari setiap blok. Kelemahannya adalah sulit untuk mendapatkan ruang kosong berurutan dengan mudah.

Pengelompokan

Menggunakan satu blok untuk menyimpan alamat blok-blok kosong di dekatnya. Jika blok telah terisi, maka akan terhapus dari blok alamat kosong.

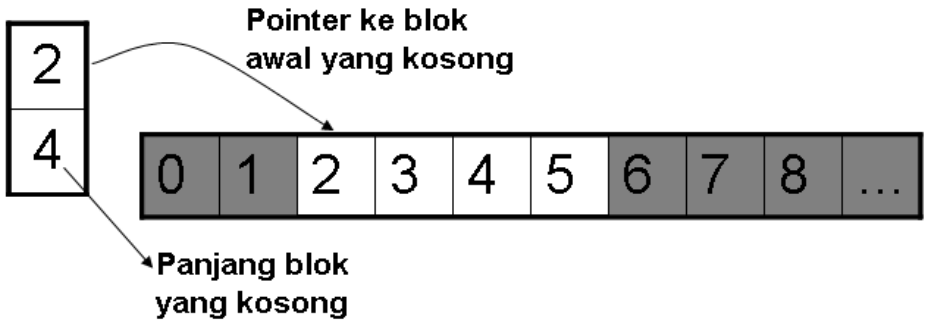
Gambar 17.11. Gambar Pengelompokan



Penghitungan

Ruang kosong list berupa urutan blok-blok kosong, maka dilakukan pendaftaran rangkaian blok kosong dengan memasukkan alamat blok kosong pertama dari rangkaian tersebut, lalu disertakan jumlah blok kosong yang bersebelahan dengannya.

Gambar 17.12. Gambar Penghitungan



17.7. *Backup*

Kita tidak pernah tau apa yang akan terjadi dengan komputer (dalam hal ini disk) kita esok hari. Bisa saja, tiba- tiba terjadi *failure* yang membuat data yang ada dalam disk berubah, bahkan terhapus. Untuk mengantisipasi ketidakonsistenan data dan terhapusya data dari disk, maka kita perlu melakukan *backup*data. *Backup* adalah menyalin isi disk kedalam media lain seperti: *floppy disc*, *magnetic tape*, *optical disk*, *external hardisk*, dll.

Setelah menyalin disk ke media sementara, maka perlu mengembalikan data tersebut ke dalam disk. Hal inilah yang dinamakan *restore*

Sebelum melakukan *backup* data, kita perlu mengecek kekosistenan data, yaitu dengan membandingkan data pada struktur direktori dengan data pada blok, lalu apabila ditemukan kesalahan,

maka program tersebut akan mencoba memperbaikinya. Pengecekan kekonsistenan data inilah yang disebut *recovery*

Ada 4 jenis backup data, yaitu:

*Backup*Penuh *(Full Backup)*

Full backup adalah menyalin semua data termasuk *folder* ke media lain. Oleh karena itu, hasil *full backup* lebih cepat dan mudah saat operasi *restore*. Namun pada saat pembuatannya membutuhkan waktu dan ruang yang sangat besar.

*Backup*Peningkatan *(Incremental Backup)*

Incremental backup adalah menyalin semua data yang berubah sejak terakhir kali melakukan *full backup* atau *differential backup*. *Incremental backup* disebut juga *differential backup*

Kelebihan:
- Membutuhkan waktu yang lebih singkat.
- Jika banyak melakukan *incremental backup*, maka data yang di *backup* semakin kecil ukurannya.
- *Backup* lebih cepat daripada *full backup* dan membutuhkan tempat sementara yang lebih kecil daripada yang dibutuhkan oleh *full backup*.

Kekurangan: Waktu untuk *restore* sangat lama.

Backup Cermin *(Mirror Backup)*

Mirror backup sama dengan *full backup*, tetapi data tidak di padatkan atau dimampatkan (dengan format .tar, .zip, atau yang lain) dan tidak bisa di lindungi dengan *password*. Dapat juga diakses dengan menggunakan *tools* seperti Windows Explorer. *Mirror backup* adalah metode *backup* yang paling cepat bila dibandingkan dengan metode yang lain karena menyalin data dan *folder* ke media tujuan tanpa melakukan pemadatan. Tapi hal itu menyebabkan media penyimpanannya harus cukup besar.

17.8. Rangkuman

Metode pengalokasian blok adalah hal yang penting dalam mengatur penyimpanan berkas pada tempat penyimpanan sekunder (disk).

Terdapat 3 macam metode pengalokasian blok, yaitu: alokasi berkesinambungan, alokasi *link* , alokasi berindeks.

Adapula variasi dari alokasi berkesinambungan, yaitu *extent-based system* serta variasi alokasi *link*, yaitu dengan penggunaan *cluster* dan FAT.

Alokasi berindeks memiliki tiga konsep penting, yaitu: skema terhubung, pengindeksan bertingkat, dan skema gabungan.

Skema gabungan atau kombinasi alokasi yang lebih dikenal dengan nama *inode* pada UNIX File System adalah metode pengalokasian yang mengkombinasikan pengindeksan menggunakan *direct blocks* dan *indirect blocks*.

Manajemen ruang kosong adalah cara mengatur ruangan pada disk yang belum terpakai atau ruang yang kosong akibat penghapusan data. Manajemen ruang kosong dilakukan dengan 4 metode, yaitu vektor bit, *linked-list* , pengelompokan, penghitungan.

Untuk menjaga kekonsistenan dan keamanan data yang disimpan pada disk pada saat terjadi *failure* pada komputer, maka dibutuhkan *backup* data. Yaitu menyimpannya ke alat penyimpanan lain, seperti *floppy disc*, *tape* magnetik, disk optis, dll.

Rujukan

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

[Tanenbaum1997] Andrew Tanenbaum, Albert Woodhull, dan Grag Gagne. 1997. *rOperating Systems Design and Implementation.* . Second Edition. Prentice-Hall..

[WEBBackup] Top Ten Reviews. 2007. *Top Ten Reviews – http://data-backup-software-review.toptenreviews.com/data-backup-definitions.html* . Diakses 26 April 2007.

[WEBBackup4all] Softland. 2007. *backup4all – http://www.backup4all.com/backup-types.php* . Diakses 26 April 2007.

Bab 18. Aneka Aspek Sistem Berkas

18.1. Pendahuluan

Penyimpanan sekunder atau biasa dikenal dengan nama *hard disk* saat ini telah menjadi kebutuhan pokok di kalangan pengguna komputer. Media penyimpanan sekunder ini terus berkembang menjadi media penyimpan yang lebih baik untuk memenuhi kebutuhan manusia akan media penyimpanan. Ada beberapa aspek atau isu yang biasanya berkembang seiring perkembangan *hard disk* . Aspek-aspek tersebut diantaranya berkaitan dengan kinerja dan efisiensi sebuah *hard disk* serta faktor-faktor yang mempengaruhinya. Dalam bab ini, hal-hal tersebut akan dibahas secara lebih mendetail lagi.

Selain itu, dalam bab ini kita juga akan membahas mengenai sistem berkas jaringan atau *Network File System* yang biasa disingkat NFS. Secara umum NFS memungkinkan terjadinya pertukaran sistem berkas antara beberapa *host* atau mesin melalui jaringan. Hal-hal yang berkaitan dengan NFS adalah *mount* NFS dan protokol NFS yang juga akan dibahas dalam bab ini.

18.2. Kinerja

Dalam bahasan kali ini, yang menjadi pokok permasalahan adalah bagaimana cara agar *hard disk* bekerja lebih baik dan lebih cepat dalam pengaksesan seperti pembacaan, penulisan atau operasi-operasi lain pada *hard disk* oleh pengguna. Di bab ini dapat kita lihat bagaimana *allocation method* mempengaruhi kinerja hard disk. Pemilihan *allocation method* yang tepat akan dapat meningkatkan efisiensi dan waktu akses blok data pada hard disk. Oleh karena itu, sangat penting bagi kita untuk memilih *allocation method* yang tepat.

Pada topik yang telah lalu, kita mengetahui ada tiga *allocation method* yaitu:

a. ***Contiguous Allocation.*** ketika mengakses data, metode ini hanya memerlukan satu kali akses untuk mendapatkan blok disk dan mengetahui alamatnya. Selanjutnya dapat mengetahui alamat dari blok selanjutnya atau blok ke-i secara langsung dan dengan cepat. Kekurangannya ialah terjadinya *fragmentation* serta sulit untuk mengatasi apabila ada file yang membesar. Alokasi ini cocok untuk sistem yang mendukung berkas dengan akses langsung.
b. ***Linked Allocation.*** Hampir sama seperti *contiguous allocation* , metode ini dapat langsung mengetahui alamat dari blok selanjutnya, tetapi memerlukan pembacaan disk sebanyak i untuk mendapatkan blok ke-i. Alokasi ini dapat dengan mudah mengatasi file yang membesar. Alokasi ini cocok untuk sistem yang mendukung berkas dengan pengaksesan secara berurutan.
c. ***Indexed Allocation.*** : Metode ini menyimpan index blok di memori. Jika index blok yang dicari ada di memori, maka dapat diakses secara langsung. Alokasi ini sangat bergantung pada struktur indeks, ukuran file, dan posisi blok.

18.3. Efisiensi

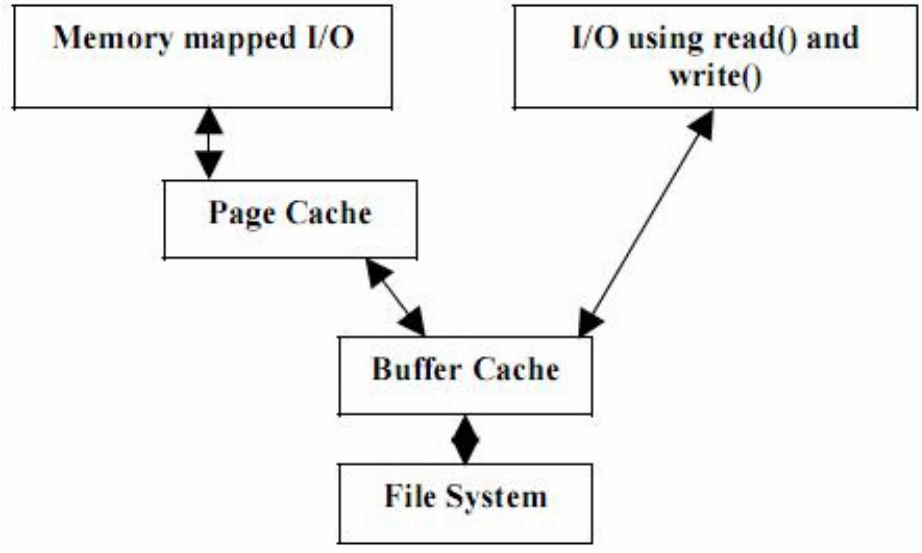
Dalam bagian ini akan dibahas mengenai efisiensi pemanfaatan disk. Tujuannya tentu saja agar disk dapat dimanfaatkan secara maksimal, baik dalam hal pemanfaatan kapasitas maupun kecepatan akses data pada *hard disk*. Efisiensi *hard disk* sangat bergantung pada metode alokasi disk dan algoritma direktori yang digunakan. Contohnya, UNIX mengembangkan kinerjanya dengan mencoba untuk menyimpan sebuah blok data berkas dekat dengan blok i-node berkas untuk mengurangi waktu perncarian.

Efisiensi juga bergantung pada tipe data yang disimpan pada masukan direktori (atau i-node). Pada umumnya, sistem menyimpan informasi mengenai tanggal terakhir kali file diakses dan tanggal penulisan terakhir untuk menentukan apakah file perlu di *back up*. Akibat dari penyimpanan informasi ini, setiap kali berkas dibaca diperlukan penulisan pada struktur direktori. Penulisan ini memerlukan blok untuk dibaca oleh memori, perubahan bagian, dan blok yang akan ditulis kembali ke disk. Hal ini tidak efisien untuk tipe berkas yang sering diakses atau untuk berkas yang diakses secara berkala.

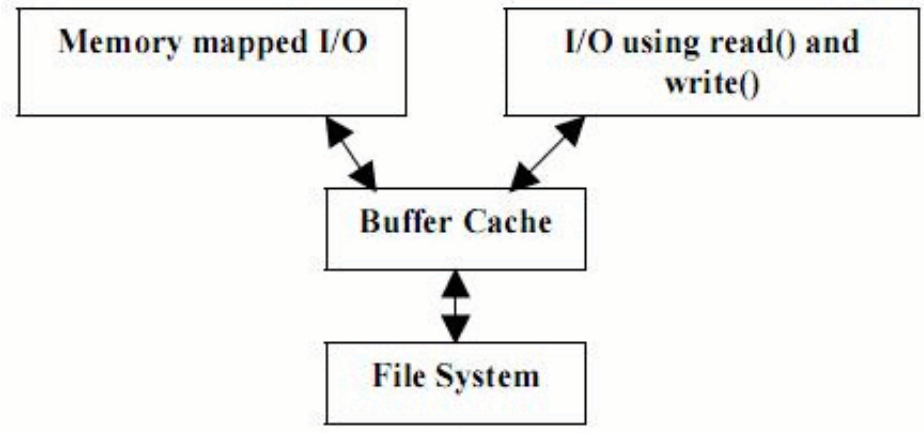
Kinerja yang dibahas pada bagian ini merupakan kelanjutan kinerja pada poin sebelumnya. Apabila algoritma yang tepat telah dipilih, kinerja masih bisa ditingkatkan lagi dengan memanfaatkan *cache*. Beberapa sistem membuat bagian yang terpisah dari memori utama untuk digunakan sebagai *disk cache*, dimana blok-blok disimpan dengan asumsi mereka akan digunakan lagi secepatnya. Sistem lainnya menyimpan data berkas menggunakan sebuah *page cache* . *Page cache* tersebut menggunakan teknik memori virtual dan menyimpan data berkas sebagai halaman-halaman, tidak sebagai blok-blok *file-system-oriented*. Hal ini dikarenakan menyimpan data berkas menggunakan alamat virtual jauh lebih efisien daripada menyimpannya melalui blok disk fisik.

Ada dua buah teknik penggunaan *cache* yang biasanya dipakai. Pertama yaitu teknik menggunakan *unified buffer cache*. Kedua yaitu dengan teknik tanpa menggunakan *unified buffer cache*. Berikut adalah ilustrasi gambar dari kedua teknik tersebut.

Gambar 18.1. Menggunakan *Unified buffer cache*



Gambar 18.2. Tanpa *Unified buffer cache*



Gambar 18.1 adalah ilustrasi teknik dengan menggunakan *unified buffer cache* dan gambar 18.2 adalah ilustrasi teknik tanpa menggunakan *unified buffer cache*. Pada gambar 18.2, jika terjadi panggilan *mapping* memori, dibutuhkan dua buah *cache*, yaitu *buffer cache* dan *page cache* . Hal ini dikarenakan pada teknik ini, *mapping* memori tidak bisa langsung berinteraksi dengan *buffer cache*, sehingga dibutuhkan *page cache*.Teknik ini sangat tidak efisien karena terjadi dua kali penyalinan berkas yang sama. Pertama pada *buffer cache*dan kedua *page cache*. Dua kali penyalinan ini biasa disebut *double caching*. *Double caching* ini tidak hanya boros memori, tetapi juga memboroskan kinerja CPU dan perputaran M/K dikarenakan perubahan data ekstra antara memori sistem. Selain itu, *double caching*

juga dapat menyebabkan korupsi berkas. Solusi untuk masalah ini adalah menggunakan *unified buffer cache* seperti pada gambar 18.1. *Buffer cache* pada teknik ini dapat berinteraksi dengan *mapping* memori sehingga dapat menghindari terjadinya *double caching*.

18.4. Struktur Log Sistem Berkas

Pemanggilan kembali struktur data sistem berkas pada disk seperti struktur-struktur direktori, penunjuk blok kosong, dan penunjuk FCB kosong, dapat menjadi tidak konsisten dikarenakan adanya sistem *crash* . Sebelum penggunaan dari teknik *log-based* di sistem operasi, perubahan biasanya digunakan pada struktur ini. Perubahan-perubahan tersebut dapat diinterupsi oleh *crash*, dengan hasil strukturnya tidak konsisten.

Ada beberapa masalah yang terjadi akibat adanya pendekatan dari menunjuk struktur untuk memecahkan dan memperbaikinya pada recovery. Salah satunya adalah ketidakkonsistenan yang tidak dapat diperbaiki. Pemeriksaan rutin mungkin tidak dapat dipakai untuk me *recover* struktur tersebut, sehingga terjadi kehilangan berkas dan mungkin seluruh direktori.

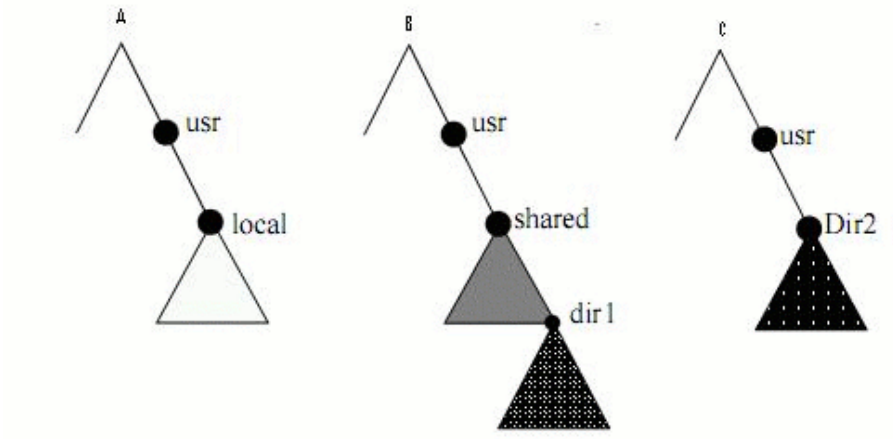
Solusinya adalah memakai teknik *log-based-recovery* seperti pada sistem berkas metadata yang terbaru. Pada dasarnya, semua perubahan metadata ditulis secara berurutan di sebuah log. Masing-masing set dari operasi-operasi yang menampilkan tugas yang spesifik adalah sebuah *transaction* . Jika sistemnya mengalami *crash* , tidak akan ada kelebihan *transactions* di berkas log. *Transactions* tersebut tidak akan pernah ditulis secara lengkap ke sistem berkas walaupun dimasukkan oleh sistem operasi sehingga harus dilengkapi. Keuntungan yang lain adalah proses-proses pembaharuan akan berlangsung lebih cepat daripada saat dipakai langsung ke struktur data pada disk.

18.5. NFS

Network File System atau sistem berkas jaringan adalah sekumpulan protokol yang digunakan untuk mengakses beberapa sistem berkas melalui jaringan. Bisa dikatakan juga bahwa NFS merupakan sebuah implementasi dan spesifikasi dari sebuah perangkat lunak untuk mengakses *remote file* melalui jaringan LAN atau WAN.

NFS yang dikembangkan oleh *Sun Micro Systems Inc.* ini menggambarkan himpunan unit-unit komputer yang saling berhubungan sebagai sebuah mesin bebas yang memiliki sistem berkas bebas. Tujuan dari NFS adalah untuk memungkinkan terjadinya pertukaran sistem berkas secara transparan antara mesin-mesin bebas tersebut. Hubungan yang terjadi di sini didasarkan pada hubungan *client-server* yang menggunakan perangkat lunak *NFS server* dan *NFS client* yang berjalan diatas *workstation*. Gambar 18.3 berikut ini menggambarkan tiga buah mesin bebas yang memiliki sistem berkas lokal masing-masing yang bebas juga.

Gambar 18.3. *Three Independent File System*



NFS didesain agar dapat beroperasi di lingkungan ataupun jaringan yang heterogen yang meliputi mesin, platform, sistem operasi, dan arsitektur jaringan. Ketidaktergantungan ini didapat dari penggunaan RPC primitif yang dibangun diatas protokol *External Data Representation* (XDR).

Jika misalnya terjadi sebuah pertukaran sistem berkas antara *server* dan *client* , maka pertukaran sistem berkas yang terjadi disini harus dipastikan hanya berpengaruh pada tingkat *client* dan tidak mempengaruhi sisi *server* , karena server dan client adalah mesin yang berbeda dan sama-sama bebas. Untuk itu, mesin *client* harus melakukan operasi *mount* terlebih dahulu agar *remote directory* dapat diakses secara transparan.

18.6. *Mount* NFS

Protokol *mount* membangun hubungan yang logis antara *client* dan *server*. Informasi penting yang didapat saat operasi *mount* dilakukan adalah nama dari direktori dan nama dari *server* yang menyimpan direktori tersebut. Hal ini menyebabkan *client* harus melakukan operasi *mount* pada *server* dan direktori yang tepat.

Ketika *client* melakukan operasi *mount* ke sebuah *server*, maka *server* akan mengirimkan sebuah *file handle* kepada *client*. *File handle* ini berfungsi sebagai kunci bagi *client* untuk melakukan pengaksesan lebih lanjut terhadap direktori yang telah di*mount* tadi.

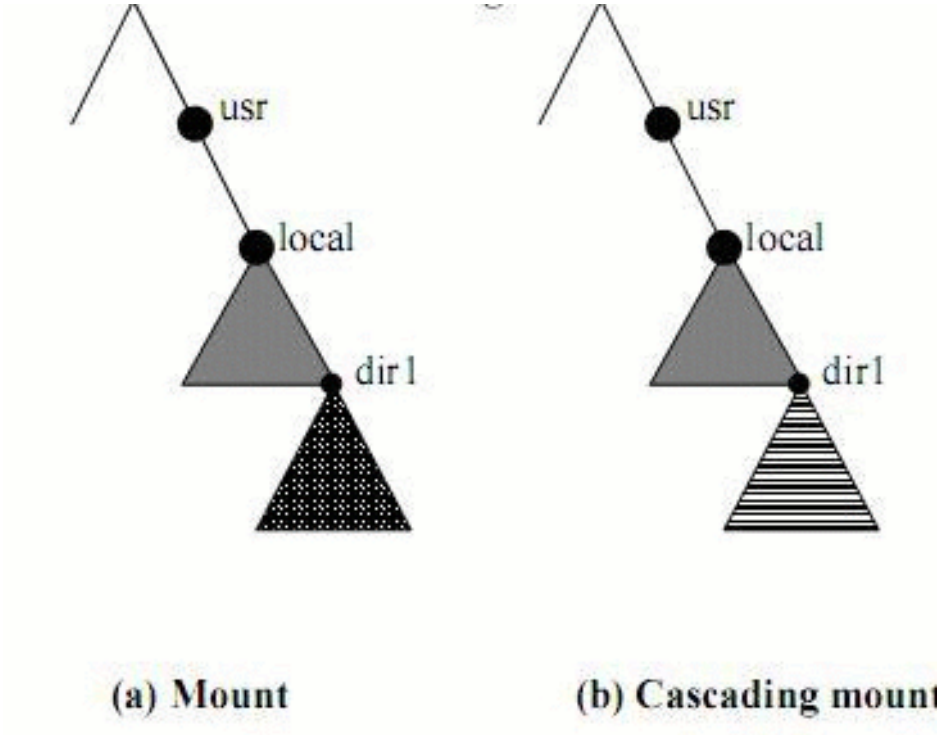
Sebagai contoh, kita akan melakukan operasi mount seperti pada gambar 18.3 (*Three Independent File System*) antara sistem berkas B/usr/shared terhadap A/usr/local (direktori *shared* pada mesin B di*mount* ke di direktori *local* pada mesin A). Kedua sistem berkas tersebut bersifat bebas dan lokal.

Seperti yang telah kita pelajari pada bahasan sebelumnya (Bab 16), *mounting* adalah memasukkan sistem berkas ke struktur direktori utama, baik ke dalam direktori kosong maupun ke dalam direktori yang sudah berisi. Isi dari direktori itu tidak bisa diakses hanya kalau dimasukkan ke direktori yang berisi. 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. Hasil operasi *mount* B/usr/shared terhadap A/usr/local pada Gambar 18.3 dapat kita lihat pada Gambar 18.4 (a).

Contoh *mount* yang baru saja kita lakukan tadi adalah contoh operasi *mount* antara dua sistem berkas lokal yang bebas. Sedangkan kondisi ketika kita ingin melakukan operasi *mount* terhadap suatu sistem berkas pada suatu sistem berkas lain yang secara *remote* telah di*mount* oleh sistem berkas lain disebut dengan istilah *cascading mount*.

Sebagai contoh, kita akan melakukan operasi *mount* antara sistem berkas C/usr/dir2 pada gambar 18.3 terhadap sistem berkas yang sebelumnya telah di*mount* , yaitu sistem berkas pada Gambar 18.4 (a) di direktori *dir1*. Maka hasilnya akan sama seperti proses *mount* sebelumnya. Hasil operasinya dapat kita lihat pada gambar 18.4 (b).

Gambar 18.4. ***Mounting in NFS***



18.7. Protokol NFS

NFS umumnya menggunakan protokol *Remote Procedure Call* (RPC) yang berjalan di atas UDP dan membuka *port* UDP dengan *port number* 2049 untuk komunikasi antara *client* dan *server* di dalam jaringan. *Client* NFS selanjutnya akan mengimpor sistem berkas *remote* dari *server* NFS, sementara *server* NFS mengekspor sistem berkas lokal kepada *client*.

Mesin-mesin yang menjalankan perangkat lunak NFS *client* dapat saling berhubungan dengan perangkat lunak NFS *server* untuk melakukan perintah operasi tertentu dengan menggunakan *request* RPC. Adapun operasi-operasi yang didukung oleh NFS adalah sebagai berikut:

a. Mencari berkas di dalam direktori.
b. Membaca kumpulan direktori.
c. Memanipulasi link dan direktori.
d. Mengakses atribut berkas.
e. Membaca dan menulis berkas.

Perlu diketahui bahwa server NFS bersifat *stateless* , yang artinya setiap *request* harus mengandung argumen yang penuh dan jelas sebab server NFS tidak menyimpan sejarah informasi *request* . Data yang dimodifikasi harus di *commit* ke *server* sebelum hasilnya di kembalikan ke *client* . NFS protokol tidak menyediakan mekanisme *concurrency-control*.

18.8. Rangkuman

Sistem berkas tersimpan secara permanen di *secondary storage* yang didesain untuk menyimpan data yang besar secara permanen. *Secondary srorage* yang cukup kita kenal adalah disk.

Perlu diingat bahwa metode alokasi ruang kosong dan algoritma direktori yang digunakan akan mempengaruhi efisiensi dari penggunaan ruang disk serta kinerja sistem berkas.

Network file system atau sistem berkas jaringan menggunakan metodologi *client-server* untuk memungkinkan *user* mengakses sistem berkas dan direktori pada mesin *remote*. Perintah-perintah

dari *client* diterjemahkan ke dalam protokol jaringan, dan diterjemahkan kembali ke dalam operasi-operasi sistem berkas pada *server*. Jaringan dan banyaknya user yang mengakses suatu sistem berkas memungkinkan terjadinya inkonsistensi data dan kinerja.

Sebelum *client* dapat mengakses sistem berkas pada *server*, terlebih dahulu perlu dibuat sebuah koneksi logis antara *client* dan *server* yang disebut dengan *mount* NFS. Ketika *client* telah melakukan operasi *mount* pada suatu *server* , maka *server* akan mengirimkan sebuah *file handle* kepada *client*.

NFS menggunakan RPC yang berjalan di atas UDP. RPC ini memungkinkan client untuk melakukan operasi-operasi tertentu. Adapun operasi-operasi yang didukung adalah:

a. Mencari berkas di dalam direktori.
b. Membaca kumpulan direktori.
c. Memanipulasi link dan direktori.
d. Mengakses atribut berkas.
e. Membaca dan menulis berkas.

Rujukan

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

[WEB2007] 2007. *Wikipedia Indonesia http://id.wikipedia.org/wiki/NFS* . Diakses 5 Mei 2007.

Bab 19. Media Disk

19.1. Pendahuluan

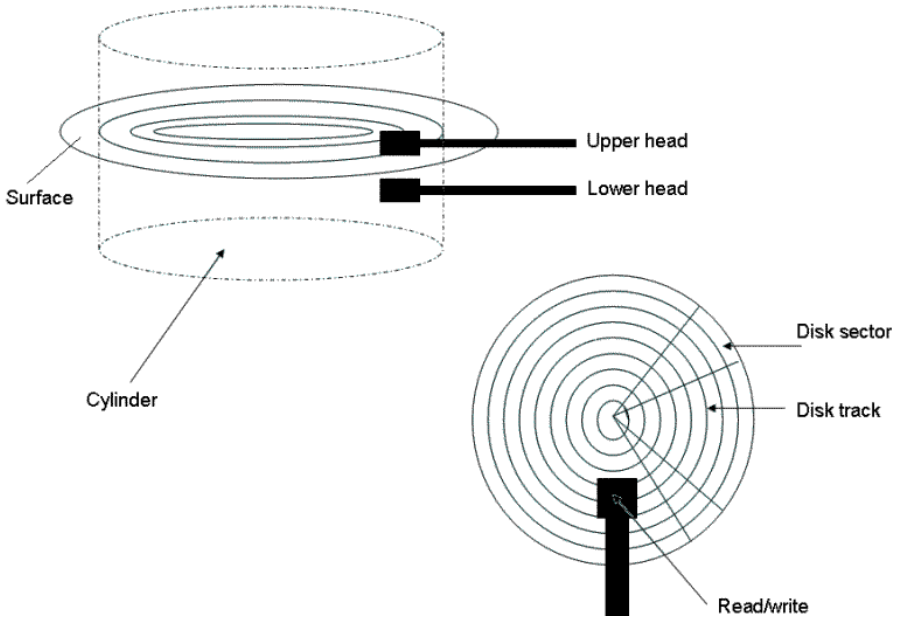
Struktur disk merupakan suatu hal yang penting bagi penyimpanan informasi. Sistem komputer modern menggunakan disk sebagai media penyimpanan sekunder. Dulu pita magnetik, yang memiliki waktu akses lebih lambat dari pada disk,digunakan sebagai media penyimpanan sekunder. Sejak digunakannya disk, *tape* digunakan untuk *back-up*, 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* satu dimensi yang besar dari blok logika. Blok logika merupakan satuan unit terkecil dari transfer. Ukuran blok logika umumnya sebesar 512 *bytes* walaupun disk dapat diformat ke level rendah (*low level formatted*) sehingga ukuran blok logika dapat ditentukan, misalnya 1024 *bytes*. *Array* satu dimensi dari blok logika tersebut dipetakan ke sektor dalam disk secara sekuensial. Sektor 0 adalah sektor pertama dari *track* pertama dari 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 yang terluar.

19.2. Struktur Disk

Seperti yang telah dikatakan sebelumnya, penulisan *disk drive* modern adalah dengan menggunakan *array* satu dimensi blok logika yang besar. Dengan menggunakan sistem pemetaan ini, Secara tiori setidaknya kita dapat mengkonversikan sebuah *logical block number* ke penulisan disk gaya lama yang berisi nomor silinder, nomor *track* di silinder, dan nomor sektor di dalam *track*. Dalam praktiknya, sangatlah sulit untuk melakukan hal ini. Ada dua alasan, yang pertama adalah kebanyakan disk memiliki beberapa sektor yang tidak sempurna, tapi pemetaan menutupi dengan cara menggantikannya dengan sektor tambahan dari tempat lain di disk. Alasan kedua adalah jumlah sektor tiap *track* berbeda pada beberapa disk.

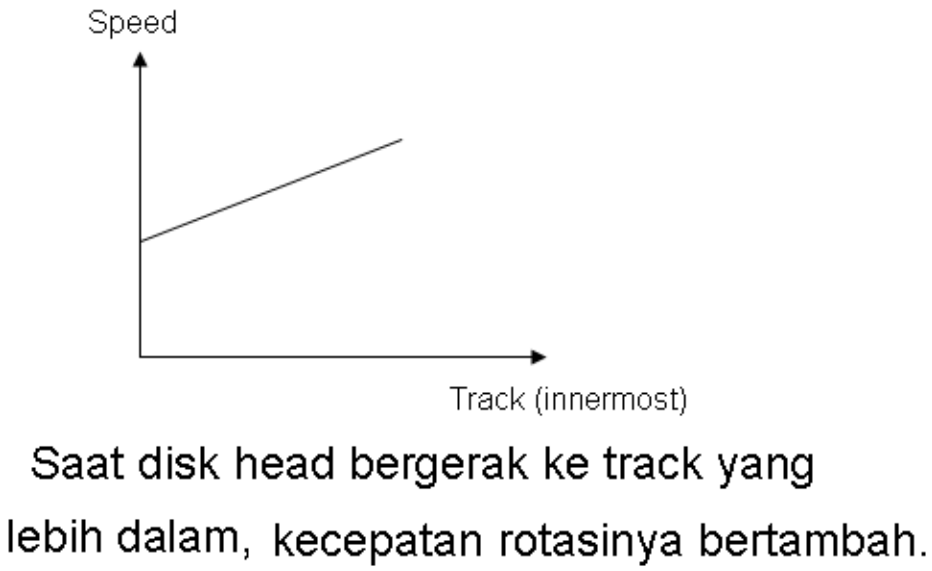
Gambar 19.1. Struktur Disk *array*



Mari kita lihat lebih dekat pada alasan kedua. Dalam media yang menggunakan *Constant Linear Velocity* (CLV), jumlah bit tiap *track* adalah sama. Semakin jauh posisi *track* dari pusat disk, jaraknya

semakin besar, sehingga semakin banyak sektor yang ada. Saat kita bergerak dari zona yang lebih luar ke zona yang lebih dalam, jumlah sektor tiap *track* menurun. *Track* di zona yang lebih luar biasanya memiliki sektor lebih banyak 40% dari *track* di zona yang lebih dalam. *Drive* meningkatkan kecepatan rotasinya saat *head* bergerak dari *track* terluar sampai *track* yang lebih dalam untuk mempertahankan kecepatan perpindahan data dibawah *head*. Metode ini digunakan pada *drive* CD-ROM dan DVD-ROM.

Gambar 19.2. CLV



Alternatif dari metode ini, kecepatan rotasi disk bisa tetap, dan jumlah (kepadatan) bit menurun dari *track* yang lebih dalam ke *track* yang lebih luar untuk mempertahankan kecepatan data tetap konstan. Metode ini digunakan dalam *hard disk* dan dikenal sebagai *Constant Angular Velocity* (CAV). Keuntungan menggunakan metode CAV adalah sebuah data bisa langsung dipetakan sesuai pada *track* dan nomor silinder yang diminta. Tetapi metode ini juga memiliki kelemahan, yaitu jumlah data yang bisa di simpan pada *track* terluar dan terdalam sama, padahal kita tahu bahwa panjang *track* bagian luar lebih panjang daripada *track* bagian dalam.

Gambar 19.3. CAV *array*

Jumlah sektor per *track* telah semakin berkembang sesuai dengan perkembangan teknologi disk, dan bagian terluar dari sebuah disk biasanya memiliki beberapa ratus sektor per *track*. Begitu pula jumlah silinder per disk semakin bertambah. Sebuah disk ukuran besar bisa memiliki puluhan ribu silinder.

19.3. HAS

Host-Attached Storage (HAS) adalah pengaksesan *storage* melalui *port* M/K lokal. *Port-port* ini menggunakan beberapa teknologi. PC biasanya menggunakan sebuah arsitektur bus M/K yang bernama IDE atau ATA. Arsitektur ini mendukung maksimal 2 *drive* per M/K bus. Arsitektur yang lebih baru yang menggunakan *simplified cabling* adalah SATA. *High-end workstation* dan server biasanya menggunakan arsitektur M/K yang lebih rumit, seperti SCSI atau *fiber channel* (FC).

SCSI adalah sebuah arsitektur bus. Medium fisiknya biasanya adalah kabel *ribbon* yang memiliki jumlah konduktor yang banyak (biasanya 50 atau 68). Protokol SCSI mendukung maksimal 16 *device* dalam bus. Biasanya, *device* tersebut termasuk sebuah *controller card* dalam *host* (SCSI *initiator*, yang meminta operasi) dan sampai 15 *storage device* (SCSI *target*, yang menjalankan perintah). Sebuah SCSI disk adalah sebuah SCSI *target* yang biasa, tapi protokolnya menyediakan kemampuan untuk menuliskan sampai 8 *logical unit* pada setiap SCSI *target*. Penggunaan *logical unit addressing* biasanya adalah perintah langsung pada komponen dari *array* RAID atau komponen dari *removable media library* .

FC adalah sebuah arsitektur seri berkecepatan tinggi yang dapat beroperasi pada serat optik atau pada kabel *copper* 4-konduktor. FC mempunyai dua varian. Pertama adalah sebuah *switched fabric* besar yang mempunyai 24-bit *space* alamat. Varian ini diharapkan dapat mendominasi di masa depan dan merupakan dasar dari *storage-area network* (SAN). Karena besarnya *space* alamat dan sifat *switched* dari komunikasi, banyak *host* dan *device* penyimpanan dapat di-*attach* pada *fabric*, memungkinkan fleksibilitas yang tinggi dalam komunikasi M/K. Varian FC lain adalah *abritrated loop* (FC-AL) yang bisa menuliskan 126 *device* (*drive* dan *controller*).

Banyak variasi dari *device* penyimpanan yang cocok untuk digunakan sebagai HAS. Beberapa diantaranya adalah *hard disk*, RAID *array*, serta *drive* CD, DVD dan *tape*. Perintah M/K yang

menginisiasikan transfer data ke HAS *device* adalah membaca dan menulis *logical data block* yang diarahkan ke unit penyimpanan teridentifikasi yang spesifik (seperti bus ID, SCSI ID, dan *target logical unit*).

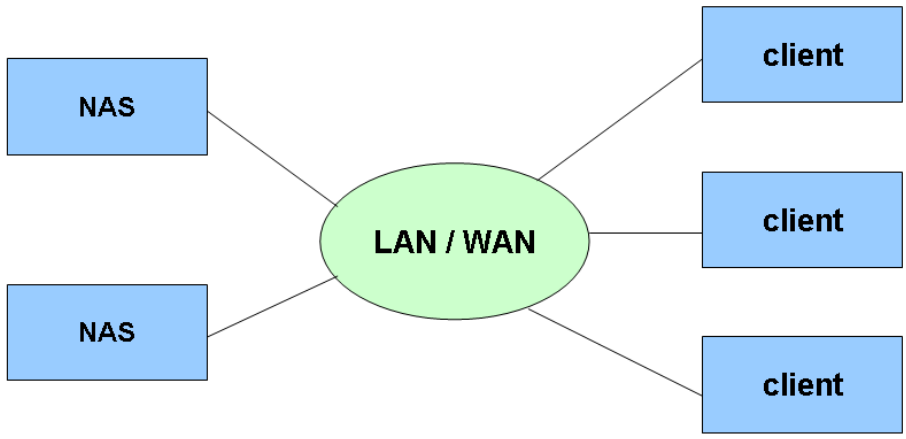
19.4. NAS dan SAN

Network-Attached Storage (NAS) *device* adalah sebuah sistem penyimpanan yang mempunyai tujuan khusus yaitu untuk diakses dari jauh melalui *data network*. Klien mengakses NAS melalui RPC (*remote-procedure-call*) seperti NFS untuk UNIX atau CIFS untuk Windows. RPC dibawa melalui TCP atau UDP (*User Datagram Protocol*) dari IP *network* biasanya dalam *local-area network* (LAN) yang sama dengan yang membawa semua lalu lintas data ke klien. Unit NAS biasanya diimplementasikan sebagai sebuah RAID *array* dengan *software* yang mengimplementasikan *interface* RPC.

NAS menyediakan jalan yang cocok untuk setiap komputer dalam sebuah LAN untuk saling berbagi *pool* penyimpanan dengan kemudahan yang sama seperti menamai dan menikmati akses seperti HAS lokal. Umumnya cenderung untuk lebih tidak efisien dan memiliki peforma yang lebih buruk dari penyimpanan *direct-attached*.

ISCSI adalah protokol NAS terbaru. Protokol ini menggunakan protokol IP *network* untuk membawa protokol SCSI. *Host* dapat memperlakukan penyimpanannya seperti *direct-attached*, tapi *storage*-nya sendiri dapat berada jauh dari *host*.

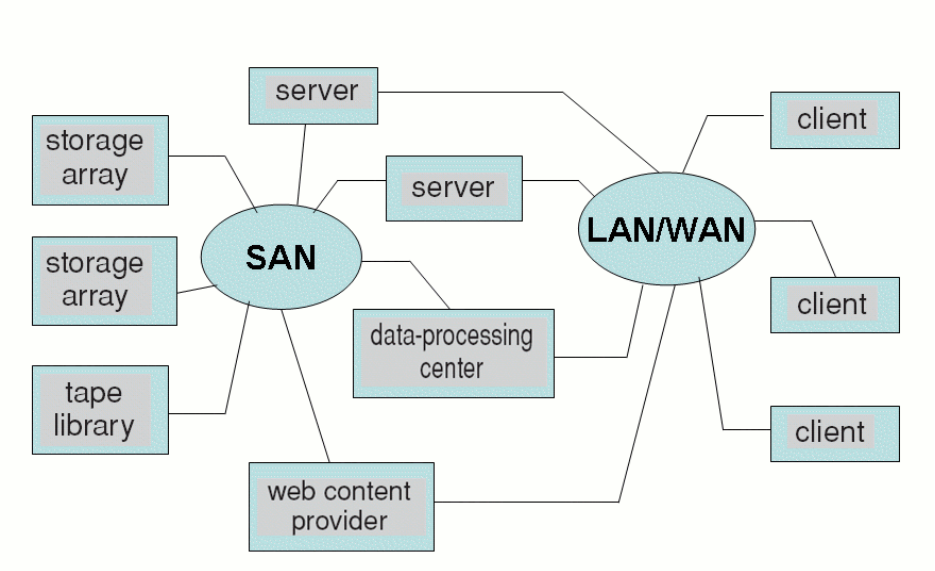
Gambar 19.4. ***Network-Attached Storage***



Storage-area Network (SAN) adalah *network private* (menggunakan protokol *storage* daripada protokol *network*) yang menghubungkan server dan unit penyimpanan. Keunggulan SAN terletak pada fleksibilitasnya. Sejumlah *host* dan *storage array* dapat di *attach* ke SAN yang sama, dan *storage* dapat dialokasikan secara dinamis pada *host*. Sebuah SAN *switch* mengizinkan atau melarang akses antara *host* dan *storage*. Sebagai contoh, apabila *host* kehabisan *disk space*, maka SAN dapat mengalokasikan *storage* lebih banyak pada *host* tersebut.

SAN memungkinkan *cluster server* untuk berbagi *storage* yang sama dan memungkinkan *storage array* untuk memasukkan beberapa koneksi *host* langsung. SAN biasanya memiliki jumlah *port* yang lebih banyak, dan *port* yang lebih murah, dibandingkan *storage array* . FC adalah interkoneksi SAN yang paling umum.

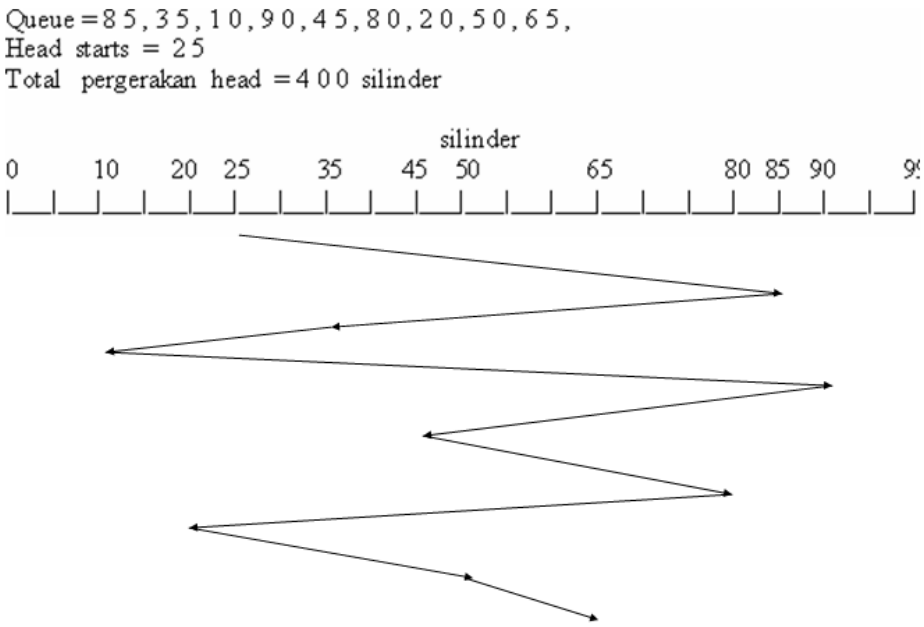
Gambar 19.5. *Storage Area Network*



19.5. Penjadwalan FCFS

Bentuk algoritma penjadwalan disk yang paling sederhana adalah *First Come First Served (FCFS)*. Sistem kerja dari algoritma ini 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 bukan merupakan algoritma dengan layanan yang tercepat. Sebagai contoh, misalnya di *queue* disk terdapat antrian permintaan blok M/K di silinder

Gambar 19.6. *FCFS*



85, 35, 10, 90, 45, 80, 20, 50, 65

secara berurutan. Jika posisi *head* awal berada pada 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 *head*-nya adalah 400 silinder.

Untuk lebih jelasnya perhatikan contoh berikut.

Tabel 19.1. Contoh FCFS

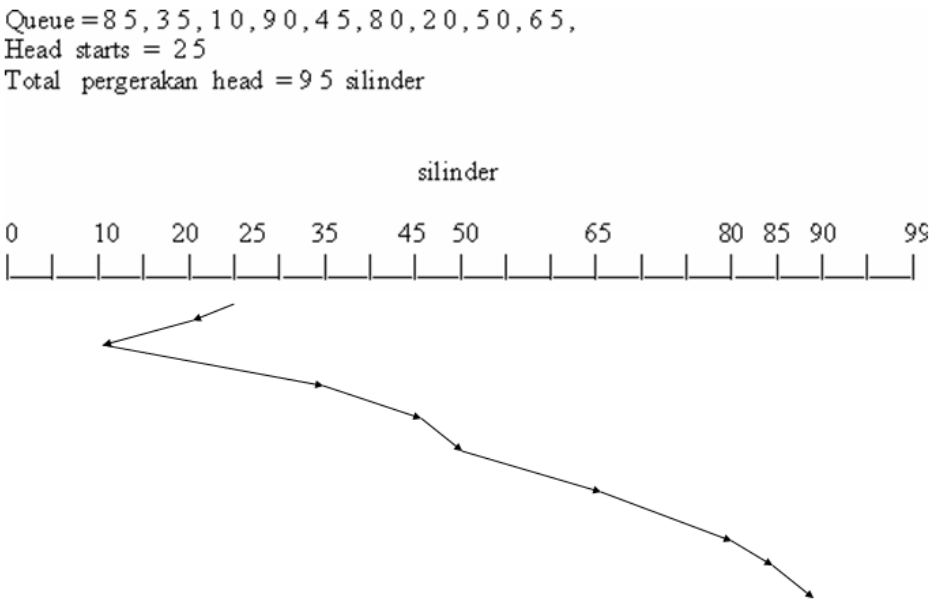
Next cylinder accessed	Number of cylinder traversed
85	60
35	45
10	25
90	80
45	45
80	35
20	60
50	30
65	15
Total pergerakan head	400 silinder

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.

19.6. Penjadwalan SSTF

Shortest-Seek-Time-First (SSTF) merupakan algoritma yang melayani permintaan berdasarkan waktu pencarian yang 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.

Gambar 19.7. *SSTF*



Untuk contoh permintaan di *queue* kita, permintaan yang terdekat dari *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 silinder berikutnya yang dilayani adalah silinder 35, lalu 45, 50, 65, 80, 85, dan akhirnya silinder 90. dengan menggunakan

algoritma ini, maka total pergerakan *head*-nya menjadi 95 silinder. Hasil yang didapat ternyata kurang dari seperempat jarak yang dihasilkan oleh penjadwalan FCFS.

Untuk lebih jelasnya perhatikan contoh berikut:

Tabel 19.2. Contoh SSTF

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 dilayani semuanya. Walaupun SSTF memberikan waktu pelayanan yang lebih cepat namun apabila dilihat dari sudut pandang keadilan bagi permintaan yang menungu di *queue*, jelas algoritma ini lebih buruk dibandingkan FCFS *scheduling*.

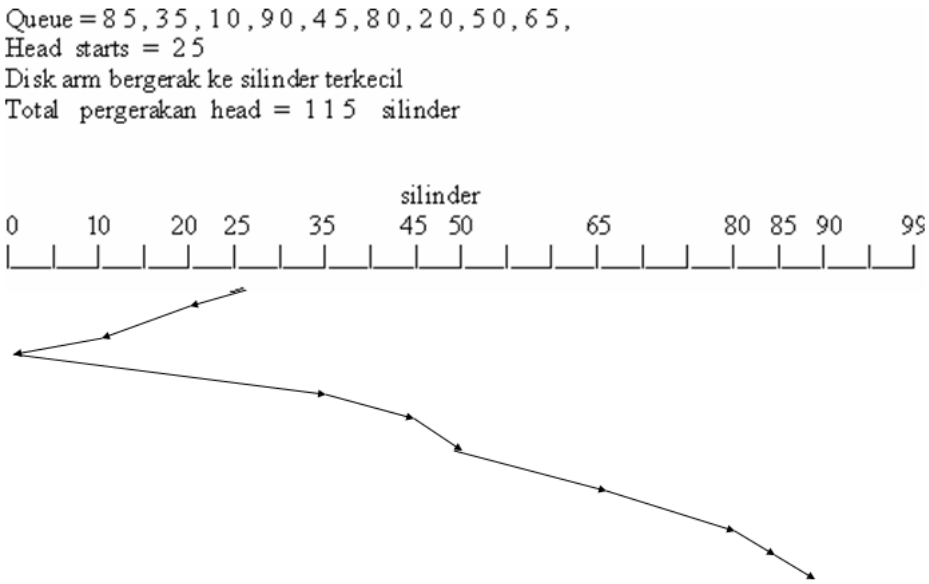
19.7. Penjadwalan SCAN dan C-SCAN

Pada algoritma SCAN, *head* bergerak ke silinder paling ujung dari disk. Setelah sampai disana 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 penggunanya 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 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 akan dilayani secara berurutan adalah 20 dan 10 lalu menuju ke silinder 0. Setelah sampai disini *head* akan berbalik arah menuju silinder yang terbesar yaitu silinder 99. Dalam pergerakannya itu secara berurutan *head* akan melayani permintaan 35, 45, 50, 65, 80, 85, dan 90. Sehingga total pergerakan *head*-nya adalah 115 silinder.

Salah satu *behavior* yang dimiliki oleh algoritma ini adalah, memiliki batas atas untuk total pergerakan *head*-nya, yaitu 2 kali jumlah silinder yang dimiliki oleh disk. Jika dilihat dari cara kerjanya yang selalu menuju ke silinder terujung, maka dapat dilihat kelemahan dari algoritma ini yaitu ketidakefisienannya. 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 algoritma LOOK yang akan dibahas di sub-bab berikutnya.

Gambar 19.8. *SCAN*



Untuk lebih jelasnya perhatikan contoh berikut:

Tabel 19.3. Contoh SCAN

Next cylinder accessed	**Number of cylinder 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

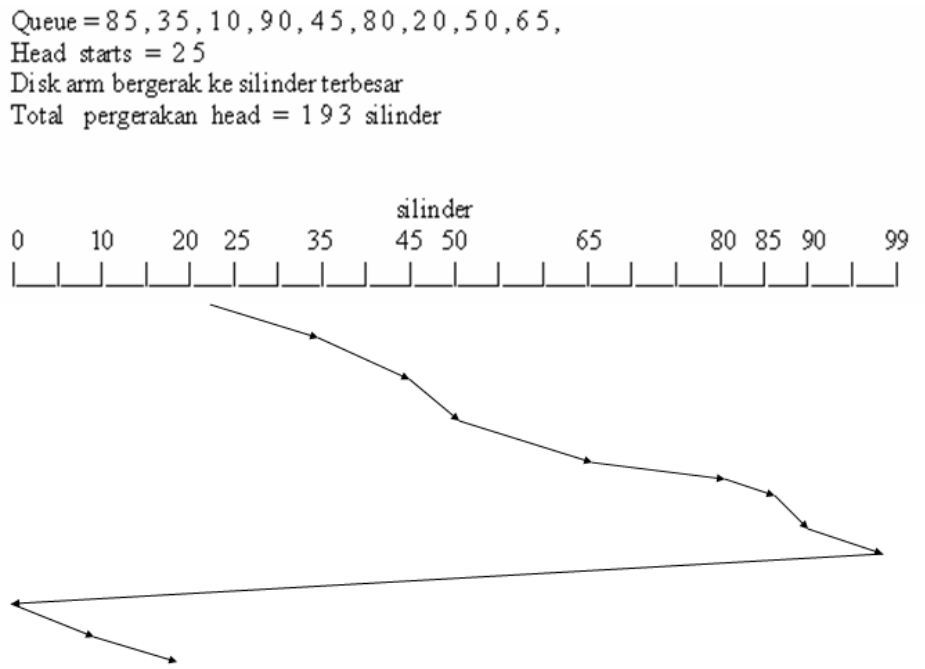
Kelemahan lain dari algoritma SCAN adalah dapat menyebabkan permintaan lama menunggu pada kondisi-kondisi tertentu. Misalkan penyebaran banyaknya permintaan yang ada di *queue* tidak sama. Permintaan yang berada di depan arah pergerakan *head* sedikit sedangkan yang berada di ujung satunya lebih banyak. 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.

Algoritma C-SCAN atau **Circular SCAN** merupakan hasil modifikasi dari SCAN untuk mengurangi kemungkinan banyak permintaan yang 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 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 akan tetap melayani permintaan saat bergerak berbalik arah menuju ujung yang lain.

Untuk contoh permintaan 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 silinder 20 dan 10. Permintaan tersebut baru dilayani ketika *head* sudah berbalik arah lagi setelah mencapai silinder 0.

Gambar 19.9. *C-SCAN*



Untuk lebih jelasnya perhatikan contoh berikut:

Tabel 19.4. Contoh C-SCAN

Next cylinder accessed	Number of cylinder traversed
35	10
45	10
50	5
65	15
80	15
85	5
90	5
99	9
0	99
10	10
20	10
Total pergerakan head	193 silinder

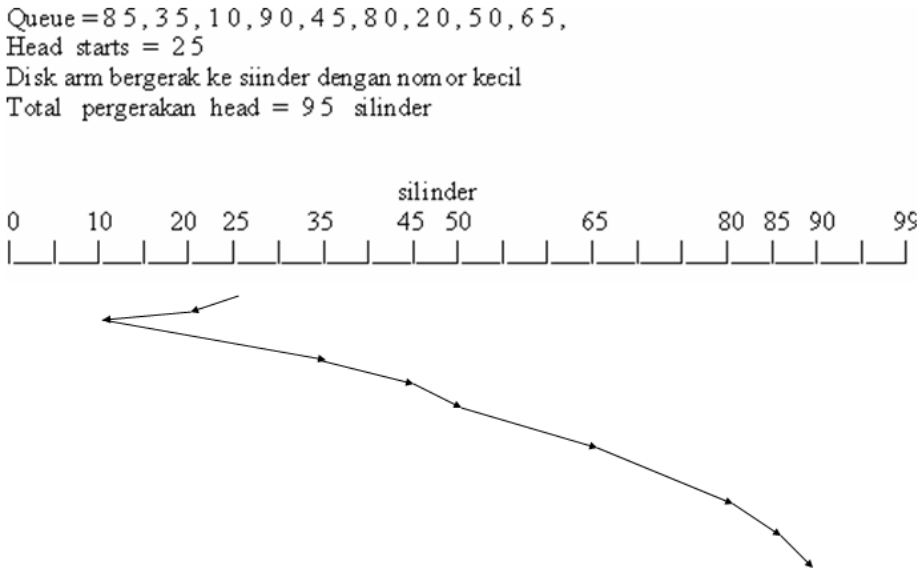
Dengan sistem kerja yang seperti itu, terlihat bahwa *head* melayani permintaan hanya dalam satu arah pergerakan saja, yaitu saat *head* bergerak ke silinder terbesar atau saat bergerak ke silinder terkecil.

sifatnya yang harus sampai ke 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.

19.8. Penjadwalan LOOK dan C-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 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 di depannya sesuai dengan arah pergerakannya.

Gambar 19.10. *LOOK*



Untuk lebih jelasnya perhatikan contoh berikut:

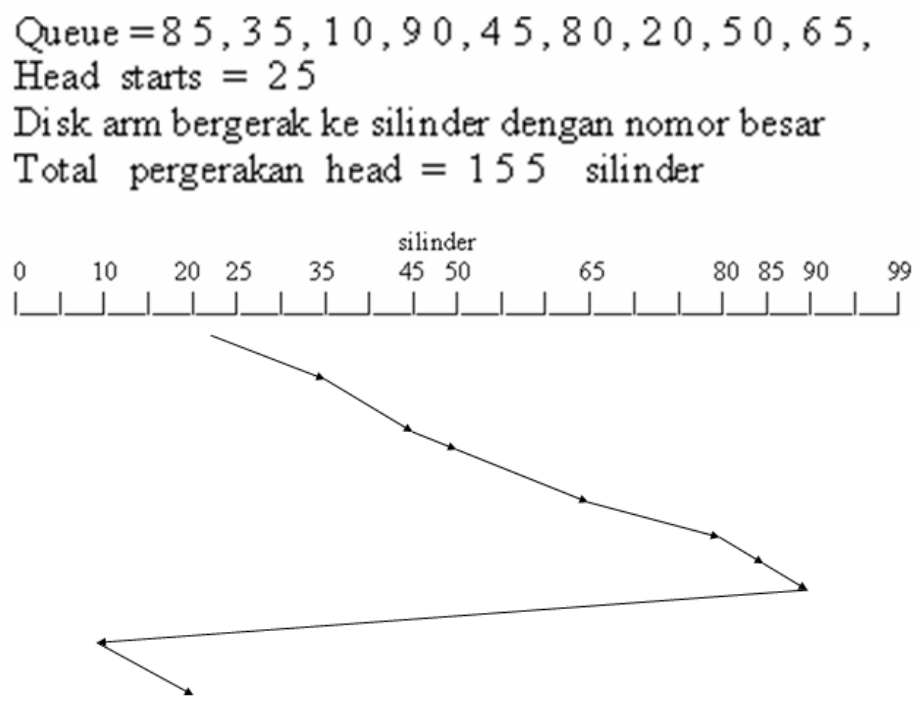
LOOK (head awal di silinder 25) Pergerakan disk arm menuju ke silinder dengan nomor yang lebih kecil (yaitu ke sebelah kiri)

Tabel 19.5. Contoh LOOK

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

Algoritma C-LOOK berhasil memperbaiki kelemahan-kelemahan algoritma SCAN, C-SCAN, dan LOOK. Algoritma C-LOOK memperbaiki kelemahan LOOK sama seperti algoritma C-SCAN memperbaiki kelemahan SCAN. 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 hanya 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 di depannya sesuai dengan arah pergerakannya.

Gambar 19.11. *C-LOOK*



Untuk lebih jelasnya perhatikan contoh berikut:

C-LOOK (head awal di silinder 25)

Tabel 19.6. Contoh C-LOOK

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 tabel tersebut, bisa dilihat bahwa *disk arm* bergerak dari permintaan 25 ke kanan sambil melayani pemintaan-permintaan yang ada di depannya 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 berada di depannya sesuai dengan arah pergerakannya yaitu ke permintaan 20.

Catatan:

Arah pergerakan *disk arm* yang bisa menuju 2 arah pada algoritma SCAN, C-SCAN, LOOK, dan C-LOOK, menuju silinder terbesar dan terkecil, diatur oleh *hardware controller*, hal ini membuat pengguna tidak bisa menentukan kemana *disk arm* bergerak.

19.9. Pemilihan Algoritma Penjadwalan

Performa dari suatu sistem biasanya tidak terlalu bergantung pada algoritma penjadwalan yang kita pakai, karena yang paling mempengaruhi kinerja dari suatu sistem adalah jumlah dan tipe dari pemintaan. 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 sistem operasi, sehingga algoritma tersebut bisa diganti dengan algoritma lain sesuai dengan jumlah dan tipe permintaan yang ada. Sistem Operasi memiliki algoritma *default* yang sering dipakai, yaitu SSTF dan LOOK.

Penerapan algoritma penjadwalan di atas berdasarkan hanya pada jarak pencarian saja. Tapi untuk disk modern, selain jarak pencarian, *rotation latency* (waktu tunggu untuk sektor yang diinginkan untuk berrotasi di bawah *disk head*) juga sangat berpengaruh. Tetapi algoritma untuk mengurangi *rotation latency* tidak dapat diterapkan oleh sistem operasi, karena pada disk modern tidak dapat diketahui lokasi fisik dari blok-blok logikanya. Tapi masalah *rotation latency* ini dapat ditangani dengan mengimplementasikan algoritma penjadwalan disk pada *hardware controller* yang terdapat dalam *disk drive*, sehingga kalau hanya kinerja M/K yang diperhatikan, maka sistem operasi dapat menyerahkan algoritma penjadwalan disk pada perangkat keras itu sendiri.

Dari seluruh algoritma yang sudah kita bahas di atas, tidak ada algoritma yang tebaik untuk semua keadaan yang terjadi. SSTF lebih umum dan memiliki perilaku yang lazim kita temui. SCAN dan C-SCAN memperlihatkan kemampuan yang lebih baik bagi sistem yang menempatkan beban pekerjaan yang berat pada disk, karena algoritma tersebut memiliki msalah *starvation* yang paling sedikit. SSTF dan LOOK sering dipakai sebagai algoritma dasar dalam sistem operasi.

19.10. Rangkuman

Bentuk penulisan *disk drive* modern adalah *array* blok logika satu dimensi yang besar. Ukuran blok logika dapat bermacam-macam. *Array* blok logika satu dimensi tersebut dipetakan dari disk ke sektor secara bertahap dan terurut. Ada dua macam aturan pemetaan, yaitu:

- **CLV**: kepadatan bit tiap *track* sama, semakin jauh sebuah *track* dari tengah disk, maka semakin besar jaraknya, dan juga semakin banyak sektor yang dimilikinya. Digunakan pada CD-ROM dan DVD-ROM.
- **CAV**: kepadatan bit dari zona terdalam ke zona terluar semakin berkurang, kecepatan rotasi konstan, sehingga aliran data pun konstan.

Host-Attached Storage (HAS) adalah pengaksesan *storage* melalui *port* M/K lokal.

Network-Attaced Storage (NAS) device adalah sebuah sistem penyimpanan yang mempunyai tujuan khusus untuk diakses dari jauh melalui *data network* .

Storage-area Network (SAN) adalah *network private*(menggunakan *protokol storage* daripada *protokol network*) yang menghubungkan server dan unit penyimpanan.

Penjadwalan disk sangat penting dalam meningkatkan efisiensi penggunaan perangkat keras. Efisiensi penggunaan disk terkait dengan kecepatan waktu akses dan besarnya *bandwith* disk. Untuk meningkatkan efisiensi tersebut dibutuhkan algoritma penjadwalan yang tepat dalam penjadwalan disk.

Algoritma penjadwalan disk ada beberapa macam, yaitu:

1. FCFS (First Come First Served)
2. Shortest-Seek-Time-First (SSTF)
3. SCAN
4. C-SCAN (Circular SCAN)
5. LOOK
6. C-LOOK (Circular LOOK)

Untuk algoritma 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

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

Bab 20. Sistem Penyimpanan Masal

20.1. Pendahuluan

Pada umumnya, penyimpanan sekunder berbentuk magnetic, kecepatan pengaksesan memori ini jauh lebih lambat dibandingkan memori utama. Pada bagian ini akan diperkenalkan konsep-konsep yang berhubungan dengan memori sekunder seperti system berkas, atribut dan operasi system berkas, struktur direktori, atribut dan operasi struktur direktori, system berkas jaringan, system berkas virtual, system berkas GNU/linux, keamanan system berkas, FHS (*File Hierarchy Systems*), serta alokasi blok system berkas.

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 memaminkan satu kaset saja.

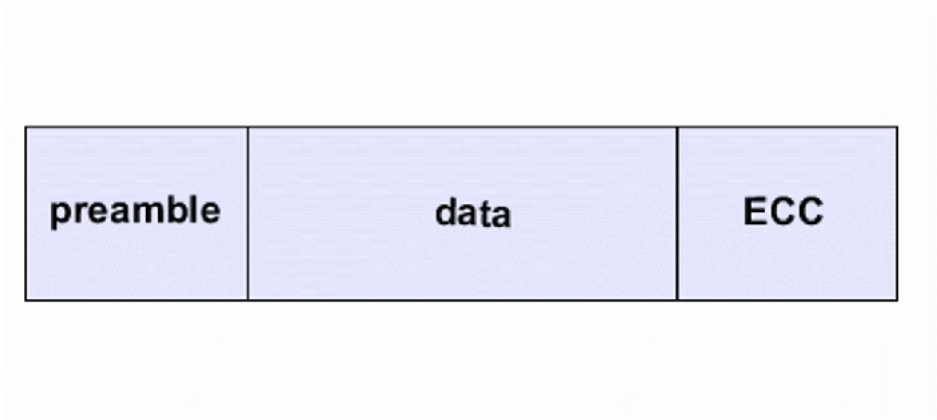
Struktur *disk* merupakan suatu hal yang penting bagi penyimpanan informasi. Sistem komputer modern menggunakan disk sebagai media penyimpanan sekunder, dulu pita makgnetik digunakan sebelum penggunaan *disk* sebagai media penyimpanan, sekunder yang memiliki waktu akses yang lebih lambat dari *disk*. Sejak digunakan disk, tape digunakan untuk beckup, 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.

20.2. Format

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 penegendali pembentukan sektor-sektor ini disebut *low level formatting* atau *physical formatting*. *low level formatting* juga akan mengisi disk dengan 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 medeteksi 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 tesimpan 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 diperlukan secara independen. *logical formatting* akan membentuk sistem berkas yang disertai pemetaan disk. terkadang sistem berkas ini dirasakan menggangu proses alokasi suatu data, sehingga diadakan sistem partisi yang tidak mengikutkan pembentukan sistem berkas, sistem raw disk

Gambar 20.1. Format sektor



Walaupun ukuran sektor dapat diatur pada proses *formatting*, namun kebanyakan disk memiliki sektor dengan ukuran 512 byte dengan alas an bahwa beberapa OS dapat menangani ukuran sektor 512. namun kemungkinan 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, begitupun sebaliknya. Bagian preamble mengandung bi-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 yang lain. Sehingga mungkin saja dirasakan mengganggu proses alokasi data. Hal ini dapat disiasati dengan system partisi lain yang tidak mengikutkan pembentukan file system, yang disebut raw disk (raw partition). Sebagai contoh, pada Windows XP ketika kita membuat sebuah patisi tanpa menyertakan suatu berkas system ke dalamnya maka partisi tersebut dapat disebut sebagai *raw partition*.

20.3. *Boot*

Booting adalah istilah teknologi informasi dalam bahasa ingris yang mengacu kepada proses awal menyalakan komputer dimana semua register prosesor disetting kosong, dan status mikroprosesor/ prosesor disetting reset. Kemudian address 0xFFFF diload di segment code *(code segment)* dan instruksi yang terdapat pada alamat address 0xFFFF tersebut dieksekusi. Secara umum program BIOS *(Basic Input Output System)*, yaitu sebuah software dasar, terpanggil. Sebab memang biasanya BIOS berada pada alamat tersebut. Kemudian BIOS akan melakukan cek terhadap semua error dalam memory, device-device yang terpasang/tersambung kepada komputer -- seperti port-port serial dan lain-lain. Inilah yang disebut dengan POST*(Power-On Self Test)*. Setelah cek terhadap sistem tersebut selesai, maka BIOS akan mencari [Sistem Operasi], memuatnya di memori dan mengeksekusinya. Dengan melakukan perubahan dalam setup BIOS (kita dapat melakukannya dengan menekan tombol tertentu saat proses booting mulai berjalan), kita dapat menentukan agar BIOS mencari Sistem operasi ke dalam floppy disk, hard disk, CD-ROM, USB dan lain-lain, dengan urutan yang kita inginkan. BIOS sebenarnya tidak memuat Sistem Operasi secara lengkap. Ia hanya memuat satu bagian dari code yang ada di sektor pertama *(first sector, disebut juga boot sector)* pada media disk yang kita tentukan tadi. Bagian/fragmen dari code Sistem Operasi tersebut sebesar 512 byte, dan 2 byte terakhir dari fragmen code tersebut haruslah 0xAA55 (disebut juga sebagai boot signature). Jika boot signature tersebut tidak ada, maka media disk dikatakan tidak bootable, dan BIOS akan mencari Sistem Operasi pada media disk berikutnya. Fragmen code yang harus berada pada boot sector tadi disebut sebagai boot-strap loader. BIOS akan memuat *boot-strap* loader tersebut ke dalam memory diawali pada alamat 0x7C00, kemudian menjalankan boot-strap loader tadi. Akhirnya sekarang kekuasaan berpindah kepada boot-

strap loader untuk memuat Sistem Operasi dan melakukan setting yang diperlukan agar Sistem Operasi dapat berjalan. Rangkaian proses inilah yang dinamakan dengan booting.

Saat sebuah computer dijalankan, system akan mencari sebuah initial program yang akan memulai segala sesuatunya. Initial programnya (initial bootstrap) bersifat sederhana dan akan menginisialisasi seluruh aspek yang dibutuhkan computer untuk beroperasi dengan baik seperti CPU Register, controller, dan terakir adalah Sistem Operasinya. Pada kebanyakan computer, bootstrap disimpan di ROM (read only memory) kanrena letaknya yang tetap dan dapat dieksekusi waktu pertama kali listrik dijalankan. Letak bootstrap di ROM juga menguntungkan karena sifatnya yang read only memungkinkan dia untuk tidak terinveksi virus. Untuk alasan praktis, bootstrap sering dibuat berbentuk kecil (tiny loader) dan diletakan di ROM yang kemungkinan akan me-load full bootstrap dari disk bagian disk yang disebut boot block. Perubahan berbentuk simple ini bertujuan jika ada perubahan yang diadakan perubahan pada bootstrap, maka stuktur ROM tidak perlu dirubah semuanya. Konsep boot block sangan erat kaitannya dengan proses booting pada sebuah computer. Ketika computer dinyalakan, sistem akan mencari sebuah initial program yang akan memulai segalah sesuatu yang berhubungan dengan proses booting. Program ini dikenal sebagai initial bootstrap dan akan menginisialisasi seluruh aspek yang dibutuhkan computer untuk beroperasi dengan baik seperti CPU Register, controller, dan terakir adalah Sistem Operasinya. Pada Pentium dan kebanyakan computer , MBR terletak pada sector 0 dan mengandung beberapa boot code dan beserta table partisi. Table partisi ini mengandung berbagai informasi misalnya letak sector pertama pada setiap partisi dan ukuran partisi. Agar isa melakukan boot dari hard disk, salah satu partisi yang terdapat pada table partisi, harus ditangdai sebagai active partition. Boot block tidak selalu mengandung kernel. Bisa saja ia derupa sebuah boot loader, misalnya LILO (Linux Loader) atau GRUB (GRand Unified Bootloader). Contoh komfigurasi sebuah inisialisasi 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" degan catatan bahwa "C:\" adalah volume atau partisi yang di-set sebagai active *partition*.

20.4. *Bad Block*

Bad block adalah satu atau lebih sektor yang cacat atau rusak. kerusakan ini akan diakibatkan karena kerentanan disk jika sering dipindah-pindah atau kemasukan benda asing. akibatnya, data didalam bed block menjadi tidak terbaca.

Pada disk sederhana seperti IDE controller, bad block akan ditangani secara manual seperti dengan perintaj format MS-DOS yang akan mencari bad block dan menulis nilai spesial ke FAT entry agar tidak mengalokasikan slidebaru. *branch rountine* ke block tersebut. SCSI mengatasi bad block dengan cara yang lebih baik. daftar bad block-nya di pertahankan oleh controller saat low-level formaating dan terus diperbaharui selama disk itu digunakan. low-leveel formating, 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 mengcopy semua isi sektor ke sektor selanjutnya secara bertahap secara satu per satu sampai ditemukan sektor kosong. misal bad sector ditemukan di sektor 7 dan sektor-sektor selanjutnya hanyalah sektor 30 dan seterusnya. Maka isi sektor 30 akan akan di-copy keruang 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 sector 8.

perlu diingat bahwa kedua algoritma diatas juga tidak selalu berhasil karena data yang telah tersimpan didalam bad block biasanya sudah tidak bisa terbaca

20.5. *Swap*

Swap space adalah istilah lain untuk backing store. Swap space dapat diletakan pada file sistem dalam bentuk swap file atau dalm partisi disk yang terpisah. dengan megunakan file sistem akan terjadi overhead yang cukup segnifikan. overhead dapat dikurangi dengan mengunakan aplikasi yang

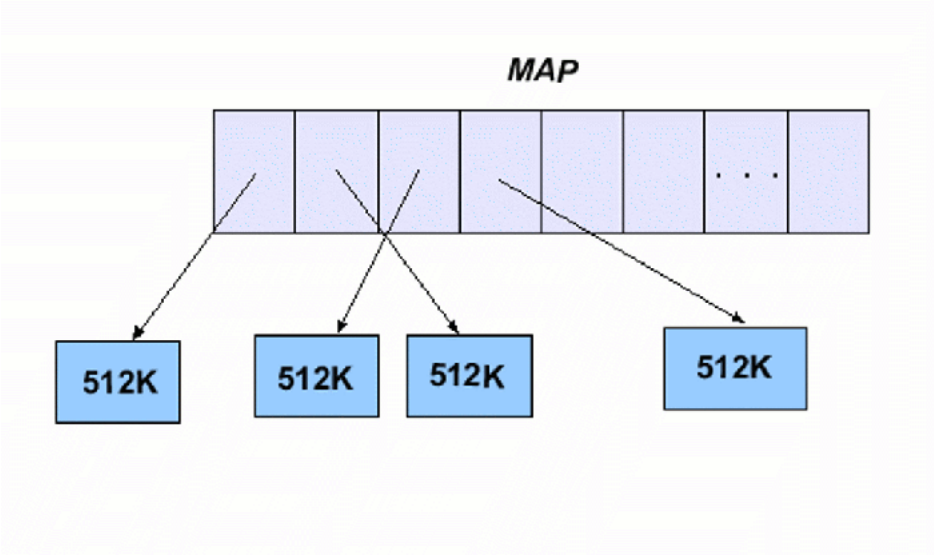
megalokasikan swap file dalam block disk yang berdekatan. lebih umum lagi, partisi yang terpisah digunakan ketika tidak ada file sisitem.solaris linux mengijinkan pertukaran antara *raw partition* dan file system.

Managemen Ruang Swap adalah salah satu low level task dari OS. Memori virtual mengunakan ruang disk sebagai perluasan dari memori utama. Tujuan utamanya adalah untuk menghasilkan output yang baik. Namun dilain pihak, pengguanaan 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 menggunakan seluruh proses trmasuk data yang segmen. Jumlah dari ruang swap yang dibutuhkan dari jumlah memori fisik, jumlah dari memeori virtual yang dijalankan, cara penggunaan memori virtual tersebut. beberapa sistem operasi seperti UNIX menggunakan banyak ruang swap, yang biasa diletakan pada disk terpisah. Ketika kita menentukan besarnya ruang swap, sebaiknya kita tidak terlalu banyak atau terlalu sedikit. Namun perlu diketahui bahwa akan lebih aman jika mengalokasikan lebih 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 lamanya akses dan pemborosan ruang disk, tetapi hal itu tidak menimbulkan resiko terhantinya proses. Sebagai contoh, Linux memperbolehkan penggunaan banyak ruang swap yang tersebar pada disk yang terpisah

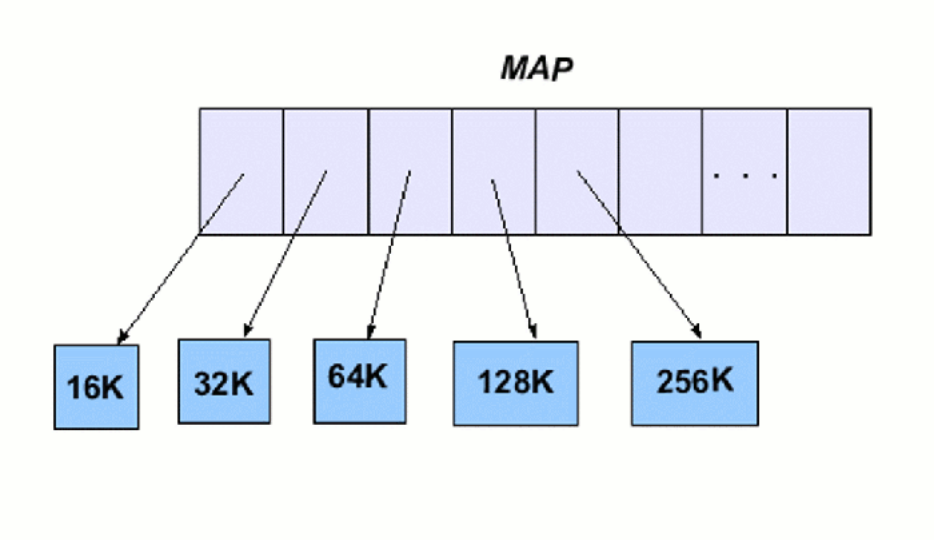
Lokasi Ruang Swap. Ruang swap dapat diletakan didalam sistem berkas normal atau dapat juga berada di partisi yang terpisah. Beberapa OS (Linux misalnya) dapat memiliki dua ruang swap sekaligus, yakni pada berkas biasa dan partisi terpisah. Jika ruang swap berukuran besar dan diletakan di sistem berkas normal, routine-nya dapat menciptakan, menamainya dan menentukan besar ruang. Walaupun lebih mudah dijalankan. cara ini cenderung tidak efisien. Pengaksesanya akan sangat memakan waktu dan akan meningkatkan fragmentasi karena pencarian data yang berulang terus selama proses baca atau tulis.

Gambar 20.2. Managemen Ruang Swap: Pemetaan Swap Segmen Teks 4.3 BSD



Ruang swap yang diletakan di partisi disk terpisah (raw partition), menggunakan manager ruang swap terpisah untuk melakukan pengalokasian ruang. manager ruang swap tersebut 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 20.3. Managemen Ruang Swap: Pemetaan Swap Segmen Data 4.3 BSD



20.6. *RAID*

Disk memiliki resiko untuk mengalami kerusakan. kerusakan ini dapat berakibat turunnya kinerja ataupun hilangnya data. meskipun terdapat beckup, tetap saja ada kemungkinan data yang hilang karena adanya perubahan yang terjadi setelah terakhir kali datadi beckup dan belum sempat di backup 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 jika tidak kecepatan CPU yang besar itu akan menjadi sia-sia.

Berbagai macam cara dilakukan untuk meningkatkan kinerja dan juga reliabilitas dari disk. bisanya untuk meningkatkan kinerja, dilibatka banyak disk sebagai satu unit penyimpanan. tiap-tiap blok data dipecah kedalam bebarapa sublok, dan dibagi-bagi kedalam 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--Redudant Array Independent (atau inexpensive) Disk. selain masalah kinerja RAID juga dapat meningkatkan reliabilitas dari disk dengan jalan mengunakan disk tambahan(redudant) untuk menyimpan paritas bit/blok ataupun sebagi mirror dari disk-disk data yang ada.

Tiga karekteristik umum dari RAID ini, yaitu:

1. RAID adalah sebuah set dari beberapa physical drive yang dipandang oleh sistem operasi sebagai sebuah logical drive.
2. Data didistribusikan kedalam array dari beberapa *physical drive*
3. Kapasitas disk yang belebih digunakan untuk menyimpan informasi paritas, yang menjamin data dapat diperbaiki jika terjadi kegagalan pada salah satu disk.

20.7. Pemilihan Tingkatan *RAID*

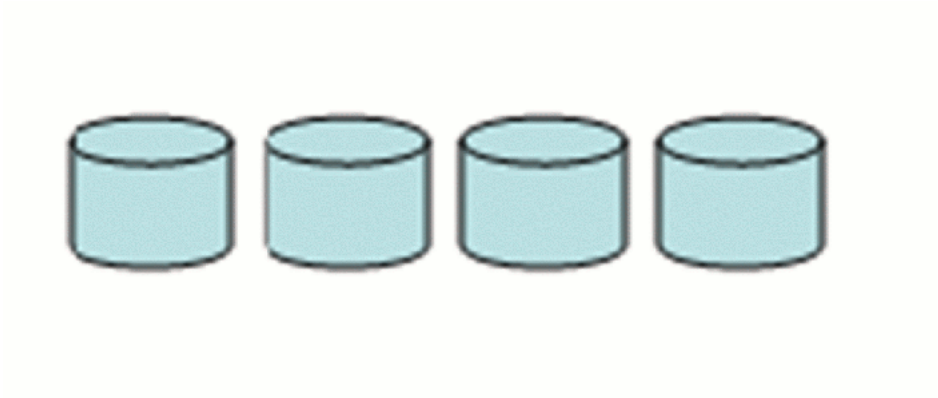
Raid terdiri dapat dibagi menjadi enam level yang berbeda:

1. Raid level 0. Menggunakan kumpulan disk dengan striping pada level blok, tanpa redundansi. jadi hanya melakukan striping blok data kedalam beberapadisk. kelebihan level ini antara lain akses beberapa blok bisa dilakukan secara paralel sehingga bis lebih cepat. kekurangan antara lain akses perblok sama saja seperti tidak ada peningkatan, kehandalan kurang karena tidak adanya pembekcupan data dengan redundancy. Berdasarkan definisi RAID sebagai redudancy array maka level ini sebenarnya tidak termasuk kedalam kelompok RAID karena tidak menggunakan redundansy untuk peningkatan kinerjanya.

2. RAID level 1. Merupakan disk mirroring, menduplikat data tanpa striping. Cara ini dapat meningkatkan kinerja disk, tapi 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 membutuhkan disk 2 kali lipat dari yang biasanya.
3. RAID level 2. Merupakan pengorganisasian dengan *error correction code* (ECC). Seperti pada memory dimana pendeteksian mengalami error mengunakan paritas bit. Sebagai contoh, misalnya misalnya setiap byte data, memiliki paritas bit yang bersesuaian yang mempresentasikan jumlah bit "1" didalm byte data tersebut dimana paritas bit = 0 jika bit genap atau paritas bit = 1 jika bit ganjil. Jadi, jika salah satu bit pada salah satu data 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 jika dibandingkan dengan level 1 (mirroring). Kelemahannya antara lain prlu adanya perhitungan paritas bit, sehingga menulis atau 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 yang interleaved. Pengorganisasian ini hamper sama dengan RAID level 2, perbedaanya adalah pada level 3 ini hanya memerlukan sebuah disk redudan, berapapun kumpulan disknya, hal ini dapt 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 membutuhakan sebuah bit paritas untuk sekumpulan bit yang mempuntai sekumpulan bit yang mempunyai posisi yang sama pada setiap dis yang berisi data. Selain itu juga menggunakan data striping dan mengakses disk-disk secara parallel. Kelebihannya antara lain kehandalan (rehabilitas) bagus, akses data lebih cepat karena pembacaan tiap bit dilakukan pada beberapa disk (parlel), hanya butuh 1 disk redudan yang tentunya lebih menguntungkan dengan level 1 dan 2. kelemahannya antara lain perlu adanya perhitungan dan penulisan parity bit akibatnya performannya lebih rendah dibandingkan yang menggunakan paritas.
5. RAID level 4. Merupakan pengorganisasian dengan paritas blok interleaved, yaitu mengunakan striping data pada level blok, menyimpan sebuah parits blok pada sebuah disk yang terpisah untuk setiap blok data pada disk-disk lain yang bersesuaian. Jka sebuah disk gagal. Blok paritas tersebut dapat digunakan untuk membentuk kembali blok-blok data pada disk yang bisa lebih cepat karena bisa parlel dan kehandalannya juga bagus karena adanya paritas blok. Kelemahannya antara lain akses perblok seperti biasa penggunaan 1 disk., bahkan untuk penulisan ke 1 blok memerlukan 4 pengaksesan untuk membaca ke disk data yag bersangkutan dan paritas disk, dan 2 lagi untuk penulisan ke 2 disk itu pula (read-modify-read)
6. RAID level 5. Merupakan pengorganisasian dengan paritas blok interleaved terbesar. Data dan paritas disebr 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 terdapt kumpulan dari 5 disk, paritas paritas blok ke n akan disimpan pada disk (n mod 5) +1, blok ke n dari 4 disk yang lain menyimpan data yang sebenarnya dari blok tersebut. Sebuah paritas blok tidak disimpan pada disk yang sama dengan lok-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 pentebaran paritas seoerti ini dapat menghindari penggunaan berlebihan dari sebuah paritas bit 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 redudansi P+Q, seperti RAID level 5, tetapi menyimpan informasi redudan 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 pada 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 data mean time to repair (MTTR). Kerugiannya yaitu penalty 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 RAID level 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 adalah RAID 1+0, dimana disk-disk mirror secara berpasangan, dan kemudian hasil pasangan mirror-nya di-stri. RAID 1+0 ini mempunyai keuntungan lebih dibandingkan dengan RAID 0+1. sebagai contoh, jika sebuah disk gagal pada RAID 0+1, seluruh disknya tidak dapat di akses, 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.

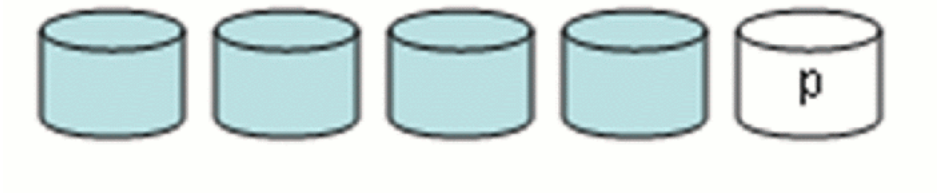
Gambar 20.4. (a) RAID 0: non-redundant striping



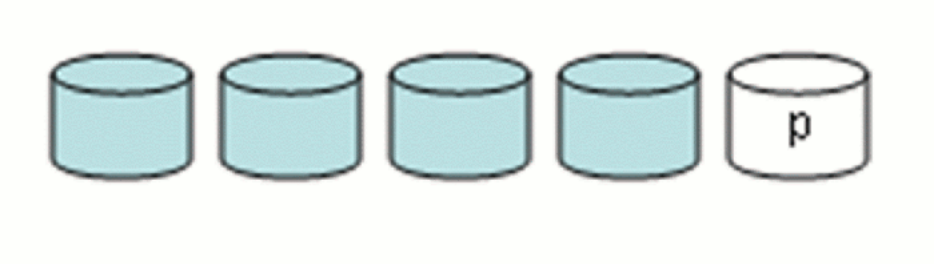
Gambar 20.5. (b) RAID 1: mirrored disk

Gambar 20.6. (c) RAID 2: memory-style error-correcting codes

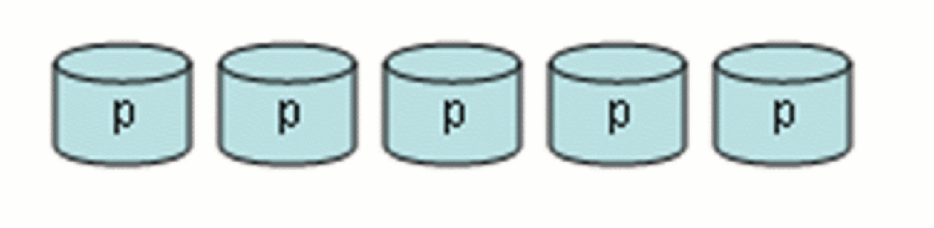
Gambar 20.7. (d) RAID 3: bit-interleavedparity



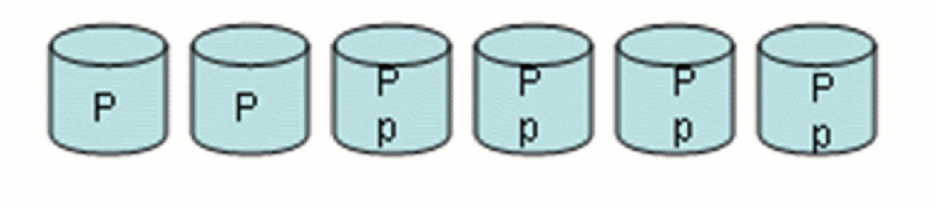
Gambar 20.8. (e) RAID 4: block-interleaved parity



Gambar 20.9. (f) RAID 5: block-interleaved distributed parity



Gambar 20.10. (g) RAID 6: P+Q redundancy



20.8. Penyimpanan Tersier

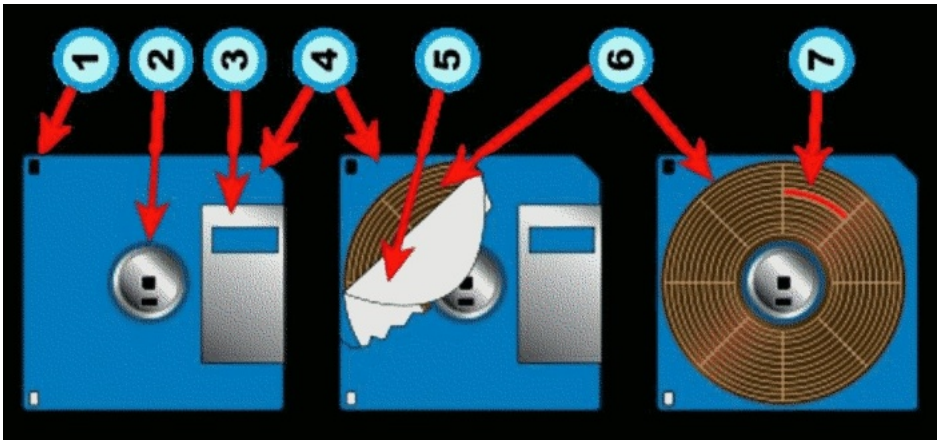
Penyimpanan tersier pada dasarnya adalah menggunakan *removable media* yang tentu saja berdampak pada biaya produksi yang lebih murah. Sebagai contoh: sebuah VCR dengan banyak kaset akan lebih murah daripada sebuah VCR yang hanya dapat memaminkan satu kaset saja.

Jenis Stuktur 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 1-2Mb saja, tetapi sekarang floppy disk dapat menyimpan data hingga 1 Gb. meskipun kecepatan akses datanya lebih lambat dari pada harddisk dan lebih rentan terhadap kerusakan permukaan disknya, floppy disk dulu sangat disukai karena harganya yang lebih murah dari pada removable disk lainnya dan dapat ditulis berkali-kali
.

Gambar 20.11. Komponen internal dasar floopy disk 3.5 inch



1.Write-protec tab; 2. Hub; 3. Shutter; 4. Plastic housting; 5. Paper ring; 6. Magnetic disk; 7. Disk sector;

Magneto-optic disk

Magneto-optic disk adalah salah satu contoh dari removable disk. Teknologi penyimpanan data pada magneto-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 bantuaan kerr effect. cara kerja kerr effect adalah ketika suatu sinar laser dipantulkan dari sebuah titik magnetik. palarisasinya 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 20.12. Magneto-Otical Disk

Optical disk

Optical disk tidak menggunakan bahan yang bersifat magneti sama sekali. Optical disk menggunakan bahan spesial yang dapat diubah oleh sinar laser menjadi memiliki spot-spot yang relatif gelap atau terang. contohnya dar optical disk ini adalah CD-RW dan DVD-RW. teknologi optical disk ini dibagi menjadi dua yaitu:

1. ***Phase-change disk*.** disk ini dilapisi oleh bahan yang dapat mengkristal(beku) menjadi crystalline(serpihan-serpihan kristal) atau menjadi amorphous state(bagian yang tak berbentuk). Bagian crytalline ini lebih transparan, karenanya tembakan laser yang mengenainya akan lebih terang 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 kedalam amorphous state, sehingga dapat digunakan untuk menulis data lagi. sinar laser dengan kekuatan sedang dipakai untuk menghapus data denga cara melelehkan permukaan disknya dan membekukannya kembali ke dalam keadaan *crytalline*, sedangakan sinar laser dengan kekuatan lemah digunakan untuk membaca data yang telah disimpan.
2. ***Dye-Polimer disk*.** Dye-polimer merekam data dengan membuat bump(gelombang) disk dilapisi dengan bahan yang dapat enyerap sinar laser. sinar laser ini membakar spot hingga spot ini memuai dan membentuk bump(gelombang). bump ini dapat dihilangakan atau didatarkan kembali dengan cara dipanasi lagi dengan sinar laser.

Gambar 20.13. DVD-RW disk pada sebuah gelondong

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 sebuat aluminium film yang dilapisi plastik dibagian bawah maupun dibagian atasnya.cara penyimapanan data pada WORM ini adalah dengan cara memanfaatkan sinar laser untuk mebuat 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 20.14. CD-R

Read only disk

Read only disk menggunakan teknologi yang mirip dengan optical disk dan WORM, tetapi penyimpanan bit-bit dilakukan demhan 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 20.15. CDROM Drive

Tapes

Harga tapes drive memang lebih mahal dari pada magnetis disk drive, tetapi harga catridge sebuah tape lebih murah dari pada equivalent penyimpanan data pada magnetic disk. sebuah tapes dapat menyimpan data lebih banyak dari pada optical disk maupun magnetik disk. tape drive dan disk drive hampir sama dalam kecepatan transfer data, tetapi dalam akses data secara random, tape jauh lebih lamabat, karena mambutuhkan 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 20.16. DDS Tape Drives

Flash memory

Flash memori adalah sejenis EEPROM (Electrically-Erasable Programmable Read-Only Memori) yang mengizinkan akses pada lokasi memori untik dihapus atau ditulis dalam satu operasi pemograman. istilah awamnya memori adalah suatu bentuk dari chip memori yang dapat ditulis, tidak seperti chip random accsess memori, dan dapt menyimpan datanya meskipun tanpa daya listrik (non-volatile). memori ini biasanya digunakan dalam kartu memori, USB flash drive(flash disk), pemutar MP3, kamera digital, telepon genggam.

Gambar 20.17. USB Drive

20.9. Dukungan Sistem Operasi

Suatu Operating System bertugas untuk mengatur *physical devices* serta menampilkan suatu abstraksi dari virtual machine ke suatu aplikasi.

OS menyediakan dua abstraksi untuk hard disk, yaitu:

1. Raw device= array dari beberapa blok.
2. File System = sistem operasi menyusun dan menjadwalkan permintaan interleaved dari beberapa aplikasi.

Operasi file pada umumnya didukung oleh sistem operasi:

- Create, yaitu menciptakan entri direktori dan mengatur sebagian atribut file, tetapi tidak menyediakan ruang untuk isi file.
- Delate, yaitu menghapus isi direktori untuk membebaskan ruang yang ditempati oleh file.

- Open, yaitu mengambil atribut file dan daftar alamat yang akan digunakan uantuk diletakan pada memori sehingga kegiatan baca/tulis data berlangsung lebih cepat.
- Close, yaitu membersihkan data pada buffer dan membebaskan memori yang digunakan untuk menyimpan atribut file dan alamat disk.
- Read, yaitu membaca sejumlah byte dari posisi tertentu pada file (yang ditentukan oleh pengguna) dan diletakan pada buffer.
- Write, yaitu menulis sejumlah byte dari buffer ke posisi tertentu pada file yang ditentukan oleh pengguna..
- Append, yaitu menambahkan data pada akhir file.
- Rename, yaitu menganti nama file.

20.10. Kinerja

Masalah kinerja sebagai salah satu komponen dari sistem operasi, penyimpanan tersier mempuyai tga aspek uatama dalam kinerja, yaitu: kecepatan, kehandalan, dan biaya.

1. **Kecepatan .** Kecepatan dari penyimpanan tersier memiliki dua aspek: benwidth dan latency, menurut silberschatsz et.at. [silberschatz2002], sustained bandwidth adalah rata-rata tingakat data pada proses transefer, 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 pada proses transfer (jumlah byte dibagi dengan lamanya proses tansfer). perhitungan rata-rata pada seluruh waktu input atau output, termasuk waktu untuk pencarian disebut dengan effective bandwidth. untuk bandwidth suatu drive, umunya yang dimaksud adalah sustained banwidth. untuk removable disk, bandwidth berkisar dri beberapa megabyte per detik untuk yang paling lambat sampai melebihi 40 MB per detik untuk yang paling cepat. tape memilki 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 dari pada tape. penyimpana disk secara esensisl berdimensi dua--semua bit berada diluar dibukaan. akses disk hanya memindahkan arm ke silinder yang dipilih dan menunggu retational latency, yang bisa memakan waktu kuarang dari pada 5 milidetik. sebaliknya, penyimpanan tape berdimensi tiga. pada sembarang waktu, hanya bagian kecil tape yang terakses ke haed, 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 mamakan waktu puluhan atau ratusan detik. jadi, kita bisa secara umum mengatakan bahwa random access dalam sebuah tape cartridge lebih dari seribu kali lebih lambat dari pada random access pada disk.
2. **Kehandalan.** Removable magnetic disk tidak begitu dapat diandalkan dibandingkan dengan fixied hard-disk karena cartridge lebih rentan terhadap lingkungan yang berbahaya seprti debu, perubahan besar pada temperatur dan kelembaban, dan gangguan mekanis seprti tekukan. Optical disk dianggap sangat dapat diandalkan karena lapisan yang menyimpan bit dilindungi oleh plastik trnsparan atau lapisan kaca. Removable magnetic sedikit kurang andal daripada fixied hard disk karena cardridge-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 diangap sangat andal, karena lapisan yang meyimpan bit dilindungi oleh plastik trnsparan atau lapisan kaca. kehandalan 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 lainya cukup lembut sehingga memungkinkan penggunaan ulang jutaan kali. dibandingakan dengan magnetic-disk head, head dalam magnetic-tape drive merupakan titik lemah. disk head melayang diatas media, tetapi head kontak langsug 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 dari pada 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, sedangakan, kalau terjadi kegagalan *(failure)* tape drive atau optical disk drive, seringkali data cardridge-nya tidak apa-apa.

3. **Harga.** Terjadi kecenderungan biaya per megabyte untuk memory DRAM, magnetic hard disk, dan tape drive. Harga dalam grafik adalah harga terendah yang ditamukan dalam iklan diberbagai 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 dibandingakan dengan pasar mainbingkai atau mini komputer. 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 membuat ribuan cardridge, storage cost didominasi oleh biaya tape cardridge. Pada tahun 2004, biaya per GB tape cardridge 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 ditunjukan sebelum tahun 1984, karena, seperti telah disebutkan, majalah yang digunakan dalam menelusuri harga ditrgetkan untuk ruang pasar komputer 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 kenyataanya, harga per megabyte magnetic disk drive mendekati sebuah tape cardridge tanpa tape drive, akibatnya, tape library yang berukuran kecil dan sedang memiliki storage cost lebih tinggi dari pada sistem disk dengan kapasitas setara.

20.11. Rangkuman

Penyimpana tersier dibangun dengan *removable media*. Penyimpanan tersier biasanya diimplementasikan sebagai jukebox dari tape atau removable disk. Kebanyakan system operasi menagani removable disk seperti fixed disk. Sementara itu, tape biasanya sitampilkan sebagai media penyimpana mentah (raw stroge medium),system berkas tidak disediakan Tiga aspek kinerja yang utama adalah kecepatan, kehandalan, dan biaya. Random access pada tape jauh lebih lama daripada disk. Kehandalan removable magnetic disk masih kurang karena masih rentan terhadap
.

Aspek penting mengenai manajemen ruang swap, yaitu
1. pengunaan ruang swap. Penguanan ruang swap tergantung pada penerapan algoritma.
2. Lokasi ruang swap.

Ruang swap dapat ditentukan di:

- sistem berkas normal
- partisi yang terpisah

Rujukan

[WEBWIKI2007] Wikipedia. 2007. *Serializability – http://en.wikipedia.org/wiki/Floopy_disk, Magneto_optical drive, DVD_Rw, CD_ROM, Tape_drive.* . Diakses 2 Mei 2007.

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

[Tanenbaum1997] Andrew S. Tanenbaum. 2001. *Modern Operating Systems, design and implementation.*. Second Edition. Prentice-Hall.

[Tanenbaum1992] Andrew S. Tanenbaum. 2001. *Modern Operating Systems*. Second Edition. Prentice-Hall.

Bab 21. Sistem Berkas Linux

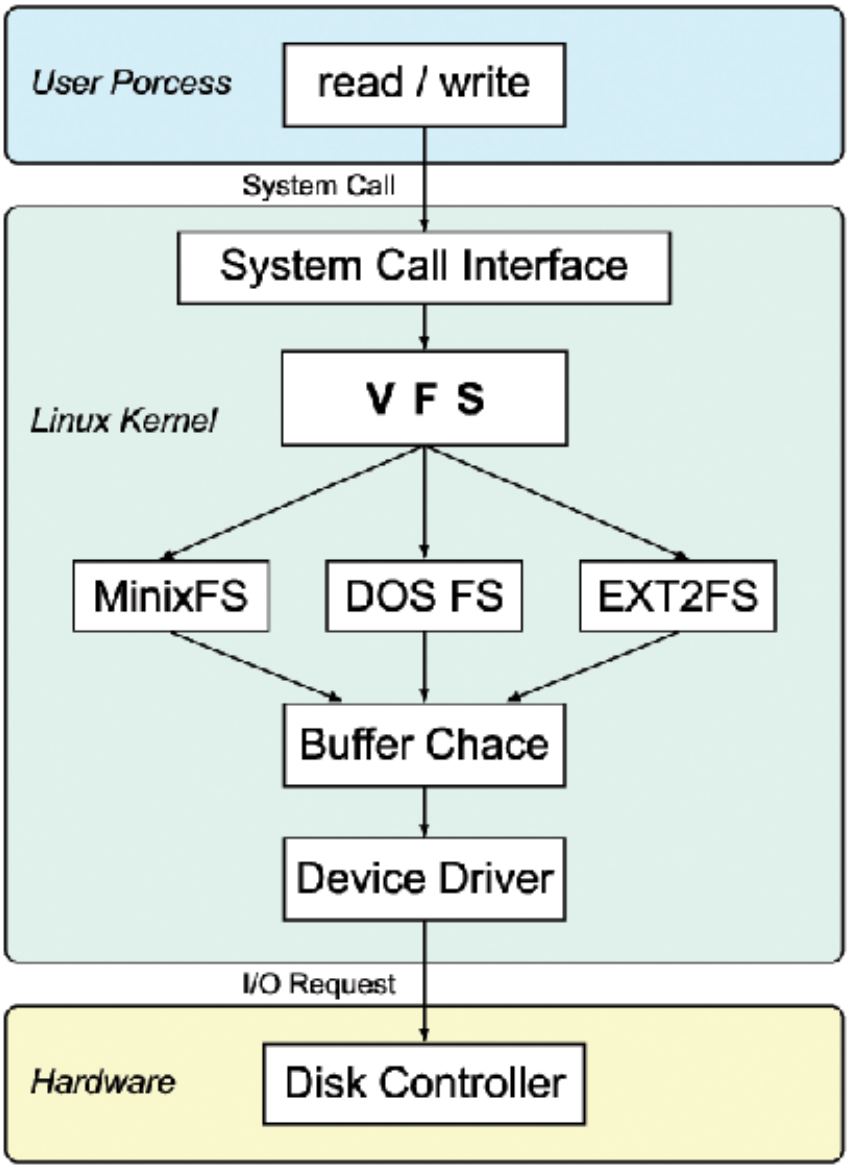
21.1. Pendahuluan

Secara umum, sistem berkas adalah suatu sistem yang bertanggung jawab dalam penyimpanan, penempatan, pengaturan, dan pencarian suatu berkas atau data lain dalam suatu tempat atau media tertentu. Sebagian besar sistem operasi yang ada saat ini memiliki dan mengimplementasikan sistem berkasnya sendiri, salah satunya Linux, yang mengimplementasikan sistem berkas yang digunakan oleh Unix.

Segala sesuatu dalam sistem berkas Linux dapat dipandang sebagai berkas; partisi adalah berkas, *directory* adalah berkas, dan berkas tentu juga berkas. Pada sistem mirip Unix contohnya Linux, beberapa sistem berkas yang terpisah tidak diakses melalui *device identifiers* seperti *device number* atau *drive name*, tetapi melalui *mount point*, yaitu *path* dimana sistem berkas itu di- *mount* oleh sistem. Pada Linux hanya terdapat sebuah struktur *tree* tunggal yang dimulai dari *root directory* (direpresentasikan dengan '/') kemudian meluas menjadi beberapa *subdirectory*. Linux menempatkan semua partisi di bawah *root directory* dengan me- *mounting*-nya atau mengaitkannya ke suatu *directory* tertentu.

Salah satu keistimewaan dari Linux adalah dukungannya terhadap beberapa sistem berkas yang berbeda. Hal ini membuat Linux sangat *flexible* dan dapat berdampingan dengan sistem operasi lain dalam satu komputer. Kernel Linux mengatasi perbedaan berbagai jenis sistem berkas tersebut dengan menyembunyikan detil implementasi dari masing-masing sistem berkas di bawah suatu lapisan abstraksi yang dikenal dengan *Virtual File System* (VFS). Beberapa sistem berkas yang didukung oleh Linux antara lain ext, ext2, ext3, ReiserFS, xia, minix, vfat, proc, smb, ncp, iso9660, sysv, hpfs, nfs, affs, dan ufs. Salah satu sistem berkas yang sangat populer di sistem Linux adalah *Extended File System* (EXTFS), yang telah mendukung salah satu fitur penting dalam suatu sistem berkas, yaitu jurnal.

Gambar 21.1. Diagram VFS



21.2. VFS

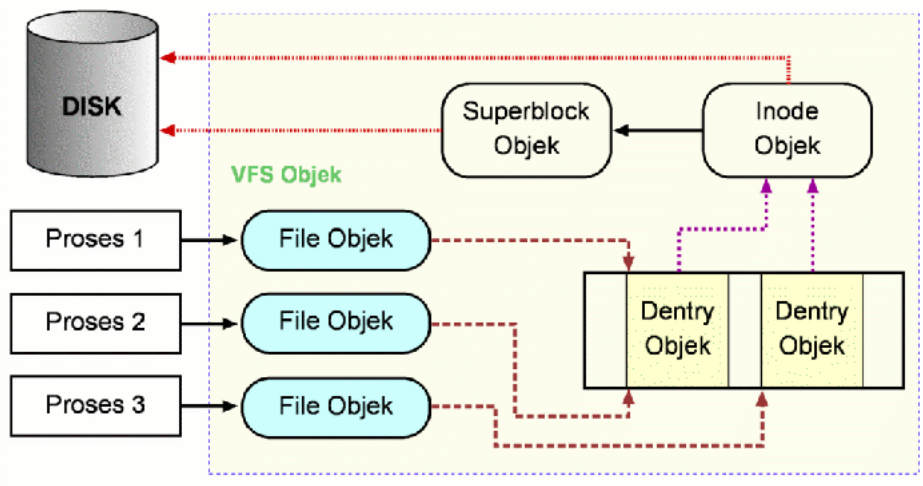
Virtual File System atau *Virtual File System Switch* adalah suatu lapisan abstrak di atas sistem berkas yang sesungguhnya, yang menangani semua *system call* yang berhubungan dengan suatu sistem berkas di bawahnya. Tujuan dari VFS yaitu agar berbagai sistem berkas yang berbeda dapat di akses oleh aplikasi komputer dengan cara yang seragam. Gambar 21.1 menunjukkan hubungan antara Linux VFS dengan sistem berkas yang sesungguhnya. *Virtual File System* menyediakan antarmuka antara *system call* dengan sistem berkas yang sesungguhnya. Keberadaan VFS tentu dapat mengatasi perbedaan berbagai sistem berkas yang digunakan oleh berbagai sistem operasi saat ini seperti Windows, Mac OS, Linux, dan sebagainya, sehingga suatu aplikasi dapat mengakses berkas dari sistem berkas yang berbeda tanpa perlu mengetahui jenis sistem berkas yang digunakan dan detil implementasi dari masing-masing sistem berkas tersebut.

Sistem Berkas yang didukung oleh Linux VFS dapat dibagi menjadi tiga kategori:
1. ***Disk-based Filesystem .*** Contohnya EXTFS, iso9660, FAT, dan sebagainya.

2. ***Network-based Filesystem .*** Contohnya NFS, Coda, AFS, CIFS, dan sebagainya.
3. **Sistem berkas khusus.** Contohnya /proc, RAMFS, dan DEVFS

Sistem berkas biasanya diimplementasikan secara *object-oriented*. Jadi, sistem berkas dapat dipandang sebagai sekumpulan objek yang terbentuk dari suatu struktur data dengan beberapa *method/function* yang berkaitan. Ide dibalik implementasi VFS yaitu suatu konsep yang disebut *common file model*. Singkatnya, setiap sistem berkas yang didukung oleh Linux harus dapat menerjemahkan *physical organization* yang diimplementasikannya ke dalam VFS *common file model*. Linux VFS *common file model* terdiri dari beberapa objek, yaitu *superblock*, *inode*, *file*, dan *dentry*.

Gambar 21.2. Interaksi antara proses dengan objek VFS

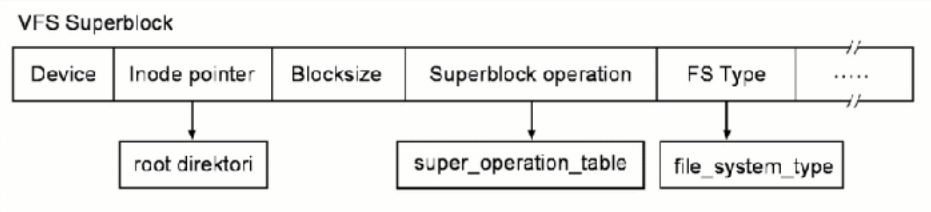


VFS *Superblock*

Setiap sistem berkas yang di-*mount* akan direpresentasikan oleh sebuah *VFS Superblock*. *Superblock* digunakan untuk menyimpan informasi mengenai partisi tersebut. Pada dasarnya *superblock* merupakan suatu struktur data yang menyimpan beberapa informasi sebagai berikut:

- ***Device*.** Merupakan suatu *device identifier*, contohnya `/dev/hda1` adalah *harddisk* pertama yang terdapat pada sistem memiliki *device identifier* `0x300`.
- ***Inode Pointer*.** Merupakan suatu *pointer* yang menunjuk ke *inode* pertama pada sistem berkas.
- ***Blocksize*.** Menunjukkan ukuran suatu *block* dari sistem berkas, contohnya 1024 bytes.
- ***Superblock operation*.** Merupakan suatu *pointer* ke sekumpulan *superblock routine* (fungsi) dari sistem berkas, contohnya *read*, *write*, dan sebagainya.
- ***File System type*.** Menunjukkan tipe dari sistem berkas, contohnya EXT2, FAT, NTFS.
- ***File System specific .*** Merupakan suatu *pointer* ke informasi yang dibutuhkan oleh sistem berkas.

Gambar 21.3. Ilustrasi VFS *Superblock*



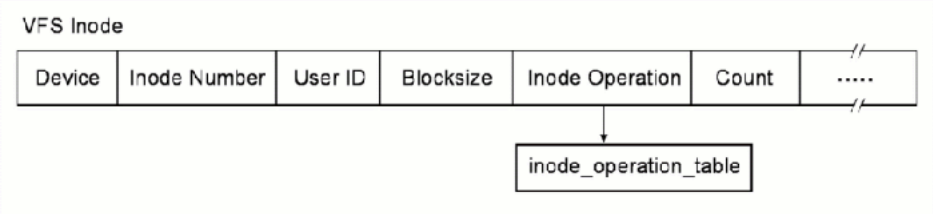
Beberapa operasi yang terdapat dalam VFS *Superblock* adalah `alloc_inode`, `read_inode`, `write_inode`, `put_inode`, `delete_inode, dan sebagainya`.

VFS *Inode*

VFS inode adalah abstraksi VFS untuk berkas. Setiap berkas, *directory*, dan data lainnya pada VFS direpresentasikan oleh satu dan hanya satu VFS *inode*. VFS *inode* hanya terdapat di memori kernel dan disimpan di *inode chace* selama masih dibutuhkan oleh sistem. Beberapa informasi yang disimpan oleh *VFS inode* yaitu:

- ***Device*.** Menunjukan *device identifier* dari suatu *device* yang menyimpan berkas, ataupun *directory*.
- ***Inode Number*.** Merupakan nomor *inode* yang unik dalam sistem berkas.
- ***Mode*.** Menggambarkan apa yang direpresentasikan oleh *VFS inode*.
- ***User ID*.** Merupakan *identifier* bagi pemilik berkas.
- ***Time*.** Menunjukkan kapan pembuatan, modifikasi, dan penulisan suatu berkas.
- ***Blocksize*.** Menunjukkan ukuran dari *block* yang digunakan oleh berkas.
- ***Inode operations*.** Merupakan pointer ke suatu *routine* yang melakukan berbagai operasi pada *inode*.
- ***Count*.** Menunjukkan berapa kali suatu sistem telah menggunakan suatu *inode*.
- ***Lock*.** Digunakan untuk mengunci *VFS inode*.
- ***Dirty*.** Mengindikasikan bahwa telah terjadi penulisan pada *VFS inode*, sehingga perubahan yang terjadi juga harus dituliskan ke sistem berkas yang sesungguhnya.
- ***File system specific information* .** Menunjukkan informasi khusus yang dibutuhkan oleh suatu *inode*.

Gambar 21.4. Ilustrasi VFS *Inode*

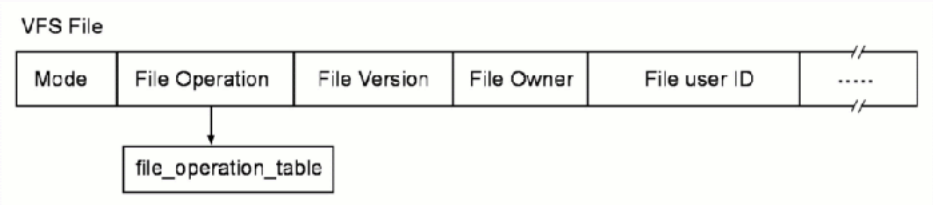


Beberapa contoh operasi yang terdapat dalam VFS *Inode* yaitu `create`, `lookup`, `mkdir`, dan `rmdir`.

VFS *File*

VFS File (berkas VFS) menyimpan informasi yang berkaitan dengan suatu *open file* dan proses. Objek ini hanya akan ada dalam VFS jika suatu proses berinteraksi dengan suatu berkas dalam disk. Beberapa data yang disimpan dalam VFS berkas adalah *mode*, *file_operation (method)*, *file_version*, *file_owner*, dan *file_userid*. Beberapa operasi dalam VFS Berkas antara lain `Read`, `Write`, `Open`, dan `Check_media_change`.

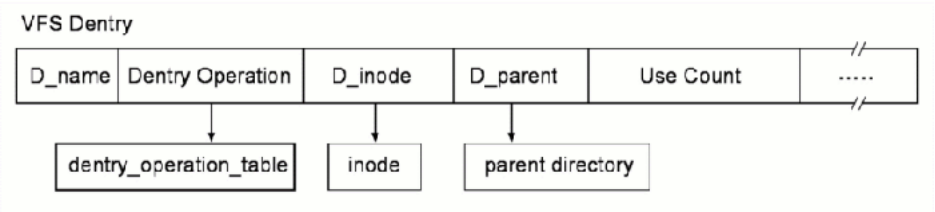
Gambar 21.5. Ilustrasi VFS *File*



VFS *Dentry*

Dentry atau *Directory Entry* merupakan suatu struktur data yang bertugas sebagai penerjemah dari suatu nama berkas ke *inode*-nya. *Dentry* disimpan dalam *chace* di VFS (*dchace*). Beberapa informasi yang disimpan dalam *dentry* adalah *name*, *pointer to inode*, *pointer to parent dentry*, *list head of children*, *chains for lots of lists*, dan *use count*. Beberapa operasi dalam VFS *dentry* antara lain `D_compare`, `D_delete`, dan `D_release`.

Gambar 21.6. Ilustrasi VFS *Dentry*



Mounting Sistem Berkas

Ketika seorang pengguna Linux me- *mounting* suatu sistem berkas pada suatu direktori, maka hal pertama yang akan dilakukan oleh VFS adalah mencari tahu apakah kernel mendukung sistem berkas tersebut. Setelah mengetahui bahwa kernel mendukung sistem berkas tersebut, maka VFS akan mencari keberadaan *superblock* dari sistem berkas tersebut agar dapat mengakses semua berkas dan direktori yang terdapat di dalamnya, tetapi jika ternyata kernel tidak mendukung sistem berkas tersebut, maka VFS akan meminta untuk memasukkan modul yang berkaitan dengan sistem berkas tersebut agar sistem berkas tersebut dapat ditangani oleh VFS. Pada kasus yang lain, jika ternyata sistem berkas tersebut telah di- *mount* pada suatu direktori tertentu dan *mount point*-nya ingin dipindahkan ke direktori lain, maka VFS akan mencari VFS *inode* dari direktori baru yang akan dijadikan tempat *mount point*. Kemudian VFS akan memeriksa apakah pada direktori tersebut tidak ditemukan sistem berkas lain yang sedang di- *mount* disana, karena sistem berkas yang berbeda tidak dapat di-*mount* pada satu direktori yang sama.

Single Virtual File System

Pada Linux, *virtual file system* diimplementasikan dengan cara membuat satu sistem berkas abstrak, dan untuk mengakses sistem berkas yang sesungguhnya hanya melalui satu berkas (*root*). *Single virtual file system* hanya mencakup semua fitur dasar dari tiap sistem berkas yang terdapat pada sistem, sehingga akses ke struktur *internal* dari tiap sistem berkas tersebut mungkin akan terbatas, terutama pada program-program yang dirancang hanya untuk memanfaatkan Single VFS dibandingkan dengan implementasi melalui *device driver* yang mengizinkan akses secara *universal*. Kekurangan lainnya adalah performa yang relatif lebih rendah dibandingkan dengan implementasi VFS lain. Performa yang rendah ini muncul karena ketika penulisan ataupun penghapusan *virtual file* yang terdapat dalam *virtual file sistem*, harus dilakukan proses *reshuffle*.

21.3. EXTFS

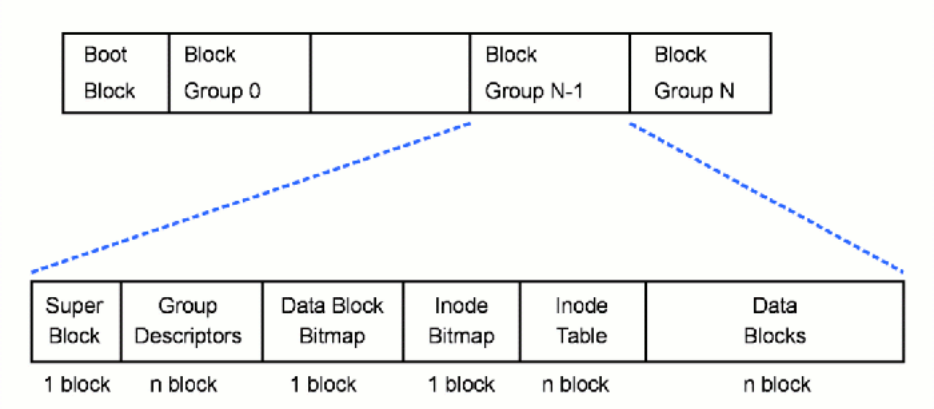
Sebelum EXTFS diperkenalkan, Linux menggunakan sistem berkas Minix, dengan beberapa keterbatasan dan performa yang kurang baik. Pada sistem berkas Minix panjang maksimal nama suatu berkas adalah 14 karakter dan ukuran berkas maksimal hanyalah 64 MBytes. Mungkin di masa itu ketika ukuran berkas masih tidak terlalu besar, 64 MBytes sudah lebih dari cukup, tetapi untuk aplikasi-aplikasi saat ini ataupun data lain yang ukurannya bisa ratusan MBytes hal ini tentu akan jadi masalah.

Untuk mengatasi permasalahan tersebut maka pada bulan April tahun 1992, diperkenalkan sistem berkas pertama yang dirancang secara khusus untuk Linux, yaitu *Extended File System* (EXTFS), dengan ukuran berkas maksimal 2 GBytes dan panjang nama suatu berkas 255 karakter. EXTFS menggunakan struktur data *linked list* untuk menandai setiap ruang yang kosong di disk, semakin

banyak data yang disimpan dalam disk tersebut, *list* menjadi tidak terurut (letak ruang yang kosong menjadi tidak berurutan), sehingga data-data menjadi terfragmentasi. Sebagai pengganti EXTFS, pada bulan Januari tahun 1993 diperkenalkanlah *Second Extended File System* (EXT2FS).

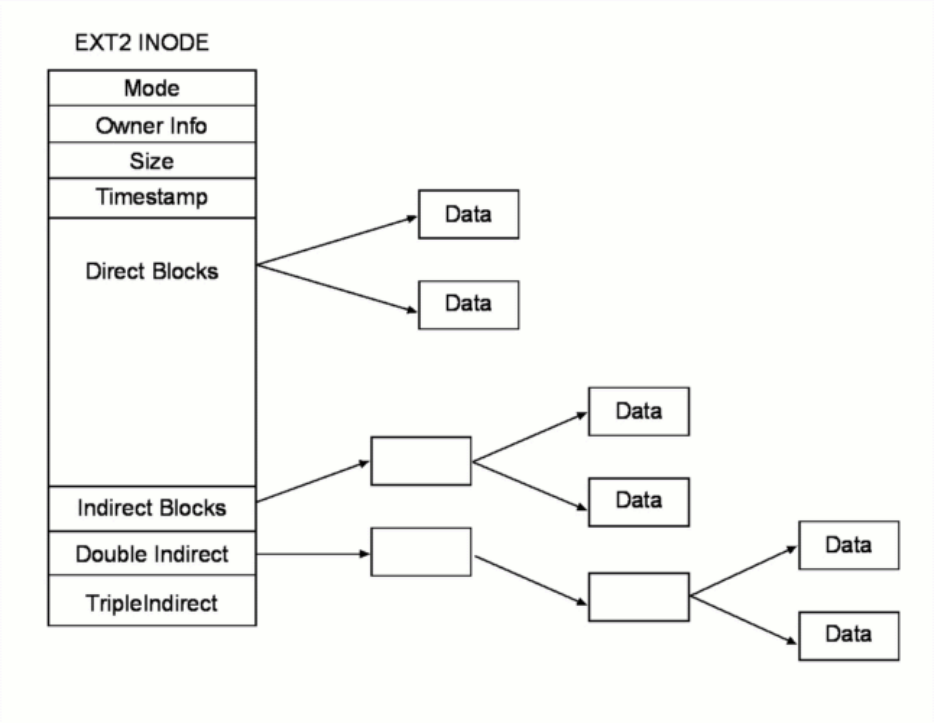
Struktur Fisik EXT2FS

Gambar 21.7. Struktur Sistem Berkas EXT2FS



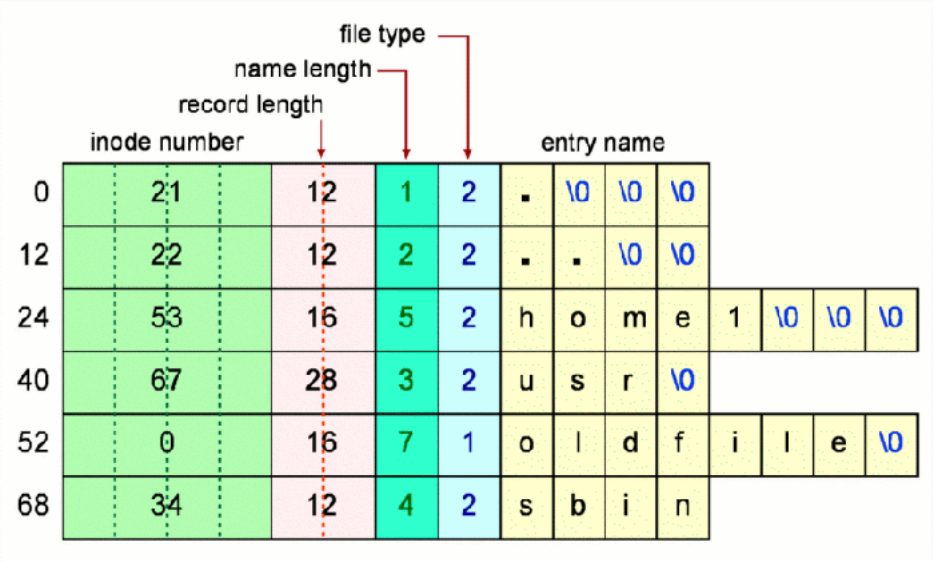
Sesuai dengan gambar struktur sistem berkas EXT2, terlihat bahwa struktur fisik EXT2FS disusun berdasarkan *block group*. Setiap *block group* berisi salinan informasi yang sangat penting mengenai sistem berkas (*Superblock* dan *Group descriptor*) dan juga berisi bagian dari sistem berkas (*block bitmap*, *inode bitmap*, *inode table*, dan *data block*). Karena informasi penting disimpan dalam masing-masing *block group*, maka jika ada *superblock* yang *corrupt*, berkas-berkas yang terdapat dalam disk tersebut dapat dengan mudah diselamatkan.

Gambar 21.8. Struktur *Inode* EXT2FS



Pada EXT2FS setiap berkas, direktori, dan berbagai data lainnya direpresentasikan dalam inode seperti pada gambar mengenai struktur *inode* EXT2FS.

Gambar 21.9. Struktur *Directory* Sistem Berkas EXT2FS



Directory dalam EXT2FS diimplementasikan sebagai suatu berkas yang berisi beberapa *entry*, setiap *entry* berisi *inode number* yang menunjuk ke tabel *inode*, *record length*, *name length*, *file type*, dan *entry name*. Tipe berkas (*file type*) menyatakan jenis berkas tersebut, yaitu:

Tabel 21.1.

Tipe Berkas	Keterangan
0	*unknown*
1	*regular file*
2	*directory*
3	*character device*
4	*block device*
5	*named pipe*
6	*socket*
7	*symbolic link*

Optimasi dan Performa EXT2FS

Pada EXT2FS beberapa optimasi dilakukan untuk mempercepat penulisan dan pembacaan data atau berkas. Pada operasi pembacaan, EXT2FS memanfaatkan suatu *buffer*. Ketika suatu *block* akan dibaca oleh sistem, sistem akan meminta beberapa *block* pada sektor selanjutnya untuk dibaca juga dan dimasukkan ke dalam suatu *buffer*, sehingga jika selanjutnya sistem ingin membaca data pada sektor selanjutnya dari *block* tersebut, data yang diinginkan telah ada di- *buffer*. Tentu ini akan menghemat waktu dalam transfer data, karena data pada *buffer* akan lebih cepat diakses dibandingkan dengan data yang ada di disk, yang mungkin akan membutuhkan waktu tambahan untuk mencari sektor yang tepat.

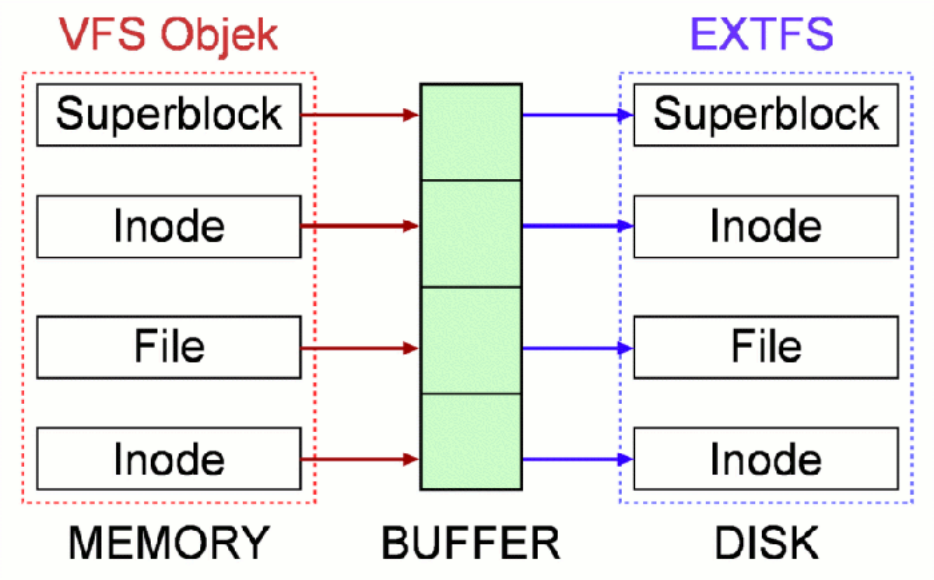
Untuk pengalokasian data atau berkas juga dilakukan optimasi seperti penggunaan *block group* untuk mengelompokkan *inode* dan berkas yang berkaitan. Hal ini akan mengurangi waktu pencarian *disk head* ketika sistem menginginkan pembacaan suatu *inode* beserta data yang bersesuaian dengan *inode* tersebut.

Optimasi lain yang dilakukan yaitu untuk penulisan berkas, ketika sistem ingin menulis data maka EXT2FS mengalokasikan 8 *block* berurutan. Pengalokasian ini dapat meningkatkan performa

dalam penulisan suatu berkas ketika berkas yang ditulis berukuran berkas. Selain itu, cara tersebut juga memungkinkan penulisan berkas dalam *block* yang terurut, sehingga meningkatkan kecepatan pembacaan karena pembacaan dilakukan secara sekuensial.

Interaksi EXTFS dengan VFS

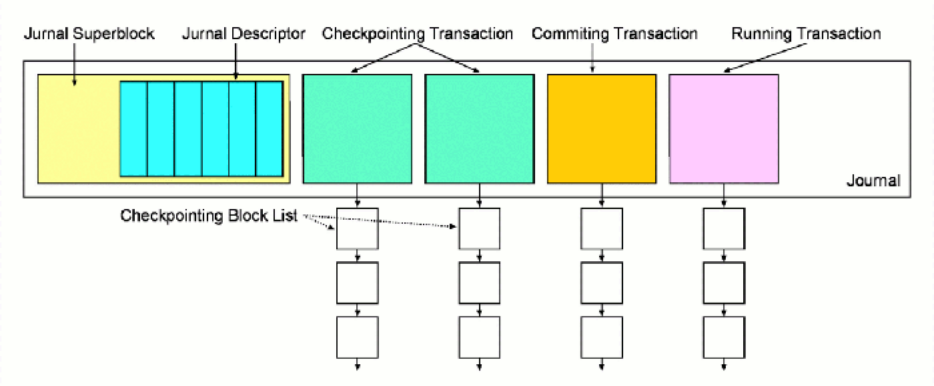
Gambar 21.10. Ilustrasi interaksi EXTFS dengan VFS



Kernel Linux berinteraksi dengan objek-objek yang terdapat di EXTFS melalui VFS. Setiap objek VFS terhubung dengan sistem berkas yang sesungguhnya, dalam hal ini EXTFS. Dalam pengaksesan data yang terdapat dalam sistem berkas, kernel akan memanfaatkan keberadaan objek di VFS yang terdapat di memori utama. Jika objek belum ditemukan, maka VFS akan melihatnya ke sistem berkas (EXTFS), lalu disalin ke memori. Biasanya objek disimpan dalam *chace* yang diimplementasikan oleh VFS, seperti *dchace* untuk menyimpan *dentry*, sehingga proses pencarian dapat dilakukan dengan lebih cepat. Lalu objek akan dihapus dari memori pada saat tertentu.

21.4. Jurnal

Gambar 21.11. Struktur *Logical* Jurnal



Transaksi adalah sekumpulan operasi yang melakukan tugas tertentu. Salah satu sifat dari transaksi adalah atomik, yang menyatakan bahwa suatu transaksi harus selesai dengan sempurna atau tidak sama sekali. Pada saat tertentu, bisa saja ketika suatu transaksi sedang melakukan perubahan data

yang terdapat dalam disk, tiba-tiba terjadi sistem *crash*. Hal ini tentu akan menyebabkan data yang terdapat dalam disk menjadi tidak konsisten. Berikut akan ditunjukkan bagaimana data dalam suatu sistem berkas menjadi tidak konsisten.

Contoh 21.1. Pembuatan Berkas Baru

Suatu transaksi yang membuat sebuah berkas, dan melalui langkah-langkah berikut:

1. Mengurangi jumlah *inode* yang dapat digunakan (*free inode*),
2. Menginisialisasi *inode* pada disk,
3. Menambah entri pada *parent* direktori dimana berkas baru tersebut dibuat.

Jika pada langkah pertama sistem *crash*, maka sistem berkas menjadi tidak konsisten, karena jumlah *inode* yang dapat digunakan (*free inode*) telah berkurang, tetapi data yang sebenarnya belum ditulis ke disk. Salah satu cara untuk mendeteksi masalah ini adalah dengan mengecek seluruh isi dari sistem berkas dengan menggunakan perintah `fsck` (*file system consistency check*). Proses pengecekan tersebut tentu akan memakan waktu yang sangat lama, jika data yang terdapat pada sistem berkas tersebut sangat banyak. Kemudian, muncul suatu ide untuk menggunakan jurnal sebagai media pencatat semua perubahan yang terjadi di dalam disk. Dengan menggunakan jurnal, konsistensi data tetap terjaga, tanpa perlu pengecekan pada keseluruhan data yang terdapat dalam suatu sistem berkas, sehingga sistem dapat dipulihkan dalam waktu yang lebih cepat.

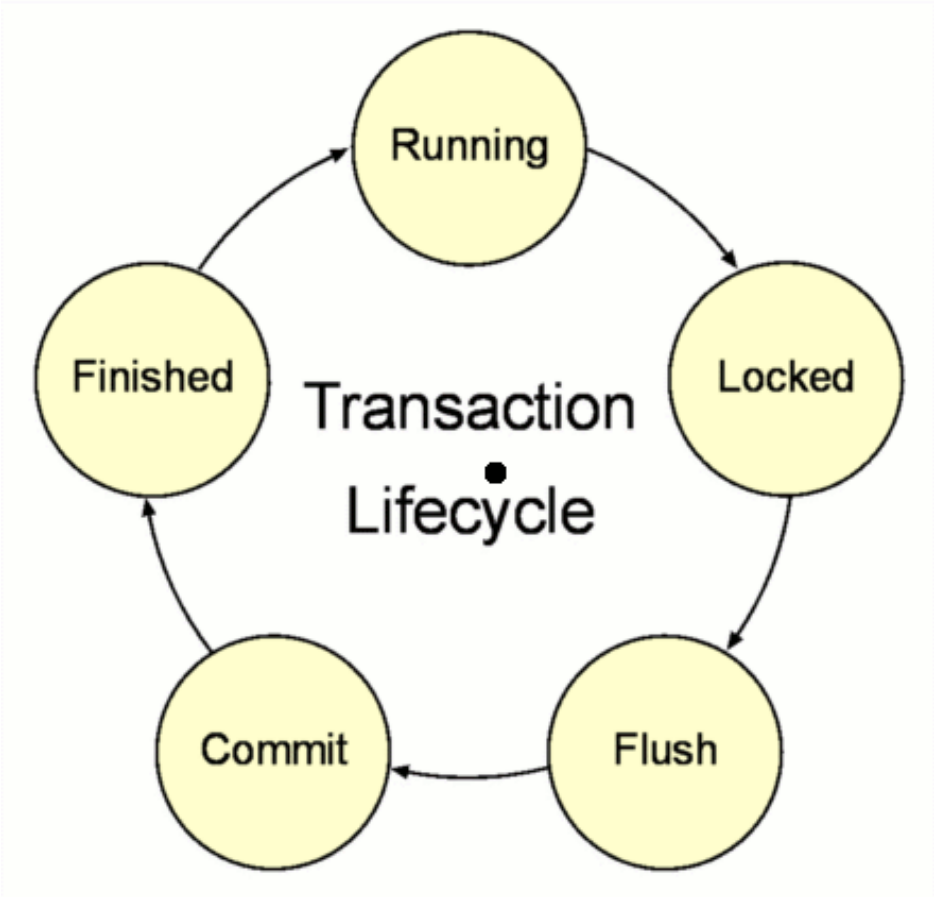
Sistem berkas berjurnal atau *journaling filesystem* menggunakan jurnal hanya sebagai tempat untuk mencatat semua informasi mengenai transaksi yang dilakukan, sedangkan disk digunakan untuk menyimpan data yang sebenarnya. Jurnal dapat saja berada pada disk yang sama dengan disk yang digunakan untuk penyimpanan data, ataupun berada pada disk yang berbeda. Beberapa sistem berkas dapat memiliki jurnalnya sendiri, tetapi dimungkinkan juga penggunaan satu jurnal bersama (*sharing journal*) untuk beberapa sistem berkas.

Journaling Block Device

Linux *Journaling Block Device* (JBD) menggunakan suatu catatan (*log*) untuk mencatat semua operasi yang mengubah konsistensi data (seperti *update*, *write*, dan sebagainya) dalam disk. Implementasi struktur data dari jurnal berbentuk seperti *circular linked list*, jadi jurnal dapat menggunakan ruang tersebut berulang-ulang jika jurnal telah penuh.

Penulisan tiap *atomic update* ke jurnal akan menyebabkan ketidakefisienan. Untuk menghasilkan performa yang lebih baik, JBD menyatukan sekumpulan *atomic update* tersebut ke dalam satu transaksi dan menulisnya ke dalam jurnal. JBD memastikan setiap transaksi adalah transaksi yang atomik. Ketika suatu transaksi diproses oleh sistem, transaksi tersebut akan melalui lima *state*.

Gambar 21.12. *Transaction State*



Keterangan:
1. ***Running*.** Transaksi sedang berjalan di sistem dan dapat menerima operasi *atomic update* lain. Hanya ada satu transaksi yang dapat berstatus running dalam sistem.
2. ***Locked*.** Transaksi tidak lagi menerima operasi atomic update, dan belum semua *atomic update* selesai dilakukan.
3. ***Flush*.** Semua *atomic update* yang terdapat dalam suatu transaksi telah selesai, sehingga transaksi dapat ditulis ke jurnal.
4. ***Commit*.** Sistem akan menulis *commit record* yang menandakan penulisan ke jurnal telah selesai.
5. ***Finished*.** Transaksi dan *commit record* telah selesai ditulis ke jurnal.

Contoh 21.2. *Transaction state*

Jika ada suatu transaksi seperti tabel berikut ini:

Tabel 21.2.

No.	***Atomic Update***
1	Write (A = 2)
2	Write (A = 3)
3	Write (A = 4)
4	Write (A = 5)

Maka transaksi tersebut akan melalui tahapan sebagai berikut:
1. Transaksi masuk *running state*. Transaksi menerima *atomic update* nomor 1, 2, 3, dan 4.
2. Transaksi berpindah ke *locked state*. Pada *state* ini transaksi tidak akan menerima *atomic update* yang baru.

3. Transaksi berpindah ke *flush state*. Semua informasi akan ditulis ke jurnal.
4. Pada tahap selanjutnya data yang terdapat di *buffer* akan ditulis ke disk. Data ditulis ke disk terlebih dahulu karena data tidak disimpan dalam jurnal. Setelah itu metadata dan juga *journal descriptor* yang menyimpan pemetaan metadata yang disimpan di jurnal ke lokasi sebenarnya di dalam disk akan ditulis ke jurnal. Jika terjadi *crash* pada tahap ini, maka sistem telah dapat dipulihkan.
5. Setelah penulisan ke jurnal selesai, maka transaksi akan berpindah ke *commit state*, dan menulis *commit record*.
6. Setelah selesai transaksi akan memasuki *finished state*. Dan dalam jangka waktu tertentu akan dilakukan *checkpoint*.

Commit dan *Checkpoint*

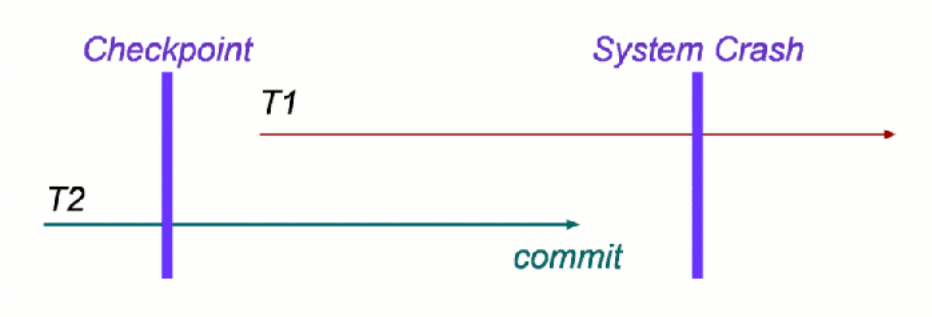
Suatu transaksi yang berjalan akan dituliskan ke disk dalam jangka waktu tertentu karena perubahan sebelumnya dilakukan hanya di *buffer/chace* yang terdapat di memori utama. Suatu proses yang menulis perubahan data ke disk sekaligus memberi tanda bahwa seluruh atau sebagian transaksi tersebut telah selesai disebut *transaction commit*.

Suatu jurnal hanya memiliki ruang yang terbatas untuk mencatat semua transaksi yang terjadi dalam sistem. Oleh karena itu, perlu adanya suatu mekanisme penggunaan kembali ruang tersebut ketika jurnal telah penuh. Untuk transaksi-transaksi yang telah menuliskan semua bloknya ke dalam disk, tidak perlu lagi ditulis ke jurnal. Suatu proses yang menulis transaksi yang telah selesai ke disk dan menandakan kembali tempat yang sesuai sebagai ruang kosong di jurnal disebut *transaction checkpoint*.

Proses *Recovery*

Pada saat sistem *crash*, sistem akan mengecek dalam jurnal dari saat terakhir *checkpoint* hingga ditemukan *commit record*. Jika tidak ditemukan *commit record* pada suatu transaksi, sistem akan mengabaikan transaksi tersebut karena data yang sebenarnya belum ditulis ke disk, namun jika sistem menemukan *commit record* pada transaksi tersebut, sistem akan melakukan proses *redo*.

Gambar 21.13. Proses *Recovery*



Berdasarkan gambar 21.13, dapat dilihat bahwa T1 belum mencapai *commit point* ketika sistem *crash*, tetapi pada T1 tidak akan dilakukan proses *undo* karena proses penulisan data masih belum dilakukan pada data yang sebenarnya. Sedangkan T2 yang telah mencapai *commit point* akan mengalamai proses *redo*, untuk memastikan data telah sempurna ditulis ke disk.

Keberadaan jurnal menjamin konsistensi data yang terdapat dalam sistem berkas. Waktu yang diperlukan untuk melihat jurnal dan melakukan proses *redo* tentu lebih cepat dibandingkan jika kita harus memeriksa keseluruhan data yang terdapat di disk dengan perintah `fsck`.

Journaling File System

Salah satu sistem berkas Linux yang mendukung journaling adalah *Third Extended File System* (EXT3FS). EXT3FS sebenarnya adalah EXT2FS yang dilengkapi dengan fitur jurnal dan beberapa

fitur lain. Salah satu keuntungan menggunakan sistem berkas ini adalah kemudahaannya dalam mengubah EXT2FS menjadi EXT3FS tanpa perlu mem- *back up* data dan me- *restore* data. Jurnal pada EXT3FS disimpan pada salah satu *inode* khusus dan dimungkinkan juga penggunaan satu jurnal untuk lebih dari satu sistem berkas (*sharing journal*).

21.5. Sistem Berkas /proc/

Linux *Process File System* atau Sistem berkas /proc adalah suatu sistem berkas semu yang digunakan untuk mengakses informasi mengenai proses dari kernel. Sistem berkas ini biasanya di-*mount* di bawah *root* dengan alamat /proc. Sistem berkas /proc tidak berisi berkas sebenarnya, tetapi berisi informasi tentang sistem, seperti sistem memori, konfigurasi perangkat keras, dan sebagainya. Untuk alasan inilah sistem berkas /proc dapat dianggap sebagai suatu kontrol dan pusat informasi untuk kernel. Pada kenyataannya, banyak perintah dalam sistem Linux yang mengambil informasi yang terdapat pada berkas dalam /proc, contohnya perintah lsmod (*list modules*) sama dengan perintah cat /proc/modules dan lspci (*list PCI bus*) sama dengan perintah cat /proc/pci. Karena sistem berkas /proc adalah VFS dan tidak berada dalam disk, tetapi dalam memori, maka setiap kali komputer dinyalakan sistem berkas /proc yang baru akan dibuat.

Jika diambil sebagian *list directory* dari *root*, maka akan muncul tampilan sebagai berikut.

```
drwxr-xr-x 14 root root 291 Oct 25 18:47 opt

dr-xr-xr-x 86 root root 0 May 09 2007 proc

drwx--x--x 16 root root 841 Nov 20 00:10 root

drwxr-xr-x 5 root root 4627 Oct 15 11:42 sbin
```

Dapat dilihat bahwa ukuran dari *directory* /proc adalah nol dan tanggal terakhir modifikasinya adalah tanggal kapan komputer dinyalakan. Hal ini menunjukkan bahwa sistem berkas /proc tidak berada pada disk tetapi terdapat pada memori utama komputer. Selain itu waktu modifikasi yang selalu berubah menunjukkan bahwa isi /proc selalu diperbarui oleh sistem.

Isi dari *directory* /proc

Jika diambil sebagian daftar berkas dan *subdirectory* yang terdapat dalam /proc, maka dapat dikelompokan menjadi dua bagian, yaitu:

1. ***Directory* yang namanya adalah bilangan .** Setiap *directory* yang namanya adalah bilangan, sebenarnya adalah proses yang sedang berjalan di sistem, angka tersebut menunjukkan *proses ID* (PID). Jika dicocokan dengan tabel proses maka akan ditemukan PID yang sesuai dengan semua nomor yang terdapat pada nama directory tersebut.
2. **Berkas yang namanya adalah *string* .**

Beberapa contoh directory yang namanya adalah bilangan/nomor : 1, 2, 3, 100, 109, 200, 462, 480, 495, 560, 570, 670, 687, 698, 777, 1002, 1200, 1302, 1666, 2000, 3005, 3444, 3455, 3566, 3766, 3877, dan sebagainya.

Isi dari tiap direktori tersebut di ataranya ditunjukkan pada tabel berikut.

Tabel 21.3.

Nama Berkas yang terdapat dalam /proc/PID	Keterangan
cmdline	*Command line arguments*
cwd	*Link to the current working directory*
environ	Nilai dari *environment variables* sistem
exe	*Link to the executable of this process*

Nama Berkas yang terdapat dalam /proc/PID	Keterangan
fd	Berisi semua *file descriptors*
maps	*Memory maps to executables and library files*
mem	Memori yang dipakai oleh proses ini
root	Pointer ke direktori *root*
stat	Status dari proses ini
statm	*Process memory status information*
status	*Process status in human readable*

Isi dari tiap direktori di atas tidak lain adalah Linux *Process Control Block* yang direalisasikan ke dalam bentuk struktur direktori.

Penjelasan mengenai beberapa berkas yang namanya berupa string ditunjukkan pada tabel berikut.

Tabel 21.4.

Nama Directory	Keterangan
/proc/apm	Informasi mengenai *Advanced Power Management*.
/proc/bus	*Directory* yang berisi informasi *bus* secara khusus.
/proc/cmdline	Kernel *command line*.
/proc/cpuinfo	Informasi mengenai prosesor (tipe, model, performa).
/proc/devices	Daftar dari *device driver* yang dipakai oleh kernel.
/proc/dma	Menunjukkan jalur DMA yang sedang digunakan pada masa tertentu.
/proc/driver	Informasi mengenai berbagai *driver* berada di sini.
/proc/fb	*Frame buffer devices*.
/proc/filesystems	Daftar sistem berkas yang didukung oleh kernel.
/proc/fs	*File System parameter*.
/proc/ide	Berisi informasi mengenai semua IDE *device*.
/proc/interrupt	Menunjukkan *interrupt* yang sedang dijalankan.
/proc/ioport	Menunjukkan *port* I/O yang sendang digunakan.
/proc/kmsg	Pesan ynag dikeluarkan oleh kernel.
/proc/ksyms	Kernel simbol tabel.
/proc/loadavg	Tiga indikator kerja yang telah dilakukan oleh sistem selama 1, 5, dan 15 menit.
/proc/lock	Kernel *lock*.
/proc/modules	Menunjukkan modul-modul yang dimasukkan ke dalam kernel.
/proc/mounts	Sistem berkas yang telah di- *mount*.
/proc/partitions	Daftar partisi yang dikenali pada suatu sistem.
/proc/pci	Informasi mengenai PCI bus.
/proc/rtc	*Real time clock*.

Nama Directory	Keterangan
`/proc/scsi`	Infomasi mengenai semua SCSI *device* (jika ada).
`/proc/swaps`	Penggunaan ruang *swap*.
`/proc/sys`	Sumber informasi yang juga dapat mengubah parameter di dalam kernel tanpa me- *restart* ulang sistem.
`/proc/version`	Versi kernel Linux yang ada.

21.6. Rangkuman

Linux adalah sistem yang mendukung banyak sistem berkas. Untuk mengatasi penggunaan banyak sistem berkas dalam satu sistem, dibuatlah satu lapisan abstrak di atas beberapa sistem berkas yang berbeda tersebut yang dikenal dengan *Virtual File System* (VFS), yang menyembunyikan detil implementasi dari suatu sistem berkas, sehingga setiap aplikasi dapat melakukan operasi pada berbagai sistem berkas tersebut dengan cara yang sama atau seragam.

Penggunaan awalnya Linux menggunakan sistem berkas Minix, kemudian digantikan oleh sistem berkas yang diciptakan khusus untuk Linux, yaitu EXTFS (*Extended File System*). Karena beberapa performa EXTFS yang masih kurang memuaskan, kemudian muncul penyempurnaan EXTFS yang dikenal dengan EXT2FS (*Second Extended File System*). Sebagai sistem berkas *standard* yang digunakan hampir di semua distribusi Linux, EXT2FS melakukan beberapa optimasi dibandingkan penduhulunya dan memiliki performa yang cukup baik dibanding sistem berkas yang ada pada saat itu.

Metode untuk menjaga konsistensi data ketika sistem *crash* ternyata masih belum didukung oleh EXT2FS. Oleh karena itu munculah *Journaling File System*. Pada sistem berkas yang memanfaatkan jurnal, semua data yang dituliskan ke disk, harus terlebih dahulu ditulis ke dalam jurnal, sehingga ketika sistem *crash*, informasi yang tersisa pada jurnal dapat membantu memulihkan dan menjaga konsistensi data yang terdapat dalam disk. Salah satu sistem berkas yang mendukung penggunaan jurnal adalah *Third Extended File System* (EXT3FS).

Sistem berkas /proc adalah sistem berkas istimewa yang menjadi bagian dari VFS Linux. Isi dari sistem berkas ini selalu diperbaharui setiap saat ketika komputer dalam keadaan menyala dan diinisialisasi ketika *boot*. Berbagai perintah Linux juga memanfaatkan sistem berkas `/proc` untuk mendapatkan informasi dari kernel. Contohnya adalah perintah `top` yang membaca isi dari berkas dan *directory* yang terdapat dalam `/proc` untuk menampilkan proses yang terdapat di sistem.

Rujukan

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

[WEBDavid1999] David A. Rusling. 1999. *http://www.tldp.org/LDP/tlk/fs/filesystem.html* . Diakses 03 Mei 2007.

[WEBWiki2007a] Wikipedia. 2007. *http://en.wikipedia.org/wiki/Ext2fs* . Diakses 03 Mei 2007.

[WEBWiki2007b] Wikipedia. 2007. *http://en.wikipedia.org/wiki/Procfs* . Diakses 03 Mei 2007.

[WEBWiki2007c] Wikipedia. 2007. *http://en.wikipedia.org/wiki/Journaling_file_system* . Diakses 03 Mei 2007.

[WEBUbuntu2007a] Ubuntu. 2007. *http://www.ubuntu-id.org/doc/edgy/xubuntu/ desktopguide/id/linux-basics.html* . Diakses 03 Mei 2007.

[WEBUbuntu2007b] Ubuntu. 2007. *http://www.ubuntu-id.org/doc/edgy/ubuntu/ desktopguide/id/directories-file-systems.html* . Diakses 03 Mei 2007.

[WEBLinuxSecurity2007c] LinuxSecurity. 2007. *http://www.linux-security.cn/ebooks/ ulk3-html/0596005652/ understandlk-CHP-12.html* . Diakses 17 Mei 2007.

[WEBLinuxSecurity2007] LinuxSecurity. 2007. *http://www.linux-security.cn/ebooks/ ulk3-html/0596005652/ understandlk-CHP-18.html* . Diakses 17 Mei 2007.

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 22. Keamanan Sistem

22.1. Pendahuluan

Pada dasarnya seorang pengguna komputer sangat membutuhkan rasa kenyamanan ketika sedang mengoperasikannya. Kenyamanan tersebut dapat diperoleh salah satunya dari keamanan sistem yang dipakai. Berbicara mengenai keamanan sistem, ada dua hal yang sering diperdebatkan yaitu mengenai istilah keamanan dan proteksi. Pertama-tama kita harus bisa membedakan antara keamanan dengan proteksi. Proteksi biasanya menyangkut faktor -faktor internal sistem yang ada di dalam komputer. Sebenarnya tujuan dari proteksi adalah untuk mencegah penggunaan akses-akses yang tidak seharusnya (*accidental access*). Akan tetapi 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 dibandingkan dengan proteksi.

Dengan mempertimbangkan aspek-aspek di atas maka dibutuhkanlah suatu keamanan sistem untuk menanggulangi kemungkinan akses data penting (rahasia) dari orang-orang yang bukan seharusnya mengakses. Namun, dalam kenyataannya tidak ada suatu sistem komputer yang memiliki sistem keamanan sempurna. Akan tetapi, setidaknya kita mempunyai suatu mekanisme tersendiri untuk mencegah ataupun mengurangi kemungkinan-kemungkinan gangguan terhadap keamanan sistem.

Di dalam bab ini kita akan membahas hal-hal yang menyangkut keamanan sistem komputer. Dengan mempelajari bab ini diharapkan nantinya kita menjadi tahu tentang keamanan sistem komputer dan penangananya ketika terjadi suatu gangguan terhadap keamanan suatu sistem komputer.

22.2. Masyarakat dan Etika

Berbicara mengenai manusia dan etika, kita mengetahui bahwa di lingkungan kita 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 diizinkan (*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.

22.3. Kebijaksanaan Keamanan

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.

Ada tiga aspek dalam mekanisme pengamanan (proteksi):
- Identifikasi user (Autentikasi), bertujuan untuk mengetahui siapa yang sedang melakukan sesuatu di dalam sistem. Sebagai contoh, kita bisa mengetahui bahwa user A sedang membuka aplikasi web browser dan mengakses situs detik.com.
- Penentuan autorisasi, sistem harus dapat mengetahui siapa user yang sedang aktif dan apa saja yang boleh ia lakukan.
- Pemakaian akses, harus dipastikan bahwa tidak terjadi penerobosan akses di dalam sistem.

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.

22.4. 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? Maksudnya adalah akses-akses yang penting seperti menginstall software hanya dimiliki oleh admin saja. User biasa tidak boleh memiliki akes ini.
2. Apakah mereka memang berhak? Hal ini juga penting untuk menjamin keamanan sistem dari orang-orang yang bermaksud tidak baik. Pembagian hak akses disesuaikan dengan kebutuhan pengguna.
3. Dapatkah sistem terlindung dari maksud dan tujuan mereka? Sebagai contoh adalah perlindungan terhadap pengaksesan data (baik untuk membaca ataupun menulis) yang ada di sistem, hanya orang-orang tertentu saja yang memilikinya.
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.

22.5. Keamaanan 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.

22.6. Keamaanan 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 tersebut 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.

22.7. 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 privasi: mencegah informasi menyebar luas tanpa izin. Akan tetapi, privaci bukan satu-satunya layanan yang disediakan kriptografi. Kriptografi dapat juga digunakan untuk mendukung *authentication* (memverifikasi identitas pengguna) 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 kunci yang secara selektif telah disebar pada komputer-komputer yang berada dalam satu jaringan dan digunakan untuk memroses suatu pesan.

22.8. 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 system 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 temasuki 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).

22.9. 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.

22.10. 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 and 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.* Menentukan bagaimana cara memeriksa 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

22.11. 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

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

[WEBWiki2007] Wikipedia. 2007. *Network Security – http://en.wikipedia.org/wiki/Network_security* . Diakses 9 Mei 2007.

[WEBWiki2007] Wikipedia. 2007. *Bussiness Continuity Plan – http://en.wikipedia.org/wiki/Business_continuity_planning* . Diakses 9 Mei 2007.

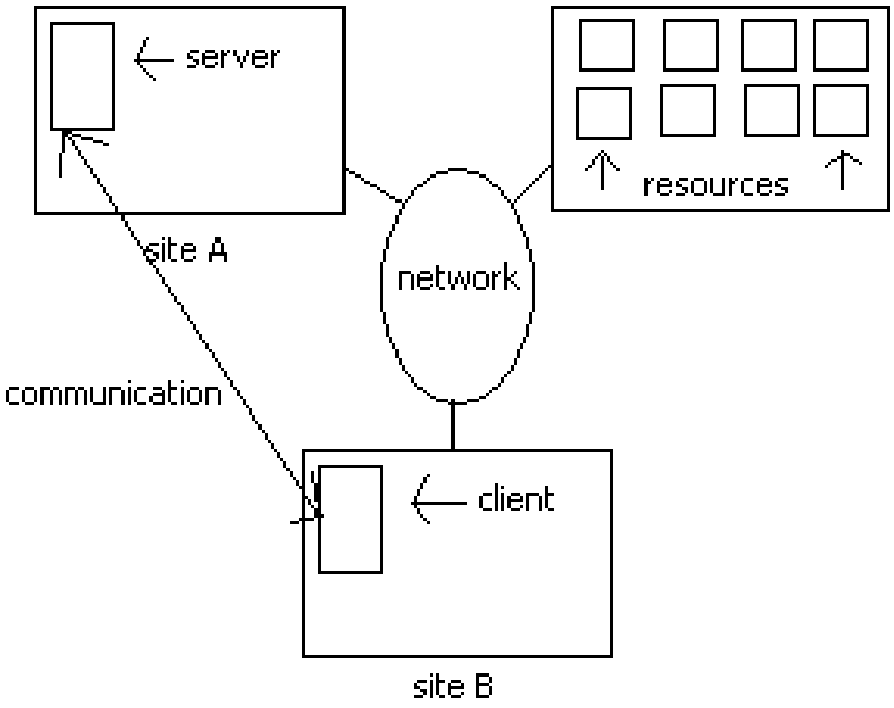
[WEBUniversityOfWisconsin2007] Barton P. Miller . 2007. *Protection and Security – http://www.cs.wisc.edu/~bart/537/lecturenotes/s29.html* . Diakses 9 Mei 2007.

Bab 23. Sistem Terdistribusi

23.1. Pendahuluan

Sistem terdistribusi adalah sekumpulan prosesor yang tidak saling berbagi memori atau *clock* dan terhubung melalui jaringan komunikasi yang bervariasi, yaitu melalui *Local Area Network* ataupun melalui *Wide Area Network*. Prosesor dalam sistem terdistribusi bervariasi, dapat berupa *small microprocessor*, *workstation*, *minicomputer*, dan lain sebagainya. Berikut adalah ilustrasi struktur sistem terdistribusi:

Gambar 23.1. Struktur Sistem Terdistribusi



Karakteristik sistem terdistribusi adalah sebagai berikut:

1. **Concurrency of components.** Pengaksesan suatu komponen/sumber daya (segala hal yang dapat digunakan bersama dalam jaringan komputer, meliputi H/W dan S/W) secara bersamaan. Contoh: Beberapa pemakai *browser* mengakses halaman *web* secara bersamaan
2. **No global clock.** Hal ini menyebabkan kesulitan dalam mensinkronkan waktu seluruh komputer/ perangkat yang terlibat. Dapat berpengaruh pada pengiriman pesan/data, seperti saat beberapa proses berebut ingin masuk ke *critical session*.
3. **Independent failures of components.** Setiap komponen/perangkat dapat mengalami kegagalan namun komponen/perangkat lain tetap berjalan dengan baik.

Ada empat alasan utama untuk membangun sistem terdistribusi, yaitu:

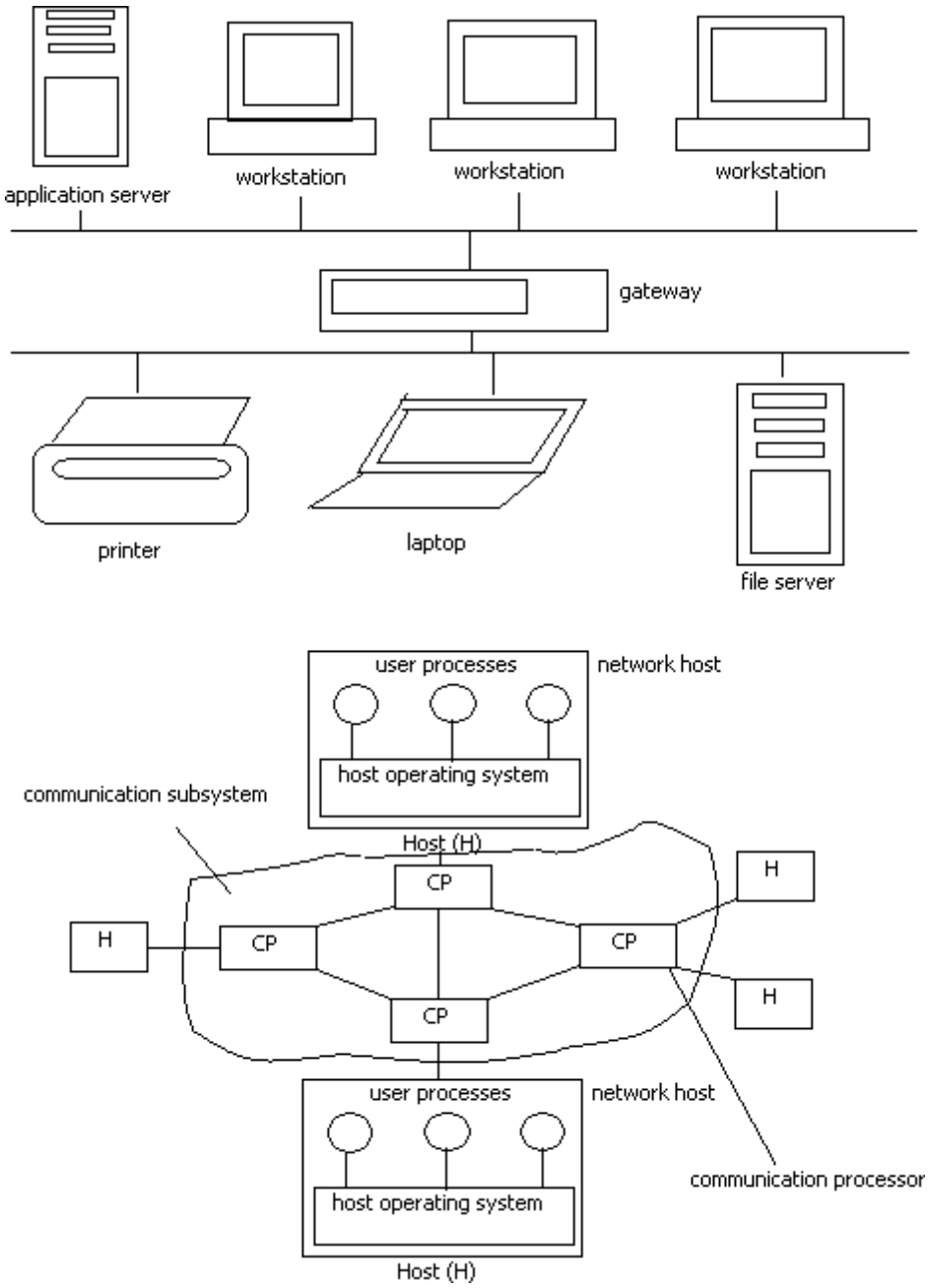
1. **Resource Sharing.** Dalam sistem terdistribusi, situs-situs yang berbeda saling terhubung satu sama lain melalui jaringan sehingga situs yang satu dapat mengakses dan menggunakan sumber daya yang terdapat dalam situs lain. Misalnya, *user* di situs A dapat menggunakan *laser printer* yang dimiliki situs B dan sebaliknya *user* di situs B dapat mengakses *file* yang terdapat di situs A.
2. **Computation Speedup.** Apabila sebuah komputasi dapat dipartisi menjadi beberapa subkomputasi yang berjalan bersamaan, maka sistem terdistribusi akan mendistribusikan subkomputasi tersebut ke situs-situs dalam sistem. Dengan demikian, hal ini meningkatkan kecepatan komputasi (*computation speedup*).

3. **Reliability.** Dalam sistem terdistribusi, apabila sebuah situs mengalami kegagalan, maka situs yang tersisa dapat melanjutkan operasi yang sedang berjalan. Hal ini menyebabkan reliabilitas sistem menjadi lebih baik.
4. **Communication.** Ketika banyak situs saling terhubung melalui jaringan komunikasi, *user* dari situs-situs yang berbeda mempunyai kesempatan untuk dapat bertukar informasi.

Tantangan-tantangan yang harus dipenuhi oleh sebuah sistem terdistribusi:

1. **Keheterogenan perangkat/multiplisitas perangkat.** Suatu sistem terdistribusi dapat dibangun dari berbagai macam perangkat yang berbeda, baik sistem operasi, H/W maupun S/W.
2. **Keterbukaan.** Setiap perangkat memiliki antarmuka (*interface*) yang di-publish ke komponen lain. Perlu integrasi berbagai komponen yang dibuat oleh *programmer* atau vendor yang berbeda
3. **Keamanan.** *Shared resources* dan transmisi informasi/data perlu dilengkapi dengan enkripsi.
4. **Penangan kegagalan.** Setiap perangkat dapat mengalami kegagalan secara independen. Namun, perangkat lain harus tetap berjalan dengan baik.
5. **Concurrency of components.** Pengaksesan suatu komponen/sumber daya secara bersamaan oleh banyak pengguna.
6. **Transparansi.** Bagi pemakai, keberadaan berbagai perangkat (multiplisitas perangkat) dalam sistem terdistribusi tampak sebagai satu sistem saja.

Gambar 23.2. ***Local Area Network***



Dalam sistem operasi terdistribusi, *user* mengakses sumber daya jarak jauh (*remote resources*) sama halnya dengan mengakses sumber daya lokal (*local resources*). Migrasi data dan proses dari satu situs ke situs yang lain dikontrol oleh sistem operasi terdistribusi.

Berikut ini adalah fitur-fitur yang didukung oleh sistem operasi terdistribusi:

1. **Data Migration.** Misalnya, *user*di situs A ingin mengakses data di situs B. Maka, transfer data dapat dilakukan melalui dua cara, yaitu dengan mentransfer keseluruhan data atau mentransfer sebagian data yang dibutuhkan untuk *immediate task.*
2. **Computation Migration.** Terkadang, kita ingin mentransfer komputasi, bukan data. Pendekatan ini yang disebut dengan *computation migration*
3. **Process Migration.** Ketika sebuah proses dieksekusi, proses tersebut tidak selalu dieksekusi di situs di mana ia pertama kali diinisiasi. Keseluruhan proses, atau sebagian daripadanya, dapat saja dieksekusi pada situs yang berbeda. Hal ini dilakukan karena beberapa alasan: *Load balancing.* Proses atau subproses-subproses didistribusikan ke jaringan untuk memeratakan beban

kerja. *Computation speedup.* Apabila sebuah proses dapat dibagi menjadi beberapa subproses yang berjalan bersamaan di situs yang berbeda-beda, maka total dari *process turnaround time* dapat dikurangi. *Hardware preference.* Proses mungkin mempunyai karakteristik tertentu yang menyebabkan proses tersebut lebih cocok dieksekusi di prosesor lain.Misalnya, proses inversi matriks, lebih cocok dilakukan di *array processor* daripada di *microprocessor Software preference.* Proses membutuhkan *software* yang tersedia di situs lain, di mana *software* tersebut tidak dapat dipindahkan atau lebih murah untuk melakukan migrasi proses daripada *software Data access.*

Sistem operasi terdistribusi (*distributed operating system*) menyediakan semua fitur di atas dengan kemudahan penggunaan dan akses dibandingkan dengan sistem operasi jaringan (*network operating system*).

Berikut adalah dua tipe jaringan yang dipakai dalam sistem terdistribusi:

- **Local Area Network (LAN).** LAN muncul pada awal tahun 1970-an sebagai pengganti dari sistem komputer *mainframe*. LAN, didesain untuk area geografis yang kecil. Misalnya, LAN digunakan untuk jaringan dalam sebuah bangunan atau beberapa bangunan yang berdekatan. Umumnya, jarak antara situs satu dengan situs yang lain dalam LAN berdekatan. Oleh karena itu, kecepatan komunikasinya lebih tinggi dan peluang terjadi kesalahan (*error rate*) lebih rendah. Dalam LAN, dibutuhkan *high quality cable* supaya kecepatan yang lebih tinggi dan reliabilitas tercapai. Jenis kabel yang biasanya dipakai adalah *twisted-pair* dan *fiber-optic*. Berikut adalah ilustrasi dari *Local Area Network*:
- **Wide Area Network.** WAN muncul pada akhir tahun 1960-an, digunakan sebagai proyek riset akademis agar tersedia layanan komunikasi yang efektif antara situs, memperbolehkan berbagi *hardware* dan *software* secara ekonomis antar pengguna. WAN yang pertama kali didesain dan dikembangkan adalah *Arpanet* yang pada akhirnya menjadi cikal bakal dari *Internet*. Situs-situs dalam WAN tersebar pada area geografis yang luas. Oleh karena itu, komunikasi berjalan relatif lambat dan reliabilitas tidak terjamin. Hubungan antara *link* yang satu dengan yang lain dalam jaringan diatur oleh *communication processor*. Berikut adalah ilustrasi dari *Wide Area Network*

23.2. Topologi Jaringan

Situs-situs dalam sistem terdistribusi dapat terhubung melalui berbagai macam cara yang ditentukan berdasarkan kriteria-kriteria sebagai berikut:

- **Biaya instalasi.** Biaya menghubungkan situs-situs dalam sistem.
- **Biaya komunikasi.** Besar waktu dan uang untuk mengirimkan pesan dari satu situs ke situs lainnya.
- **Ketersediaan/availabilitas.** Sampai sejauh mana data dapat diakses walaupun terdapat kegagalan pada beberapa *link* atau situs.

Macam-macam topologi jaringan antara lain:

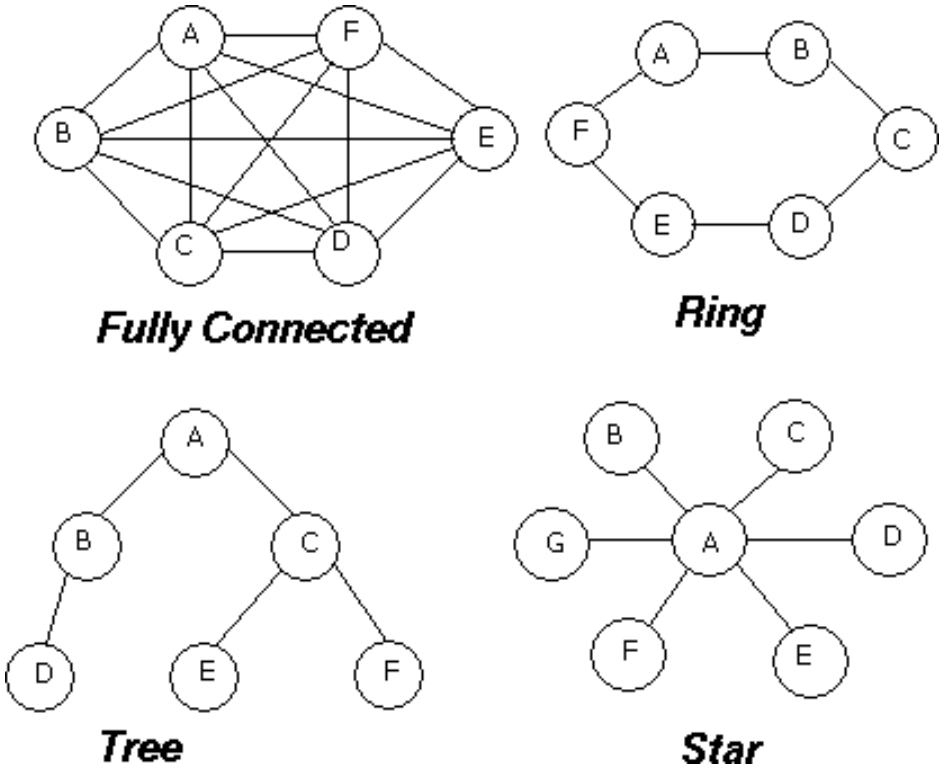
Fully Connected Network

Tiap situs dalam *Fully Connected Network* terkoneksi secara langsung dengan situs lainnya. *Link* yang ada menjadi banyak dan menyebabkan biaya instalasi besar. Topologi jenis ini tidak praktis untuk diterapkan dalam sistem yang besar.

Partially Connected Network

Link yang ada hanya antara beberapa situs sehingga biaya instalasi menjadi lebih rendah. Namun, biaya komunikasi bisa menjadi lebih mahal. Misalkan, situs A ingin mengakses data di situs E, maka jalan yang ditempuh menuju situs E harus melalui situs B terlebih dahulu karena tidak ada *link* langsung dari situs A ke situs E. Semakin jauh jalan yang ditempuh, biaya komunikasi semakin mahal. Selain itu, availibilitas atau ketersediaan data kurang baik dibandingkan dengan *Fully Connected Network*. Misalkan, jika terjadi *failure site* atau kegagalan situs di C maka akses ke situs F menjadi tidak ada.

Gambar 23.3. ***Model Network***



Partially Connected Network terdiri dari:

- **Tree-structured network.** Biaya instalasi dan komunikasi pada topologi jenis ini biasanya rendah. Namun, jika terjadi *failure link* atau *failure site* maka pengaksesan data menjadi terhambat dan mengakibatkan availibilitas/ketersediaan menjadi rendah.
- **Star network.** Biaya komunikasi rendah karena setiap situs paling banyak mengakses dua *link* ke situs lain. Namun, bila terjadi *failure site* di situs pusat maka setiap situs tidak akan dapat mengakses situs lainnya sehingga availibilitas/ketersediaan pada topologi jenis *star network* rendah.
- **Ring network.** Biaya komunikasi tinggi karena jika ingin mengakses sebuah situs bisa jadi harus menempuh banyak *link*. Misalnya, dari situs A menuju situs D, *link* yang dilewati sebanyak tiga buah. Availibilitas/ketersediaan pada topologi *ring network* lebih terjamin dibandingkan pada *star network* maupun pada *tree-structured network*.

Selain topologi jaringan, ada beberapa hal lain yang harus dipertimbangkan dalam mendesain jaringan:

1. **Naming and name resolution.** Bagaimana dua buah proses menempatkan/memposisikan satu sama lain dalam jaringan komunikasi.
2. **Routing strategies.** Bagaimana pesan dikirimkan melalui jaringan.
3. **Packet strategies.** Apakah paket dikirimkan secara individual atau sekuensial.
4. **Connection strategies.** Bagaimana dua proses mengirimkan pesan secara sekuensial.
5. **Contention.** Bagaimana memecahkan masalah permintaan yang mengalami konflik.

Perlu diketahui, bahwa dalam sistem terdistribusi terdapat berbagai macam kegagalan perangkat keras (*hardware failure*) seperti kegagalan link atau *failure link*, kegagalan situs atau *failure site*, dan kehilangan pesan atau *loss of message*.Oleh karena itu, untuk menjamin kekuatan sistem atau disebut juga *robustness* maka sistem terdistribusi harus mampu melakukan pendeteksian kegagalan, mereparasi sistem, dan mengkonfigurasinya kembali.

23.3. Sistem Berkas Terdistribusi

Sistem berkas terdistribusi adalah sebuah sistem di mana banyak pengguna dapat berbagi berkas dan sumber daya penyimpanan. *Client*, *server*, dan media penyimpanan dalam sistem terdistribusi tersebar pada perangkat-perangkat yang terdapat dalam sistem terdistribusi. *Service* dijalankan melalui

jaringan. Konfigurasi dan implementasi dari sistem berkas terdistribusi bervariasi dari sistem yang satu ke sistem yang lain.

Idealnya, sistem berkas terdistribusi tampil di depan pengguna atau *client* sebagai sistem berkas yang konvensional dan terpusat. Di mana keberagaman atau multiplisitas perangkat dibuat tidak tampak sehingga *client interface* dalam sistem berkas terdistribusi tidak dibedakan antara *local file* dan *remote file*. Sistem berkas terdistribusi yang transparan juga akan memfasilitasi mobilitas pengguna dengan membawa lingkungan pengguna, yang dimaksudkan adalah *home directory*, ke mana saja pengguna itu *login*.

Dalam sistem berkas konvensional dan terpusat, waktu yang diperlukan untuk memenuhi permintaan adalah waktu akses disk dan sedikit waktu untuk CPU *processing*. Sedangkan dalam sistem berkas terdistribusi, waktu yang diperlukan untuk memenuhi permintaan meningkat akibat *remote access* yang menambah waktu pengiriman permintaan ke *server* dan waktu penerimaan respon oleh *client*. Selain itu, dalam transfer informasi, ada tambahan waktu untuk menjalankan *software* untuk protokol komunikasi.

Dalam pengaksesan *remote file* atau *remote file access* (RFA) di dalam sistem berkas terdistribusi terdapat dua metode:

1. **Dengan *remote service*.** Permintaan akses data dikirimkan ke *server*. *Server* melakukan akses ke data dan hasilnya di-*forward* kembali ke *client*.
2. **Dengan *caching*.** Bila data yang dibutuhkan belum disimpan di *cache* maka salinan data akan dibawakan dari *server* ke *client*. Idenya adalah untuk menahan data yang baru saja diakses di *cache* sehingga akses yang berulang ke informasi yang sama dapat ditangani secara lokal. Dengan demikian, dapat mengurangi *network traffic*. Namun, masalah yang timbul adalah mengenai konsistensi *cache*, di mana seharusnya salinan *cache* tetap konsisten dengan *file-master*-nya. Dalam sistem berkas terdistribusi, replikasi berkas pada perangkat yang berbeda adalah redundansi yang berguna untuk meningkatkan availibilitas atau ketersediaan. Syarat mendasar untuk replikasi berkas adalah replika dari berkas yang sama terletak pada perangkat yang *failure-independent* sehingga ketersediaan satu replika tidak dipengaruhi oleh ketersediaan replika yang lain. Masalah utama dalam replikasi adalah *updating*. Proses *update* pada satu replika harus dilakukan juga pada replika yang lain.

23.4. Rangkuman

Sistem terdistribusi adalah sekumpulan prosesor yang tidak saling berbagi memori atau *clock*. Setiap prosesornya memiliki memori lokal tersendiri dan berkomunikasi satu sama lain melalui jaringan komunikasi, seperti LAN atau WAN. Secara umum, topologi jaringan ada dua macam, yaitu *fully connected network* dan *partially connected network* yang terbagi lagi menjadi tiga jenis, yaitu *tree-structured network*, *star network*, dan *ring network*. Dalam menentukan topologi jaringan, beberapa hal berikut patut dipertimbangkan, yaitu biaya instalasi, biaya komunikasi, dan ketersediaan atau availibilitas. Sistem berkas terdistribusi adalah sebuah sistem *file-service* di mana pengguna, server, dan media penyimpanan tersebar di berbagai situs dalam sistem terdistribusi. Keuntungan dari sistem terdistribusi adalah memberikan akses bagi pengguna untuk dapat mengembangkan sumber daya sistem, peningkatan kecepatan komputasi, dan meningkatkan availibilitas atau ketersediaan dan reliabilitas data. Sebuah sistem terdistribusi harus menyediakan mekanisme sinkronisasi proses dan komunikasi, agar terhindar dari *deadlock* serta dapat mengatasi *failure* yang tidak muncul dalam sistem terpusat.

Rujukan

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

[WEBFSF1991b] Fakultas Ilmu Komputer Universitas Indonesia. 2003. *Telaga – http://telaga.cs.ui.ac.id/ WebKuliah/ sisdis2003/ 2003-2- CharacterizationOfDistributedSystems.ppt* . Diakses 8 Mei 2007.

Bab 24. Waktu Nyata dan Multimedia

24.1. Pendahuluan

Trend saat ini di bidang teknologi adalah penggabungan data multimedia ke dalam sistem komputer. Bagaimanapun juga, sebuah sistem operasi seharusnya dapat menangani jenis data multimedia ini dengan baik. Multimedia merupakan salah satu aplikasi waktu nyata. Sistem waktu nyata yang digunakan pada aplikasi multimedia memiliki persyaratan yang berbeda dengan persyaratan yang harus dipenuhi pada sistem yang dipelajari pada bab-bab sebelumnya. Persyaratan yang harus dipenuhi pada sistem waktu nyata adalah hasil komputasi yang dihasilkan benar dan selesai pada batas waktu yang telah ditentukan. Oleh karena itu, algoritma penjadwalan yang tradisional haruslah dimodifikasi sehingga dapat memenuhi persyaratan batas waktu yang diminta.

24.2. Kernel Waktu Nyata

Sistem waktu nyata adalah sebuah sistem komputer yang tidak hanya membutuhkan hasil komputasi yang benar tetapi juga harus sesuai dengan batas waktu yang dikehendaki. Hasil dari komputasi yang dilakukan (jika benar) mungkin tidak dalam nilai real. Sistem waktu nyata banyak digunakan dalam bermacam-macam aplikasi. Sistem waktu nyata tersebut ditanam di dalam alat khusus, seperti di kamera, *mp3 players*, serta di pesawat dan mobil.

Beberapa sistem waktu nyata diidentifikasi sebagai sistem *safety-critical*, dalam skenario ini sistem waktu nyata harus merespon kejadian dalam batas waktu yang telah ditentukan, jika tidak dapat memenuhi batas waktu yang ditentukan maka akan terjadi bencana. Sistem manajemen penerbangan merupakan sebuah contoh sebuah sistem waktu nyata sebagai sistem *safety-critical*.

Sistem waktu nyata dibagi menjadi dua tipe yaitu keras dan lembut. Sistem waktu nyata keras menjamin bahwa proses waktu nyata dapat diselesaikan dalam batas waktu yang telah ditentukan. Contoh: sistem *safety-critical*. Sistem waktu nyata lembut menyediakan prioritas untuk mendahulukan proses yang menggunakan waktu nyata dari pada proses yang tidak menggunakan waktu nyata. Contoh: Linux. Karakteristik dari sistem waktu nyata:

- ***Single purpose*.** Tidak seperti PC, yang memiliki banyak kegunaan, sebuah sistem waktu nyata biasanya hanya memiliki satu tujuan, seperti mentransfer sebuah lagu dari komputer ke *mp3 player*.
- ***Small size*.** Kebanyakan sistem waktu nyata yang ada memiliki *physical space* yang terbatas.
- ***Inexpensively mass-produced*.** Sistem waktu nyata banyak ditemukan dalam peralatan rumah tangga dan peralatan lainnya. Misalnya pada kamera digital, *microprocessors*, *microwave ovens*.
- ***Specific timing requirements*.** Sistem operasi waktu nyata memenuhi persyaratan waktu yang ditentukan dengan menggunakan algoritma penjadwalan yang memberikan prioritas kepada proses waktu nyata yang memiliki penjadwalan prioritas tertinggi. Selanjutnya, *penjadwals* harus menjamin bahwa prioritas dari proses waktu nyata tidak lebih dari batas waktu yang ditentukan. Kedua, teknik untuk persyaratan waktu pengalamatan adalah dengan meminimalkan response time dari sebuah *events* seperti interupsi.

Sistem operasi waktu nyata tidak membutuhkan fitur penting (misalnya standar desktop dan sistem server pada desktop PC) karena:

- Kebanyakan sistem waktu nyata hanya melayani satu tujuan saja, sehingga tidak membutuhkan banyak fitur seperti pada desktop PC. Lagi pula, sistem waktu nyata tertentu juga tidak memasukkan *notion* pada pengguna karena sistem hanya mendukung sejumlah kecil proses saja, yang sering menunggu masukan dari peralatan perangkat keras.
- Keterbatasan *space*, menyebabkan sistem waktu nyata tidak dapat mendukung fitur standar desktop dan sistem server yang membutuhkan memori yang lebih banyak dan prosesor yang cepat.
- Jika sistem waktu nyata mendukung fitur yang biasa terdapat pada standar desktop dan sistem server, maka akan sangat meningkatkan biaya dari sistem waktu nyata.

Sebuah sistem operasi yang mendukung sistem waktu nyata harus menyediakan salah satu atau gabungan dari tiga fitur yang ada, fitur-fitur tersebut antara lain:

- Penjadwalan berdasarkan prioritas
- Kernel preemptif
- Pengurangan latensi

Ketiga fitur tersebut akan dijelaskan secara rinci di subbab selanjutnya.

24.3. Penjadwalan Berdasarkan Prioritas

Fitur yang paling penting dari sebuah sistem operasi yang mendukung sistem waktu nyata adalah merespon dengan segera sebuah proses waktu nyata secepat proses yang membutuhkan CPU. Penjadwalan untuk sistem operasi waktu nyata harus mendukung penjadwalan berdasarkan prioritas dengan *preemption*. Algoritma penjadwalan berdasarkan prioritas memberikan prioritas kepada masing-masing proses berdasarkan tingkat kepentingannya; proses yang lebih penting di berikan prioritas lebih tinggi daripada proses lain yang dianggap kurang penting. Apabila penjadwalan yang digunakan juga mendukung *preemption* dan tersedia sebuah proses berprioritas tinggi, maka sebuah proses yang berjalan sekarang ini di CPU akan didahulukan. Penjadwalan ini hanya mendukung sistem waktu nyata lembut. Contoh sistem yang mengunakan penjadwalan ini adalah Solaris, Windows XP dan Linux.

24.4. Kernel Preemptif

Kernel preemptif mengizinkan *preemption* pada sebuah proses yang berjalan di mode kernel. Implementasi dari kernel preemptif sangatlah sulit dan banyak aplikasi (*spreadsheets, word processors*, dan *web browsers*) tidak memerlukan *response time* yang cepat. Akan tetapi, untuk memenuhi persyaratan waktu pada sistem waktu nyata (terutama pada sistem waktu nyata keras) kernel preemptif menjadi sangat penting. Karena kalau tidak proses yang terdapat pada sistem waktu nyata mungkin akan menunggu dalam periode waktu yang sangat lama sampai ada proses lain yang aktif di kernel.

Ada beberapa cara untuk membuat kernel yang dapat preemptif. Salah satu pendekatannya adalah dengan menyisipkan *preemption point* pada *system call* yang berdurasi lama. *Preemption point* mengecek untuk melihat apakah proses dengan prioritas tinggi perlu untuk dijalankan atau tidak. Jika iya, *context switch* mengambil alih. Maka, ketika proses dengan prioritas tinggi *terminate*, proses yang diinterupsi akan melanjutkan *system call*. *Preemption point* akan ditempatkan hanya pada lokasi aman pada kernel, yaitu pada tempat dimana struktur data kernel belum dimodifikasi. Strategi kedua adalah dengan membuat sebuah kernel yang dapat preemptif melalui penggunaan mekanisme sinkronisasi. Dengan metode ini, kernel dapat selalu didahulukan karena beberapa data kernel yang di-*update* adalah data kernel yang di lindungi dari perubahan yang disebabkan oleh proses yang memiliki prioritas tinggi.

24.5. Pengurangan Latensi

Event latensi merupakan sejumlah waktu dari sebuah *event* mulai terjadi sampai *event* tersebut dilayani. Biasanya *event* yang berbeda memiliki persyaratan latensi yang berbeda. Dua tipe latensi yang mempengaruhi *performance* dari sebuah sistem waktu nyata yaitu:

Interrupt Latency

Interrupt latency berhubungan tentang periode waktu dari kedatangan sebuah interupsi pada CPU mulai pada *routine* yang melayani interupsi. Ketika interupsi terjadi, sistem operasi pertama kali harus melengkapi instruksi yang dieksekusinya dan menentukan tipe dari interupsi yang terjadi. Instruksi tersebut harus menyimpan *state* dari proses sekarang sebelum melayani interupsi menggunakan Interrupt Service Routine (ISR) tertentu. Waktu total yang dibutuhkan untuk melakukan task ini adalah *interrupt latency*.

Dispatch Latency

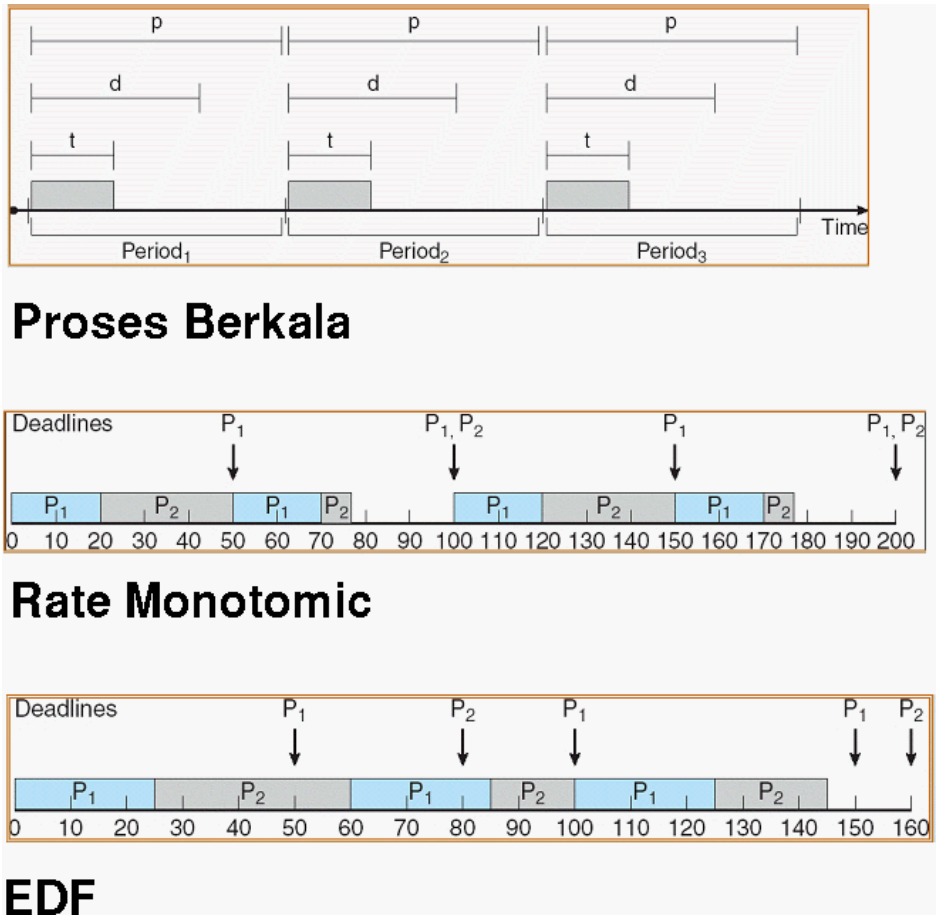
Sejumlah waktu yang dibutuhkan untuk menghentikan sebuah proses dan melanjutkan proses yang lain disebut *dispatch latency*. Tahap konflik pada *dispatch latency* memiliki dua komponen yaitu: *preemption* pada beberapa proses yang berjalan di kernel dan pelepasan *resources* oleh proses prioritas rendah yang dibutuhkan oleh proses prioritas tinggi.

24.6. Penjadwalan Proses

Ulasan kita mengenai penjadwalan sejauh ini difokuskan pada sistem waktu nyata lembut. Pada sistem waktu nyata lembut, penjadwalan untuk sistem tidak memberi jaminan kapan proses *critical* akan di jadwalkan, akan tetapi memberi jaminan bahwa proses tersebut akan didahulukan daripada proses yang tidak *critical*. Sebaliknya, pada sistem waktu nyata keras, persyaratan penjadwalan menjadi lebih ketat. Sebuah proses harus dilayani berdasarkan *deadline*-nya; sehingga jika sebuah proses dilayani setelah *deadline* berakhir, maka proses tidak akan mendapat pelayanan sama sekali.

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 t, setiap kali mendapatkan CPU, sebuah *deadline* d ketika proses tersebut harus dilayani oleh CPU, dan sebuah periode p. Hubungan antara waktu pemrosesan, *deadline*, dan periode dapat dinyatakan sebagai $0 < t < d < p$. Rate dari sebuah proses berperiodik adalah 1/p.

Gambar 24.1. Proses Berkala



Hal yang tidak biasa dari bentuk penjadwalan ini adalah proses akan mengumumkan *deadline*-nya pada penjadwal. Kemudian dengan algoritma *admission-control*, penjadwal akan menyatakan bahwa proses akan diselesaikan tepat waktu atau ditolak permohonannya karena tidak dapat menjamin bahwa proses akan dapat dilayani sesuai *deadline*.

Beberapa algoritma penjadwalan untuk sistem waktu nyata keras:

Rate Monotonic

Algoritma ini menjadwalkan proses berperiodik berdasarkan ketentuan prioritas statik dengan *preemption*. Jika proses berprioritas rendah sedang berjalan dan proses berprioritas tinggi siap untuk dijalankan, maka yang akan didahulukan adalah proses berprioritas rendah. Untuk memasuki sistem, setiap proses berperiodik mendapatkan prioritas dengan inversi berdasarkan periodenya: periode lebih pendek, prioritas tinggi, periode lebih panjang, prioritas rendah. Semakin rendah periodenya maka akan semakin tinggi prioritasnya, dan demikian pula sebaliknya.

Contoh penjadwalan menggunakan *rate monotonic* : diketahui 2 buah proses P1 dan P2, dimana periode P1 adalah 50 dan periode P2 adalah 100. Sedangkan waktu pemrosesan untuk P1 adalah t1=20 dan untuk P2 adalah t2=35. *Deadline* dari proses mempersyaratkan untuk menyelesaikan CPU burst-nya pada awal dari periode berikutnya. Utilisasi CPU dari proses P1 yang merupakan rasio t1/p1, adalah 20/50=0,40 dan utilisasi CPU dari proses P2 adalah 35/100=0,35 sehingga total utilisasi CPU-nya adalah75%. Oleh karena itu, tampaknya kita bisa menjadwal proses ini dalam sebuah cara yang dapat memenuhi *deadline* dan menyisakan *cycle* untuk CPU.

Dengan menggunakan penjadwalan *rate monotonic*, P1 akan mendapat prioritas lebih tinggi dari P2, karena periode 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. pada waktu ini, P2 di *preempted* oleh P1, walaupun P2 masih memiliki sisa 5 miliseconds di CPU burst-nya. P1 menyelesaikan CPU burst-nya di waktu 70, di waktu ini *penjadwal* melanjutkan P2. P2 menyelesaikan CPU burst-nya di waktu 75, memenuhi *deadline* pertamanya. Sistem *idle* sampai waktu 100, kemudian P1 dijadwalkan kembali.

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 menggambarkan *deadline* dari proses yang baru dapat berjalan.

Contoh penjadwalan mengunakan *Earliest Deadline First*: P1 memiliki p1=50 dan t1=25 dan P2 memiliki p2=80 dan t2=35. P1 akan mendapat prioritas awal lebih tinggi karena P1 lebih mendekati *deadline* daripada P2. Kemudian dilanjutkan oleh P2 di akhir *burst time* dari P1. apabila pada penjadwalan *rate monotonic* membiarkan P1 untuk melanjutkan kembali, maka pada EDF, P2 (*deadline* pada 80) yang melanjutkan karena lebih dekat *deadline* daripada P1 (*deadline* pada 100). Jadi, P1 dan P2 telah menemui *deadline* pertama-nya. Proses P1 mulai jalan lagi di waktu 60 dan menyelesaikan *CPU burst time* keduanya di waktu 85, juga menemui *deadline* keduanya di waktu 100. P2 mulai jalan di titik ini, hanya dapat di *preempted* oleh P1 diawal periode selanjutnya di waktu 100. P2 di *preempted* karena P1 memiliki *deadline* lebih awal (waktu 150) dari pada P2 (pada waktu 160). Di waktu 125, P1 menyelesaikan CPU burst-nya dan P2 melanjutkan eksekusinya, selesai di waktu 145 dan menemui deadline-nya. Sistem *idle* sampai waktu 150, kemudian P1 dijadwalkan untuk dijalankan lagi.

Berbeda dengan algoritma *rate-monotonic*, penjadwalan EDF tidak membutuhkan proses periodik, dan tidak juga membutuhkan jumlah waktu CPU per-*burst* yang konstan. Syarat satu-satunya dari penjadwalan EDF adalah proses mengumumkan *deadline*nya pada penjadwal ketika dapat jalan. Secara teoritis, algoritma ini optimal, yaitu dapat memenuhi semua *deadline* dari proses dan juga dapat menjadikan utilisasi CPU menjadi 100%. Namun dalam kenyataannya hal tersebut sulit terjadi karena biaya dari *context switching* antara proses dan *interrupt handler*.

Proportional Share

Penjadwalan ini akan mengalokasikan T bagian diantara 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 mendapat 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 klien terhadap sejumlah bagian apabila bagian yang diinginkan tersedia. Dicontoh kita sebelumnya, kita telah mengalokasikan 50+15+20=75 bagian dari total 100 bagian. Jika sebuah proses baru D meminta 30 bagian, *admission controller* seharusnya meniadakan *entry*D ke dalam sistem.

24.7. Penjadwalan Disk

Penjadwalan disk yang telah kita pelajari pada bab sebelumnya memfokuskan untuk menangani data yang konvensional, yang memiliki sasaran yaitu *fairness* dan *throughput*. Sedangkan pada bab ini data yang kita pelajari adalah data yang continuous.

Data *continuous* memiliki dua constraint yang tidak dimiliki oleh data konvensional yaitu: *timing deadline* dan *rate requirements*. Kedua *constraint* tersebut harus dipenuhi untuk mempertahankan jaminan QOS (*Quality Of Service*), dan algoritma penjadwalan disk harus dioptimalkan untuk *constraint*. Sayangnya, kedua *constraint* tersebut sering terjadi konflik. Data *continuous* biasanya membutuhkan kecepatan *bandwidth disk* yang sangat besar untuk memenuhi *rate-requirements* data. Karena disk memiliki transfer *rate* yang relatif rendah dan *latency rate* yang relatif tinggi maka penjadwal disk harus mengurangi waktu latensi untuk menjamin *bandwidth* yang tinggi. Bagaimanapun, mengurangi waktu latensi mungkin berakibat dalam sebuah penjadwalan *policy* yang tidak memberikan prioritas pada *deadline*.

Earliest-Deadline-First

Penjadwalan EDF yang digunakan pada penjadwalan proses pada sistem waktu nyata dapat digunakan juga untuk melakukan penjadwalan disk pada data yang *continuous*. EDF mirip dengan *shortest-seek-time-first* (SSTF), kecuali dalam melayani permintaan terdekat dengan silinder saat itu karena EDF melayani permintaan yang terdekat dengan *deadline*.

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, *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 penjadwal disk telah dapat melayani permintaan silinder 105 selama perjalanan ke silinder 183 dan masih dapat menjaga persyaratan *deadline* dari silinder 183.

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 *deadline*-nya. Apabila *deadline* tersebar merata, tumpukan dapat diatur pada grup pada ukuran tertentu. Pendekatan yang lain adalah dengan menumpuk permintaan yang *deadline*nya jatuh pada *treshold* waktu yang diberikan, misalnya 10 permintaan pertumpukan.

24.8. Manajemen Berkas

Berkas adalah kumpulan informasi yang berhubungan sesuai dengan tujuan pembuat berkas tersebut. Berkas dapat mempunyai struktur yang bersifat hierarkis (direktori, volume, dll). Sistem operasi memberikan tanggapan atas manajemen berkas untuk aktivitas-aktivitas berikut:
- Pembuatan dan penghapusan berkas
- Pembuatan dan penghapusan direktori
- Primitif-primitif yang mendukung untuk memanipulasi berkas dan direktori
- Pemetaan berkas ke secondary storage
- Melakukan back-up berkas ke media penyimpanan yang stabil (Non-volatile)

Karakteristik sistem multimedia:
- Berkas multimedia biasanya memiliki ukuran yang besar. Contoh, sebuah berkas MPEG-1 video yang berdurasi 100 menit membutuhkan kira-kira 1.125 GB ruang penyimpanan.
- Data *continouos* memerlukan *rate* yang sangat tinggi. Misalnya dalam sebuah video digital, dimana *frame* dari video yang ingin ditampilkan beresolusi 800 x 600. apabila kita menggunakan 24 bits untuk merepresentasikan warna pada setiap pixel, 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.
- Aplikasi multimedia sensitif terhadap *timing delay* selama pemutaran ulang. Setiap kali berkas *continouos* media dikirim kepada klien, pengiriman harus kontinu pada kecepatan tertentu selama pemutaran media tersebut. Hal ini dilakukan agar pada saat *user* menonton atau mendengar berkas-berkas multimedia tidak terputus-putus.

24.9. Manajemen Jaringan

Di subbab sebelumnya telah dijelaskan mengenai algoritma penjadwalan CPU dan *disk* yang difokuskan mengenai tekhnik yang digunakan agar memenuhi persyaratan kualitas *service* dari sebuah aplikasi multimedia yang lebih baik. Jika berkas multimedia dikirimkan melalui sebuah jaringan (internet), pokok persoalan yang berhubungan adalah bagaimana jaringan tersebut dapat mengirimkan data multimedia secara signifikan dan permintaan QOS juga terpenuhi.

Ketika suatu data dikirim melalui jaringan, proses 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.

Di subbab berikutnya, akan membahas 2 pendekatan lain untuk mengatasi persyaratan media continuous yang unik.

24.10. *Uni/Multicasting*

Secara umum, ada 3 metode untuk melakukan pengiriman suatu data dari server ke klien melalui sebuah jaringan:
- ***Unicasting*.** Server mengirim data ke klien tunggal. Apabila data yang ingin dikirim ke lebih dari satu klien, maka server harus membangun sebuah *unicast* (pengiriman paket informasi ke satu tujuan) yang terpisah untuk masing-masing klien.
- ***Broadcasting*.** Server mengirim data ke semua klien yang ada meskipun tidak semua klien meminta/membutuhkan data yang dkirim oleh server.
- ***Multicasting*.** Server mengirim data ke suatu grup penerima data (klien) yang menginginkan data tersebut. Metode ini merupakan metode yang berada dipertengahan metode unicasting dan broadcasting.

24.11. *Streaming Protocol*

Bagaimana cara sebuah media streaming dapat dikirimkan ke klien? Salah satu pendekatan adalah dengan mengalirkan media dari sebuah web server standar yang menggunakan *hypertext transport protocol* (HTTP). Protokol tersebut digunakan untuk mengirim dokumen dari sebuah web server. Biasanya, klien menggunakan sebuah media player, seperti QuickTime, RealPlayer, atau Windows Media Player, untuk memutar kembali media yang dialirkan oleh web server standar. Biasanya, pertama-tama klien meminta sebuah *metafile*, yang berisi lokasi sebuah berkas *streaming media*. *Metafile* tersebut dikirimkan ke *web browser* klien, dan *browser* akan membuka berkas yang dimaksud dengan memilah media player yang sesuai dengan jenis media yang dispesifikasikan di *metafile*.

Masalah yang muncul jika pengiriman *streaming* media dari sebuah *web server* standar adalah *web server* tidak dapat memelihara status koneksi dengan klien. Hal ini dapat terjadi karena HTTP merupakan protokol yang *stateless*. Akibatnya, klien akan mengalami kesulitan pada saat ia melakukan *pause* selama pengiriman *streaming media* masih berlangsung. Pelaksanaan *pause* akan menyebabkan *web server* harus mengetahui status mana yang akan dimulai kembali ketika klien memutar ulang.

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:

- ***Setup*.** *Server* mengalokasikan sumber daya kepada sesi klien.
- ***Play*.** *Server* mengirim sebuah *stream* ke sesi klien yang telah dibangun dari perintah *setup* sebelumnya.
- ***Pause*.** *Server* menunda pengiriman stream namun tetap menjaga sumber daya yang telah dialokasikan.
- ***Teardown*.** *Server* memutuskan koneksi dan membebas tugaskan sumber daya yang sebelumnya telah digunakan.

Gambar 24.2. *Finite-State Machine* yang merepresentasikan RTSP

24.12. Kompresi

Karena ukuran dan persyaratan *rate* pada sistem multimedia, berkas multimedia sering dikompresi dari ukuran aslinya ke ukuran yang lebih kecil. Sebuah berkas yang sudah dikompres, akan mengurangi *space* untuk penyimpanan dan dapat dikirim ke klien lebih cepat. Kompresi sangat berguna pada saat mengirimkan sebuah isi berkas melalui koneksi jaringan. Dalam diskusi mengenai kompresi berkas, kita sering merujuk ke ratio kompresi, yang mana ratio dari ukuran berkas asli banding ukuran berkas yang dikompres.

Sekali sebuah file telah dikompres, file tersebut harus didekompresikan sebelum dapat diakses. Fitur yang terdapat pada algoritma kompresi berkas mempengaruhi pada saat dekompresinya. Algoritma kompresi diklasifikasikan menjadi dua jenis, yaitu:

- **Algoritma kompresi *Lossy*.** Kompresi menggunakan *lossy*, beberapa bagian data asli hilang ketika berkas di *decoded*. Keuntungan dari algortima ini adalah bahwa rasio kompresi cukup tinggi. Hal ini dikarenakan cara algoritma *lossy* yang 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. Beberapa algoritma *lossy* digunakan di operasi video dengan hanya menyimpan perbedaan diantara *frame* berturut-turut. Contoh format gambar yang menggunakan algoritma *lossy* adalah JPEG.

- **Algoritma kompresi *Lossless*.** Kompresi menggunakan *lossless* menjamin bahwa berkas yang dikompresi dapat selalu dikembalikan ke bentuk aslinya. Algoritma *lossless* digunakan untuk kompresi berkas *text*, seperti program komputer (berkas zip, rar, dll), karena kita ingin mengembalikan berkas yang telah dikompres ke status aslinya. Contoh format gambar yang menggunakan algoritma *lossless* adalah GIF and PNG

24.13. Rangkuman

Sistem waktu nyata adalah sebuah sistem komputer yang mengutamakan pencapaian hasil dalam sebuah periode *deadline*; hasil yang tiba setelah periode *deadline* berlalu adalah tak berguna. Ada dua tipe dari sistem waktu nyata yaitu: sistem waktu nyata keras dan sistem waktu nyata lembut. Sistem waktu nyata keras menjamin pekerjaan yang kritis akan diselesaikan dengan tepat waktu. Sedangkan, sistem waktu nyata lembut adalah sistem yang kurang membatasi, dimana pekerjaan yang kritis mendapat prioritas setelah pekerjaan lain.

Sebuah sistem operasi yang mendukung sistem waktu nyata harus menyediakan salah satu atau gabungan dari tiga fitur yang ada, fitur-fitur tersebut antara lain:

- Penjadwalan berdasarkan prioritas
- Kernel preemptif
- Pengurangan latensi

Algoritma yang digunakan untuk penjadwalan sistem waktu nyata keras ada 3 yaitu: penjadwalan *Rate-monotonic*, *earliest deadline first* dan *proportional share*. Penjadwalan *rate monotonic* menentukan bahwa proses yang lebih sering membutuhkan CPU memiliki prioritas lebih tinggi daripada proses yang kurang sering membutuhkan CPU. Penjadwalan *Earliest-first deadline* memberi prioritas berdasarkan *deadline* yang akan datang (*deadline* lebih awal, prioritas lebih tinggi). Penjadwalan *proportional share* menggunakan teknik membagi waktu prosessor menjadi bagian dan memberi masing-masing jumlah bagian proses, kemudian menjamin waktu CPU masing-masing proses.

Aplikasi multimedia banyak digunakan di sistem komputer modern saat ini. Berkas multimedia terdiri dari berkas audio dan video, yang dapat dikirim ke sistem seperti komputer desktop, handphone, PDA. Perbedaan utama antara data multimedia dan data konvensional adalah data multimedia memiliki persyaratan *rate* tertentu dan persyaratan *deadline*.

Penjadwalan disk biasanya menggunakan persyaratan *deadline* pada sebuah berkas multimedia sebagai sebuah kriteria penjadwalan. Kedua persyaratan tersebut harus dipenuhi untuk mempertahankan jaminan QOS (Quality Of Service), dan algoritma penjadwalan disk harus dioptimalkan untuk memenuhi persyaratan tersebut. Algoritma yang digunakan untuk melakukan penjadwalan disk ada dua yaitu: penjadwalan Earliest-Deadline-First (EDF) dan SCAN-EDF.

Manajemen berkas merupakan salah satu komponen dari sebuah sistem operasi. Sistem operasi memberikan tanggapan atas manajemen file untuk aktivitas-aktivitas berikut: pembuatan dan penghapusan berkas, pembuatan dan penghapusan 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-volatile*). Berkas multimedia memiliki 3 karakteristik yaitu: memiliki ukuran berkas yang besar, memiliki *rate* yang tinggi, dan sensitif terhadap *timing delay*.

Manajemen jaringan membutuhkan protokol untuk menangani *delay* dan *jitter* yang disebabkan oleh jaringan. Ada dua cara untuk mengatasi *delay* dan *jitter* selama pengiriman berkas multimedia melalui jaringan yaitu: uni/multicasting dan *Real Time Streaming Protocol* (RTSP).

Kompresi merupakan pengukuran suatu berkas menjadi ukuran yang lebih kecil dari berkas aslinya. Kompresi sangat berguna pada saat mengirimkan sebuah isi berkas melalui koneksi jaringan. Hal ini

menyebabkan pengiriman berkas yang telah dikompresi lebih cepat jika dibandingkan dengan berkas yang belum dikompresi. Algoritma yang digunakan untuk mengkompresi sebuah berkas multimedia adalah algoritma *lossy* dan *lossless*.

Rujukan

[DONNY2005] Abas Ali Pangera dan Donny Ariyus. 2005. *Sistem operasi*. First Edition. Penerbit Andi.

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

Bab 25. Perancangan dan Pemeliharaan

25.1. 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.

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, sistem harus fleksibel, dapat diandalkan, bebas eror dan efisien.

Pemeliharaan sistem operasi penting untuk dilakukan. Hal tersebut merupakan salah satu cara untuk menjaga agar sistem operasi dapat bekerja dengan baik. Pemeliharaan sistem operasi menyangkut bagaimana nantinya sistem operasi dapat bekerja untuk memenuhi tiga kebutuhan, yaitu: fungsionalitas: apakah sistem tersebut dapat bekerja dengan baik; kecepatan: apakah sistem tersebut dapat memproses dengan cepat?; dan *fault-tolerant*: apakah sistem dapat terus bekerja apabila terjadi kesalahan pada *hardware*.

Sebelum kita berbicara tentang bagaimana merancang sebuah sistem operasi, akan sangat baik apabila kita tahu apa yang harus dilakukan oleh sebuah sistem operasi. Hal tersebut dapat dijabarkan menjadi tiga, yaitu: abstraksi *hardware*, manajemen sumber daya dan *user interface*.

Abstraksi *Hardware*

Abstraksi *Hardware* adalah proses penyederhanaan dan penyembunyian informasi. Inti dari abstraksi *hardware* adalah mendapatkan *hardware-hardware* yang kompleks dan membuatnya sesuai untuk *interface* yang simple dan terdefinisi dengan baik. Abstraksi diperlukan karena meliputi bagaimana cara untuk berhadapan dengan *failure* apabila *device* harus *fault tolerant.* Ada dua alasan mengapa abstraksi *hardware* penting: *simplicity* dan *compatibility.*

- ***Simplicity*** adalah mengenali proses-proses kompleks sehingga tidak terjadi duplikasi. Proses-proses kompleks tersebut harus dilakukan satu kali, tetapi dalam sistem dimana banyak aplikasi yang melakukan proses tersebut, akan menjadi desain yang buruk apabila setiap *developer* aplikasi mengimplementasikan ulang proses tersebut. Dan sistem operasi mengatasi hal ini dengan baik. Sistem operasi menyediakan media antarmuka yang lebih sederhana untuk aplikasi.
- ***Compatibility***, hampir semua tujuan umum sistem operasi adalah dapat mengoperasikan semua *hardware.* Akan sangat baik apabila setiap bagian kecil dari *hardware* yang melakukan hal yang sama berkomunikasi menurut standar *interface*, tetapi itu tidak benar. Beberapa bagian kecil dari sistem perlu untuk tahu apa tipe dan bagaimana cara berkomunikasi dengan suatu *hardware*. Sistem operasi melakukan hal tersebut, dan kemudian menampilkan aplikasi dengan *interface* yang sama, tidak bergantung dengan tipe *hardware*. Dengan begitu, aplikasi dapat bekerja dengan tipe *hardware* apa saja.

Manajemen Sumber Daya

Sebagai tambahan terhadap abstraksi *hardware*, umumnya sistem operasi mengatur sumber daya dari sistem dengan baik. Sumber daya adalah istilah umum yang dapat merujuk pada setiap komponen dari sistem yang sanggup melakukan perkerjaan. contohnya, *processor* adalah sumber daya, begitu juga RAM dan *disk*. *Sound card* dan *network card* juga sumber daya walaupun tidak semua *hardware* adalah sumber daya. Untuk pengaturan sumber daya yang lebih baik, sistem operasi dapat membagi sumber daya menjadi sub-sumber daya yang sejenis. Misalnya, *disk* adalah sumber daya, tetapi melalui *file system*, sistem operasi dapat membagi *disk* menjadi *file*, yang mana juga sumber daya. Manajemen

sumber daya adalah proses pemberian sumber daya kepada aplikasi untuk digunakan. Hal tersebut dilakukan dengan cara yang berbeda-beda untuk sumber daya yang berbeda, dengan perbedaan utama terletak pada *multiplexing, locking* dan *access control.*

- ***Multiplexing*** adalah berbagi sebuah sumber daya antara akses-akses yang berlangsung bersamaan. ini adalah perhatian utama untuk *multitasking* sistem operasi. Banyak sumber daya dapat di *share* untuk menyediakan akses bagi aplikasi-aplikasi yang berbeda.
- ***Locking*** beberapa *device* mungkin tidak dapat di *multiplexed*. Contohnya, *backup device* tidak dapat segera di multiplex apabila dua aplikasi mau melakukan *backup* dalam waktu yang bersamaan. Ini adalah *error*. Tetapi sistem operasi tidak seharusnya membiarkan *error* ini terjadi dan berpengaruh besar pada sistem. Untuk memastikan hal ini, sistem operasi menyediakan beberapa jenis *locking* ketika *device* tidak dapat di multiplex. *Locking* menjamin bahwa hanya satu aplikasi saja yang mempunyai akses ke sebuah *device* dalam waktu yang ditentukan.
- ***Access Control*** kadang manajemen sumber daya tidak melibatkan pemecahan masalah dari beberapa proses pengaksesan *hardware*. Malahan, manajemen sumber daya melibatkan perlindungan proses dari proses lainnya. *Security* secara khas terimplementasikan di beberapa bentuk sebagai bagian dari manajemen sumber daya. *Access control* adalah kontrol sumber daya yang disediakan oleh sistem operasi untuk menguatkan batasan-batasan *security.* Contohnya, sebuah sistem operasi mungkin akan memperbolehkan sebuah aplikasi untuk membuat *file* yang secara spesifik hanya pengguna tertentu saja yang diperbolehkan untuk menggunakannya.

User Interface

Tujuan ketiga dari sistem operasi adalah untuk menyediakan beberapa jenis media tatap muka bagi pengguna untuk mempermudah mengontrol sistem dan menjalankan aplikasi-aplikasi. Tujuan ini membawa perancangan sistem operasi jauh dari pertanyaan-pertanyaan teknis yang umumnya ada di diskusi sistem operasi dan lebih ke arah pertanyaan psikologis mengenai cara terbaik untuk berinteraksi antara manusia dan sistem. *User interface* juga aspek yang sangat modular dari perancangan sistem operasi. dengan demikian, antarmuka pengguna dapat dipaksa ke urutan atas dari perancangan-perancangan lain dengan mudah.

Adapun prinsip-prinsip dalam merancang sistem operasi adalah:

1. ***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 subsistem pada sistem operasi dapat ditambahkan tanpa mempengaruhi subsistem yang sudah ada.
2. ***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*.
3. ***Reliability*.** Adalah kemampuan sistem operasi untuk mengatasi kondisi *error*, termasuk kemampuan sistem operasi untuk memproteksi diri sendiri dan penggunanya 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.
4. ***Security*.** Sistem operasi harus memberikan keamanan terhadap data yang disimpan dalam semua *drive*.
5. ***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 berkas.

Untuk merancang sistem operasi diperlukan persiapan-persiapan yang matang. persiapan-persiapan yang diperlukan dalam merencanakan perancangan sistem operasi yaitu:

1. **Memikirkan dimana nantinya sistem operasi akan dijalankan.** ini dikarenakan banyaknya sistem komputer dan masing-masing sistem komputer tidak saling mendukung.
2. **Memikirkan kegunaan sistem operasi tersebut.** fungsi sistem operasi sebagai pelayan bagi program aplikasi dan masing-masing program aplikasi mempunyai tujuan-tujuan tertentu yang

saling berbeda. disinilah perlu dipikirkan kegunaan sistem operasi yang akan dirancang. akan tetapi, biasanya sistem operasi dirancang agar dapat menjalankan berbagai macam aplikasi. Sistem operasi ini disebut sistem operasi umum, tetapi kadang sistem operasi ditujukan bagi aplikasi-aplikasi tertentu saja, misalnya sistem operasi *database*.

3. **Bahasa pemrograman yang akan digunakan.** banyak bahasa pemrograman yang dapat digunakan untuk merancang sistem operasi dan tiap-tiap bahasa mempunyai kekurangan dan kelebihan, tetapi bahasa apapun yang jadi pilihan nantinya haruslah sudah dikuasai dengan baik.

25.2. 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 persyaratan: sebuah antarmuka harus sederhana, sebuah antarmuka harus lengkap, dan sebuah antarmuka harus memilki kinerja yang cepat.

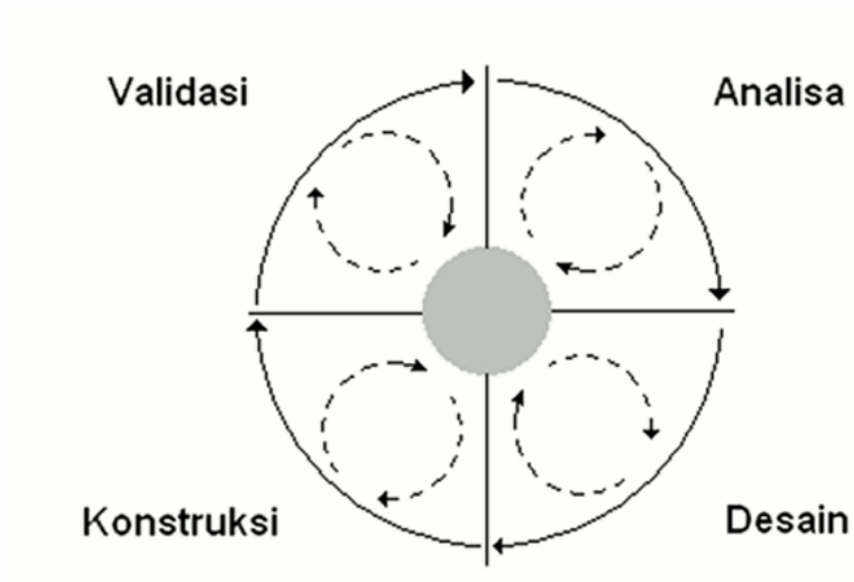
Alasan utama mengapa antarmuka sulit untuk dirancang adalah karena setiap antarmuka adalah sebuah bahasa pemrograman yang kecil: antarmuka menjelaskan sekumpulan objek-objek dan operasi-operasi yang bisa digunakan untuk memanipulasi objek.

Dalam proses pengembangan antarmuka, kita bisa atau mungkin saja tidak bisa memisahkannya dari seluruh proses pengembangan sebuah produk. Walaupun begitu, fokus dari dua proses tersebut sangatlah berbeda. Dalam proses pengembangan antarmuka, fokus haruslah terletak pada elemen-elemen antarmuka dan objek-objek yang pengguna lihat dan gunakan, dibandingkan dengan kemampuan sebuah program.

Elemen-Elemen dalam perancangan antarmuka adalah

1. **Mendefinisikan konsep.** Mengumpulkan kebutuhan-kebutuhan pengguna dan mendefinisikan desain secara konseptual.
2. **Memvalidasi konsep.** Mengevaluasi konseptual desain tersebut.
3. **Merancang.** Mengevaluasi prototype. Menandai dan memperbaiki masalah-masalah yang ditemukan.
4. **Pengembangan.** Melakukan pengujian secara berkala terhadap desain yang lebih dahulu dibuat dan desain yang paling terakhir dibuat. Menandai dan memperbaiki masalah-masalah yang ditemukan.

Gambar 25.1. Empat Tahap Proses Perancangan Antarmuka



Proses yang secara rinci menggambarkan bagaimana perancangan dan pengembangan antarmuka terlihat pada gambar di atas. Empat tahap utama dalam proses tersebut adalah:

- Mengumpulkan atau menganalisa informasi dari pengguna.
- Merancang Antarmuka.
- Mengembangkan Antarmuka.
- Memvalidasi Antarmuka.

Proses-proses tersebut independen dari *hardware* dan *software*, sistem operasi dan peralatan yang digunakan untuk merancang dan mengembangakan produk. IBM *Common User Access (CUA) interface design guide* adalah yang pertama kali mendeskripsikan proses perancangan antarmuka secara iteratif.

1. **Mengumpulkan atau Menganalisa Informasi Pengguna.** Proses perancangan antarmuka dimulai dari memahami pengguna. Sebelum merancang antarmuka, kita harus mengetahui masalah apa yang ingin pengguna selesaikan dan bagaimana mereka melakukan pekerjaan mereka. Pengumpulan dan penganalisaan aktivitas-aktivitas pada tahap pertama ini dapat dijabarkan dalam lima langkah:
   - Menentukan profil pengguna.
   - Melakukan analisa terhadap *task-task* pengguna.
   - Mengumpulkan kebutuhan-kebutuhan pengguna.
   - Menganalisa *user environments.*
   - Mencocokan kebutuhan tersebut dengan *task.*
2. **Merancang Antarmuka.** Dalam merancang antarmuka ada beberapa tahapan yang harus dilalui, yaitu:
   - Menjelaskan kegunaan dan tujuan.
   - Menetapkan icon objek, *views,* dan representasi visual.
   - Merancang objek dan jendela menu
   - Memperbaiki rancangan visual.
3. **Mengembangkan Antarmuka.** Hal pertama yang bisa dilakukan dalam membangun antarmuka adalah membangun *prototype*. Membangun *prototype* adalah cara yang berharga dalam membuat rancangan awal dan membuat demonstrasi produk dan penting untuk pengujian kegunaan antarmuka. Dari *prototype* tersebut, perancang antarmuka dapat mulai membangun antarmuka secara utuh. Ketika membuat *prototype,* sangat penting untuk diingat bahwa *prototype* harus dapat di buang setelah digunakan (*disposable*). Jangan takut untuk membuang sebuah *prototype*. Tujuan dalam membuat *prototype* adalah untuk mempercepat dan mempermudah dalam memvisualisasikan desain alternatif dan konsep, bukan untuk membangun kode yang akan digunakan sebagai bagian dari produk.
4. **Melakukan Validasi Terhadap Antarmuka.** Evaluasi kegunaan adalah bagian penting dari proses pengembangan, untuk mengetahui bagaimana tanggapan pengguna terhadap antarmuka yang telah dibuat. Evaluasi ini akan digunakan untuk memperbaiki kekurangan pada antarmuka yang telah dibangun. Aturan emas dalam perancangan antarmuka:
   - Buat Pengguna menguasai antarmuka.
   - Kurangi *user's memory load*
   - Buat antarmuka konsisten

25.3. Implementasi

Rancangan Sistem

Desain sistem memiliki masalah dalam menentukan tujuan dan spesifikasi sistem. Pada level paling tinggi, desain sistem akan dipengaruhi oleh pilihan *hardware* dan jenis sistem. Kebutuhannya akan lebih sulit untuk dispesifikasikan. Kebutuhan terdiri dari *user goal* dan *system goal*. *User* menginginkan sistem yang nyaman digunakan, mudah dipelajari, dapat diandalkan, aman, dan cepat. Namun itu semua tidaklah signifikan untuk desain sistem. Orang yang mendesain ingin sistem yang mudah didesain, diimplementasikan, fleksibel, dapat dipercaya, bebas *error*, efisien. Sampai saat ini belum ada solusi yang tepat 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.

25.4. Kinerja

Kinerja sebuah sistem ditentukan oleh komponen-komponen yang membangun sistem tersebut. Kinerja yang paling diinginkan ada pada sebuah sistem adalah bebas *error*, cepat dan *fault-tolerant*.

Fault-tolerant

Beberapa sistem perlu untuk tetap berjalan apapun yang terjadi. Sistem-sistem ini didesain untuk memulihkan secara sempurna dari masalah *hardware* dan tetap berjalan. Maka dari itu, *fault-tolerant* sistem operasi adalah menyembunyikan kegagalan *hardware* dari aplikasi yang sedang menggunakan *hardware* tersebut.

25.5. Pemeliharaan Sistem

Pemeliharaan sistem sangatlah penting bagi pengguna sistem. Karena, seringkali penggunaan sistem operasi menjadi tidak aman karena alasan-alasan seperti:
- Sistem terinfeksi *malware* aktif
- Sistem berkas *corrupt*
- Perangkat keras melemah

Untuk mencegah hal-hal tesebut, digunakanlah mOS(*maintenance Operating system*) yang berfungsi untuk:
- Manajemen *Malware* yang aktif
- Pemulihan data (*recovery*) dan perbaikan sistem berkas
- Diagnosa perangkat keras.

mOS tidak menulis ke *disk* atau menjalankan kode apapun dari *disk*, memiliki akses langsung ke perangkat keras, dan hanya membutuhkan sedikit bagian dari perangkat keras untuk bekerja dengan sempurna. Selain dengan *mOS*, kita juga dapat memelihara sistem (pada windows) dengan cara-cara yang sederhana seperti:
- Jangan pernah mematikan *power* sampai sistem benar-benar sudah *shutdown.*
- Buatlah *backup* data-data yang penting.
- Lakukan *defragment* setidaknya satu bulan sekali
- Sisakan sedikit*space* kosong di partisi tempat sistem operasi berada.
- Gunakan *firewall* jika anda terkoneksi dengan jaringan.
- Lakukan pengecekan virus secara rutin.

25.6. *Tunning*

Tuning dengan tujuan optimisasi kinerja adalah proses memodifikasi sebuah sistem untuk membuat beberapa aspek bekerja lebih efisien atau menggunakan sumber daya lebih sedikit. Dengan kata lain, *tuning* berarti mencari titik kelemahan dari suatu sistem dan memperbaikinya agar bisa jauh lebih baik lagi.

Tuning akan terpusat pada meningkatkan hanya satu atau dua aspek dari kinerja: waktu eksekusi, penggunaan memori, kapasitas *disk*, *bandwith*, konsumsi *power*, atau sumber daya lainnya. Namun peningkatan tersebut diimbangi dengan penurunan kinerja lainnya. Misalnya memperbesar ukuran *cache* mempercepat kinerja, namun meningkatkan penggunaan memori.

Salah satu contoh *tuning* yang biasa kita lakukan adalah *defragment. Defragmentation* (atau *defragging*) adalah proses mengurangi jumlah fragmentasi dalam *file system. Defragmentation* menyusun kembali konten-konten dalam *disk* dan menyimpan bagian-bagian kecil dari setiap *file* untuk tetap saling berdekatan. *Defragmentation* juga berusaha untuk membuat area untuk *free space* lebih besar dengan menggunakan *compaction* untuk menghalangi kembalinya fragmentasi. Beberapa *defragmenter* juga mencoba untuk menjaga *file-file* yang lebih kecil tetap berada dalam *singel directory*, karena mereka sering kali diaskes secara berurutan.

Selain untuk optimasi kinerja, Adalah mungkin untuk mendesain, mengkode, dan megimplementasikan sebuah sistem operasi khusus untuk satu mesin saja. Pada umumnya sistem operasi dirancang untuk dapat dijalankan pada berbagai jenis mesin, sistemnya harus dikonfigurasikan untuk setiap komputer. Proses ini terkadang disebut sebagai *System Generation.* Program *Sysgen* mendapatkan informasi mengenai konfigurasi khusus tentang sistem perangkat keras dari sebuah data, antara lain sebagai berikut:

- CPU apa yang digunakan, pilihan yang diinstall.
- Berapa banyak memori yang tersedia
- Peralatan yang tersedia
- Sistem operasi pilihan apa yang diinginkan atau parameter apa yang digunakan.

25.7. *Trend*

Trend sistem operasi sangat dipengaruhi oleh penilaian dari pengguna sistem operasi. Pengguna biasanya lebih memilih sistem operasi yang *user friendly*, performanya bagus, tidak sering mengalami *error*, tampilannya cantik, dan lain lain. *Trend* sistem operasi sampai saat ini ternyata masih dikuasai oleh keluarga *Windows*. Namun bisa diprediksi bahwa *trend* sistem operasi pada tahun-tahun mendatang akan berkaitan dengan sistem operasi yang bersifat *Open Source.* Dalam sebuah artikel di *web*, Linus Torvalds mengatakan bahwa sistem operasi yang *open source* memungkinkan penggunanya untuk menciptakan sistem yang terbaik sesuai dengan kebutuhan mereka masing-masing.

25.8. Rangkuman

Sebelum merancang sistem operasi ada baiknya melakukan persiapan-persiapan terlebih dahulu. Persiapan-persiapan tersebut antara lain: memikirkan dimana nantinya sistem operasi akan dijalankan, memikirkan kegunaan sistem operasi tersebut dan menentukan bahasa pemrograman yang akan digunakan. Selain itu dalam merancang sistem operasi juga sebaiknya berpegangan dengan prinsip-prinsip merancang sistem operasi, yaitu: *extensibility, portability, reliability, security dan high performance.*

Pemeliharaan sistem operasi dilakukan untuk menjaga agar sistem operasi tetap stabil dan dapat berkerja sebagaimana mestinya.

Trend sistem operasi pada tahun-tahun mendatang akan berkaitan dengan sistem operasi yang bersifat *Open Source.* Dalam sebuah artikel di *web*, Linus Torvalds mengatakan bahwa sistem operasi yang *open source* memungkinkan penggunanya untuk menciptakan sistem yang terbaik sesuai dengan kebutuhan mereka masing-masing.

Rujukan

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

[WEBWiki2007] Wikipedia. 2007. *Page Replacement Algortihm http://en.wikipedia.org/wiki/Page_replacement_algorithm* . Diakses 4 April 2007.

[WEBCDSmith] Chris Smith. 2007. *The Common Man's Guide to Operating System Design http://cdsmith.twu.net/professional/osdesign.html* . Diakses 9 Mei 2007.

[WEBQuirke2004] Chris Quirke. 2004. *What is a Maintenance OS? http://cquirke.mvps.org/whatmos.htm* . Diakses 9 Mei 2007.

[BudiHalusSantoso2005] Budi Halus Santoso dan Jubilee Enterprise. 2005. *Perancangan Sistem Operasi*. First Edition. ANDI Yogyakarta.

[Internet News] Internet News. 2007. *Linux Creator: Operating Systems Will Follow Internet Trends http://www.internetnews.com/dev-news/article.php/212891* . Diakses 10 Mei 2007.

[WEBWiki2007] Wikipedia. 2007. *Optimization (computer science) http://en.wikipedia.org/wiki/Optimization_(computer_science)* . Diakses 10 Mei 2007.

[WEBW3Shcools2007] W3Shcools. 2007. *OS Platform Statistics http://www.w3schools.com/browsers/browsers_os.asp* . Diakses 10 Mei 2007.

[WEBWiki2007] Wikipedia. 2007. *Maintenance OS http://en.wikipedia.org/wiki/Maintenance_OS* . Diakses 10 Mei 2007.

[WEBInfoHQ] InfoHQ. 2004. *Computer Maintenance Tips http://www.infohq.com/Computer/computer_maintenance_tip.htm* . Diakses 9 Mei 2007.

[Mandel Encyclopedia] Theo Mandel. 2002. *User/System Interface Design http://theomandel.com/docs/mandel-encyclopedia.pdf* . Diakses 4 Juni 2007.

[WEBWiki2007] Wikipedia. 2007. *Defragmentation http://en.wikipedia.org/wiki/Defragmentation* . Diakses 7 Juni 2007.

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

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 Bagian Dua

Berikut merupakan kumpulan soal Ujian Tengah Semester (UTS) dan Ujian Akhir Semester (UAS) antara 2003 dan 2008 untuk Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia. Waktu pengerjaan setiap soal [kecuali "Pasangan Konsep"] ialah 30 menit.

B.1. Memori

(B05-2004-01) Status Memori I

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?

(B05-2005-01) Status Memori II

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
```

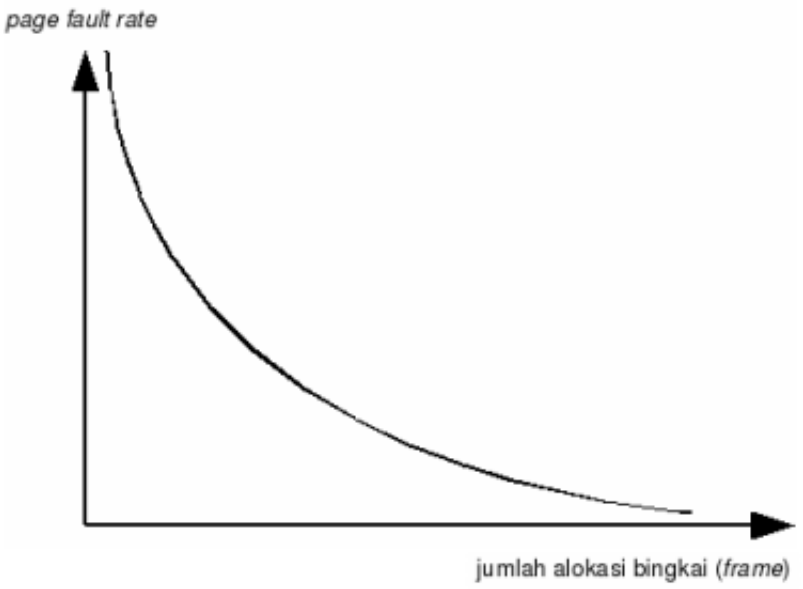
a. Terangkan secara singkat; apa yang dimaksud dengan: PID, PPID, UID, VIRT, SWAP, RES, SHR, CODE, DATA, %MEM, nFLT, nDRT, COMMAND.
b. Berapa ukuran total dari sistem "*swap*"?
c. Berapa besar bagian "*swap*" yang sedang digunakan?
d. Berapa besar ukuran bagian "*swap*" yang berada dalam partisi terpisah?
e. Berapa besar ukuran bagian "*swap*" yang berbentuk berkas biasa?

(B05-2004-02) Managemen Memori dan Utilisasi CPU

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.

(B05-2007-01) Penghitungan Kesalahan Halaman I

Terangkan bagaimana strategi *Page Fault Frequency* dapat mengatasi masalah *trashing*. Mengapa pendekatan ini lebih praktis dibandingkan pendekatan "*Working Set Model*"?



(B05-2007-04) Penghitungan Kesalahan Halaman II

a. Terangkan bagaimana pendekatan "*Working Set Model*" digunakan untuk mengalokasikan halaman memori.
b. Terangkan bagaimana pendekatan "*Page Fault Frequency*" melaksanakan hal yang sama.
c. Pendekatan mana yang lebih praktis: (a) atau (b)?
d. Jelaskan butir (c) di atas!

(B05-2002-01) Memori I

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.

(B05-2002-02) Memori II

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!

(B05-2003-01) Memori III

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!

(B05-2007-02) Memori IV

Sebuah proses secara berturut-turut mengakses alamat memori sebagai berikut:

```
1001 1002 1003 1004 1007 1301 1303 1305 1307 1251 1250 1247 1248 1235
1520 1530 1540 1550 2240 2310 2380 2450 2480 1410 1430 1450 1470 2205
2275 2345 2435 2210 1305 1306 1307 1308 3302 3333 3354 3375 3380 2305
2326 2327 2328 3110 3220 3330 3333 3120 2400 2450 2350 2250 2251 2252
```

Ukuran halaman ialah 300 bytes. Upayakan page fault seminim mungkin!

a. Tentukan *reference string* dari urutan pengaksesan memori tersebut.
b. Gunakan algoritma "*Optimal Page Replacement*". Tentukan jumlah bingkai (*frame*) minimum yang diperlukan agar terjadi "*page fault*" minimum! Berapakah jumlah "*page fault*" yang terjadi? Gambarkan dengan sebuah bagan!
c. Gunakan jumlah bingkai butir "b" di atas, namun dengan algoritma LRU (*Least Recently Used*). Berapakah jumlah "*page fault*" yang terjadi? Gambarkan dengan sebuah bagan!

(B05-2005-02) Memori Virtual

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*)

(B05-2003-02) *Multilevel Paging Memory* I

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

(B05-2004-03) *Multilevel Paging Memory* II

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

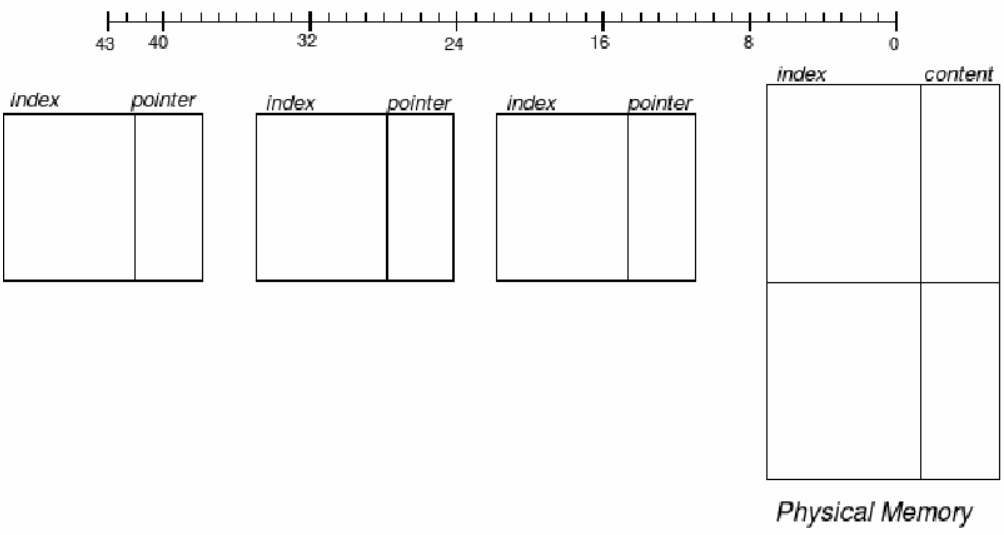
a. Berapa *byte*, kapasitas maksimum dari "*Virtual Memory*" dengan "*Logical Address Space*" tersebut?
b. Gambarkan pembagian "*Logical Address Space*" tersebut: berapa bit untuk P1/ "*Outer Page Table*", berapa bit untuk P2/ "*Page Table*", serta berapa bit untuk alokasi *offset*? 
c. Berapa *byte*, 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

(B05-2005-03) *Multilevel Paging Memory* III

[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?

(B05-2006-01) *Memori Virtual Linux Bertingkat Tiga*



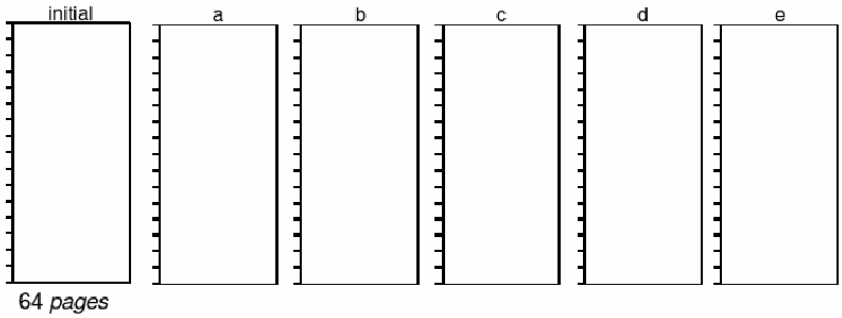
Berikut ini, 004 0200 8004(HEX), merupakan alamat virtual Linux yang sah (43 bit), dengan tiga tingkatan tabel halaman (*three level page tables*): *Global Directory* (10 bit), *Page Middle Directory* (10 bit), dan *Page Table* (10 bit).

a. Uraikan alamat virtual tersebut di atas dari basis 16 (HEX) ke basis 2.
b. Lengkapi gambar di atas seperti nama tabel-tabel, indeks tabel dalam basis heksadesimal (HEX), pointer (cukup dengan panah), alamat memori fisik (*Physical Memory*) dalam basis heksadesimal (HEX), isi memori fisik (bebas!), serta silakan menggunakan titik-titik "..." untuk menandakan "dan seterusnya".
c. Berapa ukuran bingkai memori (*memory frame*)?

(B05-2006-02) Algoritma *Buddy*

Pada prinsipnya, "Algoritma Buddy" (*Buddy Algorithm*) mengalokasikan permintaan halaman (*pages*) pada sekeping memori berkesinambungan (*contiguous memory*) berdasarkan kelipatan pangkat 2. Berikan ilustrasi pengalokasian, jika diketahui permintaan pada sekeping memori 64 halaman sebagai berikut:

a. Program A meminta 4 halaman.
b. Program B meminta 3 halaman.
c. Program C meminta 3 halaman.
d. Program B mengembalikan permintaan butir b.
e. Program D meminta 7 halaman.



(B05-2007-03) Alokasi Slab

Diketahui ukuran halaman (*page*) memori = 4 kbytes; serta slab yang dibentuk dari 4 halaman. Sistem memerlukan alokasi untuk 2 obyek kernel @ 8 kbytes; serta 2 obyek kernel @ 2 kbytes.

a. Jelaskan dengan diagram, hubungan antara slab, *cache*, dan obyek kernel.
b. Jelaskan keunggulan slab dari algoritma *buddy*

B.2. Masukan/Keluaran

(B06-2003-01) *I/O Interface*

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.

(B06-2004-01) *I/O* dan *USB*

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?

(B06-2004-02) Struktur Keluaran/Masukan Kernel (*I/O*)

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!

(B06-2005-01) Masukan/Keluaran

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.

(B06-2007-01) Prinsip Perancangan M/K I

Prinsip utama dalam perancangan perangkat lunak untuk M/K ialah "*device independence*" dan "*uniform naming*". Terangkan secara singkat, yang maksud dengan kedua prinsip tersebut!

(B06-2007-02) Prinsip Perancangan M/K II

a. Dalam perancangan penjadwalan M/K, bagian mana yang sebaiknya lebih mendapatkan prioritas: Masukan atau Keluaran? Berikan alasan secukupnya.

b. Sebutkan/terangkan salah satu dampak dari menjamurnya perangkat perangkat berbasis USB terhadap perancangan dan pengembangan perangkat lunak M/K.

(B06-2006-01) Unix SystemV STREAMS

a. Terangkan dengan singkat, apa yang dimaksud dengan Unix SystemV STREAMS!
b. Sebutkan semua komponen-komponen dari sebuah STREAMS berikut gambar/diagramnya.
c. Sebutkan sebuah kelebihan dari STREAMS.
d. Sebutkan sebuah kekurangan dari STREAMS.
e. Terangkan, mengapa proses aplikasi berkomunikasi dengan kepala STREAMS secara sinkronus (*blocking*).
f. Terangkan, mengapa antar modul STREAMS berinteraksi secara asinkronus (*non-blocking*).

B.3. Penyimpanan Sekunder

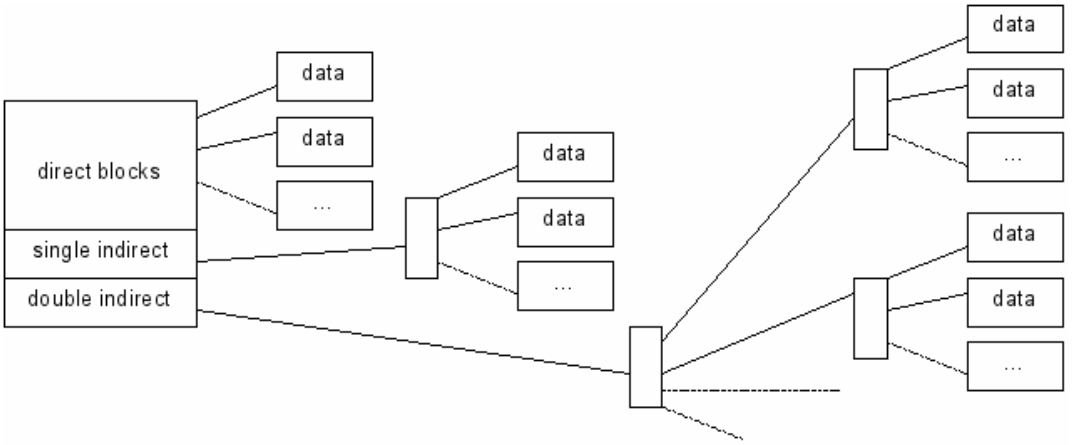
(B07-2002-01) FHS *(File Hierarchy Standards)*

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*".

(B07-2002-02) Sistem Berkas I

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?



(B07-2006-01) Sistem Berkas II

Serupa dengan B07-2002-02 di atas, namun dengan ukuran *pointer* berkas 10 *bytes*.

(B07-2003-01) Sistem Berkas III

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.

(B07-2004-01) Sistem Berkas IV

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).

(B07-2005-01) Sistem Berkas V

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!

(B07-2007-05) Sistem Berkas VI

a. Bandingkan implementasi antara sistem direktori yang "linier" dengan yang menggunakan "hash table". Terangkan kekurangan dan kelebihan masing-masing!
b. Jelaskan perbedaan antara "Very Large File System"' dan "Very Large File Size" dalam NTFS.
c. Sistem berkas modern seperti NTFS dan ext3 mengimplementasikan sistem jurnal. Terangkan apa yang dimaksud dengan "journaling" tersebut!

(B07-2003-02) Sistem Berkas "*ReiserFS*"

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!

(B07-2005-02) Sistem Berkas "*NTFS*"

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.

(B07-2004-02)(B07-2007-04) RAID (*Redudant Array of I* Disks*)

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.

(B07-2002-03) *Mass Storage System* I

Bandingkan jarak tempuh (dalam satuan silinder) antara penjadwalan 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.

(B07-2003-03) *Mass Storage System* II

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 penjadwalan "*First Come First Served*" (FCFS).
b. Hitunglah jarak tempuh (dalam satuan silinder) untuk algoritma penjadwalan "*Shortest Seek Time First*" (SSTF).

(B07-2003-04) *Mass Storage System* III

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`".

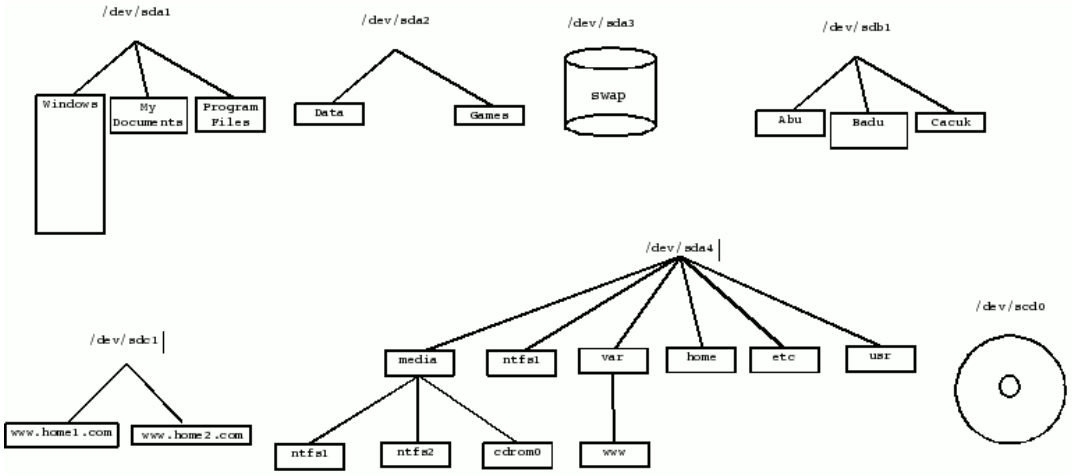
e. Bagaimana bentuk infomasi "*fdisk*" untuk "`/dev/hdc`" ini?
f. Bagaimana seharusnya isi berkas "`/etc/fstab`" setelah penambahan disk tersebut?

(B07-2007-01) *Mass Storage System* IV

Diketahui, sebuah sistem GNU/Linux dengan berkas konfigurasi "`/etc/fstab`" sebagai berikut:

```
# /etc/fstab: static file system information. ##############################
#  <fs>   <mount pointt> <type>      <options>              <dump> <pass>
proc      /proc          proc        defaults                  0  0
/dev/sda1 /media/ntfs1   ntfs        defaults,nls=utf8,umask=007 0  3
/dev/sda2 /media/ntfs2   ntfs        defaults,nls=utf8,umask=007 0  3
/dev/sda3 none           swap        sw                        0  0
/dev/sda4 /              ext3        defaults,errors=remount-ro  0  1
/dev/sdb1 /home          ext3        defaults                  0  2
/dev/sdc1 /var/www       ext3        defaults                  0  2
/dev/scd0 /media/cdrom0  udf,iso9660 user,noauto               0  0
################################################################################
```

Buat pohon (*tree*) terkait, dengan sistem-sistem berkas berikut ini:



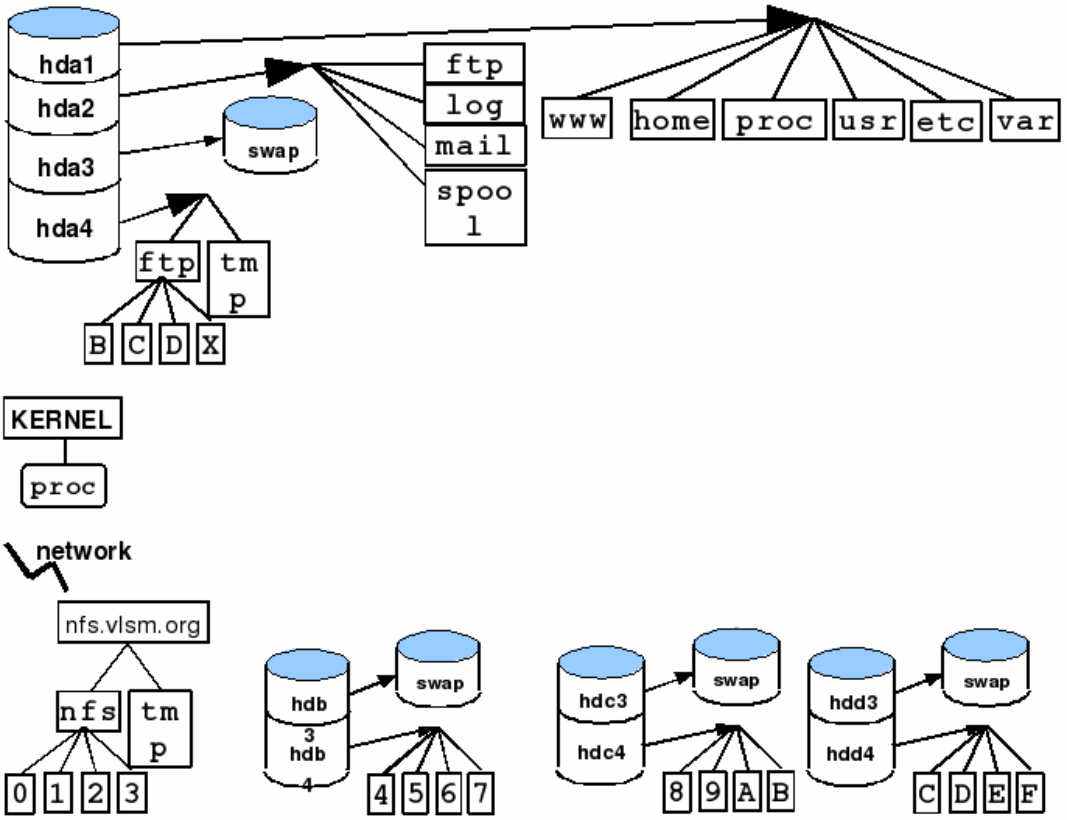
(B07-2007-03) *Mass Storage System* V

Diketahui sebuah sistem GNU/Linux dengan berkas konfigurasi "/etc/fstab" sebagai berikut:

```
################################################################################
# <filesys>      <mount point> <type>  <options>               <dump><pass>
/dev/hda3          none          swap   sw                        0  0
/dev/hdb3          none          swap   sw                        0  0
/dev/hdc3          none          swap   sw                        0  0
```

```
/dev/hdd3          none          swap   sw                          0  0
proc               /proc         proc   defaults                    0  0
/dev/hda1          /             ext3   defaults,errors=remount-ro  0  1
/dev/hda2          /var          ext3   defaults                    0  2
/dev/hda4          /www          ext3   defaults                    0  2
/dev/hdb4          /www/ftp/B    ext3   defaults                    0  2
/dev/hdc4          /www/ftp/C    ext3   defaults                    0  2
/dev/hdd4          /www/ftp/D    ext3   defaults                    0  2
nfs.vlsm.org:/nfs  /www/ftp/X    nfs    soft,ro                     0  3
##############################################################################
```

Buatkan pohon (tree) terkait, dengan menggunakan sistem-sistem berkas berikut ini:



(B07-2005-03) *CDROM*

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".

(B07-2001-01) *HardDisk* I

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?

(B07-2003-05) *HardDisk* II

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?

(B07-2004-03) *HardDisk* III

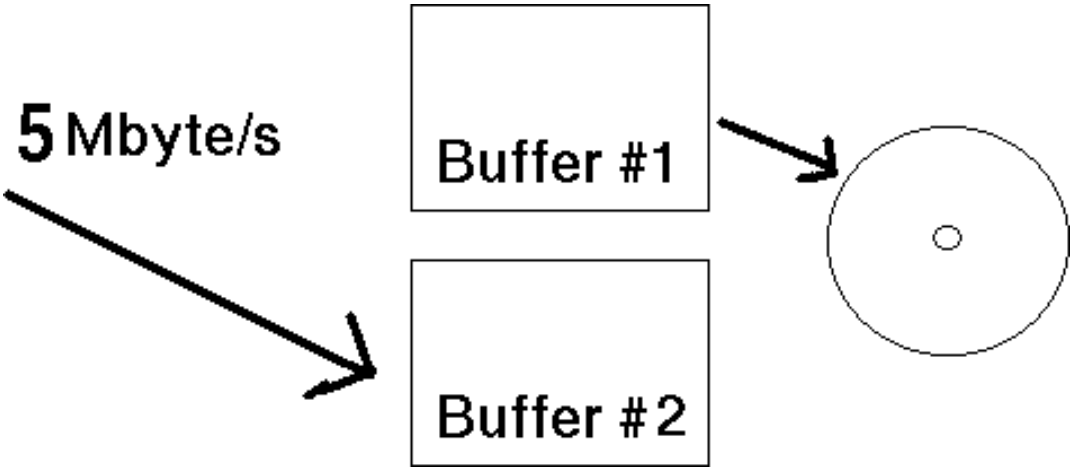
Diketahui sebuah disk dengan spesifikasi berikut ini:
- Dua (2) permukaan (muka #0 dan #1).
- Jumlah silinder: 5000 (silinder #0 - #4999).
- Kecepatan Rotasi: 6000 *rpm*.
- Kapasitas Penyimpanan: 100 *Gbytes*.
- Jumlah sektor dalam satu trak: 1000 (sektor #0 - #999).
- Waktu tempuh hingga stabil antar 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]?

(B07-2005-04) *HardDisk* IV

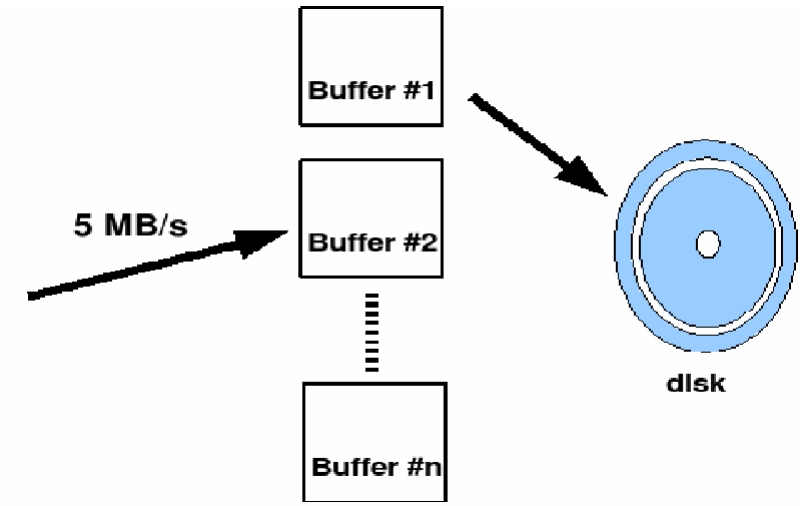
- Sebuah disk (6000 *RPM*) yang setiap *track* terdiri dari 1000 sektor @ 10 *kbytes*.
- Penulisan disk didukung dengan dua *buffer* @ 10 *kbytes* yang diisi secara bergantian dengan kecepatan 5 *Mbytes/detik* oleh sistem.
- Umpamanya, jika *buffer* #1 penuh, maka proses penulisan ke sebuah sektor disk 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?

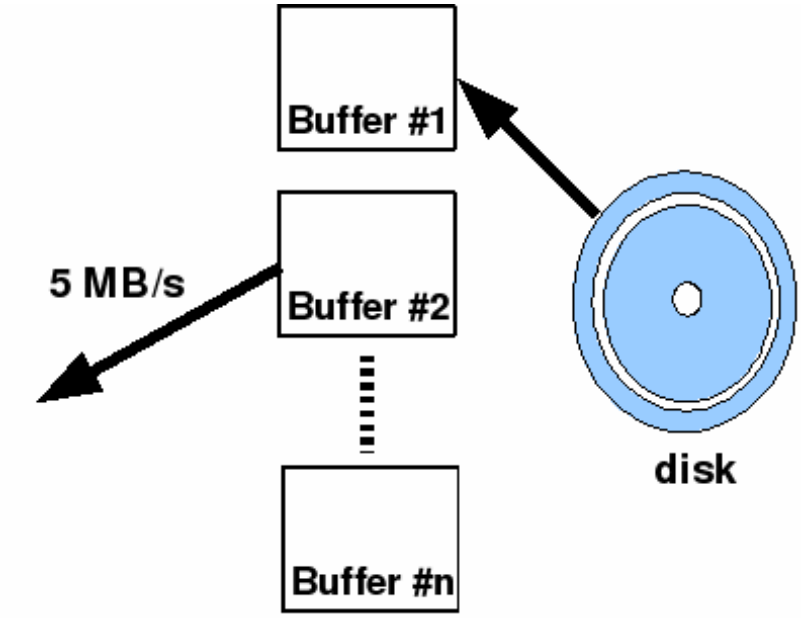
(B07-2006-02) *HardDisk* V



- Setiap TRACK pada sebuah disk (6000 RPM) terdiri dari 1000 sektor @ 10 kbytes.
- Selama ada buffer kosong (10 kbytes), sistem akan mengisi buffer secara berkesinambungan dengan kecepatan 5 Mbyte per detik.
- Setelah penuh, isi buffer akan dituliskan ke suatu sektor disk tujuan yang telah ditentukan.
- Abaikan segala tunda/delay seperti buffer switch time, seek time, kecuali Rotational Latency.
- Untuk kasus "best case", setiap buffer penuh akan langsung dituliskan ke sektor disk tujuan.
- Untuk kases "worst case", setiap buffer penuh harus menunggu satu rotasi disk sebelum dituliskan ke sektor disk tujuan.

a. BEST CASE: Berapa jumlah buffer minimum yang diperlukan, untuk memaksimalkan transfer data berkesinambungan ke disk? Berapa kecepatan transfer (efektif)?
b. WORST CASE: Berapa jumlah buffer minimum yang diperlukan, untuk memaksimalkan transfer data berkesinambungan ke disk? Berapa kecepatan transfer (efektif)?

(B07-2007-02) *HardDisk* VI



- Setiap TRACK pada sebuah disk (6000 RPM) terdiri dari 1000 sektor @ 10 kbytes.
- Saat membaca (read) disk, data dari sebuah sektor akan ditransfer ke buffer (@10 kbytes) kosong. Setelah buffer penuh, sistem akan mencari buffer kosong berikutnya.
- Data akan ditransfer dari buffer secara berkesinambungan dengan kecepatan 5 Mbyte per detik. Setelah kosong, buffer siap diisi kembali.
- Abaikan segala tunda/delay seperti buffer switch time, seek time, kecuali Rotational Latency.

a. Untuk kasus "best case", head disk selalu siap pada posisi siap baca sektor yang diinginkan.
b. Untuk kasas "worst case", head disk selalu "ketinggalan" sehingga harus menunggu satu rotasi sebelum mencapai posisi yang diinginkan.

B.4. Topik Lanjutan

(B08-2005-01) Waktu Nyata/Multimedia

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*!

(B08-2007-01) Pengaruh Perangkat Keras Terhadap Kinerja

Diketahui sebuah sistem dengan utilitasi CPU yang rendah (< 10%), namun *swap paging* yang tinggi (> 60%). Jelaskan/uraikan secara singkat, pengaruhnya terhadap utilitasi CPU dan *swap paging*; jika meng-*upgrade*/mempercanggih bagian komputer berikut ini:

a. Peningkatan kecepatan memori utama
b. Transfer Data Disk yang lebih cepat
c. Prosesor yang lebih cepat
d. Menambah Jumlah Bingkai Memori (*Memory Frame*)
e. Menambah Swap-space
f. Menambah Derajat Multiprograming

Indeks

L

M

N

P

R

S

T

U

V

W