Pengantar Sistem Operasi Komputer

Plus Ilustrasi Kernel Linux

Masyarakat Digital Gotong Royong (MDGR)

Pengantar Sistem Operasi Komputer: Plus Ilustrasi Kernel Linux

oleh Masyarakat Digital Gotong Royong (MDGR)

Diterbitkan $Date: 2005/08/26 11:42:12 $

Catatan Revisi

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, dan 81–89 Semestar Genap 2004/2005 Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia (http://rms46.vlsm.org/2/123.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 penjadualan. 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 penjadualan. 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/Penjadualan 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.

Daftar Isi

Daftar Gambar

Daftar Tabel

Daftar Contoh

Kata Pengantar

1. Kata Pengantar Revisi 3.0

Namanya juga usaha... kelihatannya versi 3.0 buku ini merupakan *face lift* dari versi sebelum, ha! Mudah-mudahan, versi 4.0 mendatang (IA 2006) lebih sempurna dari versi 3.0 ini. Semenjak 2004, buku ini dipergunakan sebagai rujukan utama Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia (http://rms46.vlsm.org/2/123.html -- http://www.cs.ui.ac.id/). Versi digital terakhir dari buku ini dapat diambil dari http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/BUKU/. Buku ini akan diperbaiki sambil jalan. Diskusi yang terkait dengan bidang Sistem Operasi secara umum, maupun yang khusus seputar buku ini, diselenggarakan melalui milis Sistem Operasi [http://groups.yahoo.com/group/sistemoperasi/]. Kritik/tanggapan/usulan juga dapat disampaikan ke `<writeme05 <at> yahoogroups dot com>`.

Pokok Bahasan

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

Para pembaca sepertinya pernah mendengar istilah "sistem operasi". Mungkin pula pernah berhubungan secara langsung atau pun tidak langsung dengan istilah tersebut. Namun, belum tentu dapat menjabarkan perihal apa yang sebetulnya dimaksud dengan kata "sistem operasi". Pembahasan buku ini diawali dengan pengenalan konsep dasar sistem komputer (Bagian I, “Konsep Dasar Perangkat Komputer”). Bagian ini, akan menguraikan secara umum komponen-komponen komputer seperti sistem operasi, perangkat keras, proteksi, keamanan, jaringan, hak kekayaan intelektual, lisensi, konsep perangkat lunak bebas (PLB), beserta tantangannya.

Aspek-aspek tersebut diperlukan untuk memahami konsep-konsep sistem operasi yang akan dijabarkan dalam buku ini. Tentunya tidak dapat diharapkan pembahasan yang dalam. Rincian lanjut, sebaiknya dilihat pada rujukan yang berhubungan dengan "Pengantar Organisasi Komputer", "Pengantar Struktur Data", serta "Pengantar Jaringan Komputer".

Bagian II, “Konsep Dasar Sistem Operasi” merupakan penjabaran secara garis besar dari sebuah sistem operasi secara umum, serta GNU/Linux secara khusus. Ini akan membantu pemahaman konsep secara terpadu, tanpa harus terganggu dengan rinciannya. Menguasai bagian ini akan mempermudah pemahanan sisa dari buku ini.

Bagian III, “Proses dan Penjadualan” mulai membahas bagian utama dari sebuah sistem operasi, yaitu proses dan penjadualannya. Dalam delapan sub-pokok bahasan (bab), akan dibahas definisi, konsep *thread*, implementasi *thread* dalam Java, konsep dan algoritma penjadualan. Bagian ini akan ditutup dengan memperkenalkan implementasi proses Linux.

Bagian IV, “Proses dan Sinkronisasi” akan membahas aspek sinkronisasi proses. Dalam tujuh sub-pokok bahasan (bab), akan dibahas konsep-konsep seperti interaksi proses, sinkronisasi, *race condition*, *critical section*, serta *deadlock*. Untuk mempermudah pemahaman konsep-konsep abtrak ini, tiga bab penutup akan merupakan ilustrasi aspek sinkronisasi dalam bahasa Java.

Membahas keempat bagian tersebut di atas memerlukan waktu sekitar tujuh minggu semester. Dengan demikian, merupakan saat yang tepat untuk melakukan ujian tengah semester.

Bagian V, “Memori” akan membahas aspek-aspek pengelolaan memori seperti, pengalamat logika dan fisik, *swap*, halaman (*page*), bingkai (*frame*), memori virtual. segmentasi, alokasi memori. Bagian ini akan ditutup dengan penguraian pengelolaan memori Linux.

Bagian VI, “Memori Sekunder” akan membahas Memori Sekunder. Bab 37 membahas Sistem Berkas, yaitu: Konsep Berkas, Atribut berkas, Jenis Berkas, Operasi Berkas, Struktur Berkas, serta Metode Akses. Bab 38 membahas Struktur Direktori, yaitu: Operasi Direktori, Direktori Satu

Tingkat, Direktori Dua Tingkat, Direktori dengan Struktur Tree, Direktori dengan Struktur Graf Asiklik, serta Direktori dengan Struktur Graf Umum. Bab 39 membahas Sistem Berkas Jaringan, yaitu: Sharing, Proteksi, serta Mounting. Bab 40 membahas Implementasi Sistem Berkas, yaitu: Struktur Sistem Berkas, Implementasi Sistem Berkas, serta Implementasi Direktori. Bab 41 membahas Filesystem Hierarchy Standard (FHS), yaitu: Sistem Berkas, Sistem Berkas Root, Hirarki /usr, serta Hirarki /var. Bab 42 membahas Konsep Alokasi Blok Sistem Berkas, yaitu: Metode Alokasi, Manajemen Ruang Kosong, Efisiensi dan Kinerja, Recovery, Log-Structured File System, Sistem Berkas Linux Virtual, Operasi-operasi Dalam Inode, Pembagian Sistem Berkas Secara Ortogonal, serta Sistem Berkas Linux.

Bagian VII, “Masukan/Keluaran (M/K)” akan membahas Masukan/Keluaran (M/K). Bab 43 membahas Perangkat Keras Keluaran/Masukan, yaitu: Perangkat M/K, Pengendali Perangkat, Polling, Interupsi, serta Direct Memory Access (DMA). Bab 44 membahas Aplikasi Antarmuka M/K; Subsistem Kernel; Operasi Perangkat Keras, yaitu: Aplikasi Antarmuka M/K, Kernel M/K Subsystem, Penanganan Permintaan M/K, M/K Streams, serta Kinerja M/K. Bab 45 membahas Managemen Disk, yaitu: Struktur Disk, Penjadualan Disk, Penjadualan FCFS, Penjadualan SSTF, Penjadualan SCAN, Penjadualan C-SCAN, Penjadualan LOOK, Penjadualan C-LOOK, serta Pemilihan Algoritma Penjadualan Disk. Bab 46 membahas Managemen Disk; Swap, Struktur RAID; Kaitan Langsung dan Jaringan; Implementasi Penyimpanan Stabil, yaitu: Managemen Disk, Managemen Ruang Swap, Struktur RAID, serta Kaitan Disk. Bab 47 membahas Perangkat Penyimpanan Tersier, yaitu: Macam-macam Struktur Penyimpanan Tersier, Future Technology, Aplikasi Antarmuka, serta Masalah Kinerja. Bab 48 membahas Keluaran/Masukan Linux, yaitu: Device Karakter, Device Blok, serta Device Jaringan.

Bagian VIII, “Topik Lanjutan” akan membahas beberapa Topik Lanjutan. Bab 49 membahas Sistem Waktu Nyata dan Multimedia, yaitu: Kernel Waktu Nyata, Penjadual Proses, Penjadual Disk, Managemen Berkas, Managemen Jaringan, serta Kompresi. Bab 50 membahas Sistem Terdistribusi, yaitu: Variasi Sistem, Topologi Jaringan, Sistem Berkas, Replikasi Berkas, Mutex, Middleware, Aplikasi, serta Kluster. Bab 51 membahas Keamanan Sistem, yaitu: Manusia dan Etika, Kebijaksanaan Pengamanan, Keamanan Fisik, Keamanan Perangkat Lunak, Keamanan Jaringan, Kriptografi, Operasional, BCP/DRP, serta Proses Audit. Bab 52 membahas Perancangan dan Pemeliharaan, yaitu: Perancangan Antarmuka, Implementasi, Implementasi Sistem, Kinerja, Pemeliharaan Sistem, serta Trend.

Setiap bab berisi soal-soal latihan agar para pembaca dapat mengulas kembali pembahasan pada bab tersebut dan mengevaluasi sejauh mana pengetahuan mengenai bab tersebut. Gambar dipilih sedemikian rupa sehingga dapat memberikan ilustrasi yang membantu pembaca untuk lebih memahami pembahasan. Selain itu, juga akan diselipkan studi kasus yaitu sistem GNU/Linux.

2. Kata Pengantar Revisi 2.0

Maaf -- selesai tidak selesai -- mulai semester ganjil 2004/2005, buku ini digunakan sebagai rujukan utama mata ajar IKI-20230/80230 Sistem Operasi [http://rms46.vlsm.org/2/117.html] di Fakultas Ilmu Komputer Universitas Indonesia. Selanjutnya, buku ini akan diperbaiki sambil jalan. Diharapkan, revisi tiga akan jauh lebih baik daripada revisi ini. Seperti biasa, silakan menyampaikan kritik/tanggapan/usulan anda ke `<writeme04 <@T> yahoogroups DOT com>`. Maaf.

3. Kata Pengantar Revisi 1.0

Buku ini merupakan hasil karya Masyarakat Digital Gotong Royong (MDGR) Fakultas Ilmu Komputer Universitas Indonesia (Fasilkom UI). Kelompok Kerja 21-28 mengawali penulisan buku ini, lalu Kelompok Kerja 41-49, 51 melakukan revisi dan perbaikan. Tujuan utama penulisan buku ini ialah untuk dimanfaatkan sendiri sebagai rujukan utama pada mata ajar IKI-20230 Sistem Operasi [http://rms46.vlsm.org/2/101.html] di Fakultas Ilmu Komputer Universitas Indonesia [http://www.cs.ui.ac.id/]. Versi digital terakhir dari buku ini dapat diambil dari http://bebas.vlsm.org/ v06/ Kuliah/ SistemOperasi/ BUKU/ [http://bebas.vlsm.org/v06/Kuliah/SistemOperasi/BUKU/].

Buku ini mencakup delapan pokok bahasan bidang Sistem Operasi. Setiap pokok bahasan dipisahkan ke dalam bab yang tersendiri, yang kemudian dibagi menjadi beberapa sub-pokok bahasan. Setiap sub-pokok bahasan dirancang untuk mengisi satu jam tatap muka kuliah. Buku yang

terdiri dari 52 sub-pokok bahasan ini, sehingga cocok untuk sebuah mata ajar dengan bobot empat Satuan Kredit Semester (SKS).

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

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

Bab 3 akan membahas Proses dan Penjadualan.

Bab 4 akan membahas Sinkronisasi dan *Deadlock*.

Bab 5 akan membahas Managemen Memori.

Bab 6 akan membahas Sistem Berkas.

Bab 7 akan membahas M/K.

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

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

Kami menyadari bahwa pada buku ini sangat *berbau* buku karya Schilberschatz dan kawan-kawan. Kebanyakan sub-pokok bahasan buku ini memang berbasis kerangka kerja (*framework*) buku tersebut. Diharapkan secara perlahan, bau tersebut akan pudar pada revisi-revisi yang mendatang. Silakan menyampaikan kritik/tanggapan/usulan anda ke `<writeme04 <@T> yahoogroups DOT com>`.

Bagian I. Konsep Dasar Perangkat Komputer

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

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

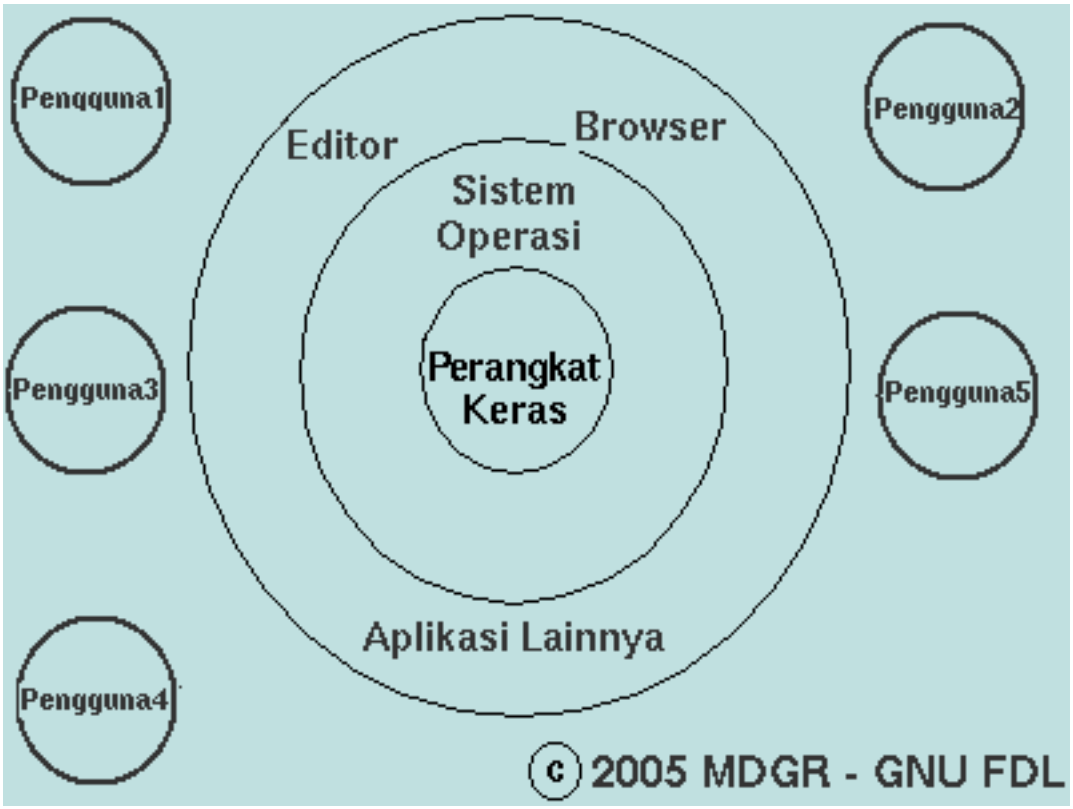
Bab 1. Hari Gini Belajar SO?

1.1. Definisi Sementara

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

Mendefinisikan istilah "sistem operasi" mungkin merupakan hal yang mudah, namun mungkin juga merupakan hal yang sangat ribet! Para pembaca sepertinya pernah mendengar istilah "sistem operasi". Mungkin pula pernah berhubungan secara langsung atau pun tidak langsung dengan istilah tersebut. Namun, belum tentu dapat menjabarkan perihal apa yang sebetulnya dimaksud dengan kata "sistem operasi". Sebaliknya, banyak pula yang pernah mendengar merek dagang "*Windows*™ [1])" atau pun istilah "GNU/Linux [2])", lalu mengidentikkan nama *Windows* ™ atau GNU/Linux dengan istilah "sistem operasi" tersebut.

Gambar 1.1. Abstraksi Komponen Sistem Komputer



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

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

keseluruhan, perangkat lunak membantu para pengguna untuk memanfaatkan sumber daya komputasi yang disediakan perangkat keras.

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

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

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

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

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

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

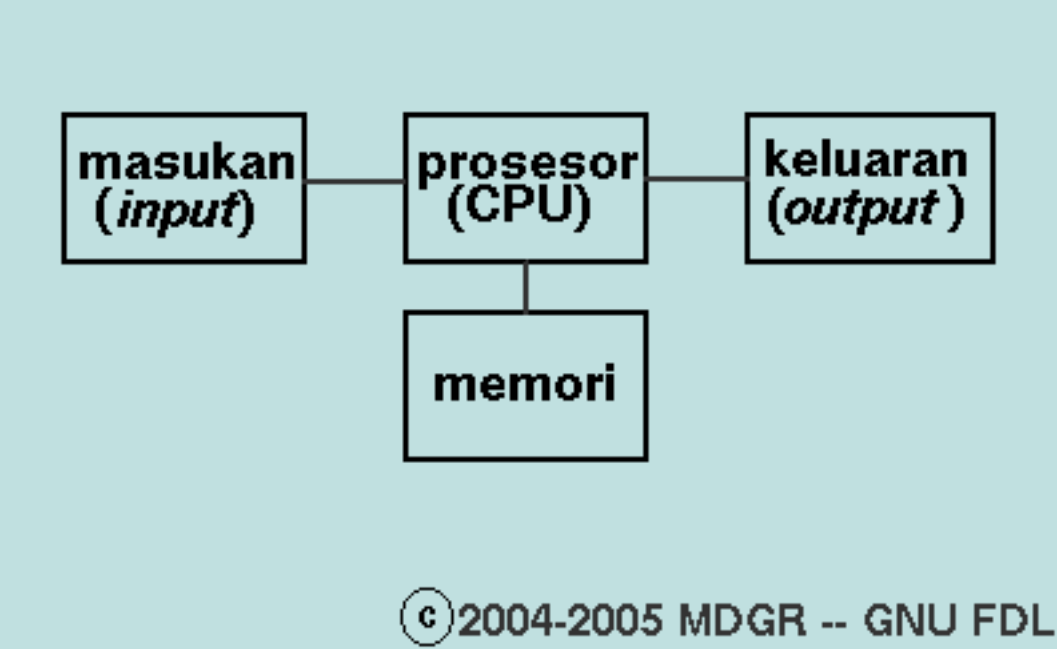
Dapat disimpulkan, bahwa sistem operasi merupakan komponen penting dari setiap sistem komputer. Akibatnya, pelajaran "sistem operasi" selayaknya merupakan komponen penting dari sistem pendidikan berbasis "ilmu komputer". Konsep sistem operasi dapat lebih mudah dipahami, jika juga memahami jenis perangkat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

[3] Internet Explorer merupakan merek dagang terdaftar dari Microsoft.

1.2. Sejarah Perkembangan

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

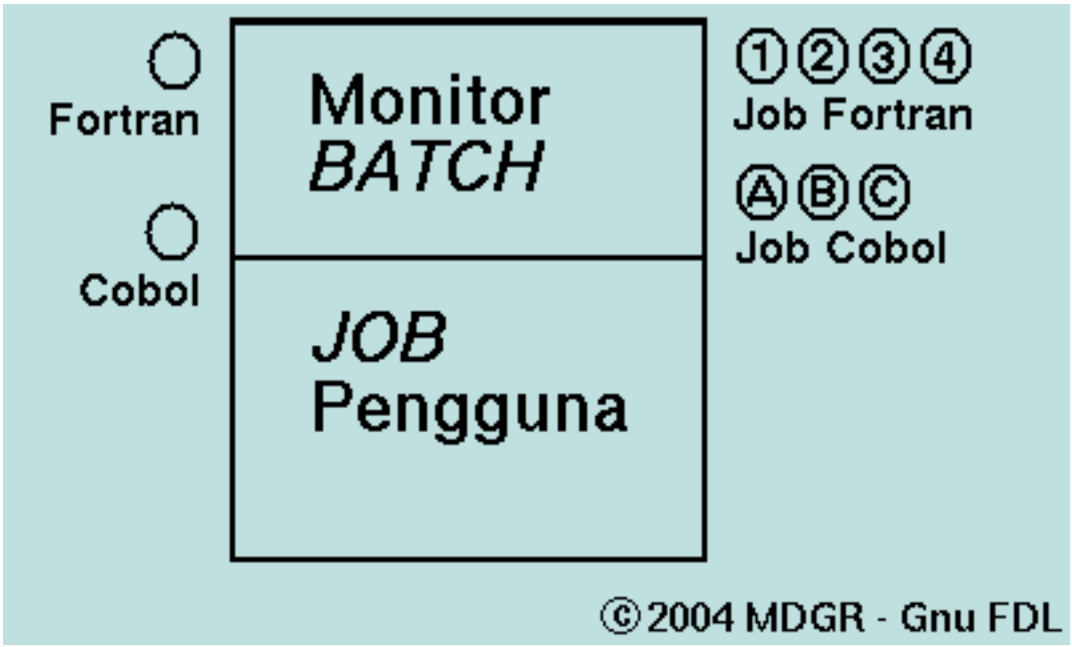
Gambar 1.2. Arsitektur Komputer von Neumann



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

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

Gambar 1.3. Bagan Memori Untuk Sistem *Monitor Batch* Sederhana



Perkembangan sistem operasi dimulai dari sini, dengan memanfaatkan sistem *batch* (Gambar 1.3, “Bagan Memori Untuk Sistem *Monitor Batch* Sederhana”). Para operator mengumpulkan *job-job* yang mirip yang kemudian dijalankan secara berkelompok. Umpama, *job* yang memerlukan kompilator Fortran akan dikumpulkan ke dalam sebuah *batch* bersama dengan *job-job* lainnya yang juga memerlukan kompilator Fortran. Setelah sebuah kelompok *job* rampung, maka kelompok *job* berikutnya akan dijalankan secara otomatis.

Gambar 1.4. Bagan Memori untuk Model *Multiprogram System*

Pada perkembangan berikutnya, diperkenalkan konsep *Multiprogrammed System*. Dengan sistem ini *job-job* disimpan di memori utama di waktu yang sama dan *CPU* dipergunakan bergantian. Hal ini

membutuhkan beberapa kemampuan tambahan yaitu: penyediaan *I/O routine* oleh sistem, pengaturan memori untuk mengalokasikan memori pada beberapa *Job*, penjadualan *CPU* untuk memilih *job* mana yang akan dijalankan, serta pengalokasian perangkat keras lain (Gambar 1.4, “Bagan Memori untuk Model *Multiprogram System*”).

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

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

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

1.3. Alasan Mempelajari Sistem Operasi

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

Tabel 1.1. Perbandingan Sistem Dahulu dan Sekarang

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

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

Terlepas dari perubahan tersebut di atas; banyak aspek yang tetap sama seperti dahulu. Komputer abad lalu menggunakan model arsitektur von-Neumann, dan demikian pula model komputer abad ini. Aspek pengelolaan sumber daya sistem operasi seperti proses, memori, masukan/keluaran (m/k),

berkas, dan seterusnya masih menggunakan prinsip-prinsip yang sama. Dengan sendirinya, mempelajari sistem operasi masih tetap serelevan abad lalu; walaupun telah terjadi berbagai perubahan fisik.

1.4. Bahan Pembahasan

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

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

1.5. Bahan Yang Tidak Akan Dibahas

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

1.6. Prasyarat

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

1.7. Sasaran Pembelajaran

Sasaran utama yang diharapkan setelah mendalami buku ini ialah:

- Mengenal komponen-komponen yang membentuk sistem operasi.
- Dapat menjelaskan peranan dari masing-masing komponen tersebut.
- Seiring dengan pengetahuan yang didapatkan dari Organisasi Komputer, dapat menjelaskan atau meramalkan kinerja dari aplikasi yang berjalan di atas sistem operasi dan perangkat keras tersebut.
- Landasan/fondasi bagi mata ajar lainnya, sehingga dapat menjelaskan konsep-konsep bidang tersebut.

1.8. Rangkuman

Sistem operasi telah berkembang selama lebih dari 40 tahun dengan dua tujuan utama. Pertama, sistem operasi mencoba mengatur aktivitas-aktivitas komputasi untuk memastikan pendayagunaan yang baik dari sistem komputasi tersebut. Kedua, menyediakan lingkungan yang nyaman untuk

pengembangan dan jalankan dari program.

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

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

1.9. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:
   - *OS View: "Resource Allocator" vs. "Control Program"*.
   - *Operating System Goal: "Convenient" vs. "Efficient"*.
   - *Job: "Batch system" vs. "Time-Sharing System"*.
2. Sebutkan tiga tujuan utama dari sebuah Sistem Operasi!
3. Apakah keuntungan utama dari *multiprogramming*?
4. Apakah perbedaan utama antara *Mainframe* dengan *PC*?

Rujukan

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

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

Bab 2. Perangkat Lunak Bebas

2.1. Pendahuluan

Era "Ekonomi Lama" telah melintas ambang senja – era "Ekonomi Baru" (*New Economy*) yang berbasis Teknologi Informasi dan Komunikasi (TIK) pun mulai menyingsing! Sebuah kekonyolan transisi yang memalukan – "gelembung **dot com** menjelang akhir 1990-an." – pun telah kita alami. Walaupun demikian, TIK masih tetap berperan penting dalam era baru ini. Demikian pula, sebuah fenomena TIK yaitu pemanfaatan Perangkat Lunak Bebas (PLB) dan/atau *Open Source Software* (OSS).

Buku ini berisi banyak ilustrasi Sistem Operasi GNU/Linux yang berbasis PLB. Untuk itu, bab ini akan membahas secara khusus konsep PLB (*Free Software*) tersebut. PLB ini sering disalah-kaprahkan sebagai serupa OSS, namun sebetulnya terdapat beberapa berbedaan yang mendasar. Pembahasan ini bukan bertujuan sebagai indoktrinasi faham tersebut! Justru yang diharapkan, bab ini akan meluruskan persepsi-persepsi keliru terhadap PLB dan/atau OSS, seperti:

- Para penulis program komputer tidak berhak digaji layak.
- PLB tidak boleh dijual/dikomersialkan.
- PLB wajib disebarluaskan.
- Perbedaan dasar antara PLB dan OSS

Setelah menyimak bab ini, diharapkan akan lebih memahami dan lebih menghargai makna PLB secara umum, terutama Sistem Operasi yang bebas.

2.2. Hak Kekayaan Intelektual

Latar belakang

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

``Hak" itu sendiri dapat dibagi menjadi dua. Pertama, ``Hak Dasar (Azasi)", yang merupakan hak mutlak yang tidak dapat diganggu-gugat. Umpama, hak untuk hidup, hak untuk mendapatkan keadilan, dan sebagainya. Kedua, ``Hak Amanat Aturan/Perundangan" yaitu hak karena dibarikan/diatur oleh masyarakat melalui peraturan/perundangan. Di berbagai negara, termasuk Amrik dan Indonesia, HKI merupakan "Hak Amanat Aturan", sehingga masyarakatlah yang menentukan, seberapa besar HKI yang diberikan kepada individu dan kelompok.

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

Tumbuhnya konsepsi kekayaan atas karya-karya intelektual pada akhirnya juga menimbulkan untuk melindungi atau mempertahankan kekayaan tersebut. Pada gilirannya, kebutuhan ini melahirkan konsepsi perlindungan hukum atas kekayaan tadi, termasuk pengakuan hak terhadapnya. Sesuai

dengan hakekatnya pula, HKI dikelompokan sebagai hak milik perorangan yang sifatnya tidak berwujud (Intangible).

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

Aneka Ragam HKI

Hak Cipta (*Copyright*)

Berdasarkan pasal 1 ayat 1 Undang-Undang Nomor 19 Tahun 2002 Tentang Hak Cipta: Hak Cipta adalah hak eksklusif bagi Pencipta atau penerima hak untuk mengumumkan atau memperbanyak ciptaannya atau memberikan izin untuk itu dengan tidak mengurangi pembatasan-pembatasan menurut peraturan perundang-undangan yang berlaku.

Paten (*Patent*)

Berdasarkan Pasal 1 ayat 1 Undang-Undang Nomor 14 Tahun 2001 Tentang Paten: Paten adalah hak eksklusif yang diberikan oleh Negara kepada Inventor atas hasil Invensinya di bidang teknologi, yang untuk selama waktu tertentu melaksanakan sendiri Invensinya tersebut atau memberikan persetujuannya kepada pihak lain untuk melaksanakannya.

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

Merk Dagang (*Trademark*)

Berdasarkan pasal 1 ayat 1 Undang-Undang Nomor 15 Tahun 2001 Tentang Merek: Merek adalah tanda yang berupa gambar, nama, kata, huruf-huruf, angka-angka, susunan warna, atau kombinasi dari unsur-unsur tersebut yang memiliki daya pembeda dan digunakan dalam kegiatan perdagangan barang atau jasa.

Merk dagang digunakan oleh pebisnis untuk mengidentifikasikan sebuah produk atau layanan. Merk dagang meliputi nama produk atau layanan, beserta logo, simbol, gambar yang menyertai produk atau layanan tersebut. Contoh merk dagang misalnya adalah Kentucky Fried Chicken. Yang disebut merk dagang adalah urut-urutan kata-kata tersebut beserta variasinya (misalnya KFC), dan logo dari produk tersebut. Jika ada produk lain yang sama atau mirip, misalnya Ayam Goreng Kentucky, maka itu adalah termasuk sebuah pelanggaran merk dagang. Berbeda dengan HKI lainnya, merk dagang dapat digunakan oleh pihak lain selain pemilik merk dagang tersebut, selama merk dagang tersebut digunakan untuk mereferensikan layanan atau produk yang bersangkutan. Sebagai contoh, sebuah artikel yang membahas KFC dapat saja menyebutkan Kentucky Fried Chicken di artikelnya, selama perkataan itu menyebut produk dari KFC yang sebenarnya. Merk dagang diberlakukan setelah pertama kali penggunaan merk dagang tersebut atau setelah registrasi. Merk dagang berlaku pada negara tempat pertama kali merk dagang tersebut digunakan atau didaftarkan. Tetapi ada beberapa perjanjian yang memfasilitasi penggunaan merk dagang di negara lain. Misalnya adalah sistem Madrid. Sama seperti HKI lainnya, merk dagang dapat diserahkan kepada pihak lain, sebagian atau seluruhnya. Contoh yang umum adalah mekanisme franchise. Pada franchise, salah satu kesepakatan adalah penggunaan nama merk dagang dari usaha lain yang sudah terlebih dahulu sukses.

Rahasia Dagang (*Trade Secret*)

Menurut pasal 1 ayat 1 Undang-Undang Nomor 30 Tahun 2000 Tentang Rahasia Dagang: Rahasia Dagang adalah informasi yang tidak diketahui oleh umum di bidang teknologi dan/atau bisnis, mempunyai nilai ekonomi karena berguna dalam kegiatan usaha, dan dijaga kerahasiaannya oleh pemilik Rahasia Dagang.

Berbeda dari jenis HKI lainnya, rahasia dagang tidak dipublikasikan ke publik. Sesuai namanya, rahasiadagang bersifat rahasia. Rahasia dagang dilindungi selama informasi tersebut tidak dibocorkan olehpemilik rahasia dagang. Contoh dari rahasia dagang adalah resep minuman Coca Cola. Untuk beberapa tahun, hanya Coca Cola yang memiliki informasi resep tersebut. Perusahaan lain tidak berhak untuk mendapatkan resep tersebut, misalnya dengan membayar pegawai dari Coca Cola. Cara yang legal untuk mendapatkan resep tersebut adalah dengan cara rekayasa balik (reverse engineering). Sebagai contoh, hal ini dilakukan oleh kompetitor Coca Cola dengan menganalisis kandungan dari minuman Coca Cola. Hal ini masih legal dan dibenarkan oleh hukum. Oleh karena itu saat ini ada minuman yang rasanya mirip dengan Coca Cola, semisal Pepsi atau RC Cola.

Service Mark

Adalah kata, prase, logo, sysmbol, warna, suara, bau yang digunakan oleh sebuah bisnis untuk mengindentifikasi sebuah layanan dan membedakannya dari kompetitornya. Pada prakteknya legal protection untuk trademark sedang service mark untuk identitasnya.

Desain Industri

Berdasarkan pasal 1 ayat 1 Undang-Undang Nomor 31 Tahun 2000 Tentang Desain Industri: Desain Industri adalah suatu kreasi tentang bentuk, konfigurasi, atau komposisi garis atau warna, atau garis dan warna, atau gabungan daripadanya yang berbentuk tiga dimensi atau dua dimensi yang memberikan kesan estetis dan dapat diwujudkan dalam pola tiga dimensi atau dua dimensi serta dapat dipakai untuk menghasilkan suatu produk, barang, komoditas industri, atau kerajinan tangan.

Desain Tata Letak Sirkuit Terpadu

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

Ayat 1: Sirkuit Terpadu adalah suatu produk dalam bentuk jadi atau setengah jadi, yang di dalamnya terdapat berbagai elemen dan sekurang-kurangnya satu dari elemen tersebut adalah elemen aktif, yang sebagian atau seluruhnya saling berkaitan serta dibentuk secara terpadu di dalam sebuah bahan semikonduktor yang dimaksudkan untuk menghasilkan fungsi elektronik.

Ayat 2: Desain Tata Letak adalah kreasi berupa rancangan peletakan tiga dimensi dari berbagai elemen, sekurang-kurangnya satu dari elemen tersebut adalah elemen aktif, serta sebagian atau semua interkoneksi dalam suatu Sirkuit Terpadu dan peletakan tiga dimensi tersebut dimaksudkan untuk persiapan pembuatan Sirkuit Terpadu.

Indikasi Geografis

Berdasarkan pasal 56 ayat 1 Undang-Undang No. 15 Tahun 2001 Tentang Merek: Indikasi-geografis dilindungi sebagai suatu tanda yang menunjukkan daerah asal suatu barang yang karena faktor lingkungan geografis termasuk faktor alam, faktor manusia, atau kombinasi dari kedua faktor tersebut, memberikan ciri dan kualitas tertentu pada barang yang dihasilkan.

2.3. HKI Perangkat Lunak

Bagaimana dengan HKI Perangkat Lunak? Di Indonesia, ini termasuk ke dalam kategori Hak Cipta (*Copyright*). Beberapa negara, memperbolehkan pematenan perangkat lunak. Pada industri

perangkat lunak, sangat umum perusahaan besar memiliki portfolio paten yang berjumlah ratusan, bahkan ribuan. Sebagian besar perusahaan-perusahaan ini memiliki perjanjian cross-licensing, artinya Saya izinkan anda menggunakan paten saya asalkan saya boleh menggunakan paten anda. Akibatnya hukum paten pada industri perangkat lunak sangat merugikan perusahaan-perusahaan kecil yang cenderung tidak memiliki paten. Tetapi ada juga perusahaan kecil yang menyalahgunakan hal ini. Misalnya Eolas yang mematenkan teknologi plug-in pada web browser. Untuk kasus ini, Microsoft tidak dapat menyerang balik Eolas, karena Eolas sama sekali tidak membutuhkan paten yang dimiliki oleh Microsoft. Eolas bahkan sama sekali tidak memiliki produk atau layanan, satu-satunya hal yang dimiliki Eolas hanyalah paten tersebut. Oleh karena itu, banyak pihak tidak setuju terhadap paten perangkat lunak karena sangatmerugikan industri perangkat lunak. Sebuah paten berlaku di sebuah negara. Jika sebuah perusahaan ingin patennya berlaku di negara lain, maka perusahaan tersebut harus mendaftarkan patennya di negara lain tersebut. Tidak seperti hak cipta, paten harus didaftarkan terlebih dahulu sebelum berlaku.

Berikut akan diuraikan beberapa jenis lisensi perangkat lunak.

Perangkat Lunak Berpemilik

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

Perangkat lunak komersial adalah perangkat lunak yang dikembangkan oleh kalangan bisnis untuk memperoleh keuntungan dari penggunaannya. ``Komersial'' dan ``kepemilikan'' adalah dua hal yang berbeda! Kebanyakan perangkat lunak komersial adalah berpemilik, tapi ada perangkat lunak bebas komersial, dan ada perangkat lunak tidak bebas dan tidak komersial. Harap sebarkan ke khalayak, perangkat lunak bebas komersial merupakan sesuatu yang mungkin. Sebaiknya, anda jangan mengatakan ``komersial'' ketika maksud anda ialah ``berpemilik''.

Perangkat Lunak Semi-Bebas

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

Program semi-bebas memiliki batasan-batasan tambahan, yang dimotivasi oleh tujuan pribadi semata. Sangat mustahil untuk menyertakan perangkat lunak semi-bebas pada sistem operasi bebas. Hal ini karena perjanjian distribusi untuk sistem operasi keseluruhan adalah gabungan dari perjanjian distribusi untuk semua program di dalamnya. Menambahkan satu program semi-bebas pada sistem akan membuat keseluruhan sistem menjadi semi-bebas. Perangkat lunak semi-bebas adalah perangkat lunak yang tidak bebas, tapi mengizinkan setiap orang untuk menggunakan, menyalin, mendistribusikan, dan memodifikasinya (termasuk distribusi dari versi yang telah dimodifikasi) untuk tujuan non-laba. PGP adalah salah satu contoh dari program semi-bebas. Perangkat lunak semi-bebas jauh lebih baik dari perangkat lunak berpemilik, namun masih ada masalah, dan seseorang tidak dapat menggunakannya pada sistem operasi yang bebas.

Public Domain

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

Freeware

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

Shareware

Shareware ialah perangkat lunak yang mengizinkan orang-orang untuk meredistribusikan salinannya, tetapi mereka yang terus menggunakannya diminta untuk membayar biaya lisensi. Shareware bukan perangkat lunak bebas atau pun semi-bebas. Ada dua alasan untuk hal ini, yakni: Sebagian besar shareware, kode programnya tidak tersedia; jadi anda tidak dapat memodifikasi program tersebut sama sekali. *Shareware* tidak mengizinkan seseorang untuk membuat salinan dan memasangnya tanpa membayar biaya lisensi, tidak juga untuk orang-orang yang terlibat dalam kegiatan nirlaba. Dalam prakteknya, orang-orang sering tidak mempedulikan perjanjian distribusi dan tetap melakukan hal tersebut, tapi sebenarnya perjanjian tidak mengizinkannya.

Copylefted/ Non-Copylefted

Perangkat lunak *copylefted* merupakan perangkat lunak bebas yang ketentuan pendistribusinya tidak memperbolehkan untuk menambah batasan-batasan tambahan – jika mendistribusikan atau memodifikasi perangkat lunak tersebut. Artinya, setiap salinan dari perangkat lunak, walaupun telah dimodifikasi, haruslah merupakan perangkat lunak bebas. Dalam proyek GNU, kami meng-*copyleft*-kan hampir semua perangkat lunak yang kami buat, karena tujuan kami adalah untuk memberikan kebebasan kepada semua pengguna seperti yang tersirat dalam istilah ``perangkat lunak bebas". Copyleft merupakan konsep yang umum. Jadi, untuk meng-*copyleft*-kan sebuah program, anda harus menggunakan ketentuan distribusi tertentu. Terdapat berbagai cara untuk menulis perjanjian distribusi program *copyleft*.

Perangkat lunak bebas non-*copyleft* dibuat oleh pembuatnya yang mengizinkan seseorang untuk mendistribusikan dan memodifikasi, dan untuk menambahkan batasan-batasan tambahan dalamnya. Jika suatu program bebas tapi tidak *copyleft*, maka beberapa salinan atau versi yang dimodifikasi bisa jadi tidak bebas sama sekali. Perusahaan perangkat lunak dapat mengkompilasi programnya, dengan atau tanpa modifikasi, dan mendistribusikan file tereksekusi sebagai produk perangkat lunak yang berpemilik. Sistem X Window menggambarkan hal ini. Konsorsium X mengeluarkan X11 dengan ketentuan distribusi yang menetapkannya sebagai perangkat lunak bebas non-*copyleft*. Jika anda menginginkannya, anda dapat memperoleh salinan yang memiliki perjanjian distribusi dan juga bebas. Namun ada juga versi bebasnya, dan ada workstation terkemuka serta perangkat grafik PC, dimana versi yang tidak bebas merupakan satu-satunya yang dapat bekerja disini. Jika anda menggunakan perangkat keras tersebut, X11 bukanlah perangkat lunak bebas bagi anda.

FM Perangkat Lunak Seharusnya Tidak ada Pemiliknya! (FSF)

GNU General Public License (GNU/GPL)

GNU GPL (General Public License) (20k huruf) merupakan sebuah kumpulan ketentuan pendistribusian tertentu untuk meng-copyleft-kan sebuah program. Proyek GNU menggunakannya sebagai perjanjian distribusi untuk sebagian besar perangkat lunak GNU. Pembatasan dari copyleft dirancang untuk melindungi kebebasan bagi semua pengguna. Bagi pihak GNU, satu-satunya alasan untuk membatasi substantif dalam menggunakan program – ialah melarang orang lain untuk menambahkan batasan lain.

Sistem GNU

Sistem GNU merupakan sistem serupa Unix yang seutuhnya bebas. Sistem operasi serupa Unix terdiri dari berbagai program. Sistem GNU mencakup seluruh perangkat lunak GNU, dan juga paket program lain, seperti sistem X Windows dam TeX yang bukan perangkat lunak GNU. Pengembangan sistem GNU ini telah dilakukan sejak tahun 1984. Pengedaran awal (percobaan) dari sistem GNU lengkap dilakukan tahun 1996. Sekarang (2001), sistem GNU ini bekerja secara handal, serta orang-orang bekerja dan mengembangkan GNOME, dan PPP dalam sistem GNU. Pada saat bersamaan sistem GNU/Linux, merupakan sebuah terobosan dari sistem GNU yang menggunakan

Linux sebagai kernel dan mengalami sukses luar biasa. Berhubung tujuan dari GNU ialah untuk kebebasan, maka setiap komponen dalam sistem GNU harus merupakan perangkat lunak bebas. Namun tidak berarti semuanya harus copyleft; setiap jenis perangkat lunak bebas dapat sah-sah saja jika menolong memenuhi tujuan teknis. Seseorang dapat menggunakan perangkat lunak non-copyleft seperti sistem X Window. Program GNU setara dengan perangkat lunak GNU. Program Anu adalah program GNU jika ia merupakan perangkat lunak GNU.

Perangkat Lunak GNU

Perangkat lunak GNU merupakan perangkat lunak yang dikeluarkan oleh proyek GNU. Sebagian besar perangkat lunak GNU merupakan copyleft, tapi tidak semuanya; namun, semua perangkat lunak GNU harus merupakan perangkat lunak bebas. Jika suatu program adalah perangkat lunak GNU, seseorang juga dapat menyebutnya sebagai program GNU. Beberapa perangkat lunak GNU ditulis oleh staf dari Free Software Foundation (FSF, Yayasan Perangkat Lunak Bebas), namun sebagian besar perangkat lunak GNU merupakan kontribusi dari para sukarelawan. Beberapa perangkat lunak yang dikontribusikan merupakan hak cipta dari Free Software Foundation; beberapa merupakan hak cipta darikontributor yang menulisnya.

GNU GPL (General Public License) (20k huruf) merupakan sebuah kumpulan ketentuan pendistribusian tertentu untuk meng-*copyleft*-kan sebuah program. Proyek GNU menggunakannya sebagai perjanjian distribusi untuk sebagian besar perangkat lunak GNU.

Sebagai contoh adalah lisensi GPL yang umum digunakan pada perangkat lunak Open Source. GPL memberikan hak kepada orang lain untuk menggunakan sebuah ciptaan asalkan modifikasi atau produk derivasi dari ciptaan tersebut memiliki lisensi yang sama. Kebalikan dari hak cipta adalah public domain. Ciptaan dalam public domain dapat digunakan sekehendaknya oleh pihak lain. Sebuah karya adalah public domainjika pemilik hak ciptanya menghendaki demikian. Selain itu, hak cipta memiliki waktu kadaluwarsa. Sebuah karya yang memiliki hak cipta akan memasuki public domain setelah jangka waktu tertentu. Sebagai contoh, lagu-lagu klasik sebagian besar adalah public domain karena sudah melewati jangka waktu kadaluwarsa hak cipta. Lingkup sebuah hak cipta adalah negara-negara yang menjadi anggota WIPO. Sebuah karya yang diciptakan di sebuah negara anggota WIPO secara otomatis berlaku di negara-negara anggota WIPO lainnya. Anggota non WIPO tidak mengakui hukum hak cipta. Sebagai contoh, di Iran, perangkat lunak Windows legal untuk didistribusikan ulang oleh siapapun.

2.4. Perangkat Lunak Bebas

PL Tanpa Kepemilikan

FM

Perangkat lunak bebas ialah perangkat lunak yang mengizinkan siapa pun untuk menggunakan, menyalin, dan mendistribusikan, baik dimodifikasi atau pun tidak, secara gratis atau pun dengan biaya. Perlu ditekankan, bahwa source code dari program harus tersedia. Jika tidak ada kode program, berarti bukan perangkat lunak.

Perangkat Lunak Bebas mengacu pada kebebasan para penggunanya untuk menjalankan, menggandakan, menyebarluaskan, mempelajari, mengubah dan meningkatkan kinerja perangkat lunak. Tepatnya, mengacu pada empat jenis kebebasan bagi para pengguna perangk at lunak:

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

merupakan suatu prasyarat juga.

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

Anda juga seharusnya memiliki kebebasan untuk memodifikasi (merubah), serta menggunakan untuk keperluan anda pribadi dalam pekerjaan anda, atau untuk main-main, tanpa perlu menyatakan kerberadaan program tersebut. Jika mengedarkan perubahan tersebut, anda seharusnya tidak perlu memberitahu siapa pun dengan cara apa pun.

Kebebasan untuk menggunakan sebuah program berarti kebebasan bagi siapa pun – perorangan atau pun organisasi – menggunakan pada komputer jenis apa pun, untuk kegiatan apa pun, tanpa perlu membe ritahu para pengembang atau pun pihak-pihak lainnya secara khusus.

Kebebasan untuk menyebarluaskan hasil penggandaan, harus termasuk bentuk biner (eksekusi), atau pun kode program, yang termodifikasi maupun yang belum. Tidak apa-apa, jika tidak disertakan cara memproduksi bentuk biner tersebut, namun perlu ada kebebasan penyebarluasannnya, jika dikemudian hari ditemukan cara untuk memproduksinya.

Agar terdapat kebebasan melakukan perubahan – mempublikasikan versi yang lebih baik – arti, anda harus memiliki akses pada kode program tersebut. Jadi, memiliki akses tersebut merupakan syarat mutlak untuk perangkat lunak bebas.

Agar dapat menjadi nyata, kebebasan ini tidak boleh dibatalkan selama anda tidak melakukan suatu kesalahan. Jika pengembang perangkat lunak tersebut mempunyai hak untuk mencabut lisensi, tanpa anda melakukan apa-apa yang menyebabkan seperti itu, maka program tersebut tidak dapat disebut sebagai perangkat lunak bebas.

Walau pun demikian, aturan tertentu mengenai tata cara pendistribusian perangkat lunak bebas dapat diterima, selama tidak bertentangan dengan hakikat inti dari kebebasan itu sendiri. Umpamanya, "copyleft" (pada garis besarnya), tidak mengizinkan penambahan aturan pelarangan atau pembatasan hak orang lain yang tidak sesuai dengan hakikat inti dari kebebasan. Hal ini tidak bertentangan dengan hakikat inti dari kebebas an itu sendiri, justru aturan ini melindunginya.

Jadi, anda mungkin harus membayar untuk mendapatkan perangkat lunak GNU, atau mungkin juga anda mendapatkannya secara cuma-cuma. Terlepas dari cara mendapatkan perangkat lunak tersebut, anda akan selalu bebas untuk menyalin dan mengubah perangkat lunak tersebut, atau pun untuk menjualnya.

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

Aturan perihal cara mengemas perangkat lunak bebas hasil modifikasi pun dapat diterima, jika tidak secara efektif menghalangi kebebasan anda untuk mempublikasikan ulang modifikasinya. Demikian pula aturan, "Jika anda membuat program tersedia dalam cara tertentu, maka anda juga harus membuatnya tersedia dalam cara tertentu lainnya,juga dapat diterima dengan ketentuan yang sama (Perhatikan bahwa aturan tersebut masih memberikan anda pilihan untuk menentukan apakah program itu akan dipublikasikan atau tidak).

ZCZCOLD

FIXME: Konsep Perangkat Lunak Bebas

FIXME: Definisi Perangkat Lunak Bebas

FIXME: Konsep Copyleft

FIXME: Konsep Lisensi PLB dan berbagai PLB

FIXME: Berbisnis dengan PLB

2.5. *Open Source Software*

Konsep open source pada intinya adalah membuka "source code" dari sebuah software. Konsep ini terasa aneh pada awalnya dikarenakan source code merupakan kunci dari sebuah software. Dengan diketahui logika yang ada di source code, maka orang lain semestinya dapat membuat software yang sama fungsinya. Open source hanya sebatas itu. Artinya, dia tidak harus gratis. Kita bisa saja membuat software yang kita buka source codenya, mempatenkan algoritmanya, medaftarkan hak cipta atau copyright, dan tetap menjual software tersebut secara komersial (alias tidak gratis). definisi open source yangasli seperti tertuang dalam OSD (Open Source Definition) yaitu:

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

Secara sederhana Open Source adalah sistem pengembangan yang tidak dikoordinasi oleh suatu orang/lembaga pusat, tetapi oleh para pelaku yang bekerja sama dengan memanfaatkan source-code yang tersebar dan tersedia bebas (menggunakan fasilitas komunikasi internet). Pola pengembangan ini mengambil model ala bazaar, sehingga pola Open Source ini memiliki ciri bagi komunitasnya yaitu adanya dorongan yang bersumber dari gift culture, yang artinya ketika suatu komunitas menggunakan sebuah program Open Source dan telah menerima sebuah manfaat kemudian akan termotivasi untuk menimbulkan sebuah pertanyaan apa yang bisa pengguna berikan balik kepada orang banyak.

Pola Open Source lahir karena kebebasan berkarya, tanpa intervensi berpikir dan mengungkapkan apa yang diinginkan dengan menggunakan pengetahuan dan produk yang cocok. Kebebasan menjadi pertimbangan utama ketika dilepas ke publik. Komunitas yang lain mendapat kebebasan untuk belajar, mengutak-ngatik, merevisi ulang, membenarkan ataupun bahkan menyalahkan, tetapi kebebasan ini juga datang bersama dengan tanggung jawab, bukan bebas tanpa tanggung jawab. Prinsip dasar dari pengembangan ala Open Source ini adalah: "rapid code evolution and massive independent peer review".

Pergerakan perangkat lunak bebas dan open source saat ini membagi pergerakannya dengan pandangan dan tujuan yang berbeda. Open source adalah pengembangan secara metodelogy, perangkat lunak tidak bebas adalah solusi suboptimal. Untuk pergerakan perangkat lunak bebas,perangkat lunak tidak bebas adalah masalah sosial dan perangkat lunak bebas adalah solusi.

2.6. Berbisnis PLB

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

Mendistribusi ulang perangkat lunak bebas merupakan kegiatan yang baik dan sah; jika anda melakukannya, silakan juga menarik keuntungan.

Beberapa bentuk model bisnis yang dapat dilakukan dengan Open Source:

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

2.7. Tantangan PLB

Ada 4 tantangan perangkat lunak bebas:

Perangkat Keras Rahasia

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

Library tidak bebas

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

Paten perangkat Lunak

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

Dokumentasi bebas

FM

2.8. Sistem Operasi GNU/Linux

LINUX (Kernel-nya saja) adalah sistem operasi komputer yang bermula dari proyek hobi Linus Torvalds, seorang mahasiswa dari Helsinki University, Finlandia. Linus sendiri terinspirasi Minix, suatu sistem UNIX kecil yang dikembangkan Prof. Andrew Tanenbaum dari der Frein University, Amsterdam. Linux versi 0.01 dikerjakan sekitar bulan Agustus 1991. Pada 5 Oktober 1991, Linus mengumumkan versi resmi Linux, yaitu 0.02. Versi ini hanya dapat menjalankan Bash (GNU Bourne Again Shell) dan gcc (GNU C Compiler). Saat ini, Linux telah berkembang demikian cepat sehingga dilengkapi banyak program, mulai dari Office Suite semacam StarOffice hingga server web (seperti Apache), email (Sendmail), database (PostgreSQL dan MySQL), dan lainnya sehingga jadi GNU/Linux. Linux didistribusikan secara bebas bersama program GNU (Gnu is Not Unix) lainnya dengan model lisensi GPL (General Public License). GNU/Linux atau yang selanjutnya disebut Linux saja adalah UNIX Clone, sebuah sistem operasi komputer yang mirip seperti UNIX yang merupakan implementasi independen dari POSIX. Saat ini linux adalah system UNIX yang sangat lengkap, bias digunakan untuk jaringan,pengembangan software dan bahka untuk pekerjaan sehari-hari.

2.9. Rangkuman

Arti bebas yang salah, telah menimbulkan persepsi masyarakat bahwa perangkat lunak bebas merupakan perangkat lunak yang gratis. Perangkat lunak bebas ialah perihal kebebasan, bukan harga. Konsep kebebasan yang dapat diambil dari kata bebas pada perangkat lunak bebas adalah seperti kebebasan berbicara bukan seperti bir gratis. Maksud dari bebas seperti kebebasan berbicara adalah kebebasan untuk menggunakan, menyalin, menyebarluaskan, mempelajari, mengubah, dan meningkatkan kinerja perangkat lunak. Suatu perangkat lunak dapat dimasukkan dalam kategori perangkat lunak bebas bila setiap orang memiliki kebebasan tersebut. Hal ini berarti, setiap pengguna perangkat lunak bebas dapat meminjamkan perangkat lunak yang dimilikinya kepada orang lain untuk dipergunakan tanpa perlu melanggar hukum dan disebut pembajak. Kebebasan yang diberikan perangkat lunak bebas dijamin oleh copyleft, suatu cara yang dijamin oleh hukum untuk melindungi kebebasan para pengguna perangkat lunak bebas. Dengan adanya copyleft maka suatu perangkat lunak bebas beserta hasil perubahan dari kode sumbernya akan selalu menjadi perangkat lunak bebas. Kebebasan yang diberikan melalui perlindungan copyleft inilah yang membuat suatu program dapat menjadi perangkat lunak bebas. Keuntungan yang diperoleh dari penggunaan perangkat lunak bebas adalah karena serbaguna dan efektif dalam keanekaragaman jenis aplikasi. Dengan pemberian source code-nya, perangkat lunak bebas dapat disesuaikan secara khusus untuk kebutuhan pemakai. Sesuatu yang tidak mudah untuk terselesaikan dengan perangkat lunak berpemilik. Selain itu, perangkat lunak bebas didukung oleh milis-milis pengguna yang dapat menjawab pertanyaan yang timbul karena permasalahan pada penggunaan perangkat lunak bebas.

2.10. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:

   - *Software License: "Free Software" vs. "Copyleft".*

2. Apa beda hak cipta, paten, trade mark, service mark?

3. Apa beda copyleft dan copyright?

4. Terangkan dengan singkat antara PLB vs open souce?

5. Sebutkan tantangan PLB kedepan?

Rujukan

[WEBFSF1991a] Free Software Foundation . 1991. *GNU General Public License – http://gnui.vLSM.org/licenses/gpl.txt [http://gnui.vLSM.org/licenses/gpl.txt]* . Diakses 16 Agustus 2005.

[WEBFSF2001a] Free Software Foundation . 2001. *Definisi Perangkat Lunak Bebas – http://gnui.vlsm.org/philosophy/free-sw.id.html [http://gnui.vlsm.org/philosophy/free-sw.id.html]* . Diakses 16 Agustus 2005.

[WEBFSF2001b] Free Software Foundation . 2001. *Frequently Asked Questions about the GNU GPL – http://gnui.vlsm.org/licenses/gpl-faq.html [http://gnui.vlsm.org/licenses/gpl-faq.html]* . Diakses 16 Agustus 2005.

[WEBHuham2005] Departemen Hukum dan Hak Asasi Manusia Republik Indonesia . 2005. *Kekayaan Intelektual – http://www.dgip.go.id/article/archive/2 [http://www.dgip.go.id/article/archive/2]* . Diakses 17 Agustus 2005.

[WEBRamelan1996] Rahardi Ramelan . 1996. *Hak Atas Kekayaan Intelektual Dalam Era Globalisasi http://leapidea.com/presentation?id=6 [http://leapidea.com/presentation?id=6]* . Diakses 16 Agustus 2005.

[WEBSamik2003a] Rahmat M Samik-Ibrahim . 2003. *Pengenalan Lisensi Perangkat Lunak Bebas – http://rms46.vlsm.org/ 1/ 70.pdf [http://rms46.vlsm.org/1/70.pdf]* . vLSM.org. Pamulang. Diakses 17 Agustus 2005.

[WEBStallman1994a] Richard M Stallman . 1994. *Mengapa Perangkat Lunak Seharusnya Tanpa Pemilik – http://gnui.vlsm.org/philosophy/why-free.id.html [http://gnui.vlsm.org/philosophy/why-free.id.html]* . Diakses 16 Agustus 2005.

[WEBWiki2005a] From Wikipedia, the free encyclopedia . 2005. *Intellectual property – http://en.wikipedia.org/wiki/Intellectual_property [http://en.wikipedia.org/wiki/Intellectual_property]* . Diakses 16 Agustus 2005.

[WEBWIPO2005] World Intellectual Property Organization . 2005. *About Intellectual Property – http://www.wipo.int/about-ip/en/ [http://www.wipo.int/about-ip/en/]* . Diakses 17 Agustus 2005.

Undang-Undang Nomor 30 Tahun 2000 Tentang Rahasia Dagang.

Undang-Undang Nomor 31 Tahun 2000 Tentang Desain Industri.

Undang-Undang Nomor 32 Tahun 2000 Tentang Desain Tata Letak Sirkuit Terpadu;

Undang-Undang Nomor 14 Tahun 2001 Tentang Paten.

Undang-Undang Nomor 15 Tahun 2001 Tentang Merek.

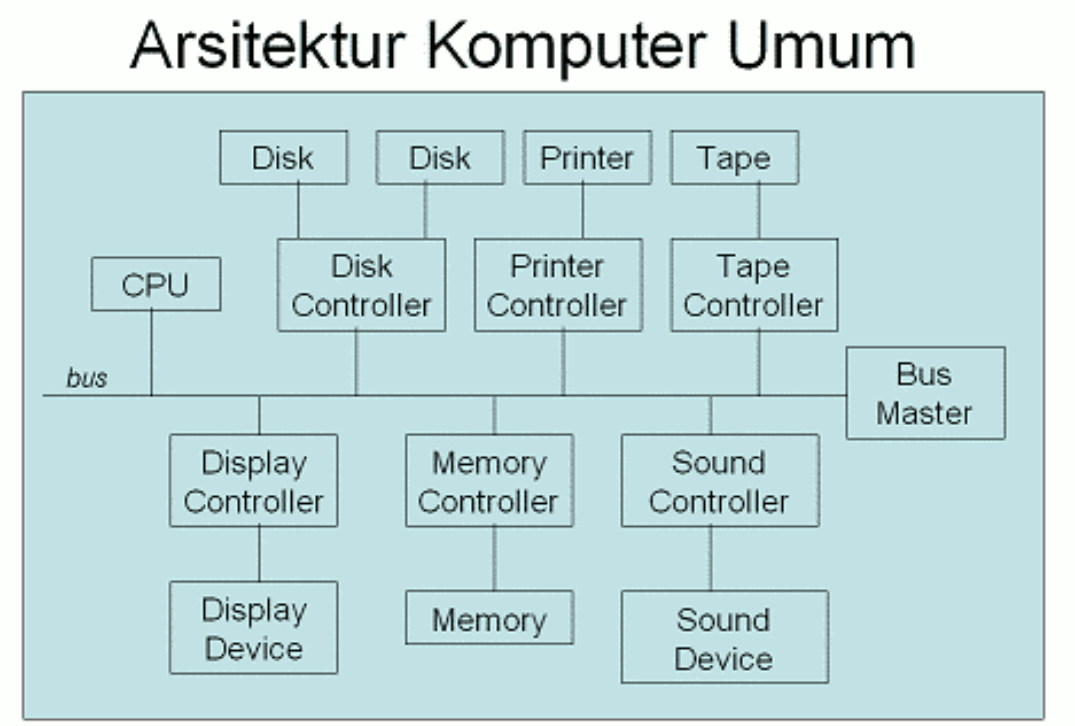
Undang-Undang Nomor 19 Tahun 2002 Tentang Hak Cipta.

Bab 3. Perangkat Keras Komputer

3.1. Pendahuluan

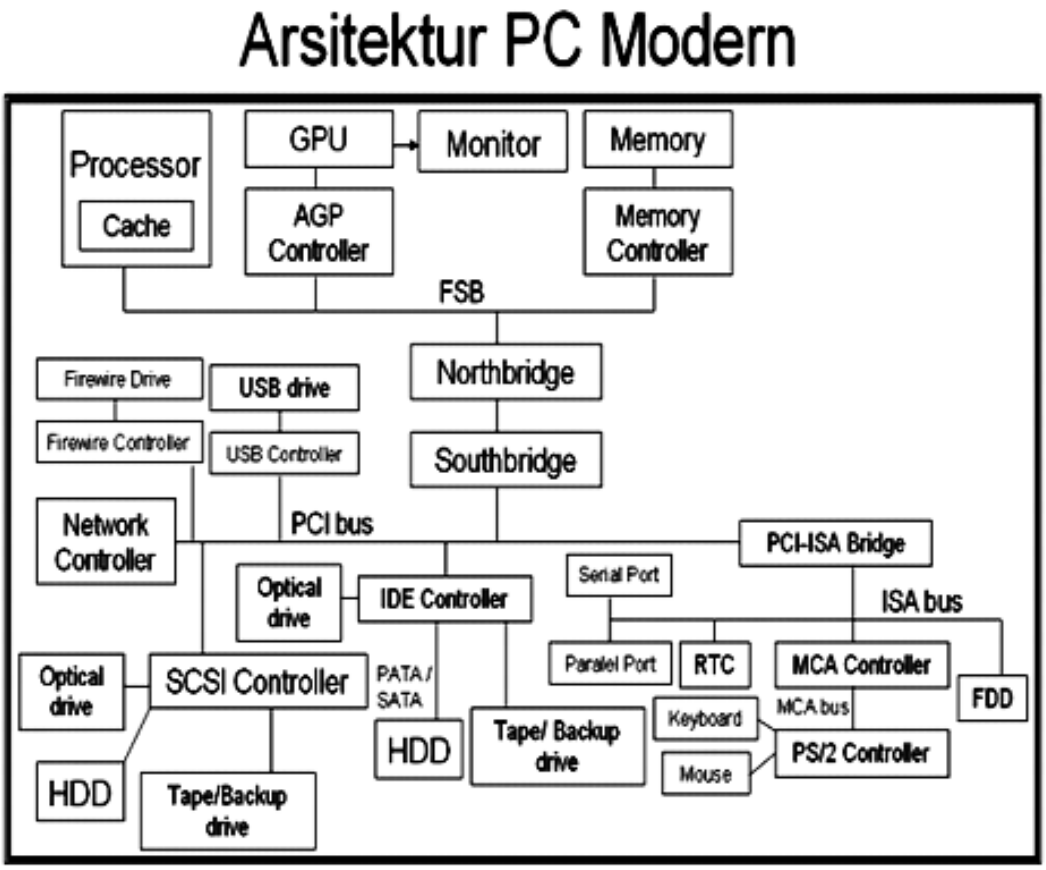
Tidak ada suatu ketentuan khusus tentang bagaimana seharusnya struktur sistem sebuah komputer. Setiap ahli dan desainer arsitektur komputer memiliki pandangannya masing-masing. Akan tetapi, untuk mempermudah kita memahami detail dari sistem operasi di bab-bab berikutnya, kita perlu memiliki pengetahuan umum tentang struktur sistem komputer.

Gambar 3.1. Arsitektur Umum Komputer



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

Gambar 3.2. Arsitektur PC Modern



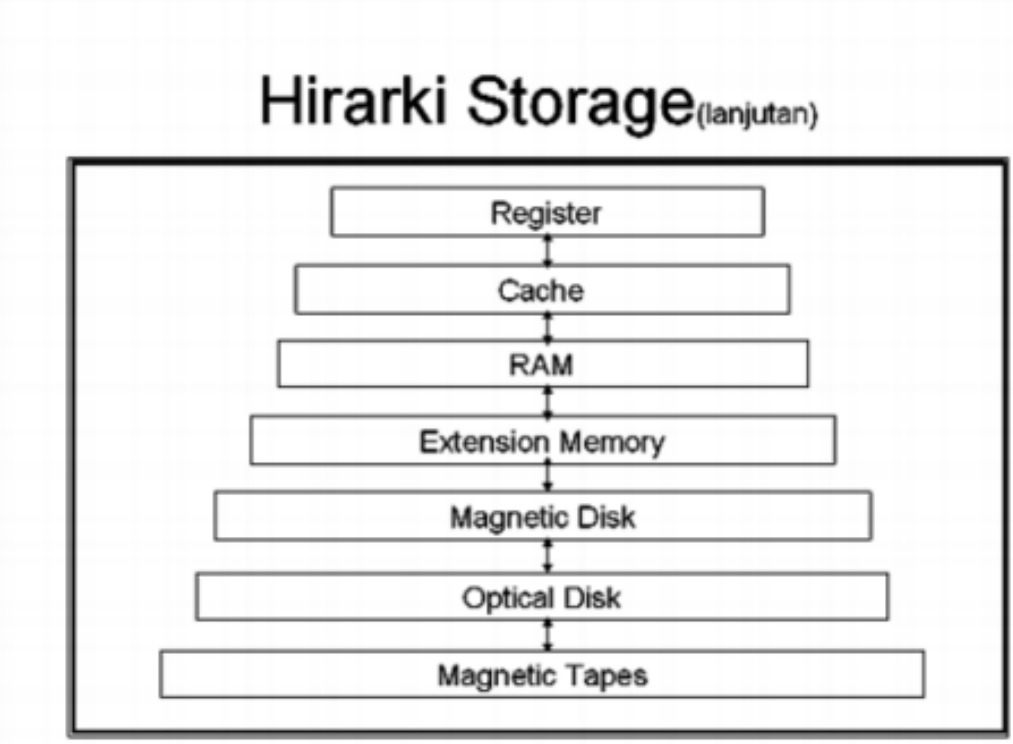
3.2. Prosesor

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

3.3. Memori Utama

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

Gambar 3.3. Penyimpanan Hirarkis



Register

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

Cache Memory

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

Random Access Memory

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

Memori Ekstensi

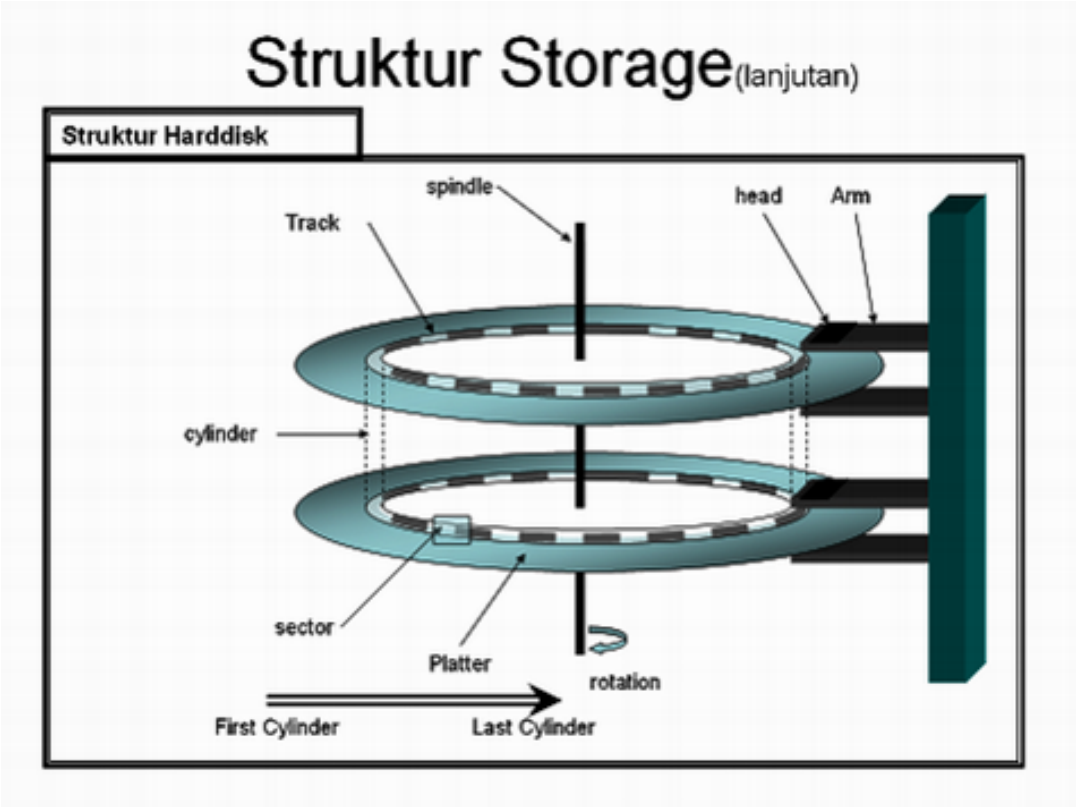
Tambahan memori yang digunakan untuk membantu proses-proses dalam komputer, biasanya berupa buffer. Peranan tambahan memori ini sering dilupakan akan tetapi sangat penting artinya untuk efisiensi. Biasanya tambahan memori ini memberi gambaran kasar kemampuan dari perangkat tersebut, sebagai contoh misalnya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

Direct Memory Access

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

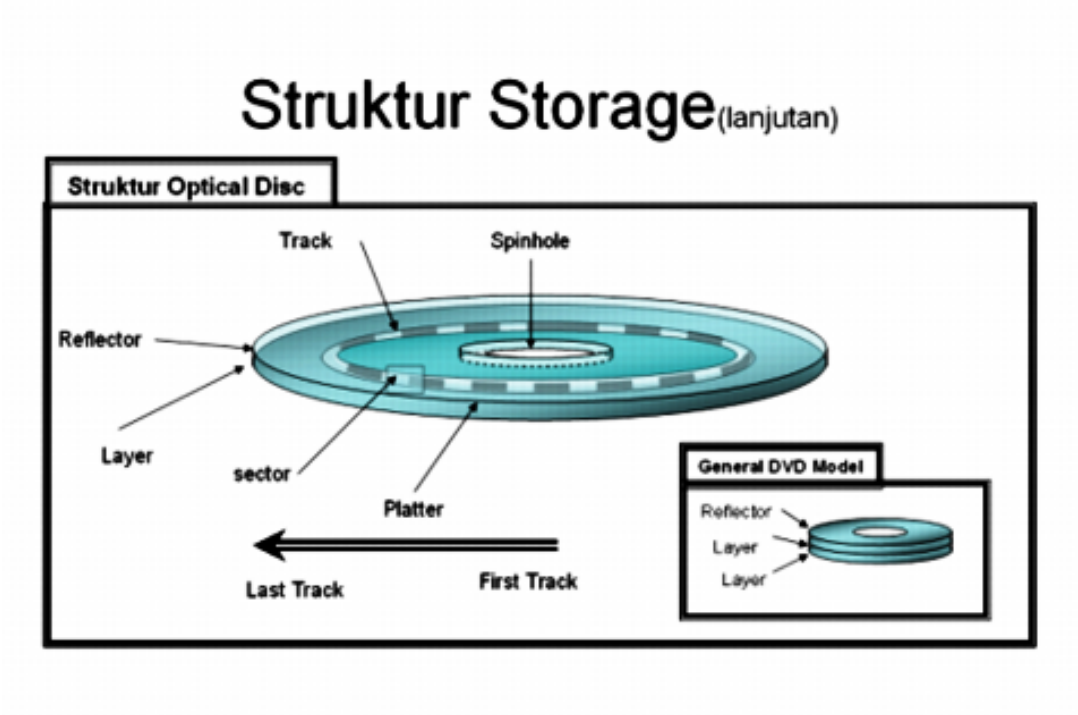
3.4. Memori Sekunder

Gambar 3.4. Struktur *Harddisk*



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

Gambar 3.5. Struktur *Optical Drive*



3.5. Memori Tersier

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

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

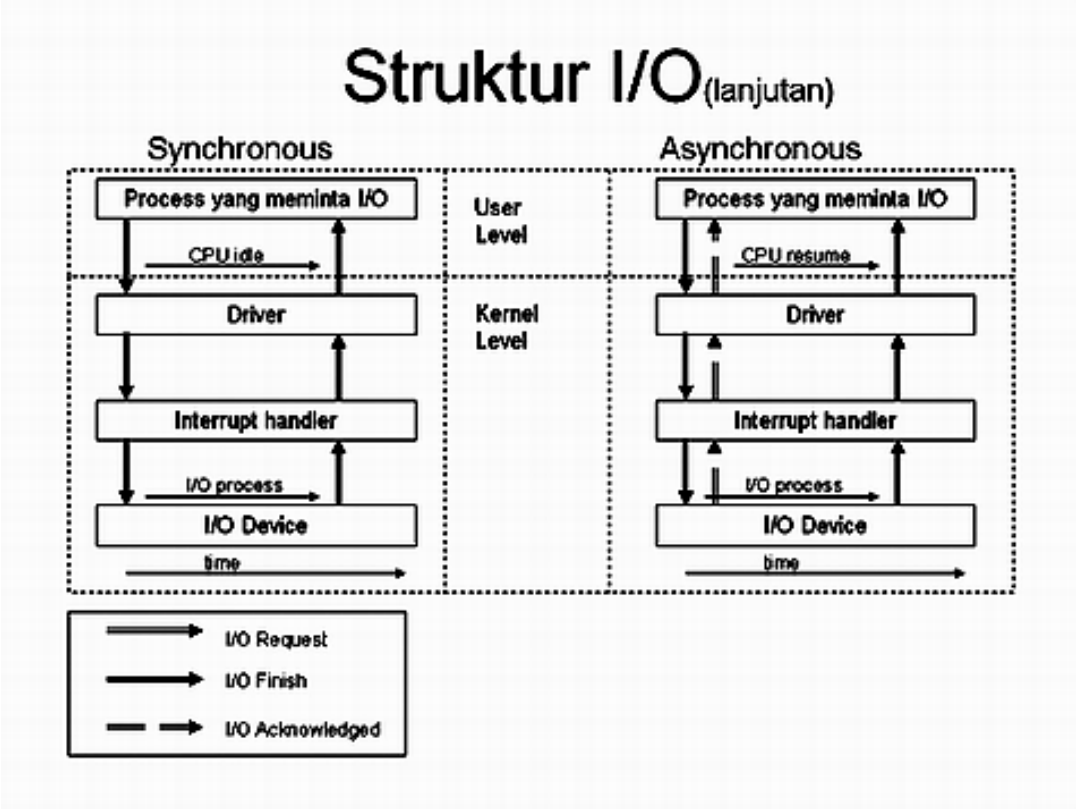
3.6. Struktur Keluaran/Masukan (M/K)

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

Setelah proses M/K dimulai, kendali akan kembali ke user program saat proses M/K selesai *(Synchronous).* Instruksi wait menyebabkan CPU idle sampai interupsi berikutnya. Akan terjadi *Wait loop* (untuk menunggu akses berikutnya). Paling banyak satu proses M/K yang berjalan dalam satu waktu.

Setelah proses M/K dimulai, kendali akan kembali ke user program tanpa menunggu proses M/K selesai (Asynchronous). System call permintaan pada sistem operasi untuk mengizinkan user menunggu sampai M/K selesai.Device-status table mengandung data masukkan untuk tiap M/K device yang menjelaskan tipe, alamat, dan keadaannya. Sistem operasi memeriksa M/K device untuk mengetahui keadaan device dan mengubah tabel untuk memasukkan interupsi. Jika M/K device mengirim/mengambil data ke/dari memory hal ini dikenal dengan nama *Direct Memory Access* (DMA).

Gambar 3.6. Struktur M/K



3.7. BUS

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

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

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

3.8. Interupsi

Kejadian ini pada komputer modern biasanya ditandai dengan munculnya interupsi dari software atau hardware, sehingga Sistem Operasi ini disebut *Interrupt-driven. Interrupt* dari *hardware* biasanya dikirimkan melalui suatu signal tertentu, sedangkan *software* mengirim interupsi dengan cara menjalankan *system call* atau juga dikenal dengan istilah *monitor call. System/Monitor call* ini

akan menyebabkan *trap* yaitu interupsi khusus yang dihasilkan oleh software karena adanya masalah atau permintaan terhadap layanan sistem operasi.

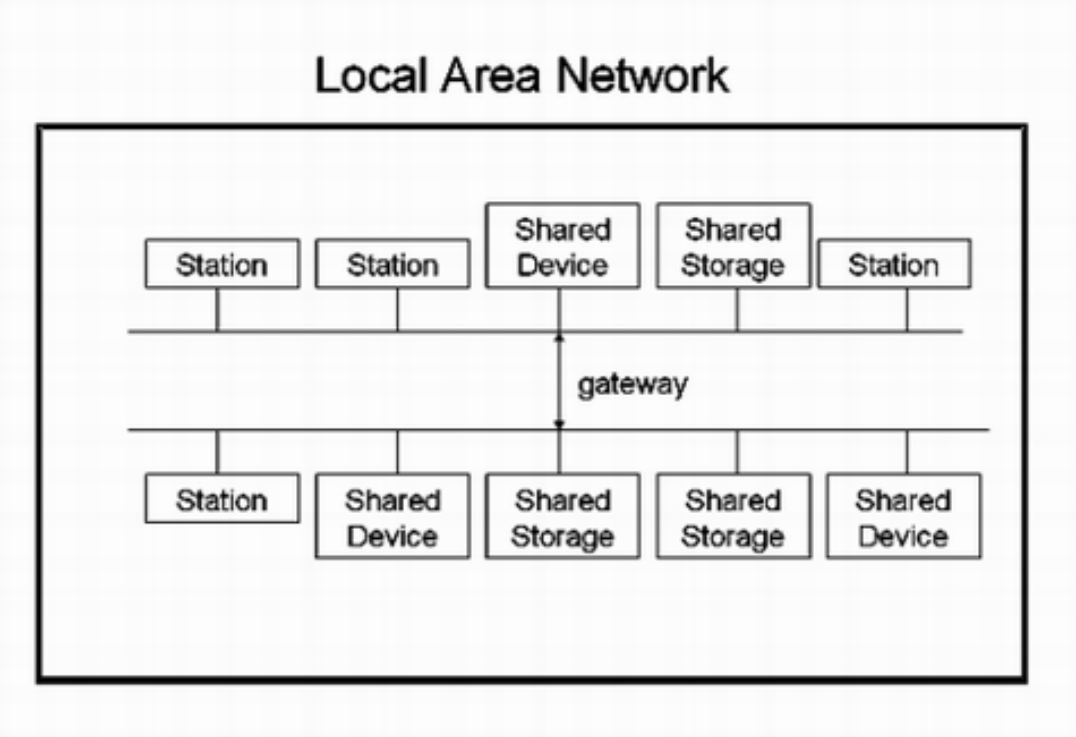
Trap ini juga sering disebut sebagai *exception*.

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

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

3.9. *Local Area Network*

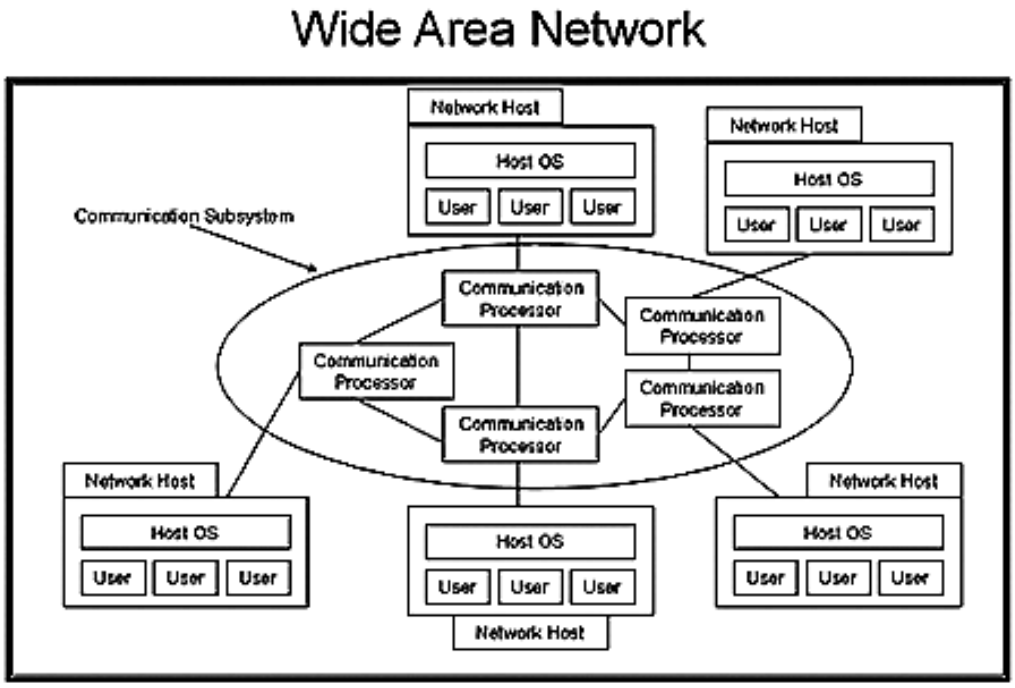
Gambar 3.7. ***Local Area Network***



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

3.10. *Wide Area Network*

Gambar 3.8. *Wide Area Network*



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

3.11. Rangkuman

Memori utama adalah satu-satunya tempat penyimpanan yang besar yang dapat diakses secara langsung oleh prosessor, merupakan suatu *array* dari *word* atau *byte*, yang mempunyai ukuran ratusan sampai jutaan ribu. Setiap word memiliki alamatnya sendiri. Memori utama adalah tempat penyimpanan yang volatile, dimana isinya hilang bila sumber energinya (energi listrik) dimatikan. Kebanyakan sistem komputer menyediakan secondary storage sebagai perluasan dari memori utama. Syarat utama dari secondary storage adalah dapat menyimpan data dalam jumlah besar secara permanen.

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

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

3.12. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan

persamaan pasangan konsep tersebut:

- *"Random Access Memory" vs. "Magnetic Disk"*.

2. Apakah keuntungan utama dari *multiprogramming*?

Rujukan

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

Bab 4. Proteksi Perangkat Keras

4.1. Pendahuluan

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

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

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

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

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

4.2. Proteksi Fisik

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

4.3. Proteksi Media

Dalam keseharian kita ada beberapa jenis media yang digunakan untuk penyimpanan data, antara lain tape, disket, CD, USB flash disk, dan lainnya. Untuk menjamin keamanan data yang tersimpan dalam media-media tersebut, maka perlu sebuah mekanisme untuk menanganinya. Mekanisme proteksi antara satu media dengan media yang lain tidak sama.

Sebagai contoh CD yang sering kita kenal sebagai media untuk installer memiliki mekanisme untuk menjamin CD perangkat lunak yang digunakan oleh consumer ber-lisensi. mekanisme yang digunakan yaitu dengan menggunakan CD key number. Cara kerja dari mekanisme ini, ketikapengguna ingin menginstal program maka pengguna harus memasukkan kode dari key number, jika tidak maka program tersebut tidak akan bisa diinstal. Key number bisanya didapatkan dilabel yang disertakan pada CD atau pada dokumentasi program.

Tetapi mekanisme ini tidak efektif dalam mencegah menyebarnya CD program yang tidak berlisenikarena sangat mudah untuk di cari Key number-nya, dengan menggunakan key generator,

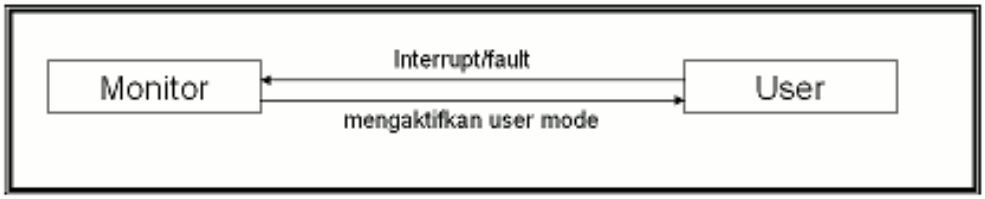
yang bisa didapatkan di intrnet secara gratis.

Lain halnya dengan disket mekanisme yang di gunakan dari media ini mirip dengan kaset dan USB Flash diskyaitu dengan memproteksi fisiknya. Mekanisme proteksinya dengan menggunakan katup yang dapat di geser, jika katupnya dibuka maka disket bisa ditulis dan di baca, jika ditutup maka disket hanya bisa dibaca tetapi tidak bisa ditulis.

4.4. Konsep Mode Operasi Ganda (*Dual Mode Operation*)

Membagi sumber daya sistem yang memerlukan sistem operasi untuk menjamin bahwa program yang salah tidak menyebabkan program lain berjalan salah juga. Menyediakan dukungan perangkat keras untuk membedakan minimal dua mode operasi yaitu: *User Mode* - Eksekusi dikendalikan oleh pengguna; *Monitor/Kernel/System Mode* - Eksekusi dikendalikan oleh sistem operasi. Instruksi tertentu hanya berjalan di mode ini (*Privileged Instruction*). Ditambahkan sebuah bit penanda operasi. Jika terjadi *interrupt*, maka perangkat keras berpindah ke *monitor mode*.

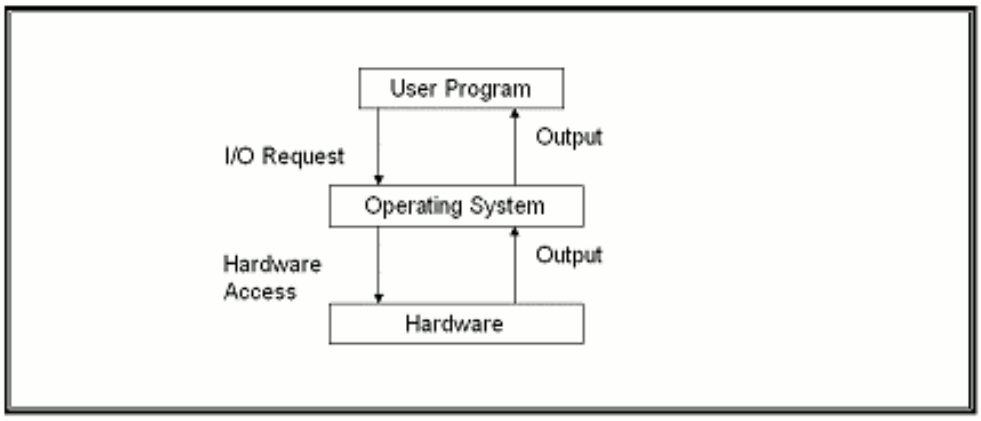
Gambar 4.1. *Dual Mode Operation*



4.5. Proteksi Masukan/Keluaran

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

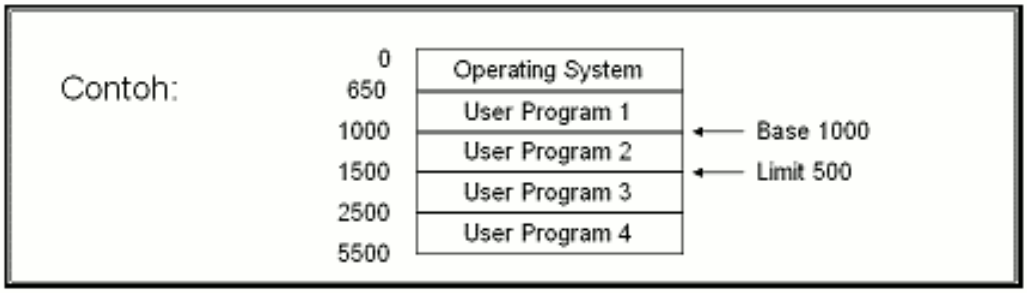
Gambar 4.2. Proteksi M/K



4.6. Proteksi Memori

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

Gambar 4.3. ***Memory Protection***



4.7. Proteksi CPU

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

4.8. Rangkuman

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

4.9. Latihan

1. Terangkan kegunaan dari *DMA*!
2. Apakah perbedaan antara *trap* dan interupsi? Sebutkan penggunaan dari setiap fungsi tersebut
3. Sebutkan tiga kelas komputer menurut jenis datanya!

Rujukan

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

Bagian II. Konsep Dasar Sistem Operasi

Daftar Isi

Bab 5. Komponen Sistem Operasi

5.1. Pendahuluan

Tidak semua sistem operasi mempunyai struktur yang sama. Namun menurut Avi Silberschatz, Peter Galvin, dan Greg Gagne, umumnya sebuah sistem operasi modern mempunyai komponen sebagai berikut:

- Managemen Proses.
- Managemen Memori Utama.
- Managemen Berkas.
- Managemen Sistem Masukan/Keluaran.
- Managemen Penyimpanan Sekunder.
- Sistem Proteksi.
- Jaringan.
- *Command-Interpreter System*.

Sedangkan menurut A.S. Tanenbaum, sistem operasi mempunyai empat komponen utama, yaitu:

- Managemen proses,
- Masukan/Keluaran
- Managemen Memori, dan
- Sistem Berkas.

5.2. Managemen Proses

Proses adalah sebuah program yang sedang dieksekusi. Sebuah proses membutuhkan beberapa sumber daya untuk menyelesaikan tugasnya. Sumber daya tersebut dapat berupa *CPU time*, memori, berkas-berkas, dan perangkat-perangkat Masukan/Keluaran. Sistem operasi mengalokasikan sumber daya-sumber daya tersebut saat proses itu diciptakan atau sedang diproses/dijalankan. Ketika proses tersebut berhenti dijalankan, sistem operasi akan mendapatkan kembali semua sumber daya yang bisa digunakan kembali.

Sistem operasi bertanggung-jawab atas aktivitas-aktivitas yang berkaitan dengan managemen proses seperti:

- Membuat dan menghapus proses pengguna dan sistem proses.
- Menunda atau melanjutkan proses.
- Menyediakan mekanisme untuk proses sinkronisasi.
- Menyediakan mekanisme untuk proses komunikasi.
- Menyediakan mekanisme untuk penanganan *deadlock*.

5.3. Managemen Memori Utama

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

Sistem operasi bertanggung-jawab atas aktivitas-aktivitas yang berkaitan dengan managemen memori seperti:

- Menjaga *track* dari memori yang sedang digunakan dan siapa yang menggunakannya.
- Memilih program yang akan di-*load* ke memori.

5.4. Managemen Berkas

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

Sistem operasi bertanggung-jawab dalam aktivitas yang berhubungan dengan managemen berkas:

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

5.5. Managemen Sistem Masukan/Keluaran

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

Komponen Sistem Operasi untuk sistem Masukan/Keluaran:

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

5.6. Managemen Penyimpanan Sekunder

Data yang disimpan dalam memori utama bersifat sementara dan jumlahnya sangat kecil. Oleh

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

Sistem operasi bertanggung-jawab atas aktivitas-aktivitas yang berkaitan dengan managemen disk seperti:

- *free-space management*.
- alokasi penyimpanan.
- penjadualan disk.

5.7. Sistem Proteksi

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

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

5.8. Jaringan

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

5.9. *Command-Interpreter System*

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

5.10. Rangkuman

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

5.11. Latihan

1. Sebutkan komponen-komponen Sistem Operasi!
2. Sebutkan aktivitas yang dilakukan oleh Sistem Operasi yang berkaitan dengan managemen proses!

3. Sebutkan aktivitas yang dilakukan oleh Sistem Operasi yang berkaitan dengan managemen berkas!

Rujukan

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

Bab 6. Sudut Pandang Alternatif

6.1. Layanan Sistem Operasi

Layanan sistem operasi dirancang untuk membuat pemrograman menjadi lebih mudah.

1. Pembuatan Program

   Sistim operasi menyediakan berbagai fasilitas yang membantu programer dalam membuat program seperti editor. Walaupun bukan bagian dari sistim operasi, tapi layanan ini diakses melalui sistim operasi.

2. Eksekusi Program

   Sistem harus bisa me-*load* program ke memori, dan menjalankan program tersebut. Program harus bisa menghentikan pengeksekusiannya baik secara normal maupun tidak (ada *error*).

3. Operasi Masukan/Keluaran

   Program yang sedang dijalankan kadang kala membutuhkan Masukan/Keluaran. Untuk efisiensi dan keamanan, pengguna biasanya tidak bisa mengatur peranti Masukan/Keluaran secara langsung, untuk itulah sistem operasi harus menyediakan mekanisme dalam melakukan operasi Masukan/Keluaran.

4. Manipulasi Sistem Berkas

   Program harus membaca dan menulis berkas, dan kadang kala juga harus membuat dan menghapus berkas.

5. Komunikasi

   Kadang kala sebuah proses memerlukan informasi dari proses yang lain. Ada dua cara umum dimana komunikasi dapat dilakukan. Komunikasi dapat terjadi antara proses dalam satu komputer, atau antara proses yang berada dalam komputer yang berbeda, tetapi dihubungkan oleh jaringan komputer. Komunikasi dapat dilakukan dengan *share-memory* atau *message-passing*, dimana sejumlah informasi dipindahkan antara proses oleh sistem operasi.

6. Deteksi *Error*

   Sistem operasi harus selalu waspada terhadap kemungkinan *error*. *Error* dapat terjadi di CPU dan memori perangkat keras, Masukan/Keluaran, dan di dalam program yang dijalankan pengguna. Untuk setiap jenis *error* sistem operasi harus bisa mengambil langkah yang tepat untuk mempertahankan jalannya proses komputasi. Misalnya dengan menghentikan jalannya program, mencoba kembali melakukan operasi yang dijalankan, atau melaporkan kesalahan yang terjadi agar pengguna dapat mengambil langkah selanjutnya.

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

1. Alokasi Sumber Daya

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

2. *Accounting*

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

3. Proteksi

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

6.2. *System Program*

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

Dapat dikategorikan:

- Managemen/manipulasi berkas

  Membuat, menghapus, *copy*, *rename*, *print*, memanipulasi berkas dan direktori.

- Informasi status

  Beberapa program meminta informasi tentang tanggal, jam, jumlah memori dan disk yang tersedia, jumlah pengguna dan informasi lain yang sejenis.

- Modifikasi berkas

  Membuat berkas dan memodifikasi isi berkas yang disimpan pada disk atau tape.

- Pendukung bahasa pemrograman

  Kadang kala *compiler*, *assembler*, *interpreter* dari bahasa pemrograman diberikan kepada pengguna dengan bantuan sistem operasi.

- *Loading* dan eksekusi program

  Ketika program di-*assembly* atau di-*compile*, program tersebut harus di-*load* ke dalam memori untuk dieksekusi. Untuk itu sistem harus menyediakan *absolute loaders*, *relocatable loaders*, *linkage editors*, dan *overlay loaders*.

- Komunikasi

  Menyediakan mekanisme komunikasi antara proses, pengguna, dan sistem komputer yang berbeda. Sehingga pengguna bisa mengirim pesan, *browse* web pages, mengirim e-mail, atau mentransfer berkas.

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

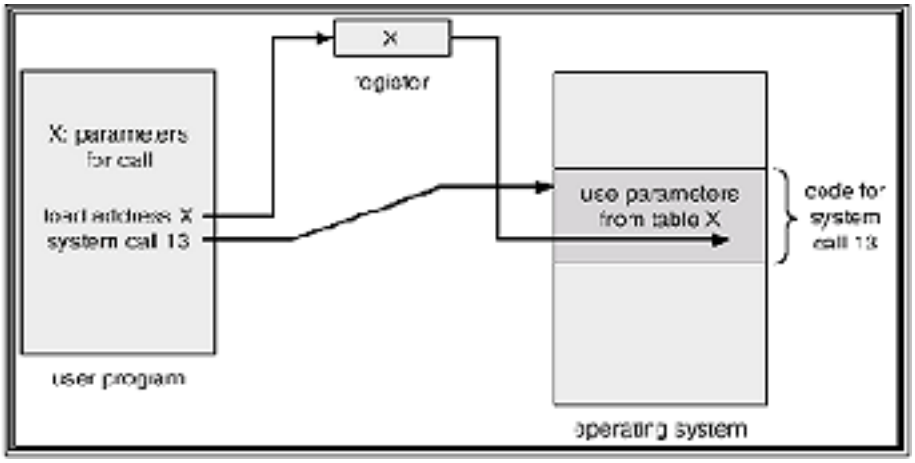
6.3. *System Calls*

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

Tiga metoda umum yang digunakan dalam memberikan parameter kepada sistem operasi:

- Melalui *register*.
- Menyimpan parameter dalam *block* atau tabel pada memori dan alamat *block* tersebut diberikan sebagai parameter dalam *register*.
- Menyimpan parameter (*push*) ke dalam *stack* oleh program, dan melakukan *pop off* pada *stack* oleh sistem operasi.

Gambar 6.1. Memberikan parameter melalui tabel



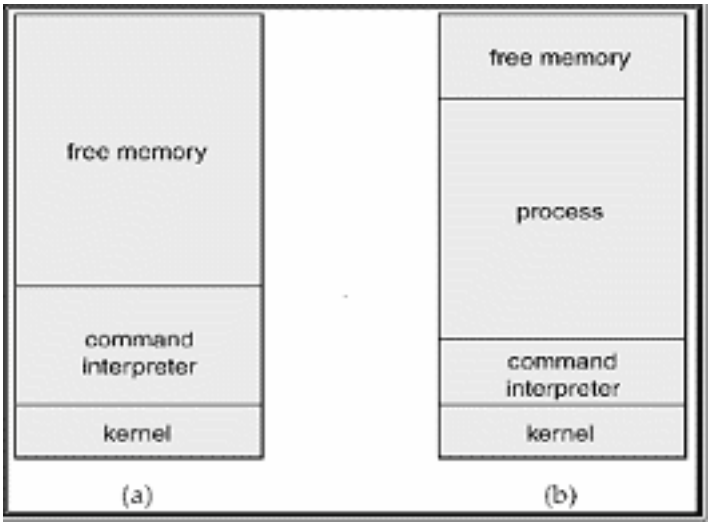
Sumber: Silberschatz, 2003, hal. 65.

Jenis *System Calls*

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

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

Gambar 6.2. Eksekusi MS-DOS



Sumber: Silberschatz, 2003, hal. 68.

6.4. *System Calls* Managemen Proses

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

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

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

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

Rujukan :

http://www.cs.vu.nl/~ast/books/mos2/ tanenbaum

6.5. *System Calls* Managemen Berkas

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

Ada juga *system calls* yang menyediakan mekanisme lain yang berhubungan dengan direktori atau sistim berkas secara keseluruhan. Jadi bukan hanya berhubungan dengan satu spesifik berkas.

Contohnya membuat atau menghapus suatu direktori, dan lain-lain.

6.6. *System Calls* Managemen Peranti

Program yang sedang dijalankan kadang kala memerlukan tambahan sumber daya. Jika banyak pengguna yang menggunakan sistem, maka jika memerlukan tambahan sumber daya maka harus meminta peranti terlebih dahulu. Dan setelah selesai penggunakannnya harus dilepaskan kembali. Ketika sebuah peranti telah diminta dan dialokasikan maka peranti tersebut bisa dibaca, ditulis, atau direposisi.

6.7. *System Calls* Informasi *Maintenance*

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

6.8. *System Calls* Komunikasi

Dua model komunikasi:

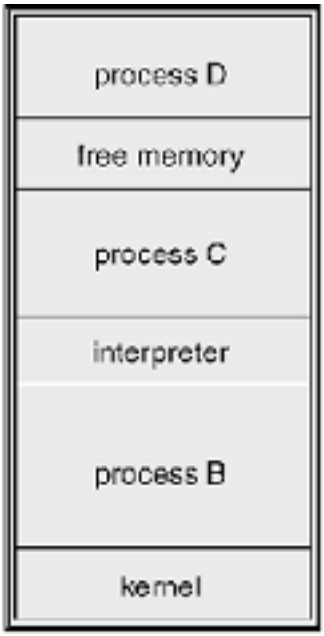
- *Message-passing*

  Pertukaran informasi dilakukan melalui fasilitas komunikasi antar proses yang disediakan oleh sistem operasi.

- *Shared-memory*

  Proses menggunakan memori yang bisa digunakan oleh berbagai proses untuk pertukaran informasi dengan membaca dan menulis data pada memori tersebut.

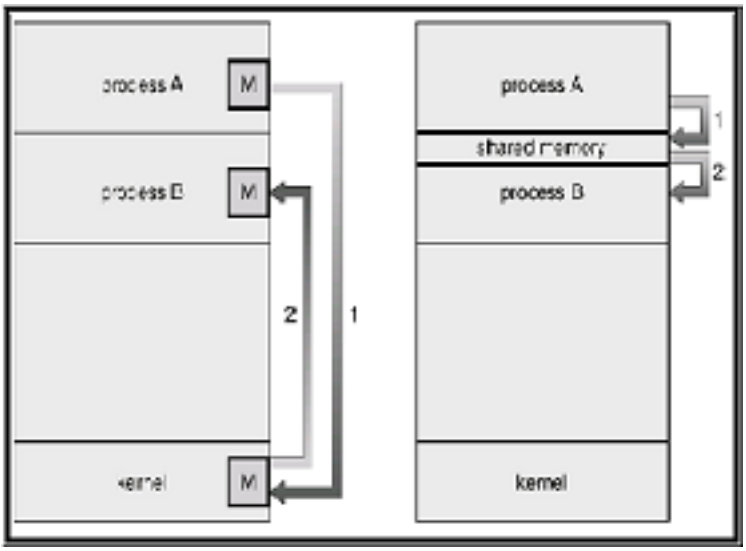
Gambar 6.3. Multi program pada Unix



Sumber: Silberschatz, 2003, hal. 69.

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

Gambar 6.4. Mekanisme komunikasi



Sumber: Silberschatz, 2003, hal. 72.

6.9. Rangkuman

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

6.10. Latihan

1. Jelaskan apa yang dimaksud dengan *Command-Interpreter System!*
2. Apakah tujuan dari *System Calls*?
3. Sebutkan lima layanan yang disediakan oleh Sistem Operasi!
4. Sebutkan tujuan dari *system program*!
5. Jelaskan dua model komunikasi pada Sistem Operasi!

Rujukan

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

[Stallings2001] William Stallings. 2001. *Operating Systems: Internal and Design Principles*. Fourth

Edition. Edisi Keempat. Prentice-Hall International. New Jersey.

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

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

Bab 7. Struktur Sistem Operasi

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

- Struktur Sederhana
- Pendekatan *Berlapis*
- Kernel Mikro

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

7.1. Struktur Sederhana

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

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

7.2. Pendekatan Berlapis

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

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

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

- Lapisan 0

  Mengatur alokasi prosesor, pertukaran antar proses ketika interupsi terjadi atau waktu habis. Lapisan ini mendukung dasar *multi-programming* pada CPU.

- Lapisan 1

  Mengalokasikan ruang untuk proses di memori utama dan pada 512 kilo word *drum* yang digunakan untuk menahan bagian proses ketika tidak ada ruang di memori utama.

- Lapisan 2

Menangani komunikasi antara masing-masing proses dan operator *console*. Pada lapis ini masing-masing proses secara efektif memiliki opertor *console* sendiri.

- Lapisan 3

  Mengatur peranti M/K dan menampung informasi yang mengalir dari dan ke proses tersebut.

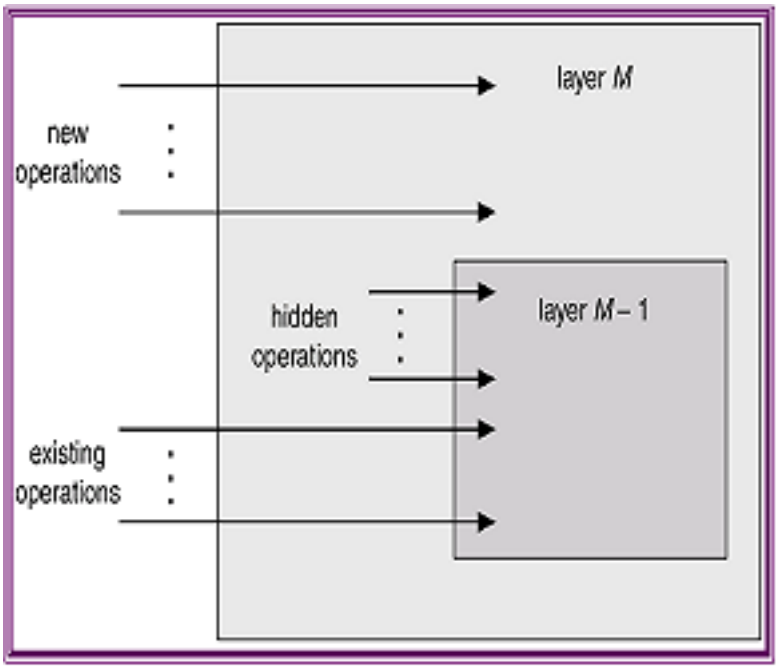
- Lapisan 4

  Tempat program pengguna. Pengguna tidak perlu memikirkan tentang proses, memori, *console*, atau managemen M/K.

- Lapisan 5

  Merupakan operator sistem.

Gambar 7.1. Lapisan pada Sistem Operasi



Sumber: Silberschatz, 2003, hal. 77.

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

- Level 1

  Terdiri dari sirkuit elektronik dimana obyek yang ditangani adalah *register memory cell*, dan gerbang logika. Operasi pada obyek ini seperti membersihkan register atau membaca lokasi memori.

- Level 2

  Pada level ini adalah set instruksi pada prosesor. Operasinya adalah instruksi bahasa-mesin, seperti menambah, mengurangi, *load* dan *store*.

- Level 3

  Tambahan konsep prosedur atau subrutin ditambah operasi *call* atau *return*.

- Level 4

  Mengenalkan interupsi yang menyebabkan prosesor harus menyimpan perintah yang baru dijalankan dan memanggil rutin penanganan interupsi.

Gambar 7.2. Tabel Level pada Sistem Operasi

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

Sumber: Stallings, 2001, hal. 69.

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

- Level 5

  Level ini mengenalkan ide proses dalam mengeksekusi program. Kebutuhan-kebutuhan dasar pada sistem operasi untuk mendukung proses ganda termasuk kemampuan men-*suspend* dan me-*resume* proses. Hal ini membutuhkan register perangkat keras untuk menyimpan agar

eksekusi bisa ditukar antara satu proses ke proses lainnya.

- Level 6

  Mengatasi penyimpanan sekunder dari komputer. Level ini untuk menjadualkan operasi dan menanggapi permintaan proses dalam melengkapi suatu proses.

- Level 7

  Membuat alamat logik untuk proses. Level ini mengatur alamat virtual ke dalam blok yang bisa dipindahkan antara memori utama dan memori tambahan. Cara-cara yang sering dipakai adalah menggunakan ukuran halaman yang tetap, menggunakan segmen sepanjang variabelnya, dan menggunakan cara keduanya. Ketika blok yang dibutuhkan tidak ada dimemori utama, alamat logis pada level ini meminta transfer dari level 6.

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

- Level 8

  Mengatasi komunikasi informasi dan pesan-pesan antar proses. Dimana pada level 5 disediakan mekanisme penanda yang kuno yang memungkinkan untuk sinkronisasi proses, pada level ini mengatasi pembagian informasi yang lebih banyak. Salah satu peranti yang paling sesuai adalah *pipe* (pipa) yang menerima output suatu proses dan memberi input ke proses lain.

- Level 9

  Mendukung penyimpanan jangka panjang yang disebut dengan berkas. Pada level ini, data dari penyimpanan sekunder ditampilkan pada tingkat abstrak, panjang variabel yang terpisah. Hal ini bertentangan tampilan yang berorientasikan perangkat keras dari penyimpanan sekunder.

- Level 10

  Menyediakan akses ke peranti eksternal menggunakan antarmuka standar.

- Level 11

  Bertanggung-jawab mempertahankan hubungan antara internal dan eksternal *identifier* dari sumber daya dan obyek sistem. Eksternal *identifier* adalah nama yang bisa dimanfaatkan oleh aplikasi atau pengguna. Internal *identifier* adalah alamat atau indikasi lain yang bisa digunakan oleh level yang lebih rendah untuk meletakkan dan mengontrol obyek.

- Level 12

  Menyediakan suatu fasilitator yang penuh tampilan untuk mendukung proses. Hal ini merupakan lanjutan dari yang telah disediakan pada level 5. Pada level 12, semua info yang dibutuhkan untuk managemen proses dengan berurutan disediakan, termasuk alamat virtual di proses, daftar obyek dan proses yang berinteraksi dengan proses tersebut serta batasan interaksi tersebut, parameter yang harus dipenuhi proses saat pembentukan, dan karakteristik lain yang mungkin digunakan sistem operasi untuk mengontrol proses.

- Level 13

  Menyediakan antarmuka dari sistem operasi dengan pengguna yang dianggap sebagai *shell* atau dinding karena memisahkan pengguna dengan sistem operasi dan menampilkan sistem operasi dengan sederhana sebagai kumpulan servis atau pelayanan.

Dari ketiga sumber diatas dapat kita simpulkan bahwa lapisan sistem operasi secara umum terdiri atas 4 bagian, yaitu:

1. Perangkat keras

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

2. Sistem operasi

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

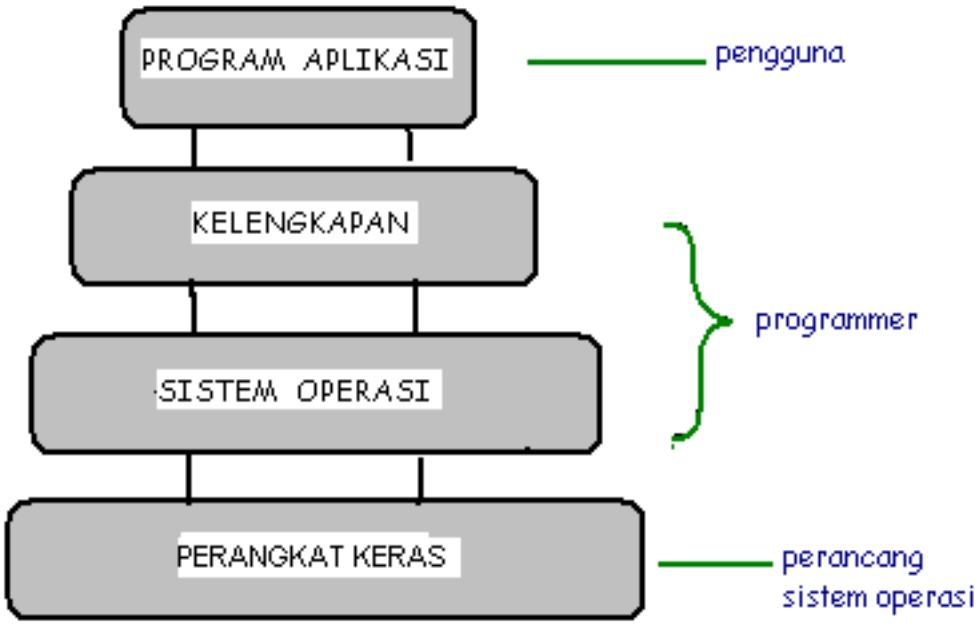
3. Kelengkapan

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

4. Program aplikasi

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

Gambar 7.3. Lapisan Sistem Operasi secara umum



Sumber: Stallings, 2001, hal. 46.

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

7.3. Kernel-mikro

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

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

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

7.4. Boot

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

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

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

7.5. *Tuning*

Adalah mungkin untuk mendesign, mengkode, dan megimplementasikan sebuah sistem operasi khusus untuk satu mesin di suatu *site*. Pada umumnya sistem operasi dibuat untuk berjalan pada beberapa kelas mesin di berbagai *site* dan berbagai konfigurasi *peripheral*. Kemudian, sistem dikonfigurasikan untuk masing-masing komputer, untuk *site* yang spesifik. Proses ini terkadang disebut sebagai ***System Generation***.

Sistem program membaca dari berkas yang diberikan atau mungkin bertanya pada operator tentang informasi yang berhubungan dengan perangkat keras tersebut, antara lain adalah sebagai berikut:

- CPU apa yang digunakan, pilihan yang diinstall?
- Berapa banyak memori yang tersedia?
- Peralatan yang tersedia?
- Pilihan Sistem operasi apa yang diinginkan atau parameter yang digunakan?

Satu kali informasi didapat, bisa digunakan dengan berbagai cara.

7.6. Kompail Kernel

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

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

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

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

- Download kernel

  Tempat untuk mendownload kernel ada di beberapa situs internet. Silakan dicari sendiri. Tetapi biasanya di "kambing.vlsm.org" ada versi-versi kernel terbaru. Source kernel tersebut biasanya dalam format linux-X.Y.ZZ.tar.gz, di mana X.Y.ZZ menunjukkan nomor versi kernel.

  Misalnya 2.6.11. Nomor tersebut dibagi menjadi tiga bagian, yaitu Major number, Minor number, Revision number.

  Pada contoh versi kernel di atas (2.6.40), angka 2 menunjukkan major number. Angka ini jarang berubah. Perubahan angka ini menandakan adanya perubahan besar (upgrade) pada kernel. Kemudian angka 6 menunjukkan minor number. Angka ini menunjukkan stabilitas.

  - Angka genap (0, 2, 4, 6, dst.) menunjukkan kernel tersebut telah stabil.
  - Angka ganjil menandakan bahwa kernel tersebut dalam tahap pengembangan Kernel ganjil mengandung experimental-code atau fitur terbaru yang ditambahkan oleh developernya.

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

- Kompilasi Kernel

  Kompilasi akan memakan waktu lama, dan seperti telah diberitahukan diatas, sangat mungkin untuk menimbulkan kegagalan. Di direktori /usr/src/linux, jalankan : make dep; make clean; make zImage. Perintah make dep : membaca file konfigurasi dan membentuk dependency tree. proses ini mengecek apa yang dikompilasi dan apa yang tidak. make clean : menghapus seluruh jejak kompilasi yang telah dilakukan sebelumnya. Ini memastikan agar tidak ada fitur lama yang tersisa. make zImage : Kompilasi yang sesungguhnya. Jika tidak ada kesalahan akan terbentuk kernel terkompresi dan siap dikompilasi. Sebelum dikompilasi, modul-modul yang berhubungan perlu dikompilasi juga dengan make modules. Cek lokasi kernel, umumnya nama kernel dimulai dengan vmlinuz, biasanya ada di direktori /boot, atau buka /etc/lilo.conf untuk memastikannya. Di sini tidak akan dijelaskan secara mendetail langkah-langkah dalam mengompilasi kernel. Langkah-langkah ini dapat dilihat di banyak situs, salah satunya situs "bebas.vlsm.org/v09/onno-ind-1/network/ppt-linux-ethernet-10-2000.ppt" yang dibuat oleh Onno W. Purbo dengan slide. Sebelum kernel modul diinstalasi, sebaiknya back-up dulu modul lama. Keuntungan memback-up modul lama adalah bila nanti modul baru tidak berjalan baik, maka modul lama bisa digunakan lagi dengan menghapus modul baru. Setelah tahap ini selesai, jalankan lilo, reboot sistem dan lihat hasilnya.

- Konfigurasi Kernel

  Konfigurasi kernel adalah tahap terpenting yang menentukan kualitas sebuah kernel. Mana yang harus diikutkan, dan mana yang harus ditinggal sesuai tuntutan hardware dan keperluan. Konfigurasi dimulai dari directori /usr/src/linux. Ada tiga cara: make config, berupa text base interface, cocok untuk user yang memiliki terminal mode lama dan tidak memiliki setting termcap. make menuconfig, berupa text base juga tapi memiliki puldown menu berwarna, digunakan untuk user yang memiliki standar console. make xconfig, interface menggunakan layar grafik penuh, untuk user yang sudah memiliki X Window. Ada sekitar 14 menu pilihan dimulai dari Code maturity level options sampai kernel hacking. Masing-masing memiliki submenu bila dipilih dan pilihan yes(y), module(m), atau no(n). Setiap pilihan dimuat/kompilasi ke dalam kernel akan memperbesar ukuran kernel. Karena itu pilih fitur-fitur yang sering digunakan atau jadikan module untuk fitur yang tidak sering digunakan. Dan jangan memasukkan fitur-fitur yang tidak dibutuhkan. Setelah selesai melakukan pilihan konfigurasi, simpanlah sebelum keluar dari layar menu konfigurasi.

- Patch Kernel

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

7.7. Komputer Meja

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

Baik komputer meja maupun sistem operasinya, sudah sangat popular sehingga mungkin tidak perlu perkenalan lebih lanjut. Bahkan, mungkin banyak yang belum menyadari bahwa masih terdapat banyak jenis komputer dan sistem operasi lainnya. Dewasa ini (2005), komputer meja lebih canggih ribuan kali dibandingkan IBM PC yang pertama (1981, prosesor 8088, 4.77 MHz). Sedangkan PC pertama tersebut, beberapa kali lebih canggih dibandingkan *main-frame* tahun 1960-an.

Titik fokus perancangan sistem operasi jenis komputer meja agak berbeda dibadingkan dengan sistem operasi "*main-frame*". *Pertama*, kinerja serta derajat kerumitan komponen perangkat keras komputer meja jauh lebih sederhana (dan murah). Karena itu, "*utilisasi*" perangkat keras tidak lagi menjadi masalah utama. *Kedua*, para pengguna komputer meja tidak selalu merupakan "pakar", sehingga kemudahan penggunaan menjadi prioritas utama dalam perancangan sistem operasinya. Ketiga, akibat dari butir kedua di atas, "keamanan" dan "perlindungan" kurang mendapatkan perhatian. Dewasa ini (2005), "virus" dan "cacing" (*worm*) telah menjadi masalah utama yang dihadapi para pengguna sistem operasi komputer meja tertentu.

Yang juga termasuk keluarga komputer meja ini ialah komputer jinjing. Pada dasarnya, tidak terdapat banyak perbedaan, kecuali:

a. sistem *portable* ini pada dasarnya mirip dengan sistem komputer meja, namun harganya relatif lebih mahal.

b. penggunaan catu daya internal (baterei) agar catu daya dapat bertahan selama mungkin (rata-rata 3-6 jam).

c. bobot komputer yang lebih ringan, serta ukuran komputer yang nyaman untuk dijinjing. Sistem ini nyaman digunakan untuk bekerja di perjalanan atau pekerjaan yang menuntut fleksibilitas tempat.

7.8. Sistem Prosesor Jamak

Pada umumnya, setiap komputer dilengkapi dengan satu buah prosesor (CPU). Namun, dewasa ini (2005) mulai umum, jika sebuat sistem komputer memiliki lebih dari satu prosesor (*multi-procesor*). Terdapat dua jenis sistem prosesor jamak, yaitu *Symmetric MultiProcessing (SMP)* dan *Asymmetric MultiProcessing (ASMP)*. Dalam *SMP* setiap prosesor menjalankan salinan identik dari sistem operasi dan banyak job yang dapat berjalan di suatu waktu tanpa pengurangan kinerja. Sementara itu dalam *ASMP* setiap prosesor diberikan suatu tugas yang spesifik. Sebuah prosesor bertindak sebagai *Master processor* yang bertugas menjadualkan dan mengalokasikan pekerjaan pada prosesor lain yang disebut *slave processors*. Umumnya ASMP digunakan pada sistem yang besar.

Sistem Operasi Jamak memiliki beberapa keunggulan [Silbeschatz 2004]:

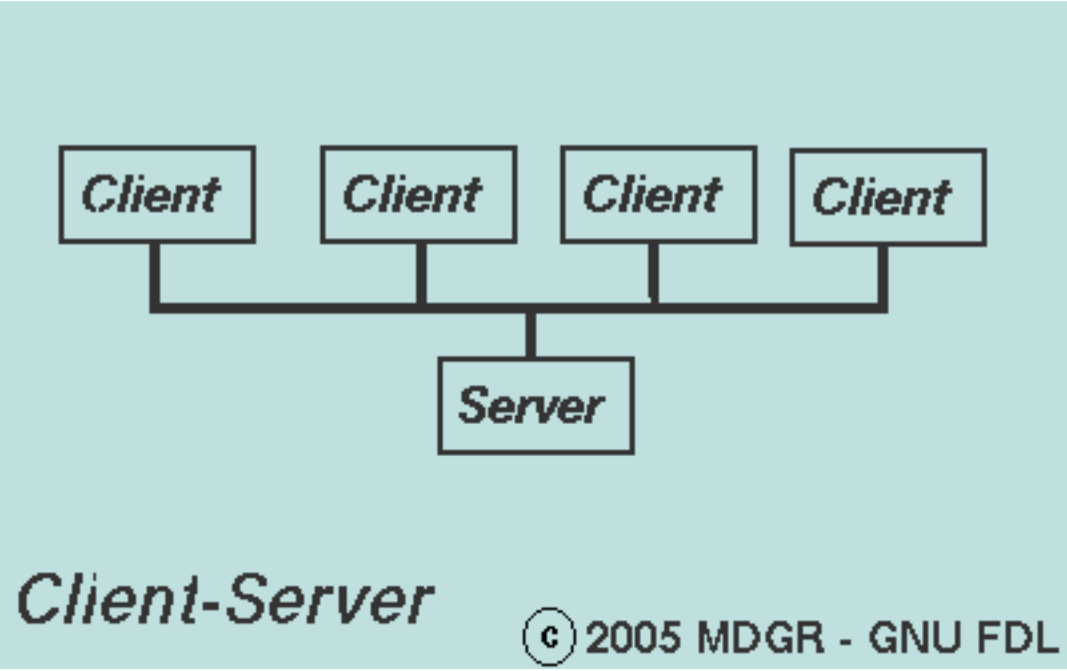
a. Peningkatan *throughput* karena lebih banyak proses/*thread* yang dapat dijalankan sekali gus. Perlu diingat hal ini tidak berarti daya komputasinya menjadi meningkat sejumlah prosesornya. Yang meningkat ialah jumlah pekerjaan yang bisa dilakukannya dalam waktu tertentu.

b. Economy of Scale: Ekonomis dalam peralatan yang dibagi bersama. Prosesor-prosesor terdapat dalam satu komputer dan dapat membagi *peripheral* (ekonomis) seperti disk dan catu daya listrik.

c. Peningkatan Kehandalan: Jika satu prossor mengalami suatu gangguan, maka proses yang terjadi masih dapat berjalan dengan baik karena tugas prosesor yang terganggu diambil alih oleh prosesor lain. Hal ini dikenal dengan istilah *Graceful Degradation*. Sistemnya sendiri dikenal bersifat *fault tolerant* atau *fail-soft system*.

7.9. Sistem Terdistribusi dan Terkluster

Gambar 7.4. Sistem Terdistribusi



Melaksanakan komputasi secara terdistribusi diantara beberapa prosesor. Hanya saja komputasinya bersifat *loosely coupled system* yaitu setiap prosesor mempunyai memori lokal sendiri. Komunikasi terjadi melalui bus atau jalur telepon. Keuntungannya hampir sama dengan prosesor jamak

multiprocessor), yaitu adanya pembagian sumber daya dan komputasi lebih cepat. Namun, pada distributed system juga terdapat keuntungan lain, yaitu memungkinkan komunikasi antar komputer.

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

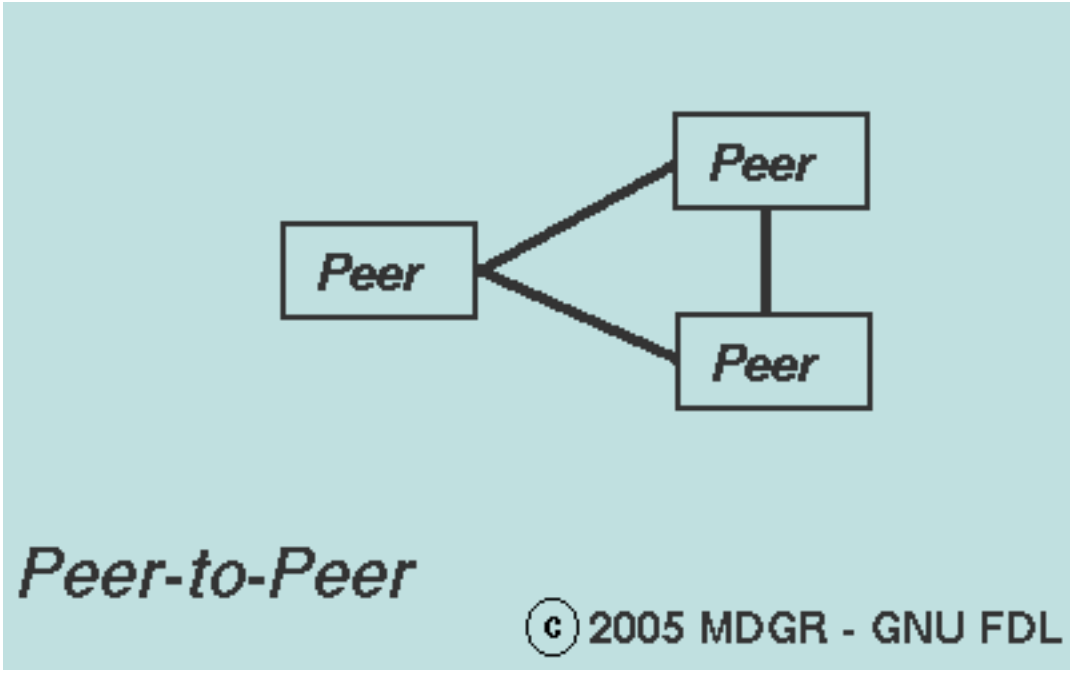
Terdapat sekurangnya tiga model dalam sistem terdistribusi ini. Pertama, sistem *client*/*server* yang membagi jaringan berdasarkan pemberi dan penerima jasa layanan. Pada sebuah jaringan akan didapatkan: *file server*, *time server*, *directory server*, *printer server*, dan seterusnya. Kedua, sistem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

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

Sistem kluster ialah gabungan dari beberapa sistem individual (komputer) yang dikumpulkan pada suatu lokasi, saling berbagi tempat penyimpanan data (*storage*), dan saling terhubung dalam jaringan lokal (*Local Area Network*). Sistem kluster memiliki persamaan dengan sistem paralel dalam hal menggabungkan beberapa CPU untuk meningkatkan kinerja komputasi. Jika salah satu mesin mengalami masalah dalam menjalankan tugas maka mesin lain dapat mengambil alih pelaksanaan tugas itu. Dengan demikian, sistem akan lebih andal dan *fault tolerant* dalam melakukan komputasi.

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

Gambar 7.5. Sistem Terdistribusi



Dalam ruang lingkup jaringan lokal, sistem kluster memiliki beberapa model dalam pelaksanaannya: asimetris dan simetris. Kedua model ini berbeda dalam hal pengawasan mesin yang

sedang bekerja. Pengawasan dalam model asimetris menempatkan suatu mesin yang tidak melakukan kegiatan apa pun selain bersiap-siaga mengawasi mesin yang bekerja. Jika mesin itu masalah maka pengawas akan segera mengambil alih tugasnya. Mesin yang khusus bertindak pengawas ini tidak diterapkan dalam model simetris. Sebagai gantinya, mesin-mesin yang melakukan komputasi saling mengawasi keadaan mereka. Mesin lain akan mengambil alih tugas mesin yang sedang mengalami masalah.

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

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

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

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

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

7.10. Sistem Waktu Nyata

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

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

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

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

Menurut Morgan [MORG92], terdapat sekurangnya lima karakteristik dari sebuah sistem waktu nyata

- deterministik, dapat ditebak berapa waktu yang dipergunakan untuk mengeksekusi operasi.
- responsif, kapan secara pasti eksekusi dimulai serta diakhiri.
- kendali pengguna, dengan menyediakan pilihan lebih banyak daripada sistem operasi biasa.
- kehandalan, sehingga dapat menanggulangi masalah-masalah pengecualian dengan derajat tertentu.
- penanganan kegagalan, agar sistem tidak langsung *crash*.

7.11. Aspek Lain

Masih terdapat banyak aspek sistem operasi yang lain; yang kurang cocok diuraikan dalam bab pendahuluan ini. Sebagai penutup dari sub-pokok bahasan ini; akan disinggung secara singkat perihal:

- Sistem Multimedia
- *Embeded System*
- Komputasi Berbasis Jaringan
- PDA dan Telepon Seluler
- *Smart Card*

Sistem MultiMedia

Sistem multimedia merupakan sistem yang mendukung sekali gus berbagai medium seperti gambar tidak bergerak, video (gambar bergerak), data teks, suara, dan seterusnya. Sistem operasi yang mendukung multimedia seharusnya memiliki karakteristik sebagai berikut:

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

Embeded System

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

Mengacu pada sistem komputer yang bertugas mengendalikan tugas spesifik dari suatu alat seperti

mesin cuci digital, tv digital, radio digital. Terbatas dan hampir tak memiliki *user-interface*. Biasanya melakukan tugasnya secara *real-time* merupakan sistem paling banyak dipakai dalam kehidupan.

Komputasi Berbasis Jaringan

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

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

PDA dan Telepon Seluler

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

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

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

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

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

Smart Card

Smart Card (Kartu Pintar) merupakan sistem komputer dengan ukuran kartu nama. Kemampuan komputasi dan kapasitas memori sistem ini sangat terbatas sehingga optimasi merupakan hal yang paling memerlukan perhatian. Umumnya, sistem ini digunakan untuk menyimpan informasi rahasia untuk mengakses sistem lain. Umpamanya, telepon seluler, kartu pengenal, kartu bank, kartu kredit, sistem wireless, uang elektronis, dst.

Dewasa ini (2005), *smart card* dilengkapi dengan prosesor 8 bit (5 MHz), 24 kB ROM, 16 kB EEPROM, dan 1 kB RAM. Namun kemampuan ini meningkat drastis dari waktu ke waktu.

7.12. Rangkuman

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

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

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

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

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

7.13. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:
   - *"Symetric Multiprocessing" vs. "Asymetric Multiprocessing"*.
   - *"Distributed Systems" vs. "Clustered Systems"*.
   - *"Microkernels" vs. "Virtual Machines"*.
   - *"Hard Real-time" vs "Soft Real-time"*.
2. Sebutkan keuntungan dalam penggunaan sistem lapisan! Jelaskan!
3. Jelaskan salah satu kesulitan besar dalam penggunaan sistem lapisan !
4. Sebutkan beberapa obyek yang ada pada level dua sistem lapisan!
5. Sebutkan cara-cara yang dipakai untuk membuat alamat logis yang dilakukan pada level tujuh!
6. Sebutkan salah satu keuntungan kernelmikro!

7.14. Rujukan

Abraham Silberschatz, Peter Galvin, Greg Gagne.2003. *Operating System Concepts, Sixth Edition*.John Wiley & Sons.

Andrew S Tanenbaum, Albert S Woodhull.1997.*Operating System Design and Implementation, Second Edition*. Prentice Hall.

Andrew S Tanenbaum.2001.*Modern Operating System, Second Edition*.Prentice Hall.

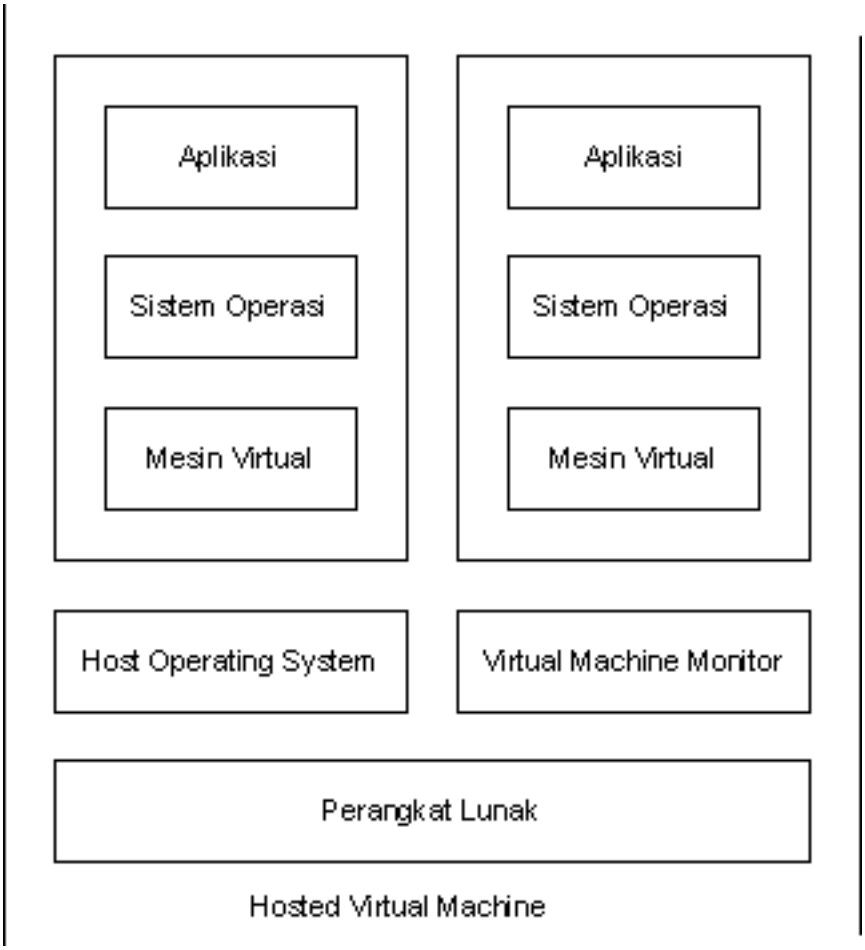
William Stallings.2001.*Operating Systems, Fourth Edition.* Prentice Hall.

http://www.linux.org.uk/~davej/docs/post-halloween-2.6.txt per 13 Desember 2004.

Bab 8. Mesin Virtual Java

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

Gambar 8.1. Struktur Mesin Virtual



Sumber: http://utenti.lycos.it/ yanorel6/ 2/ ch52.htm

8.1. Konsep Mesin Virtual

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

- Perangkat keras

- Kernel
- Sistem program

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

Kelemahan Mesin Virtual

Kesulitan utama dari konsep VM adalah dalam hal sistem penyimpanan dan pengimplementasian. Sebagai contoh, kesulitan dalam sistem penyimpanan adalah sebagai berikut. Andaikan kita mempunyai suatu mesin yang memiliki 3 disk drive namun ingin mendukung 7 VM. Keadaan ini jelas tidak memungkinkan bagi kita untuk dapat mengalokasikan setiap disk drive untuk tiap VM, karena perangkat lunak untuk mesin virtual sendiri akan membutuhkan ruang disk secara substansi untuk menyediakan memori virtual dan spooling. Solusinya adalah dengan menyediakan disk virtual, atau yang dikenal pula dengan minidisk, di mana ukuran daya penyimpanannya identik dengan ukuran sebenarnya. Sistem disk virtual mengimplementasikan tiap minidisk dengan mengalokasikan sebanyak mungkin track dari disk fisik sebanyak kebutuhan minidisk itu. Secara nyata, total kapasitas dari semua minidisk harus lebih kecil dari kapasitas disk fisik yang tersedia. Dengan demikian, pendekatan VM juga menyediakan sebuah antarmuka yang identik dengan underlying bare hardware. VM dibuat dengan pembagian sumber daya oleh physical computer. Pembagian minidisk sendiri diimplementasikan dalam perangkat lunak.

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

Keunggulan Mesin Virtual

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

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

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

Contoh Mesin Virtual

Contoh penerapan VM saat ini terdapat pada sistem operasi Linux. Mesin virtual saat ini

memungkinkan aplikasi Windows untuk berjalan pada komputer yang berbasis Linux. VM juga berjalan pada aplikasi Windows dan sistem operasi Windows.

8.2. Konsep Bahasa Java

Sun Microsystems mendesain bahasa Java, yang pada mulanya dikenal dengan nama Oak. James Gosling, sang pencipta Oak, menciptakannya sebagai bagian dari bahasa C++. Bahasa ini harus cukup kecil agar dapat bertukar informasi dengan cepat di antara jaringan kabel perusahaan dan pertelevisian dan cukup beragam agar dapat digunakan lebih dari satu jaringan kabel.

Sun Microsystems lalu merubah nama Oak menjadi Java, kemudian membuatnya tersedia di dalam Internet. Perkenalan dengan Java di Internet ini dimulai pada tahun 1995.

Java didesain dengan tujuan utama portabilitas, sesuai dengan konsep write once run anywhere. Jadi, hasil kompilasi bahasa Java bukanlah native code, melainkan bytecode. Bytecode dieksekusi oleh interpreter Java yang juga merupakan Java Virtual Machine. Penjelasan mengenai Java Virtual Machine (JVM) akan dijelaskan pada bab 8.3.

Ada beberapa hal yang membedakan Java dengan bahasa pemrograman lain yang populer pada saat ini, yakni:

- Bersifat portable, artinya program Java dapat dijalankan pada platform yang berbeda tanpa perlu adanya kompilasi ulang.
- Memiliki garbage collection yang berfungsi untuk mendelokasi memori secara otomatis.
- Menghilangkan pewarisan ganda, yang merupakan perbaikan dari bahasa C++.
- Tidak ada penggunaan pointer, artinya bahasa Java tidak membolehkan pengaksesan memori secara langsung.

Teknologi Java terdiri dari tiga komponen penting, yakni:

- ???
- Application Programming Interface (API)
- Spesifikasi mesin virtual

Penjelasan lebih lanjut mengenai komponen Java adalah sebagai berikut.

Bahasa Pemrograman

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

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

Contoh 8.1. Contoh penggunaan class objek dalam Java

```
01 class Objek1
02 {
03      private int attribut1;
04      private String attribut2;
05
06      public void changeAttribut1()
07      {
08              // melakukan sesuatu terhadap attribut1 harus dengan
09              // method ini. Jadi variabel attribut1 aman di dalam
10              // objeknya, tidak mudah diakses begitu saja...
11      }
12 }
13
14 class Objek2
15 {
16      private int attribut1;
17      private String attribut2;
18
19      public Objek1 objekSatu;
20
21      public void interfensi()
22      {
23              objekSatu.changeAttribut1();
24              // valid karena akses modifiernya public
25
26              System.out.print( objekSatu.attribut1 );
27              // invalid karena akses modifiernya private
28      }
29 }
```

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

API

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

API dalam Java

Terdiri dari tiga bagian, yaitu:

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

Contoh 8.2. Contoh penggunaan Java API

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

8.3. Mesin Virtual Java

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

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

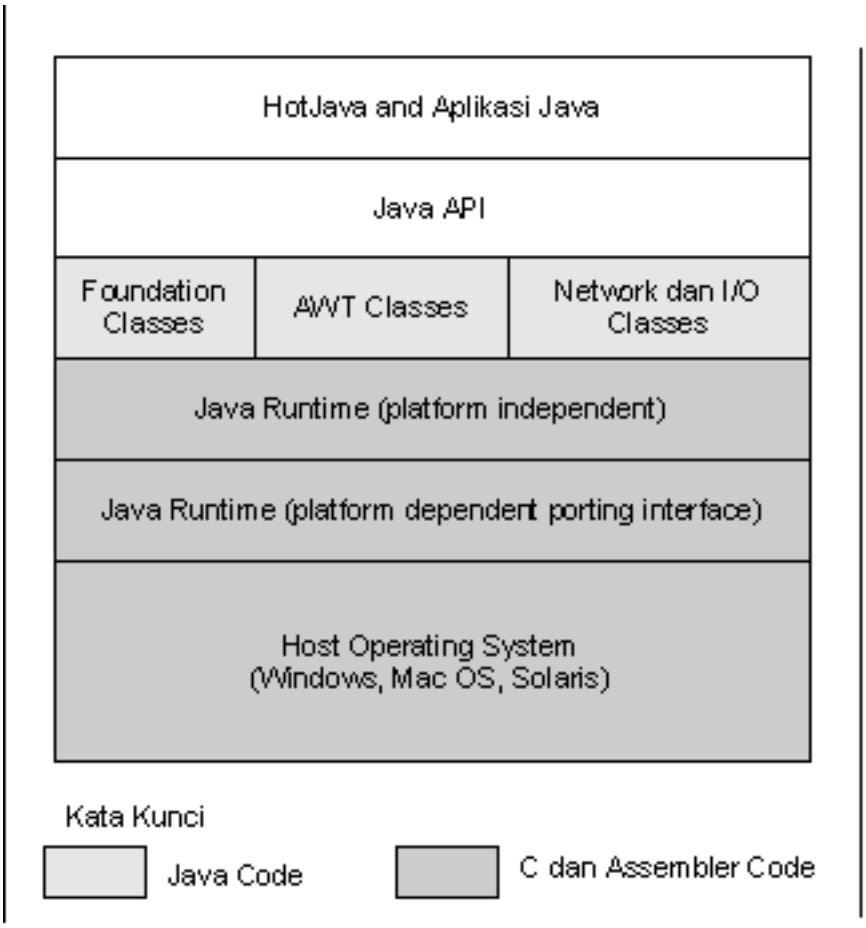
Pada applet, JVM menciptakan method main() sebelum membuat applet itu sendiri. Java Development Environment terdiri dari sebuah Compile- Time Environment dan Runtime Environment. Compile berfungsi mengubah sourcecode Java menjadi bytecode, sedangkan Runtime merupakan Java Platform untuk sistem Host.

ZCZCOLD

Pada dasarnya, sistem komputer dibangun atas lapisan-lapisan (*layers*). Perangkat keras adalah lapisan terendah. Kernel yang berada di lapisan atasnya menggunakan instruksi perangkat keras untuk menciptakan seperangkat *system calls* yang dapat digunakan oleh komponen-komponen lain yang berada pada level di atasnya. Sistem program pada level di atasnya dapat menggunakan *system call* dan perangkat keras seolah-olah mereka berada pada level yang sama.

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

Gambar 8.2. Gambar ...



Sumber: http://utenti.lycos.it/ yanorel6/ 2/ ch52.htm

Meski sangat berguna, VM sulit untuk diimplementasikan. Banyak hal yang dibutuhkan untuk menyediakan duplikat yang tepat dari *underlying machine*. VM dapat dieksekusi pada *only user mode* sehingga kita harus mempunyai *virtual user mode* sekaligus *virtual memory mode* yang keduanya berjalan di *physical user mode*. Ketika instruksi yang hanya membutuhkan *virtual user mode* dijalankan, ia akan mengubah isi register yang berefek pada *virtual monitor mode* sehingga dapat memulai ulang VM tersebut. Sebuah instruksi M/K yang membutuh waktu 100 ms, dengan menggunakan VM bisa dieksekusi lebih cepat karena *spooling* atau lebih lambat karena interpreter. Terlebih lagi, CPU menjadi *multiprogrammed* di antara banyak VM. Jika setiap user diberi 1 VM, dia akan bebas menjalankan sistem operasi (kernel) yang diinginkan pada VM tersebut.

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

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

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

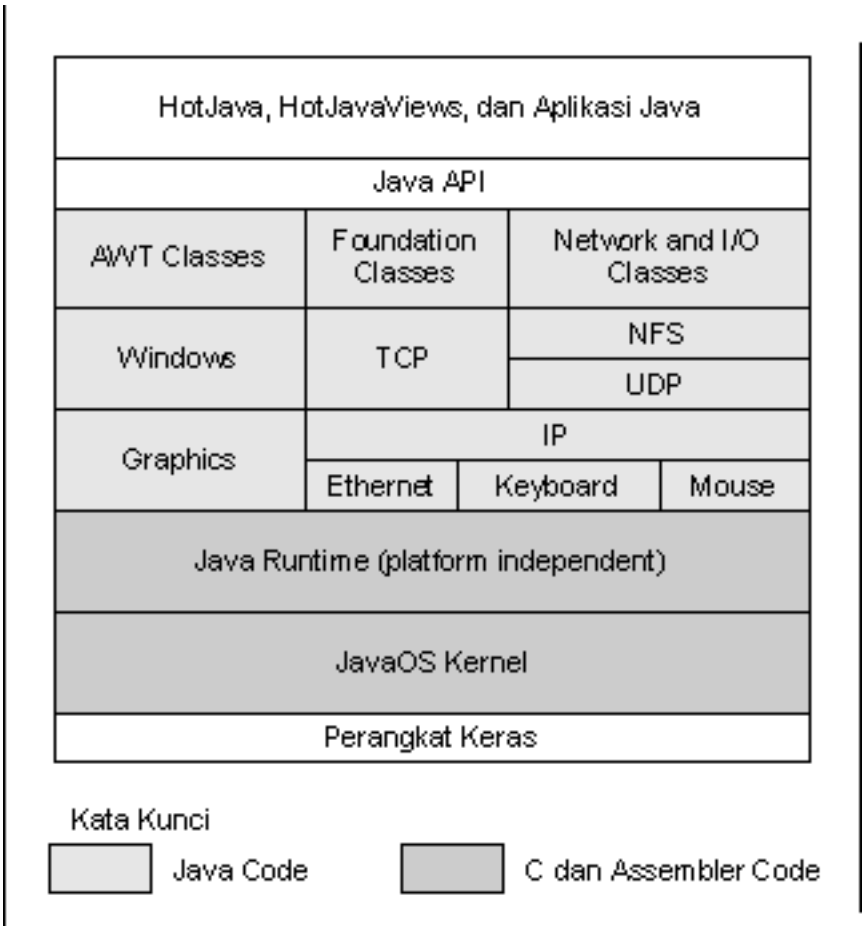
8.4. Sistem Operasi Java

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

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

Sistem Operasi JavaOS

Gambar 8.3. Struktur sistem operasi JavaOS



Sumber: http://utenti.lycos.it/ yanorel6/ 2/ ch52.htm

JavaOS adalah satu-satunya sistem yang mencoba untuk mengimplementasi fungsi sistem operasi dalam bahasa Java secara lengkap. JavaOS mengimplementasi platform Java agar dapat menjalankan aplikasi atau applet yang mengakses fasilitas dari beberapa objek. Selain itu, JavaOS

juga mengimplementasikan JVM dan lapisan fungsionalitas untuk windowing, jaringan, dan sistem berkas tanpa membutuhkan dukungan dari sistem operasi lokal. JavaOS mendefinisikan platform seperti halnya CPU, memori, bus, dan perangkat keras lainnya. Platform independen dari sistem operasinya disebut JavaOS runtime, sedangkan bagian platform yang non-independen dari sistem operasinya disebut JavaOS kernel.

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

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

Kernel JavaOS

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

Gambar 8.4. PL3

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

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

Komponen JavaOS Runtime

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

Sistem Operasi JX

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

Berikut penjelasan lebih lanjut mengenai arsitektur sistem operasi JX:

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

Kelemahan sistem operasi berbasis bahasa pemrograman

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

8.5. Rangkuman

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

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

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

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

ZCZCOLD

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

untuk masing-masing komputer.

8.6. Latihan

1. Sebutkan keuntungan dan kelebihan menggunakan mesin virtual!
2. Jelaskan masalah utama dari desain sistem!
3. Jelaskan perbedaan mekanisme dan kebijakan!
4. Mengapa mesin virtual sulit untuk diimplementasikan?
5. Sebutkan 2 keunggulan bahasa Java!
6. Sebutkan tiga komponen penting dalam teknologi Java!
7. Jelaskan secara singkat fungsi dari API pada bahasa Java!
8. Jelaskan tugas class loader pada mesin virtual Java!
9. Sebutkan beberapa fungsi yang disediakan oleh kernel JavaOS!
10. Jelaskan mekanisme arsitektur sistem operasi JX!
11. Apakah kelemahan dari desain sistem operasi yang berbasis bahasa pemrograman? Jelaskan secara singkat!

8.7. Rujukan

http://utenti.lycos.it/yanorel6/2/ch52.htm

http://casl.csa.iisc.ernet.in/OperatingSystems/JavaOS/

http://www.javasoft.com/products/javaos/javaos.white.html

http://www.jxos.org/publications/jx-usenix.pdf

Abraham Silberschatz, Peter Galvin, Greg Gagne.2003. *Operating System Concepts, Sixth Edition*.John Wiley & Sons.

[Silberschatz2004] Avi Silberschatz, Peter Galvin, and Greg Gagne, 2004, Operating System Concepts with Java, 6th Edition, John Wiley & Sons.

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

Bab 9. Sistem GNU/Linux

9.1. Sejarah Kernel Linux

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

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

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

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

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

Satu tahun setelah versi 1.0 dirilis, kernel 1.2 keluar. Kernel versi 1.2 ini mendukung variasi perangkat keras yang lebih luas. Pengembang telah memperbaharui networking stack untuk menyediakan support bagi protokol IPX, dan membuat implementasi IP lebih lengkap dengan memberikan fungsi accounting dan firewalling. Kernel 1.2 ini merupakan kernel Linux terakhir yang PC-only. Konsentrasi lebih diberikan pada dukungan perangkat keras dan memperbanyak implementasi lengkap pada fungsi-fungsi yang ada.

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

Semenjak Desember 2003, telah diluncurkan kernel versi 2.6, yang dewasa ini (2005) telah mencapai *patch* versi 2.6.12.3. Hal-hal yang berubah dari versi 2.6 ini ialah:

- Subsitem IO yang dipercanggih.
- Kernel yang pre-emptif.

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

9.2. Sistem dan Distribusi GNU/Linux

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

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

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

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

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

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

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

Sejak dirilisnya Slackware, sejumlah besar distribusi komersial dan non-komersial Linux telah

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

9.3. Lisensi Linux

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

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

9.4. Linux Saat Ini

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

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

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

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

- The Tetragrid, sebuah *mega computer* dari Amerika yang dapat menghitung lebih dari 13 trilyun kalkulasi per detik (13.6 TeraFLOPS -- *FLoating Operations Per Second*). Tetragrid dapat dimanfaatkan untuk mencari solusi dari masalah matematika kompleks dan simulasi, dari astronomi dan riset kanker hingga ramalan cuaca.

- Evolocity, juga dari Amerika, dapat berjalan dengan kecepatan maksimum 9.2 TeraFLOPS, menjadikannya sebagai salah satu dari lima *supercomputer* tercepat di dunia.

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

Gambar 9.1. Logo Linux. Sumber: . . .

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

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

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

9.5. Prinsip Rancangan Linux

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

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

Akibat dari semakin berkembangnya teknologi PC, kernel Linux juga semakin lengkap dalam mengimplementasikan fungsi UNIX. Tujuan utama perancangan Linux adalah cepat dan efisien, tetapi akhir-akhir ini konsentrasi perkembangan Linux lebih pada tujuan rancangan yang ketiga yaitu standarisasi. Standar POSIX terdiri dari kumpulan spesifikasi dari beberapa aspek yang berbeda kelakuan sistem operasi. Ada dokumen POSIX untuk fungsi sistem operasi biasa dan untuk

ekstensi seperti proses untuk thread dan operasi *real-time*. Linux dirancang agar sesuai dengan dokumen POSIX yang relevan. Sedikitnya ada dua distribusi Linux yang sudah memperoleh sertifikasi ofisial POSIX.

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

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

9.6. Kernel

Sistem Linux terdiri dari tiga bagian kode penting:

- Kernel: Bertanggung-jawab memelihara semua abstraksi penting dari sistem operasi, termasuk hal-hal seperti memori virtual dan proses-proses.
- Perpustakaan sistem: menentukan kumpulan fungsi standar dimana aplikasi dapat berinteraksi dengan kernel, dan mengimplementasi hampir semua fungsi sistem operasi yang tidak memerlukan hak penuh atas kernel.
- Utilitas sistem: adalah program yang melakukan pekerjaan managemen secara individual.

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

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

9.7. Perpustakaan Sistem

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

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

9.8. Utilitas Sistem

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

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

9.9. Modul Kernel Linux

Pengertian Modul Kernel Linux

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

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

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

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

Managemen Modul Kernel Linux

Managemen modul akan mengatur pemanggilan modul ke dalam memori dan berkomunikasi dengan bagian lainnya dari kernel. Memanggil sebuah modul tidak hanya memasukkan isi binarinya ke dalam memori kernel, namun juga harus dipastikan bahwa setiap rujukan yang dibuat oleh modul ke simbol kernel atau pun titik masukan diperbaharui untuk menunjuk ke lokasi yang benar di alamat kernel. Linux membuat tabel simbol internal di kernel. Tabel ini tidak memuat semua simbol

yang didefinisikan di kernel saat kompilasi, namun simbol-simbol tersebut harus diekspor secara eksplisit oleh kernel. Semua hal ini diperlukan untuk penanganan rujukan yang dilakukan oleh modul terhadap simbol-simbol.

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

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

Registrasi Driver

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

Hal-hal yang masuk dalam tabel registrasi adalah:

- *device driver*
- sistem berkas
- protokol jaringan
- format binari

Resolusi Konflik

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

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

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

9.10. Rangkuman

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

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

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

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

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

9.11. Latihan

1. Jelaskan perbedaan antara proses dan thread di Linux!

2. Jabarkan fitur dan keunggulan/kekurangan dari kernel Linux v0.01, v1.0, v2.0.

3. Sebutkan tiga komponen utama sistem Linux. Jelaskan.

4. Apakah kelebihan dan kekurangan protokol IPX, TCP/IP, dan UDP?

5. GNU/Linux

   a) Sebutkan perbedaan utama antara kernel linux versi 1.X dan versi 2.X !

   b) Terangkan, apa yang disebut dengan "Distribusi (distro) Linux"? Berikan empat contoh distro!

   c) Berikut merupakan sebagian dari keluaran menjalankan perintah "top b n 1" pada server "bunga.mhs.cs.ui.ac.id" p ada tanggal 10 Juni 2003 yang lalu.

```
16:22:04 up 71 days, 23:40,  8 users,  load average: 0.06, 0.02, 0.00
58 processes: 57 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  15.1% user,   2.4% system,    0.0% nice,  82.5% idle
Mem:    127236K total,   122624K used,     4612K free,     2700K buffers
Swap:   263160K total,     5648K used,   257512K free,    53792K cached

  PID USER   PRI NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
    1 root     0  0   112   72    56 S     0.0  0.0   0:11 init
    2 root     0  0     0    0     0 SW    0.0  0.0   0:03 kflushd
    4 root     0  0     0    0     0 SW    0.0  0.0 156:14 kswapd
    ...
```

```
14953 root     0  0   596  308   236 S     0.0  0.2  19:12 sshd
31563 daemon   0  0   272  256   220 S     0.0  0.2   0:02 portmap
 1133 user1   18  0  2176 2176  1752 R     8.1  1.7   0:00 top
 1112 user1    0  0  2540 2492  2144 S     0.0  1.9   0:00 sshd
 1113 user1    7  0  2480 2480  2028 S     0.0  1.9   0:00 bash
30740 user2    0  0  2500 2440  2048 S     0.0  1.9   0:00 sshd
30741 user2    0  0  2456 2456  2024 S     0.0  1.9   0:00 bash
30953 user3    0  0  2500 2440  2072 S     0.0  1.9   0:00 sshd
30954 user3    0  0  2492 2492  2032 S     0.0  1.9   0:00 bash
 1109 user3    0  0  3840 3840  3132 S     0.0  3.0   0:01 pine
...
 1103 user8    0  0  2684 2684  1944 S     0.0  2.1   0:00 tin
```

9.12. Rujukan

Abraham Silberschatz, Peter Galvin, Greg Gagne. 2003. *Operating System Concepts, Sixth Edition*. John Wiley & Sons.

http://www.linux.org.uk/~davej/docs/post-halloween-2.6.txt ; per 13 Desember 2004.

Bagian III. Proses dan Penjadualan

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

Bab 10. Konsep Proses

Jika kita berdiskusi mengenai sistem operasi, maka akan timbul sebuah pertanyaan yaitu mengenai istilah apa yang tepat untuk menyebut semua kegiatan yang dilakukan oleh CPU. Sistem *batch* mengeksekusi *jobs* sebagaimana suatu sistem *time-share* menggunakan program pengguna (*user programs*) atau *tasks*. Bahkan pada sistem dengan pengguna tunggal pun, seperti pada *Microsoft Windows* dan Macintosh OS, seorang pengguna mampu menjalankan beberapa program pada saat yang sama, contohnya *Word Processor*, *Web Browser*, dan paket *e-mail*. Bahkan jika pengguna hanya dapat menjalankan satu program pada satu waktu, sistem operasi perlu untuk mendukung aktivitas program internalnya sendiri, seperti managemen memori. Dalam banyak hal, seluruh aktivitas ini adalah serupa, maka kita menyebut seluruh program itu proses-proses.

Istilah *job* dan proses digunakan hampir dapat dipertukarkan pada tulisan ini. Walau kami sendiri lebih menyukai istilah proses, banyak teori dan terminologi sistem operasi dikembangkan selama suatu waktu ketika aktivitas utama sistem operasi adalah *job processing*. Akan membingungkan jika kita menghindari penggunaan istilah yang telah diterima oleh masyarakat yang memasukkan kata *job* hanya karena proses memiliki istilah *job* sebagai pengganti atau pendahulunya.

10.1. Definisi Proses

Secara tidak langsung, proses merupakan program yang sedang dieksekusi. Menurut Silberschatz, suatu proses adalah lebih dari sebuah kode program, yang terkadang disebut ***text section***. Proses juga mencakup ***program counter***, yaitu sebuah ***stack*** untuk menyimpan alamat dari instruksi yang akan dieksekusi selanjutnya dan register. Sebuah proses pada umumnya juga memiliki sebuah ***stack*** yang berisikan data-data yang dibutuhkan selama proses dieksekusi seperti parameter metoda, alamat return dan variabel lokal, dan sebuah ***data section*** yang menyimpan variabel global.

Sama halnya dengan Silberschatz, Tanenbaum juga berpendapat bahwa proses adalah sebuah program yang dieksekusi yang mencakup *program counter*, register, dan variabel di dalamnya.

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

Untuk mempermudah kita membedakan program dengan proses, kita akan menggunakan analogi yang diberikan oleh Tanenbaum. Misalnya ada seorang tukang kue yang ingin membuat kue ulang tahun untuk anaknya. Tukang kue tersebut memiliki resep kue ulang tahun dan bahan-bahan yang dibutuhkan untuk membuat kue ulang tahun di dapurnya seperti: tepung terigu, telur, gula, bubuk vanila dan bahan-bahan lainnya. Dalam analogi ini, resep kue ulang tahun adalah sebuah program, si tukang kue tersebut adala prosesor (CPU), dan bahan-bahan untuk membuat kue tersebut adalah data input. Sedangkan proses-nya adalah kegiatan sang tukang kue untuk membaca resep, mengolah bahan, dan memanggang kue tersebut.

Walau dua proses dapat dihubungkan dengan program yang sama, program tersebut dianggap dua urutan eksekusi yang berbeda. Sebagai contoh, beberapa pengguna dapat menjalankan salinan yang berbeda pada mail program, atau pengguna yang sama dapat meminta banyak salinan dari program editor. Tiap-tiap proses ini adakah proses yang berbeda dan walau bagian *text-section* adalah sama, *data section*-nya bervariasi. Adalah umum untuk memiliki proses yang menghasilkan banyak proses begitu ia bekerja. Hal ini akan dijelaskan lebih detail pada bagian berikutnya.

10.2. Pembuatan Proses

Sebuah proses dibuat melalui system call create-process yang dilakukan oleh parent process. Setiap proses anakan (child process) dapat juga membuat proses baru.

Ketika sebuah proses dibuat maka proses tersebut dapat memperoleh resource (waktu CPU, memory, berkas atau perangkat I/O) secara langsung dari sistem operasi atau proses tersebut berbagi

resource dengan resource orang tuanya. Orang tua proses tersebut dapat membagi-bagi resource yang dimilikinya atau menggunakan secara bersama- sama resource yang dimilikinya dengan proses anakannya.

Ketika sebuah proses membuat proses baru maka terdapat dua kemungkinan dalam pelaksanaannya:

1. orang tua proses tersebut berjalan secara konkuren dengan proses anaknya.
2. orang tua proses tersebut menunggu hingga beberapa atau seluruh proses anakannya selesai.

Juga terdapat dua kemungkinan dalam pemberian ruang alamat (address space) proses yang baru:

1. proses tersebut merupakan duplikasi orang tuanya.
2. proses tersebut memiliki program yang diload ke ruang alamatnya.

Dalam sistem operasi UNIX setiap proses diidentifikasikan dengan process identifier yang berupa bilangan unik. Pembuatan proses baru dilakukan dengan system call fork. Proses baru memiliki salinan ruang alamat proses awal (orang tua proses tersebut). Mekanisme ini membuat orang tua proses tersebut dapat berkomunikasi dengan anaknnya secara mudah. Setelah sytem call fork dipanggil, orang tua proses dan anaknya dapat berjalan bersamaan, tetapi nilai kembalian (return code) kedua proses tersebut berbeda. Untuk proses anakan nilai kembaliannya adalah nol. Sedangkan nilai kembalian orang tua proses tersebut adalah bilangan selain nol (tetapi tidak negatif).

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

10.3. Terminasi Proses

Suatu proses diterminasi ketika proses tsb telah selesai mengeksekusi statement terakhir dan meminta sistem operasi untuk menghapus statement tsb dengan menggunakan system call exit. Pada saat itu, proses dapat mengembalikan data (output) kepada proses parent-nya (melalui system call wait). Semua resource yang digunakan oleh proses akan dialokasikan kembali ke tempat asalnya oleh sistem operasi.

Suatu proses dapat menyebabkan terminasi proses lain melalui system call abort. Biasanya hanya parent dari proses yang akan diterminasi yang dapat menjalankan system call abort ini. Parent sebaiknya mengetahui identitas dari children-nya. Pada saat suatu proses menciptakan sebuah proses baru, identitas dari proses yang baru diciptakan ini diperoleh dari parent-nya.

Suatu parent dapat mengakhiri eksekusi salah satu children-nya untuk alasan-alasan seperti:

- Child melampaui penggunaan resource yang telah dialokasikan. Dalam keadaan ini, parent perlu mempunyai mekanisme untuk memeriksa status children-nya.
- Task yang ditugaskan kepada child tidak lagi diperlukan.
- Parent berakhir dan sistem operasi tidak memperbolehkan suatu child untuk tetap menjalankan proses jika parent-nya sudah tidak ada. Jadi, jika suatu proses berakhir, maka semua children-nya juga harus diterminasi. Fenomena yang disebut cascading termination ini biasanya dimulai oleh sistem operasi.

Dalam UNIX, suatu proses dapat diterminasi dengan system call exit; proses parent-nya dapat menunggu proses terminasi child dengan system call wait. System call wait akan mengembalikan process identifier dari child yang diterminasi, sehingga parent dapat mengetahui child mana yang telah diterminasi. Jika parent diterminasi, semua child-nya telah diberikan proses init sehingga children tetap memiliki parent untuk mengumpulkan statistik status dan eksekusi proses.

10.4. Status Proses

Bila sebuah proses dieksekusi, maka statusnya akan berubah-ubah. Status dari sebuah proses mencerminkan aktivitas atau keadaan dari proses itu sendiri. Berikut ini adalah status-status yang mungkin dimiliki sebuah proses menurut Tanenbaum:

- *Running*: pada saat menggunakan CPU pada suatu waktu.
- *Ready*: proses diberhentikan sementara karena menunggu proses lain untuk dieksekusi.
- *Blocked*: tidak dijalankan sampai event dari luar, yang berhubungan dengan proses tersebut terjadi.

Sedangkan menurut Silberschatz, terdapat lima macam jenis status yang mungkin dimiliki oleh suatu proses:

- *New*: status yang dimiliki pada saat proses baru saja dibuat.
- *Running*: status yang dimiliki pada saat instruksi-instruksi dari sebuah proses dieksekusi.
- *Waiting*: status yang dimiliki pada saat proses menunggu suatu event (contohnya: proses M/K).
- *Ready*: status yang dimiliki pada saat proses siap untuk dieksekusi oleh prosesor.
- *Terminated*: status yang dimiliki pada saat proses telah selesai dieksekusi.

Nama-nama tersebut adalah berdasar opini, istilah tersebut bervariasi di sepanjang sistem operasi. Keadaan yang mereka gambarkan ditemukan pada seluruh sistem. Namun, pada sistem operasi tertentu lebih baik menggambarkan keadaan/status proses. Penting untuk diketahui bahwa hanya satu proses yang dapat berjalan pada prosesor mana pun pada satu waktu. Namun, banyak proses yang dapat berstatus ready atau waiting. Keadaan diagram yang berkaitan dengan keadaan tersebut dijelaskan pada Gambar 10.1, "*Process Control Block*".

Ada tiga kemungkinan bila sebuah proses memiliki status *running*:

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

10.5. *Process Control Block*

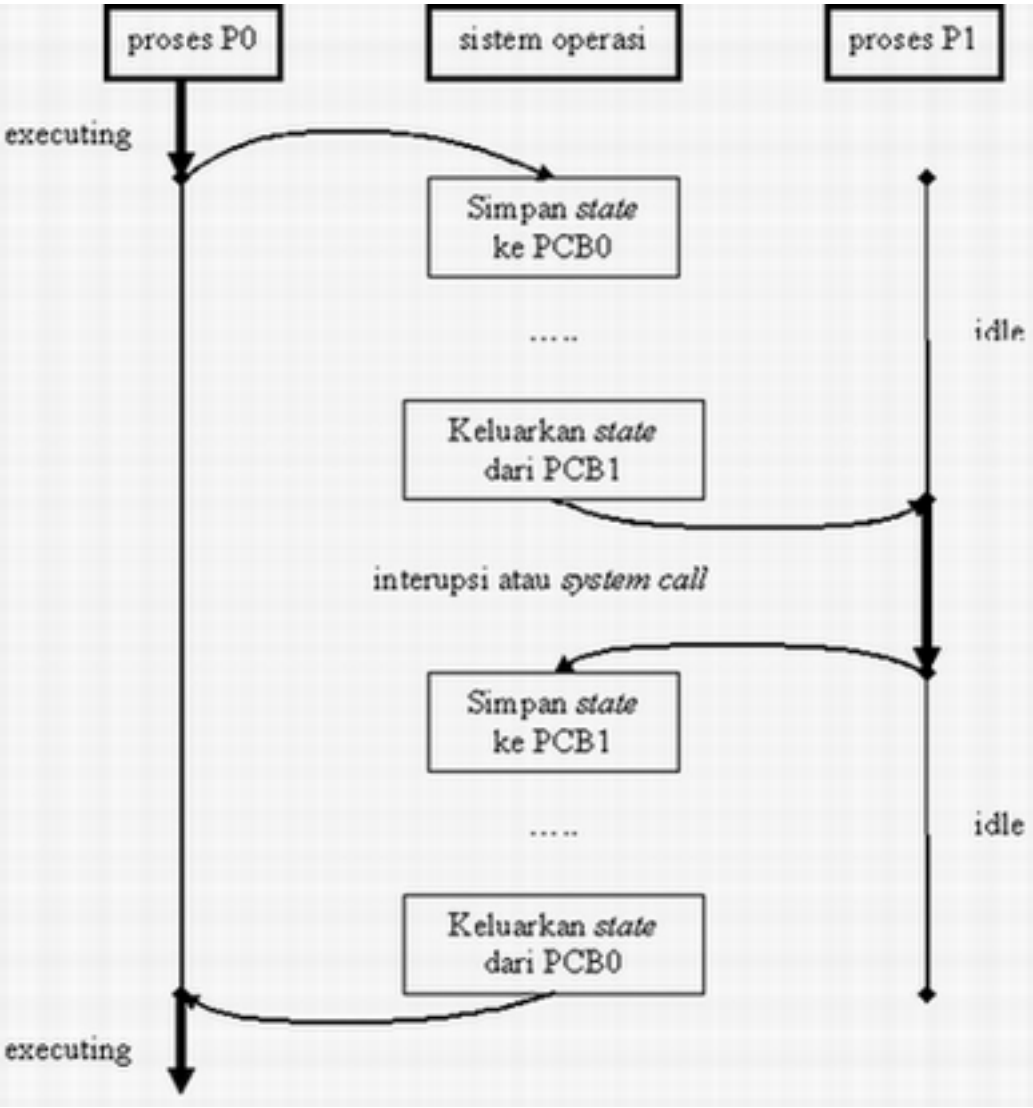
Tiap proses digambarkan dalam sistem operasi oleh sebuah ***process control block*** (PCB) - juga disebut sebuah *control block*. Sebuah PCB ditunjukkan dalam Gambar 10.1, "*Process Control Block*". PCB berisikan banyak bagian dari informasi yang berhubungan dengan sebuah proses yang spesifik, termasuk hal-hal di bawah ini:

- Status proses: status mungkin, *new*, *ready*, *running*, *waiting*, *halted*, dan juga banyak lagi.
- *Program counter*: suatu *stack* yang berisi alamat dari instruksi selanjutnya untuk dieksekusi untuk proses ini.
- *CPU register*: Register bervariasi dalam jumlah dan jenis, tergantung pada rancangan komputer. Register tersebut termasuk *accumulator*, register indeks, *stack pointer*, *general-purposes register*, ditambah *code information* pada kondisi apa pun. Beserta dengan *program counter*, keadaan/status informasi harus disimpan ketika gangguan terjadi, untuk memungkinkan proses tersebut berjalan/bekerja dengan benar setelahnya (lihat Gambar 10.2, “Status Proses”).
- Informasi managemen memori: Informasi ini dapat termasuk suatu informasi sebagai nilai dari dasar dan batas register, tabel page/halaman, atau tabel segmen tergantung pada sistem memori yang digunakan oleh sistem operasi (lihat Bagian V, “Memori”).
- Informasi pencatatan: Informasi ini termasuk jumlah dari CPU dan waktu riil yang digunakan, batas waktu, jumlah akun jumlah *job* atau proses, dan banyak lagi.
- Informasi status M/K: Informasi termasuk daftar dari perangkat M/K yang di gunakan pada proses ini, suatu daftar berkas-berkas yang sedang diakses dan banyak lagi.
- PCB hanya berfungsi sebagai tempat penyimpanan informasi yang dapat bervariasi dari proses yang satu dengan yang lain.

Gambar 10.1. *Process Control Block*

Sumber:...

Gambar 10.2. Status Proses



10.6. Hirarki Proses

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

Hieraki proses biasanya tidak sangat deep (lebih dari tiga tingkatan adalah tidak wajar), dimana hierarki berkas umumnya empat atau lima. Hierarki proses typically short-lived, kebanyakan umumnya cuma beberapa menit saja, tapi hierarki direktorinya dapat exist sampai bertahun-tahun. Pemilikan dan perlindungan juga membedakan antara proses dan berkas-berkas. Biasanya hanya sebuah parent proses dapat mengontrol atau bahkan mengakses sebuah proses anak, tapi mekanismenya membolehkan berkas-berkas dan direktori dibaca oleh a wider gorup daripada hanya owner.

Pada UNIX, proses-proses dibuat dengan FORK system call, yang membuat salinan identik dari

calling proses. Setelah fork di panggil, parent meneruskanprosesnya dan pararel dengan proses anak. UNIX menyebutnya "grup proses".

10.7. Rangkuman

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

10.8. Latihan

1. Sebutkan lima aktivitas Sistem Operasi yang merupakan contoh dari suatu managemen proses.

10.9. Rujukan

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000. John Wiley & Sons, Inc.

William Stallings: Operating Systems -- Fourth Edition, Prentice Hall, 2001.

Bab 11. Konsep *Thread*

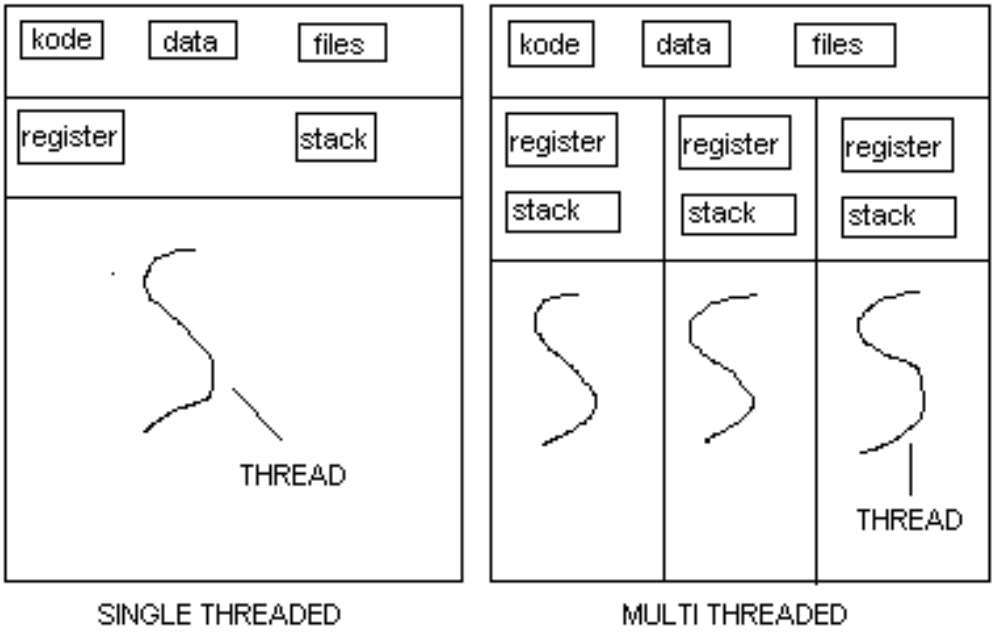
11.1. Pendahuluan

Model proses yang didiskusikan sejauh ini telah menunjukkan bahwa suatu proses adalah sebuah program yang menjalankan eksekusi *thread* tunggal. Sebagai contoh, jika sebuah proses menjalankan sebuah program *Word Processor*, ada sebuah *thread* tunggal dari instruksi-instruksi yang sedang dilaksanakan.

Kontrol *thread* tunggal ini hanya memungkinkan proses untuk menjalankan satu tugas pada satu waktu. Banyak sistem operasi modern telah memiliki konsep yang dikembangkan agar memungkinkan sebuah proses untuk memiliki eksekusi *multi-threads*, agar dapat secara terus menerus mengetik dan menjalankan pemeriksaan ejaan didalam proses yang sama, maka sistem operasi tersebut memungkinkan proses untuk menjalankan lebih dari satu tugas pada satu waktu.

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

Gambar 11.1. *Thread*



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

Kadang kala ada situasi dimana sebuah aplikasi diperlukan untuk menjalankan beberapa tugas yang serupa. Sebagai contohnya sebuah *web server* dapat mempunyai ratusan klien yang mengaksesnya

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

Java mempunyai pengunaan lain dari *thread*. Perlu diketahui bahwa Java tidak mempunyai konsep *asynchronous*. Sebagai contohnya kalau program java mencoba untuk melakukan koneksi ke server maka ia akan berada dalam keadaan block state sampai koneksinya jadi (dapat dibayangkan apa yang terjadi apabila servernya mati). Karena Java tidak memiliki konsep *asynchronous* maka solusinya adalah dengan membuat *thread* yang mencoba untuk melakukan koneksi ke server dan *thread* lain yang pertamanya tidur selamabeberap waktu (misalnya 60 detik) kemudian bangun. Ketika waktu tidurnya habis maka ia akan bangun dan memeriksa apakah *thread* yang melakukan koneksi ke server masih mencoba untuk melakukan koneksi ke server, kalau *thread* tersebut masih dalam keadaan mencoba untuk melakukan koneksi ke server maka ia akan melakukan interrupt dan mencegah *thread* tersebut untuk mencoba melakukan koneksi ke server.

11.2. Keuntungan *Thread*

Keuntungan dari program yang *multithreading* dapat dipisah menjadi empat kategori:

1. Responsi: Membuat aplikasi yang interaktif menjadi *multithreading* dapat membuat sebuah program terus berjalan meski pun sebagian dari program tersebut diblok atau melakukan operasi yang panjang, karena itu dapat meningkatkan respons kepada pengguna. Sebagai contohnya dalam *web browser* yang *multithreading*, sebuah *thread* dapat melayani permintaan pengguna sementara *thread* lain berusaha menampilkan image.

2. Berbagi sumber daya: *thread* berbagi memori dan sumber daya dengan *thread* lain yang dimiliki oleh proses yang sama. Keuntungan dari berbagi kode adalah mengizinkan sebuah aplikasi untuk mempunyai beberapa *thread* yang berbeda dalam lokasi memori yang sama.

3. Ekonomi: dalam pembuatan sebuah proses banyak dibutuhkan pengalokasian memori dan sumber daya. Alternatifnya adalah dengan penggunaan *thread*, karena *thread* berbagi memori dan sumber daya proses yang memilikinya maka akan lebih ekonomis untuk membuat dan *context switch thread*. Akan susah untuk mengukur perbedaan waktu antara proses dan *thread* dalam hal pembuatan dan pengaturan, tetapi secara umum pembuatan dan pengaturan proses lebih lama dibandingkan *thread*. Pada Solaris, pembuatan proses lebih lama 30 kali dibandingkan pembuatan *thread*, dan *context switch* proses 5 kali lebih lama dibandingkan *context switch thread*.

4. Utilisasi arsitektur *multiprocessor*: Keuntungan dari multithreading dapat sangat meningkat pada arsitektur *multiprocessor*, dimana setiap *thread* dapat berjalan secara pararel di atas processor yang berbeda. Pada arsitektur processor tunggal, CPU menjalankan setiap *thread* secara bergantian tetapi hal ini berlangsung sangat cepat sehingga menciptakan ilusi pararel, tetapi pada kenyataannya hanya satu *thread* yang dijalankan CPU pada satu-satuan waktu (satu-satuan waktu pada CPU biasa disebut *time slice* atau *quantum*).

11.3. *User* dan *Kernel Threads*

User Thread

User *thread* didukung di atas kernel dan diimplementasi oleh *thread* library pada user level. *Library* menyediakan fasilitas untuk pembuatan *thread*, penjadualan *thread*, dan managemen *thread* tanpa dukungan dari kernel. Karena kernel tidak menyadari user-level *thread* maka semua pembuatan dan penjadualan *thread* dilakukan di user space tanpa intervensi dari kernel. Oleh karena itu, user-level *thread* biasanya cepat untuk dibuat dan diatur. Tetapi user *thread* mempunyai kelemahan yaitu apabila kernelnya merupakan *thread* tunggal maka apabila salah satu user-level *thread* menjalankan *blocking system call* maka akan mengakibatkan seluruh proses diblok walaupun ada *thread* lain

yang dapat jalan dalam aplikasi tersebut. Contoh *user-thread libraries* adalah POSIX Pthreads, Mach C-threads, dan Solaris threads.

Kernel Thread

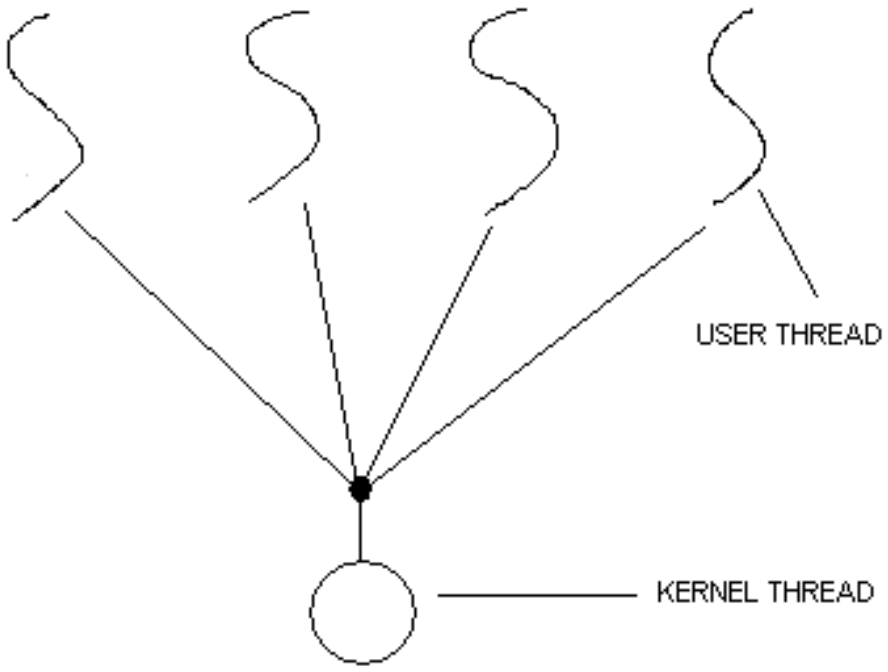
Kernel *thread* didukung langsung oleh sistem operasi. Pembuatan, penjadualan, dan managemen *thread* dilakukan oleh kernel pada *kernel space*. Karena pengaturan *thread* dilakukan oleh sistem operasi maka pembuatan dan pengaturan kernel *thread* lebih lambat dibandingkan user *thread*. Keuntungannya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

11.4. *Multithreading Models*

Many-to-One Model

Many-to-One model memetakan banyak user-level *thread* ke satu kernel *thread*. Pengaturan *thread* dilakukan di *user space*, oleh karena itu ia efisien tetapi ia mempunyai kelemahan yang sama dengan user *thread*. Selain itu karena hanya satu *thread* yang dapat mengakses *thread* pada suatu waktu maka *multiple thread* tidak dapat berjalan secara pararel pada *multiprocessor*. User-level *thread* yang diimplementasi pada sistem operasi yang tidak mendukung kernel *thread* menggunakan Many-to-One model.

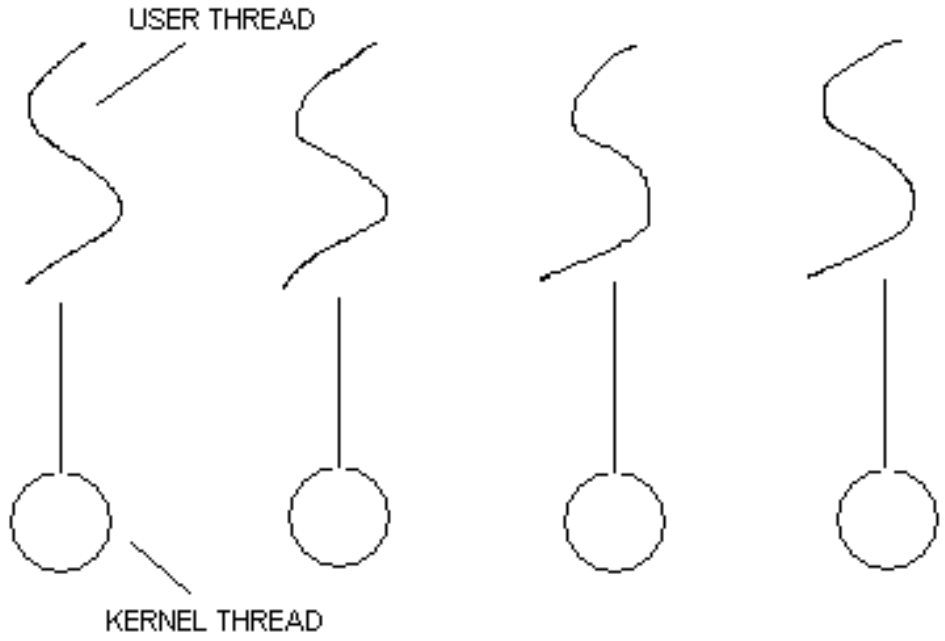
Gambar 11.2. ***Many-To-One***



One-to-One Model

One-to-One model memetakan setiap user *thread* ke kernel *thread*. Ia menyediakan lebih banyak *concurrency* dibandingkan Many-to-One model. Keuntungannya sama dengan keuntungan kernel *thread*. Kelemahannya model ini adalah setiap pembuatan user *thread* membutuhkan pembuatan kernel *thread*. Karena pembuatan *thread* dapat menurunkan performa dari sebuah aplikasi maka implmentasi dari model ini membatasi jumlah *thread* yang dibatasi oleh sistem. Contoh sistem operasi yang mendukung One-to-One model adalah Windows NT dan OS/2.

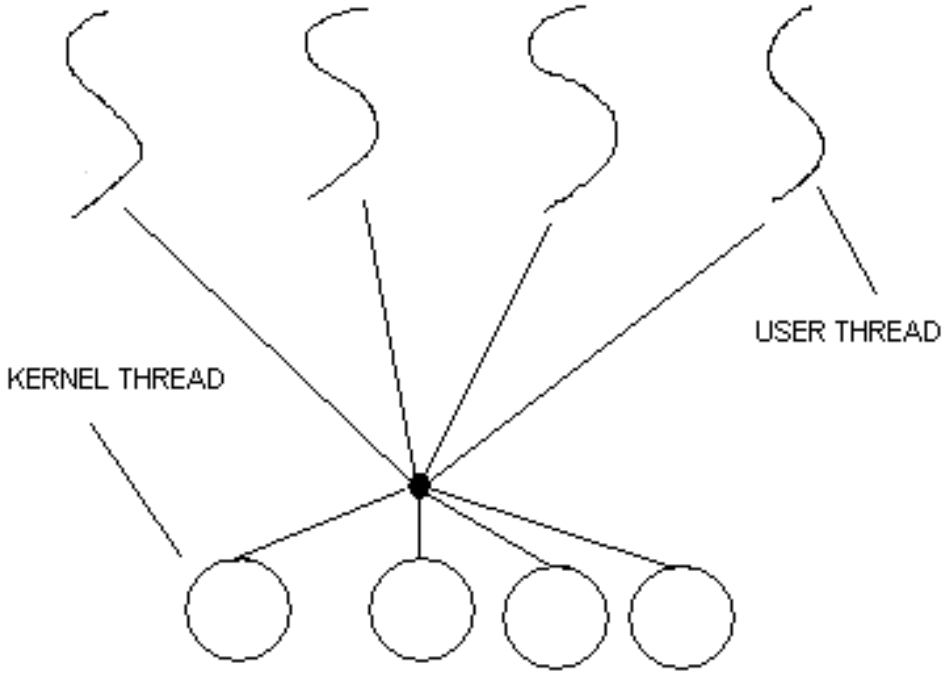
Gambar 11.3. *One-To-One*



Many-to-Many Model

Many-to-many model *multiplexes* banyak *user-level thread* ke kernel *thread* yang jumlahnya lebih kecil atau sama banyaknya dengan user-level *thread*. Jumlah kernel *thread* dapat spesifik untuk sebagian aplikasi atau sebagian mesin. Many-to-One model mengizinkan developer ntuk membuat user *thread* sebanyak yang ia mau tetapi *concurrency* tidak dapat diperoleh karena hanya satu *thread* yang dapat dijadual oleh kernel pada suatu waktu. One-to-One menghasilkan concurrency yang lebih tetapi developer harus hati-hati untuk tidak menciptakan terlalu banyak *thread* dalam suatu aplikasi (dalam beberapa hal, developer hanya dapat membuat *thread* dalam jumlah yang terbatas). Many-to-Many model tidak menderita kelemahan dari 2 model di atas. Developer dapat membuat user *thread* sebanyak yang diperlukan, dan kernel *thread* yang bersangkutan dapat bejalan secara pararel pada *multiprocessor*. Dan juga ketika suatu *thread* menjalankan *blocking system call* maka kernel dapat menjadualkan *thread* lain untuk melakukan eksekusi. Contoh sistem operasi yang mendukung model ini adalah Solaris, IRIX, dan Digital UNIX.

Gambar 11.4. ***Many-To-Many***



11.5. *Fork* dan *Exec System Call*

Ada dua kemungkinan dalam system UNIX jika *fork* dipanggil oleh salah satu *thread* dalam proses:

1. Semua *thread* diduplikasi.
2. Hanya *thread* yang memanggil *fork*.

Kalau *thread* memanggil *exec System Call* maka program yang dispesifikasi di parameter exec akan mengganti keseluruhan proses termasuk *thread* dan LWP.

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

11.6. *Cancellation*

Thread cancellation adalah tugas untuk memberhentikan *thread* sebelum ia menyelesaikan tugasnya. Sebagi contohnya jika dalam program java kita hendak mematikan *Java Virtual Machine* (JVM) maka sebelum JVM-nya dimatikan maka seluruh *thread* yang berjalan dihentikan terlebuh dahulu. Thread yang akan diberhentikan biasa disebut target *thread*.

Pemberhentian target *thread* dapat terjadi melalui dua cara yang berbeda:

1. *Asynchronous cancellation*: suatu *thread* seketika itu juga memberhentikan target *thread*.

2. *Defered cancellation*: target *thread* secara perodik memeriksa apakah dia harus berhenti, cara ini memperbolehkan target *thread* untuk memberhentikan dirinya sendiri secara terurut.

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

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

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

11.7. Penanganan Sinyal

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

Semua sinyal (*asynchronous* dan *synchronous*) mengikuti pola yang sama:

1. Sebuah sinyal dimunculkan oleh kejadian dari suatu persitiwa.

2. Sinyal yang dimunculkan tersebut dikirim ke proses.

3. Sesudah dikirim, sinyal tersebut harus ditangani.

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

Setiap sinyal dapat ditangani oleh salah satu dari dua penerima sinyal:

1. Penerima sinyal yang merupakan set awal dari sistem operasi.

2. Penerima sinyal yang didefinisikan sendiri ole user.

Penanganan sinyal pada program yang hanya memakai *thread* tunggal cukup mudah yaitu hanya dengan mengirimkan sinyal ke prosesnya. Tetapi mengirimkan sinyal lebih rumit pada program yang *multithreading*, karena sebuah proses dapat memiliki beberapa *thread*.

Secara umum ada empat pilihan kemana sinyal harus dikirim:

1. Mengirimkan sinyal ke *thread* yang dituju oleh sinyal tersebut.

2. Mengirimkan sinyal ke setiap *thread* pada proses tersebut.

3. Mengirimkan sinyal ke *thread* tertentu dalam proses.

4. Menugaskan *thread* khusus untuk menerima semua sinyal yang ditujukan pada proses.

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

11.8. *Thread Pools*

Pada *web server* yang *multithreading* ada dua masalah yang timbul:

1. Ukuran waktu yang diperlukan untuk menciptakan *thread* untuk melayani permintaan yang diajukan terlebih pada kenyataannya *thread* dibuang ketika ia seketika sesudah ia menyelesaikan tugasnya.

2. Pembuatan *thread* yang tidak terbatas jumlahnya dapat menurunkan performa dari sistem.

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

Keuntungan *thread* pool:

1. Biasanya lebih cepat untuk melayani permintaan dengan *thread* yang ada dibanding dengan menunggu *thread* baru dibuat.

2. Thread pool membatasi jumlah *thread* yang ada pada suatu waktu. Hal ini pentingpada sistem yang tidak dapat mendukung banyak *thread* yang berjalan secara *concurrent*.

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

11.9. *Thread Specific Data*

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

11.10. *Pthreads*

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

11.11. Rangkuman

Thread adalah sebuah alur kontrol dari sebuah proses. Suatu proses yang multithreaded mengandung beberapa perbedaan alur kontrol dengan ruang alamat yang sama. Keuntungan dari multithreaded meliputi peningkatan respon dari pengguna, pembagian sumber daya proses, ekonomis, dan kemampuan untuk mengambil keuntungan dari arsitektur multiprosesor. *Thread* tingkat pengguna adalah thread yang tampak oleh programer dan tidak diketahui oleh kernel. *Thread* tingkat pengguna secara tipikal dikelola oleh sebuah library *thread* di ruang pengguna. *Thread* tingkat *kernel* didukung dan dikelola oleh *kernel* sistem operasi. Secara umum, *thread* tingkat pengguna lebih cepat dalam pembuatan dan pengelolaan dari pada *kernel thread*. Ada 3 perbedaan tipe dari model yang berhubungan dengan pengguna dan *kernel thread* yaitu one-to one model, many-to-one model, many-to-many model.

- Model many to one: memetakan beberapa pengguna level *thread* hanya ke satu buah *kernel thread*.
- Model one to one: memetakan setiap *thread* pengguna ke dalam satu *kernel thread* berakhir.
- Model many to many: mengijinkan pengembang untuk membuat *thread* pengguna sebanyak mungkin, konkurensi tidak dapat tercapai karena hanya satu *thread* yang dapat dijadualkan oleh kernel dalam satu waktu.

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

Pemberhentian target *thread* dapat terjadi melalui 2 cara yang berbeda

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

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

Keuntungan *Thread Pools*

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

11.12. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:

   - *Multithread Model: "One to One" vs. "Many to Many"*.
   - *Scheduling Process: "Short Term" vs. "Long Term"*.
   - *Scheduling Algorithm: "FCFS (First Come First Serve)" vs. "SJF (Shortest Job First)"* .
   - *"Preemptive Shortest Job First" vs. "Non-preemptive Shortest Job First"*.

2. Tunjukkan dua contoh pemrograman dari multithreading yang dapat meningkatkan sebuah solusi thread tunggal.

3. Tunjukkan dua contoh pemrograman dari multithreading yang tidak dapat meningkatkan sebuah solusi thread tunggal.

4. Sebutkan dua perbedaan antara user level thread dan kernel thread. Saat kondisi bagaimana salah satu dari thread tersebut lebih baik

5. Jelaskan tindakan yang diambil oleh sebuah kernel saat context switch antara kernel level thread.

6. Sumber daya apa sajakah yang digunakan ketika sebuah thread dibuat? Apa yang membedakannya dengan pembentukan sebuah proses.

7. Tunjukkan tindakan yang diambil oleh sebuah thread library saat context switch antara user level thread.

8. Definisikan perbedaan antara penjadualan secara preemptive dan nonpreemptive!

9. Jelaskan mengapa penjadualan strict nonpreemptive tidak seperti yang digunakan di sebuah komputer pusat.

11.13. Rujukan

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000. John Wiley & Sons, Inc.

William Stallings: Operating Systems -- Fourth Edition, Prentice Hall, 2001.

Bab 12. *Thread* Java

12.1. Pendahuluan

Dewasa ini (2005), banyak sistem operasi yang telah mendukung proses *multithreading*. Setiap sistem operasi memiliki konsep tersendiri dalam mengimplementasikannya ke dalam sistem. Sistem operasi mendukung *thread* pada tingkat *kernel* atau tingkat pengguna. Java merupakan salah satu dari sedikit bahasa pemrograman yang mendukung *thread* di tingkat bahasa untuk pembuatan dan managemen *thread*. Karena *thread* dalam Java diatur oleh *Java Virtual Machine* (JVM), tidak dengan *user level library* atau pun *kernel*, sulit mengelompokkan *thread* di Java apakah di tingkat pengguna atau *kernel*.

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

12.2. Pembuatan *Thread*

Ada dua cara untuk membuat *thread* dalam Java. Pertama, *thread* dapat dibuat secara eksplisit dengan cara membuat obyek baru dari class yang telah meng-*extends class Thread* yang menyebabkan class tersebut mewarisi *method-method* dan *field* dari *class super*. Dalam kasus ini, sebuah *class* hanya dapat meng-*extends* sebuah class. Keterbatasan ini dapat diatasi dengan cara kedua yaitu meng-*implements interface Runnable*, yang merupakan cara yang paling sering digunakan untuk membuat *thread*, sehingga class tersebut dapat meng-*extends class* lain.

Contoh 12.1. Thread

```
public class TestThread1 {
    public static void main (String[] args) {
        BuatThread1 b = new BuatThread1();
        for(int i = 0; i < angka; i++) {
            b.start();
        }
    }
}

class BuatThread1 extends Thread {
    public void run() {
        try {
            System.out.println("Thread baru dibuat.");
        }
        catch (InterruptedException e) {
        }
    }
}
```

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

- Pengalokasian memori dan menginisialisasikan sebuah *thread* baru dalam JVM.

- Memanggil *method run*, yang sudah di-*override*, membuat *thread* dapat dijalankan oleh JVM.

Catatan: *Method run* dijalankan jika *method start* dipanggil. Memanggil *method run* secara langsung hanya menghasilkan sebuah *single-thread* tambahan selain *main thread*.

12.3. Status *Thread*

Thread pada Java dapat terdiri dari beberapa status, yaitu:

1. Baru

   Sebuah thread berstatus baru berarti thread tersebut adalah sebuah objek thread yang kosong , belum ada sumber daya sistem yang dialokasikan kepada thread tersebut, oleh karena itu method thread yang dapat dipanggil hanyalah start(), apabila dipanggil method thread yang lain akan menyebabkan IllegalThreadStateException.

2. Dapat Dijalankan

   Sebuah thread dapat memasuki status dapat dijalankan apabila method thread start() telah dijalankan. Method start() mengalokasikan sumber daya sistem yang dibutuhkan oleh thread, menentukan penjadwalan thread tersebut, serta menjalankan method run(), akan tetapi thread tersebut benar-benar berjalan apabila sesuai dengan jadwalnya.

3. Tidak Dapat Dijalankan

   Sebuah thread memasuki status tidak dapat dijalankan apabila:

   - Method sleep() dari thread tersebut dijalankan.
   - Method wait() dari thread tersebut dijalankan.
   - Thread tersebut tersangkut dalam proses tunggu M/K sumber daya sistem untuk melakukan operasi input atau pun output.

   Thread tersebut dapat berada pada status dapat dijalankan lagi apabila:

   - Waktu yang ditetapkan oleh method sleep() telah berlalu.
   - Kondisi menunggu dari thread tersebut telah berubah dan telah menerima pesan notify() ataupun notifyAll().
   - Sumber daya sistem yang dibutuhkan telah tersedia dan proses I/O nya telah selesai dilakukan.

   Perubahan status sebuah thread antara dapat dijalakan dan tidak dapat dijalankan dapat dijelaskan secara garis besar sebagai akibat dari penjadwalan ataupun programmer control.

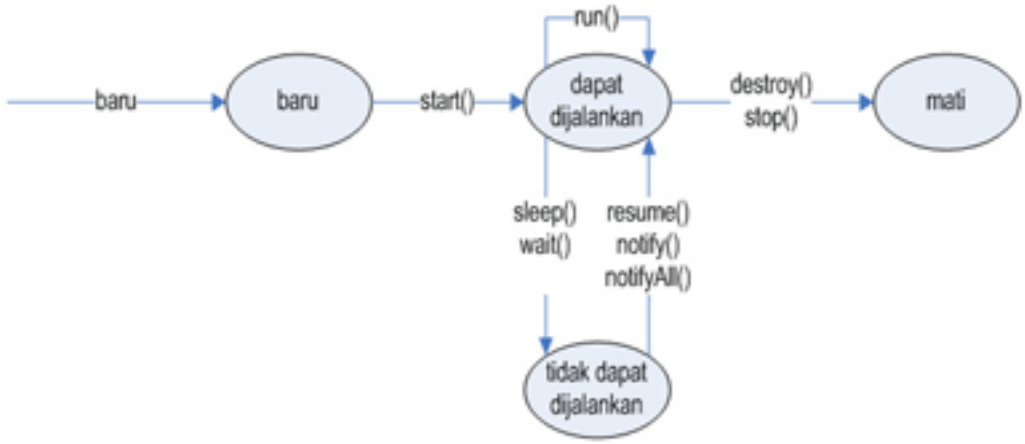
4. Mati

   Sebuah thread berada dalam status mati apabila:

   - Method run() telah selesai dieksekusi.
   - Method destroy() dijalankan, namun method ini tidak membuat thread tersebut melepaskan objek-objeknya yang telah dikunci.

Status-status ini dapat dilihat pada gambar berikut ini

Gambar 12.1. P3



Sumber: Silberschatz

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

12.4. Penggabungan *Thread*

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

Keuntungan yang dapat diperoleh dari penggunaan thread yaitu kita dapat memadukan antara keamanan dan kenyamanan. Dikatakan aman karena threads dalam sebuah groups tidak dapat mengakses parent threads dari group tersebut. Hal ini mengakibatkan suatu thread dapat diisolasi dan dapat mencegah thread yang ada dalam sebuah group dari saling mengubah satu sama lainnya.

Pada java untuk mengorganisasikan thread kedalam bentuk groups, diwakili dengan ThreadGroup class. Sebuah ThreadGroup terdiri dari beberapa individual treads, atau thread groups yang lain, untuk membentuk sebuah thread hirarcy. Hirarcy yang terbentuk yaitu parent dan children.

Contoh 12.2. XXXX

```
public class AllThreads {
     public static void main(String[] args) {
     ThreadGroup top = Thread.currentThread().getThreadGroup();
     while(true)
     {
       if (top.getParent() != null) top = top.getParent();
       else break;
     }
     Thread[] theThreads = new Thread[top.activeCount()];
     top.enumerate(theThreads);
     for (int i = 0; i < theThreads.length; i++)
     {
         System.out.println(theThreads[i]);
     }
}
```

Pada program java untuk jika kita ingin mencetak seluruh thread yang ada maka dapat digunakan method getThreadGroup() dari java.lang.Thread. jika kita ingin melihat level paling atas pada hirarcy dari sebua Threads groups maka kita dapat menggunakan method getParent() dari java.lang.ThreadGroup. kita juga dapat mendaftar seluruh thread yang ada di thread utama beserta childrennya dengan menggunakan enumerate.

Keluaran dari program diatas:

Gambar 12.2. Program ...

```
Thread[clock handler,11,system]
Thread[idle thread,0,system]
Thread[Async Garbage Collector,1,system]
Thread[Finalizer thread,1,system]
Thread[main,1,main]
Thread[Thread-0,5,main]

Thread[Finalizer thread,1,system]
Thread[main,1,main]
Thread[Thread-0,5,main]
```

12.5. Terminasi *Thread*

FIXME

12.6. JVM dan *Host Operating System*

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

12.7. Solusi *Multi-Threading* (FM)

FIXME

12.8. Rangkuman

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

Java adalah unik karena telah mendukung *thread* didalam tingkatan bahasanya. Semua program Java sedikitnya terdiri dari kontrol sebuah *thread* tunggal dan mempermudah membuat kontrol untuk *multiple thread* dengan program yang sama. JAVA juga menyediakan *library* berupa API

untuk membuat thread, termasuk method untuk suspend dan resume suatu thread, agar *thread* tidur untuk jangka waktu tertentu dan menghentikan *thread* yang berjalan. Sebuah java *thread* juga mempunyai empat kemungkinan keadaan, diantaranya: *New*, *Runnable*, *Blocked* dan *Dead*. Perbedaan API untuk mengelola *thread* seringkali mengganti keadaan *thread* itu sendiri.

12.9. Latihan

1. FIXME

12.10. Rujukan

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000. John Wiley & Sons, Inc.

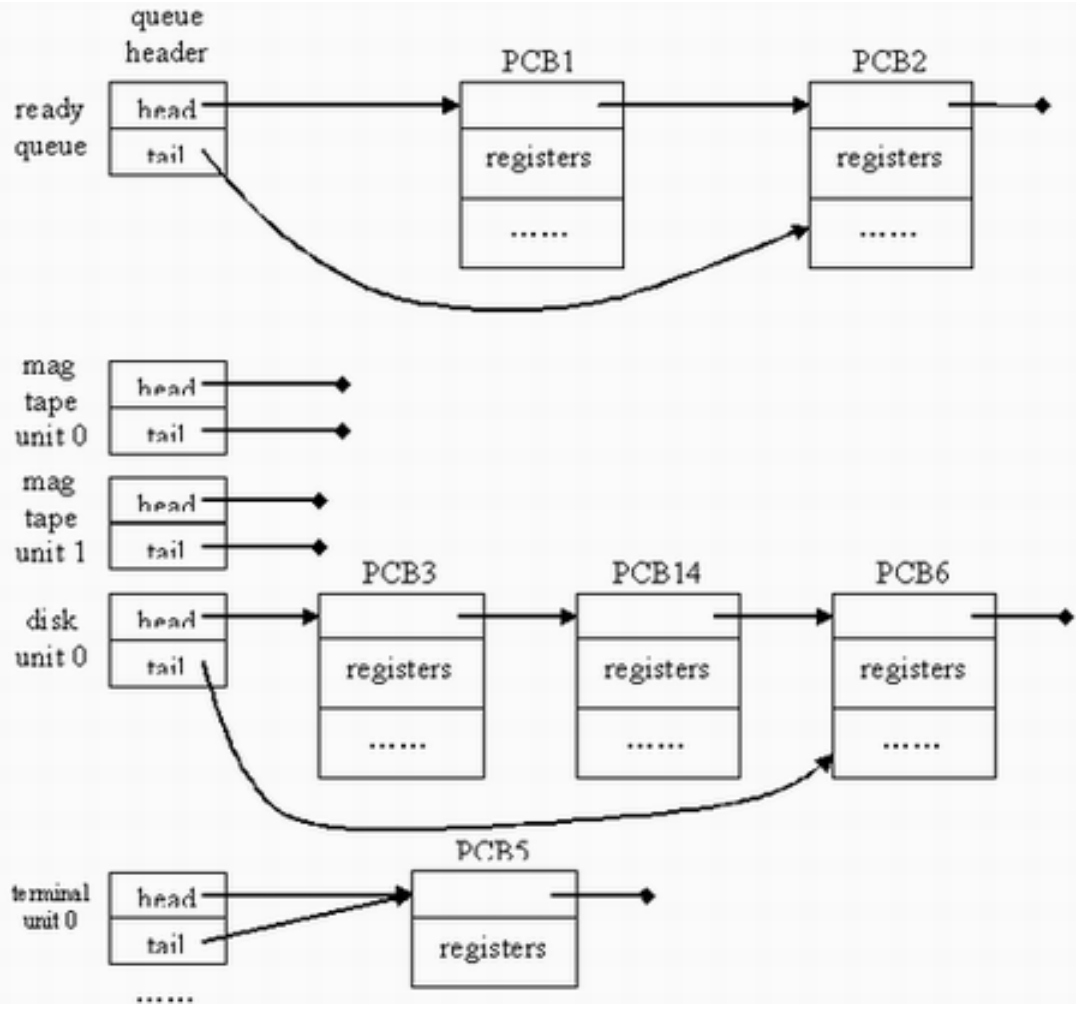
William Stallings: Operating Systems -- Fourth Edition, Prentice Hall, 2001.

Bab 13. Konsep Penjadualan

Multiprogramming bertujuan untuk memaksimalkan penggunaan CPU dengan cara mengatur alokasi waktu yang digunakan oleh CPU, sehingga proses berjalan sepanjang waktu dan memperkecil waktu *idle*. Untuk sistem yang bersifat prosesor tunggal (*uniprosesor*), hanya ada satu proses yang dapat berjalan setiap waktunya. Jika proses yang ada lebih dari satu, maka proses yang lain harus menunggu sampai CPU bebas dan siap untuk dijadualkan kembali.

13.1. *Queue Scheduling*

Gambar 13.1. ***Device Queue***



Sumber:...

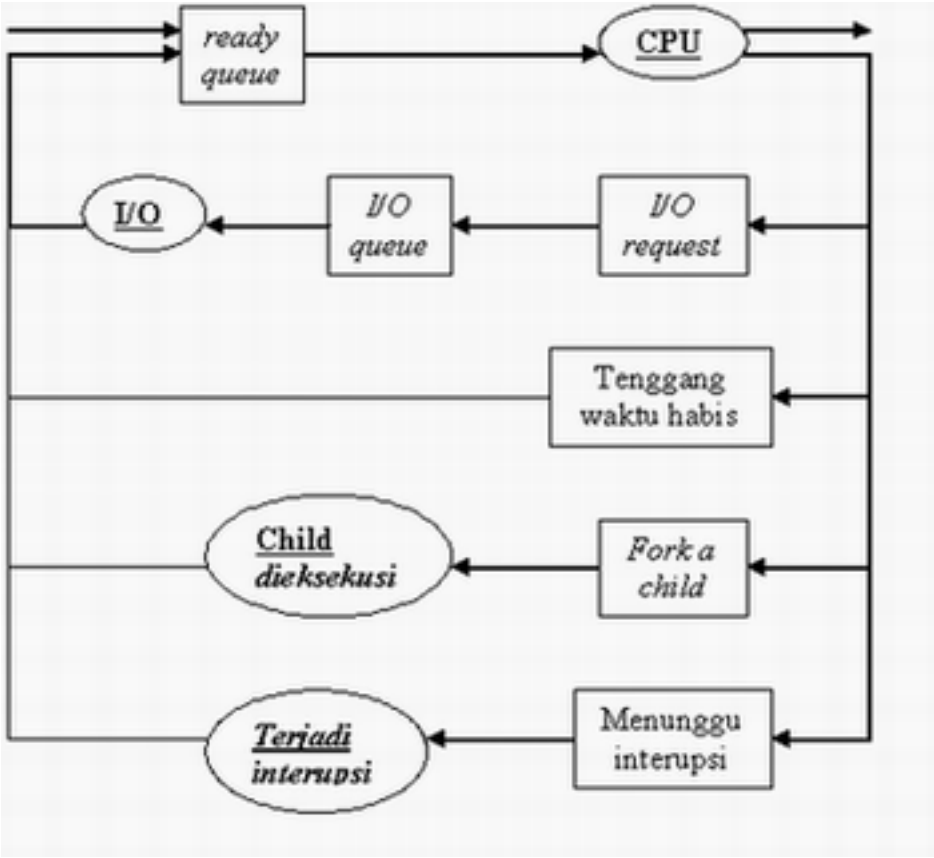
Ketika sebuah proses memasuki sistem, proses itu diletakkan di dalam ***job queue***. Pada antrian ini terdapat seluruh proses yang berada dalam sistem. Sedangkan proses yang berada pada memori utama, siap dan menunggu untuk mengeksekusi disimpan dalam sebuah daftar yang bernama ***ready queue***. Antrian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

Sistem operasi juga memiliki antrian lain. Ketika sebuah proses dialokasikan ke CPU, proses tersebut berjalan sebentar lalu berhenti, di-interupsi, atau menunggu suatu hal tertentu, seperti

selesainya suatu permintaan M/K. Dalam permintaan M/K, dapat saja yang diminta itu adalah *tape drive*, atau peralatan yang di-*share* secara bersama-sama, seperti disk. Karena ada banyak proses dalam sistem, disk dapat saja sibuk dengan permintaan M/K dari proses lainnya. Untuk itu proses tersebut mungkin harus menunggu disk tersebut. Daftar dari proses-proses yang menunggu peralatan M/K tertentu disebut dengan ***device queue***. Tiap peralatan memiliki *device queue*-nya masing-masing (Gambar 13.1, "*Device Queue*").

Penjadualan proses dapat direpresentasikan secara umum dalam bentuk diagram antrian, seperti yang ditunjukkan oleh Gambar 13.2, "Diagram Antrian". Setiap kotak segi empat menunjukkan sebuah antrian. Dua tipe antrian menunjukan antrian yang siap dan seperangkat *device queues*. Lingkaran menunjukkan sumber daya yang melayani antrian, dan tanda panah mengindikasikan alur dari proses-proses yang ada dalam sistem.

Gambar 13.2. Diagram Antrian



...

Sebuah proses baru pertama-tama diletakkan dalam *ready queue*. Proses tersebut menunggu di dalam *ready* antrian sampai dia dipilih untuk eksekusi, atau dengan kata lain di-*dispatched*. Begitu proses tersebut dialokasikan ke CPU dan sedang berjalan, beberapa kemungkinan di bawah ini dapat terjadi:

- Proses tersebut mengeluarkan permintaan M/K, lalu ditempatkan dalam sebuah antrian M/K.
- Proses tersebut dapat membuat sub proses yang baru dan menunggu untuk di-terminasi.
- Proses tersebut dapat dikeluarkan secara paksa dari CPU, sebagai hasil dari suatu interupsi, dan diletakkan kembali dalam *ready queue*.

Pada dua kemungkinan pertama (proses meminta M/K atau membuat sub proses baru), proses berganti keadaan dari *waiting state* menjadi *ready state*, lalu diletakkan kembali dalam *ready queue*. Proses akan meneruskan siklus ini sampai dia di-terminasi, yaitu saat dimana proses tersebut dikeluarkan dari seluruh antrian yang ada dan memiliki PCB-nya sendiri dan seluruh sumber daya yang dia gunakan dialokasikan kembali.

13.2. *Scheduler*

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

Dalam sistem *batch*, seringkali proses yang diserahkan lebih banyak daripada yang dapat dilaksanakan dengan segera. Proses-proses ini disimpan pada suatu *mass-storage device* (disk), dimana proses tersebut disimpan untuk eksekusi di lain waktu. ***Long-term scheduler***, atau ***job scheduler***, memilih proses dari tempat ini dan mengisinya ke dalam memori. Sedangkan ***short-term scheduler***, atau ***CPU scheduler***, hanya memilih proses yang sudah siap untuk melakukan eksekusi, dan mengalokasikan CPU untuk proses tersebut.

Hal yang cukup jelas untuk membedakan kedua jenis *scheduler* ini adalah frekuensi dari eksekusinya. *Short-term scheduler* harus memilih proses baru untuk CPU sesering mungkin. Sebuah proses dapat mengeksekusi hanya dalam beberapa milidetik sebelum menunggu permintaan M/K. Seringkali, *short-term scheduler* mengeksekusi paling sedikit sekali setiap 100 milidetik. Karena durasi waktu yang pendek antara eksekusi-eksekusi tersebut, *short-term scheduler* seharusnya cepat. Jika memerlukan waktu 10 mili detik untuk menentukan suatu proses eksekusi selama 100 mili detik, maka 10/(100 + 10) = 9 persen dari CPU sedang digunakan (atau terbuang) hanya untuk pekerjaan penjadualan.

Long-term scheduler, pada sisi lain, mengeksekusi jauh lebih jarang. Mungkin ada beberapa menit waktu yang dibutuhkan untuk pembuatan proses baru dalam sistem. *Long-term scheduler* mengontrol *degree of multiprogramming* (jumlah proses dalam memori). Jika *degree of multiprogramming* stabil, maka tingkat rata-rata penciptaan proses harus sama dengan tingkat rata rata proses meninggalkan sistem. Maka dari itu *long-term scheduler* mungkin dipanggil hanya ketika suatu proses meninggalkan sistem. Karena interval yang lebih panjang antara eksekusi, *long-term scheduler* dapat menggunakan waktu yang lebih lama untuk menentukan proses mana yang harus dipilih untuk dieksekusi.

Sangat penting bagi *long-term scheduler* membuat seleksi yang hati-hati. Secara umum, proses dapat dibedakan atas dua macam, yaitu proses ***I/O bound*** dan proses ***CPU bound***. Proses *I/O bound* adalah proses yang lebih banyak menghabiskan waktunya untuk mengerjakan M/K dari pada melakukan komputasi. Proses *CPU-bound*, di sisi lain, jarang melakukan permintaan M/K, dan menggunakan lebih banyak waktunya untuk melakukan komputasi. Oleh karena itu, *long-term scheduler* harus memilih gabungan proses yang baik antara proses *I/O bound* dan *CPU bound*. Jika seluruh proses adalah *I/O bound*, *ready queue* akan hampir selalu kosong, dan *short-term scheduler* akan memiliki sedikit tugas. Jika seluruh proses adalah *CPU bound*, *I/O waiting queue* akan hampir selalu kosong, peralatan akan tidak terpakai, dan sistem akan menjadi tidak seimbang. Sistem dengan kinerja yang terbaik akan memiliki kombinasi yang baik antara proses *CPU bound* dan *I/O bound*.

Pada sebagian sistem, *long-term scheduler* dapat jadi tidak ada atau kerjanya sangat minim. Sebagai contoh, sistem *time-sharing* seperti UNIX sering kali tidak memiliki *long-term scheduler*. Stabilitas sistem-sistem seperti ini bergantung pada keterbatasan fisik (seperti jumlah terminal yang ada) atau pada penyesuaian sendiri secara alamiah oleh manusia sebagai pengguna. Jika kinerja menurun pada tingkat yang tidak dapat diterima, sebagian pengguna akan berhenti.

Sebagian sistem operasi, seperti sistem *time-sharing*, dapat memperkenalkan sebuah *scheduler* tambahan, yaitu ***medium-term scheduler***. *Scheduler* ini digambarkan pada Gambar 13.3, "*Medium-term Scheduler*". Ide utama atau kunci dari *scheduler* ini terkadang akan menguntungkan untuk memindahkan proses dari memori (dan dari pengisian aktif dari CPU), dan akibatnya *degree of multiprogramming* akan berkurang. Di kemudian waktu, proses dapat dibawa kembali dalam memori dan eksekusinya dapat dilanjutkan pada keadaan dimana proses itu dipindahkan tadi. Skema ini disebut *swapping*. Proses di-*swapped out* dan di-*swapped in* oleh *scheduler* ini. *Swapping*

mungkin diperlukan untuk meningkatkan mutu penggabungan proses, atau karena perubahan dalam kebutuhan memori yang mengakibatkan memori harus dibebaskan. *Swapping* dibahas dalam Bab 29, *Alokasi Memori*.

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

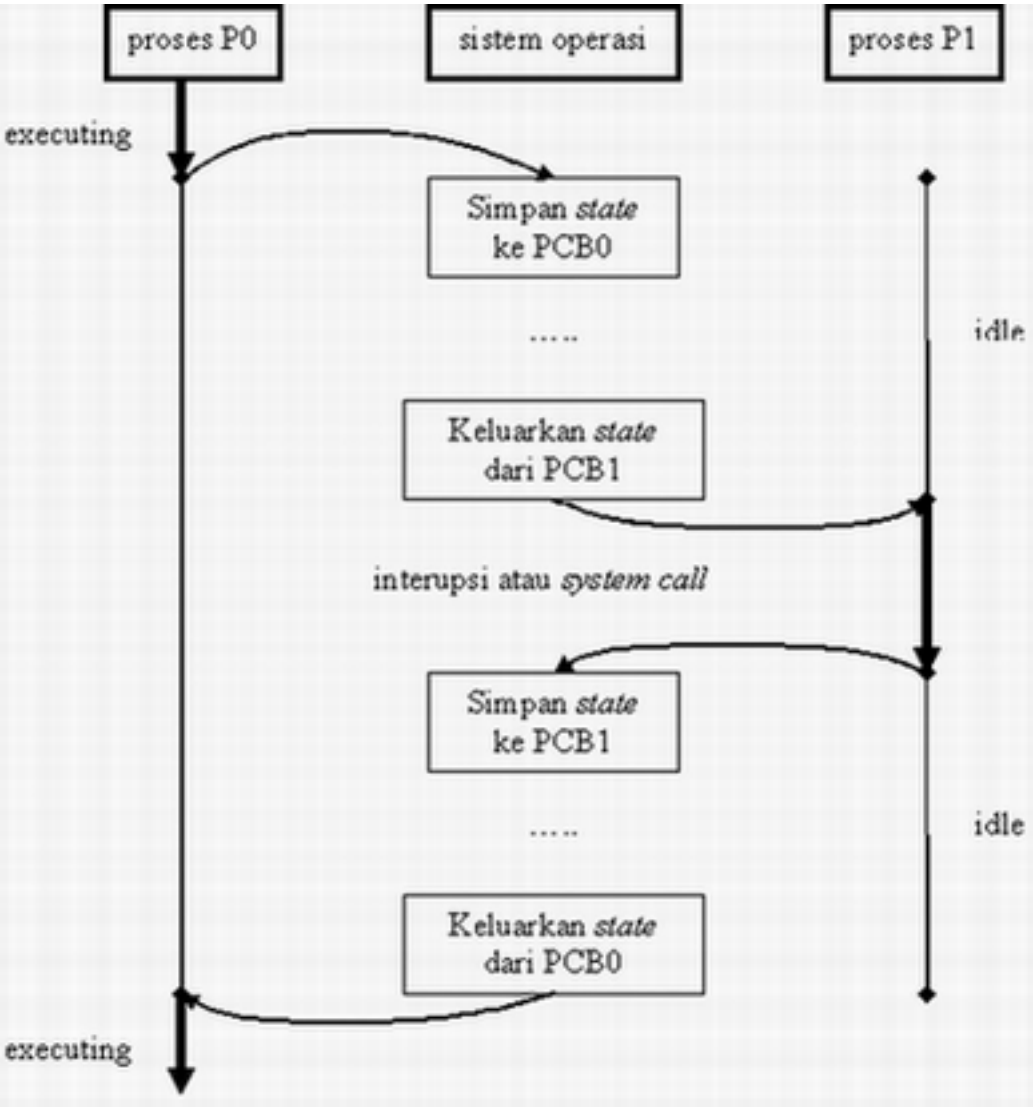
...

13.3. *Context Switch*

Mengganti CPU ke proses lain memerlukan penyimpanan keadaan dari proses lama dan mengambil keadaan dari proses yang baru. Hal ini dikenal dengan sebutan ***context switch***. *Context switch* sebuah proses direpresentasikan dalam PCB dari suatu proses; termasuk nilai dari CPU register, status proses (dapat dilihat pada Gambar 13.4, "*Context Switch*") dan informasi managemen memori. Ketika *context switch* terjadi, *kernel* menyimpan data dari proses lama ke dalam PCB nya dan mengambil data dari proses baru yang telah terjadual untuk berjalan. Waktu *context switch* adalah murni *overhead*, karena sistem melakukan pekerjaan yang tidak begitu berarti selama melakukan pengalihan. Kecepatannya bervariasi dari mesin ke mesin, bergantung pada kecepatan memori, jumlah register yang harus di-*copy*, dan ada tidaknya instruksi khusus (seperti instruksi tunggal untuk mengisi atau menyimpan seluruh register). Tingkat kecepatan umumnya berkisar antara 1 sampai 1000 mikro detik.

Waktu *context switch* sangat bergantung pada dukungan perangkat keras. Sebagai contoh, prosesor seperti UltraSPARC menyediakan beberapa set register. Sebuah proses *context switch* hanya memasukkan perubahan *pointer* ke set register yang ada saat itu. Tentu saja, jika proses aktif yang ada lebih banyak daripada proses yang ada pada set register, sistem menggunakan bantuan untuk meng-copy data register dari dan ke memori, sebagaimana sebelumnya. Semakin kompleks suatu sistem operasi, semakin banyak pekerjaan yang harus dilakukan selama *context switch*. Dapat dilihat pada Bagian V, "Memori", teknik managemen memori tingkat lanjut dapat mensyaratkan data tambahan untuk diganti dengan tiap data. Sebagai contoh, ruang alamat dari proses yang ada saat itu harus dijaga sebagai ruang alamat untuk proses yang akan dikerjakan berikutnya. Bagaimana ruang alamat dijaga, berapa banyak pekerjaan dibutuhkan untuk menjaganya, tergantung pada metode managemen memori dari sistem operasi. Akan kita lihat pada Bagian V, "Memori", *context switch* terkadang dapat menyebabkan *bottleneck*, dan programer menggunakan struktur baru (*threads*) untuk menghindarinya kapan pun memungkinkan.

Gambar 13.4. ***Context Switch***



...

13.4. Rangkuman

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

13.5. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:

- *Process Bound: "I/O Bound" vs. "CPU Bound".*
- *"Process State" vs. "Process Control Block".*
- *"Waiting Time" vs. "Response Time".*
- *Process Type: "Lightweight" vs. "Heavyweight".*

13.6. Rujukan

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000. John Wiley & Sons, Inc.

William Stallings: Operating Systems -- Fourth Edition, Prentice Hall, 2001.

Bab 14. Penjadual CPU

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

14.1. Konsep Dasar

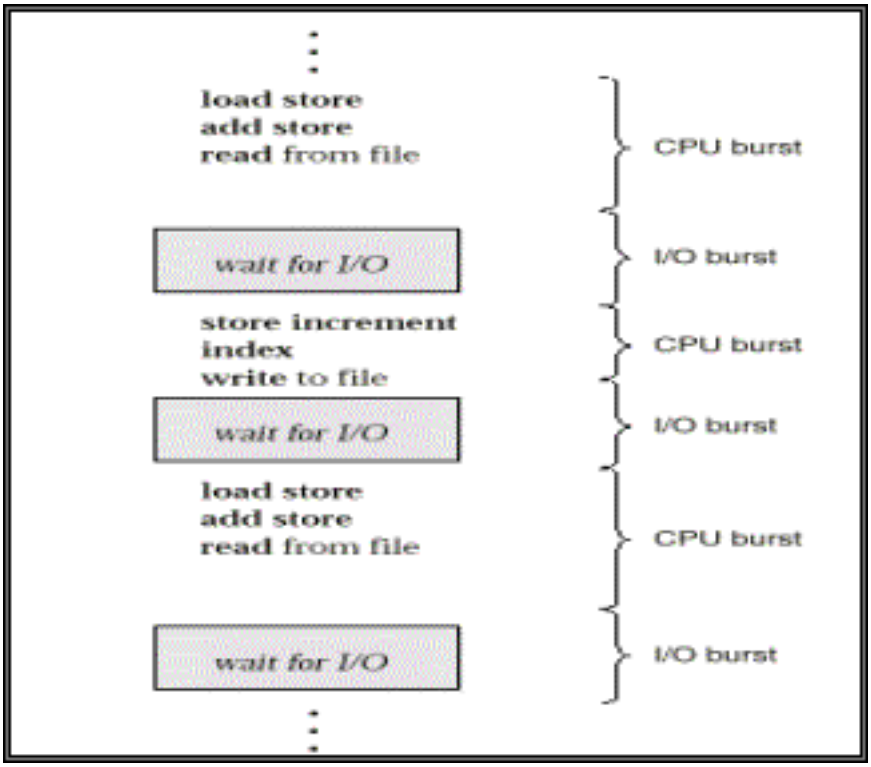
Tujuan dari multi-programming adalah untuk mempunyai proses berjalan secara bersamaan, untuk memaksimalkan kinerja dari CPU. Pada sistem prosesor tunggal, tidak pernah ada proses yang berjalan lebih dari satu. Bila ada proses yang lebih dari satu maka proses yang lain harus mengantri sampai CPU bebas proses.

Ide dari multi-progamming sangat sederhana. Ketika sebuah proses dieksekusi maka proses yang lain harus menunggu sampai proses pertama selesai. Pada sistem komputer yang sederhana CPU akan banyak dalam posisi idle. Sehingga waktu CPU ini sangat terbuang. Akan tetapi dengan multiprogamming, kita mencoba menggunakan waktu secara produktif. Beberapa proses di simpan di memori dalam satu waktu. Ketika suatu proses harus menuggu, Sistem operasi dapat saja akan menghentikan CPU dari suatu proses yang sedang diekseskusi dan memberikan sumberdaya kepada proses yang lainnya. Begitu seterusnya.

Penjadualan adalah fungsi dasar dari suatu sistem operasi. Hampir semua sumber komputer dijadualkan sebelum digunakan. CPU salah satu sumber dari komputer yang penting yang menjadi sentral dari sentral penjadualan di sistem operasi.

14.2. Siklus *Burst* CPU-M/K

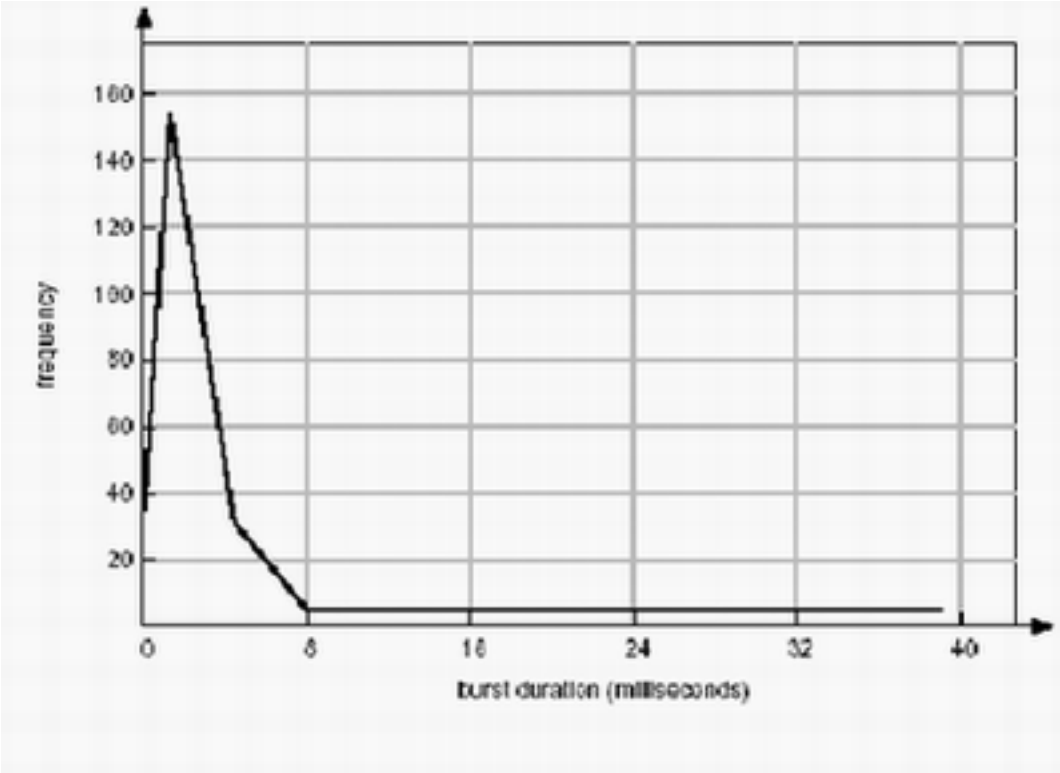
Gambar 14.1. Siklus *Burst*



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

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

Gambar 14.2. *Burst*



14.3. Penjadualan CPU

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

14.4. Penjadualan *Preemptive*

Penjadualan CPU mungkin akan dijalankan ketika proses:

1. Berubah dari running ke waiting state
2. Berubah dari running ke ready state

3. Berubah dari waiting ke ready

4. Terminates

Penjadualan dari no 1 sampai 4 non premptive sedangkan yang lain premptive. Dalam penjadualan nonpreemptive sekali CPU telah dialokasikan untuk sebuah proses, maka tidak dapat di ganggu, penjadualan model seperti ini digunakan oleh windows 3.X; windows 95 telah menggunakan penjadualan preemptive.

14.5. Penjadualan *Non-Preemptive*

Penjadualan non-preemptive terjadi ketika proses hanya:

1. berjalan dari running state sampai waiting state

2. dihentikan

Ini berarti cpu menjaga proses sampai proses itu pindah ke waiting state ataupun dihentikan (proses tidak diinterupt). Metode ini digunakan oleh Microsoft Windows 3.1 dan Macintosh. Ini adalah metode yang dapat digunakan untuk platforms hardware tertentu, karena tidak memerlukan perangkat keras khusus (misalnya timer yang digunakan untuk menginterupt pada metode penjadwalan preemptive).

14.6. *Dispatcher*

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

1. *Switching context*

2. *Switching to user mode*

3. Lompat dari suatu bagian di progam user untuk mengulang progam.

Dispatcher seharusnya secepat mungkin.

14.7. Kriteria Penjadualan

Algoritma penjadualan CPU yang berbeda mempunyai *property* yang berbeda. Dalam memilih algoritma yang digunakan untuk situasi tertentu, kita harus memikirkan properti yang berbeda untuk algoritma yang berbeda. Banyak kriteria yang dianjurkan untuk membandingkan penjadualan CPU algoritma.

Kritria yang biasanya digunakan dalam memilih adalah:

1. *CPU utilization*: kita ingin menjaga CPU sesibuk mungkin. CPU utilization akan mempunyai range dari 0 ke 100 persen. Di sistem yang sebenarnya seharusnya ia mempunyai range dari 40 persen samapi 90 persen

2. *Throughput*: jika CPU sibuk mengeksekusi proses, jika begitu kerja telah dilaksanakan. Salah satu ukuran kerja adalah banyak proses yang diselesaikan per unit waktu, disebut througput. Untuk proses yang lama mungkin satu proses per jam ; untuk proses yang sebentar mungkin 10 proses perdetik.

3. *Turnaround time*: dari sudur pandang proses tertentu, kriteria yang penting adalah berapa lama untuk mengeksekusi proses tersebut. Interval dari waktu yang dijinkan dengan waktu yang

dibutuhkan untuk menyelesaikan sebuah prose disebut *turn around time*. *Turn around time* adalah jumlah periode untuk menunggu untuk dapat ke memori, menunggu di ready queue, eksekusi di CPU, dan melakukan M/K

4. *Waiting time*: algoritma penjadualan CPU tidak mempengaruhi waktu untuk melaksanakan proses tersebut atau M/K; itu hanya mempengaruhi jumlah waktu yang dibutuhkan proses di antrian ready. Waiting time adalah jumlah periode menghabiskan di antrian ready.

5. *Response time*: di sistem yang interaktif, *turnaround time* mungkin bukan waktu yang terbaik untuk kriteria. Sering sebuah proses dapat memproduksi *output* diawal, dan dapat meneruskan hasil yang baru sementara hasil yang sebelumnya telah diberikan ke user. Ukuran yang lain adalah waktu dari pengiriamn permintaan sampai respon yang pertama di berikan. Ini disebut response time, yaitu waktu untuk memulai memberikan respon, tetapi bukan waktu yang dipakai *output* untu respon tersebut.

Biasanya yang dilakukan adalah memaksimalkan CPU *utilization* dan *throughput*, dan minimalkan *turnaround time*, *waiting time*, dan *response time* dalam kasus tertentu kita mengambil rata-rata.

14.8. Rangkuman

Penjadualan CPU adalah pemilihan proses dari antrian ready untuk dapat dieksekusi. Algoritma yang digunakan dalam penjadulan CPU ada bermacam-macam. Diantaranya adalah *First Come First Serve* (FCFS), merupakan algoritma sederhana dimana proses yang datang duluan maka dia yang dieksekusi pertama kalinya. Algoritma lainnya adalah *Shorthest Job First* (SJF), yaitu penjadulan CPU dimana proses yang paling pendek dieksekusi terlebih dahulu.

Kelemahan algoritma SJF adalah tidak dapat menghindari starvation. Untuk itu diciptakan algoritma Round Robin (RR). Penjadulan CPU dengan Round Robin adalah membagi proses berdasarkan waktu tertentu yaitu waktu quantum q. Setelah proses menjalankan eksekusi selama q satuan waktu maka akan digantikan oleh proses yang lain. Permasalahannya adalah bila waktu quantumnya besar sedang proses hanya membutuhkan waktu sedikit maka akan membuang waktu. Sedang bila waktu quantum kecil maka akan memakan waktu saat context-switch.

Penjadualan FCFS adalah nonpreemptive yaitu tidak dapat diinterupsi sebelum proses dieksekusi seluruhnya. Penjadualan RR adalah preemtive yaitu dapat dieksekusi saat prosesnya masih dieksekusi. Sedangkan penjadualan SJF dapat berupa nonpreemtive dan preemtive.

14.9. Latihan

1. Status Proses

   a) Gambarkan sebuah model bagan status proses (process state diagram) dengan minimum lima (5) status.

   b) Sebutkan serta terangkan semua nama status proses (process states) tersebut.

   c) Sebutkan serta terangkan semua nama kejadian (event) yang menyebabkan perubahan status proses.

   d) Terangkan perbedaan antara proses "I/O Bound" dengan proses "CPU Bound" berdasarkan bagan status proses tersebut.

2. Apa yang dimaksud dengan marshaling, jelaskan kegunaanya!

14.10. Rujukan

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000.

John Wiley & Sons, Inc.

William Stallings: Operating Systems -- Fourth Edition, Prentice Hall, 2001.

Bab 15. Algoritma Penjadualan I

Proses yang belum mendapat jatah alokasi dari CPU akan mengantri di ready queue. Di sini algoritma diperlukan untuk mengatur giliran proses-proses tersebut. Berikut ini adalah algoritmanya.

15.1. *First-Come, First-Served*

Algoritma ini merupakan algoritma yang paling sederhana. Dari namanya, kita dapat menebak kalau algoritma ini akan mendahulukan proses yang lebih dulu datang. Jadi proses akan mengantri sesuai waktu kedatangannya.

Kelemahan algoritma ini adalah *waiting time* rata-rata yang cukup lama. Sebagai contoh; ada tiga algoritma yang datang berturut-turut. Algoritma pertama mempunyai burst time 7 milidetik, sedangkan yang kedua dan ketiga masing-masing 5 milidetik dan 1 milidetik. *Waiting time* proses pertama ialah 0 milidetik (proses pertama tak perlu menunggu). *Waiting time* proses kedua ialah 7 milidetik (menunggu proses pertama), dan yang ketiga 12 milidetik (menunggu proses pertama dan kedua). Jadi *waiting time* rata-rata sebesar (0+7+12)/3 = 6,33 milidetik. Bandingkan jika proses datang dengan urutan terbalik (yang terakhir datang pertama dan kebalikannya). *Waiting time* rata-ratanya hanya sebesar (0+1+6)/3 = 2,33 milidetik (jauh lebih kecil). Bayangkan selisih yang mungkin terjadi jika beda *burst time* tiap proses sangat besar.

Munculah istilah *convoy effect*, dimana proses lain menunggu satu proses besar mengembalikan sumber daya CPU. Tentu kemungkinan utilisasi CPU akan lebih baik jika proses yang lebih singkat didahulukan.

Algoritma ini *nonpreemptive*. Setelah CPU dialokasikan ke suatu proses, hanya proses tersebut yang dapat mengembalikannya.

15.2. *Shortest-Job First*

Algoritma ini mempunyai cara yang berbeda untuk mengatur antrian di *ready queue*. Proses diatur menurut panjang *CPU burst* berikutnya (lebih tepatnya *shortest next CPU burst*).

Waiting time rata-rata dari algoritma ini sangat kecil, sehingga layak disebut optimal. Perbandingan algoritma ini dengan algoritma pertama telah kita lihat di bagian sebelumnya (*shortest job first*), di mana proses yang memiliki *CPU burst* terkecil jika didahulukan akan mengurangi *waiting time* rata-ratanya. Kelemahan algoritma ini yaitu kita tak pernah tahu secara pasti panjang *CPU burst* proses berikutnya. Kita hanya dapat mengira-ngira nilainya.

Algoritma ini dapat merupakan *preemptive* atau *nonpreemptive*. Jika preemptive, jika ada proses datang dengan sisa *CPU burst* yang lebih kecil daripada yang sedang dieksekusi, maka proses tersebut akan menggantikan proses yang sedang dieksekusi. Contoh: 2 proses datang bersamaan dengan *CPU burst* masing-masing sebesar 4 dan 5 milidetik. Algoritma ini akan mengalokasikan CPU untuk proses yang memiliki *CPU burst* 4 milidetik, sementara satu lagi akan menunggu di *ready queue*. Baru 1 milidetik proses pertama dieksekusi, ada proses lain datang dengan besar *CPU burst* 2 milidetik. Alokasi CPU segera diberikan pada proses baru tersebut karena mempunyai sisa waktu terkecil yaitu 2 milidetik(proses yang dieksekusi mempunyai sisa waktu 3 milidetik karena telah mendapat alokasi CPU selama 1 milidetik), dan kedua proses yang datang pertama kembali menunggu di *ready queue*. Bandingkan *waiting time* rata-ratanya, yang *nonpreemptive* sebesar (0+4+9)/3 = 4,33 milidetik, dan yang *preemptive* sebesar ((3-1)+6+(1-1))/3 = 2,66 milidetik.

15.3. *Priority*

Algoritma ini memberikan skala prioritas kepada tiap proses. Proses yang mendapat prioritas terbesar akan didahulukan. Skala diberikan dalam bentuk integer. Beberapa sistem menggunakan integer kecil untuk prioritas tertinggi, beberapa sistem menggunakan integer besar.

Algoritma ini dapat *preemptive* maupun *nonpreeemptive*. Jika *preemptive* maka proses dapat diinterupsi oleh proses yang prioritasnya lebih tinggi.

Kelemahan dari algoritma ini adalah proses dengan prioritas kecil tidak akan mendapat jatah CPU. Hal ini dapat diatasi dengan *aging*, yaitu semakin lama menunggu, prioritas semakin tinggi.

15.4. *Round-Robin*

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

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

15.5. *Multilevel Queue*

Algoritma ini mengelompokkan antrian dalam beberapa buah antrian. Antrian-antrian tersebut diberi prioritas.Antrian yang lebih rendah tak boleh mendapat alokasi selama ada antrian tinggi yang belum kebagian. Tiap antrian boleh memiliki algoritma yang berbeda. Kita juga dapat menjatah waktu CPU untuk tiap antrian. Semakin tinggi tingkatannya, semakin besar jatah waktu CPU-nya.

15.6. *Multilevel Feedback Queue*

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

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

Algoritma ini didefinisikan melalui beberapa parameter, antara lain:

- Jumlah antrian
- Algoritma penjadualan tiap antrian
- Kapan menaikkan proses ke antrian yang lebih tinggi
- Kapan menurunkan proses ke antrian yang lebih rendah
- Antrian mana yang akan dimasuki proses yang membutuhkan

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

15.7. Rangkuman

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

- First come first serve

  Algoritma ini mendahulukan proses yang lebih dulu datang. Kelemahannya,waiting time rata-rata cukup lama.

- Shortest job first

  Algoritma ini mendahulukan proses dengan CPU burst terkecil yang akan mengurangi waiting time rata-rata.

- Priority

  Algoritma ini mendahulukan prioritas terbesar. Kelemahannya, prioritas kecil tidak mendapat jatah CPU. Hal ini dapat diatasi dengan aging,yaitu semakin lama menunggu,prioritas semakin tinggi.

- Round Robin

  Algoritma ini menggilir proses-proses yang ada diantrian dengan jatah time quantum yang sama. Jika waktu habis,CPU dialokasikan keproses selanjutnya.

- Multilevel Queue

  Algoritma ini membagi beberapa antrian yang akan diberi prioritas berdasarkan tingkatan. Tingkatan lebih tinggi menjadi prioritas utama.

- Multilevel Feedback Queue

  Pada dasarnya sama dengan Multilevel Queue,bedanya pada algoritma ini diizinkan untuk pindah antrian.

15.8. Latihan

Tabel 15.1. Tabel untuk soal 4 - 5

Proses	Burst Time	Prioritas
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

1. Berikut merupakan sebagian dari keluaran hasil eksekusi perintah ``**top b n 1**" pada sebuah sistem GNU/Linux yaitu "**bunga.mhs.cs.ui.ac.id**" beberapa saat yang lalu.

```
15:34:14 up 28 days, 14:40, 53 users,  load average: 0.28, 0.31, 0.26
265 processes: 264 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:   5.9% user,   1.8% system,   0.1% nice,  92.2% idle
Mem:    126624K total,   113548K used,    13076K free,      680K buffers
Swap:   263160K total,    58136K used,   205024K free,    41220K cached

PID   USER  PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
    1 root    8   0   460  420   408 S     0.0  0.3   0:56 init
    2 root    9   0     0    0     0 SW    0.0  0.0   0:02 keventd
    3 root   19  19     0    0     0 SWN   0.0  0.0   0:02 ksoftirqd_CPU0
```

```
.....
17353 user1   9   0  2500 2004  2004 S     0.0  1.5   0:00 sshd
17354 user1   9   0  1716 1392  1392 S     0.0  1.0   0:00 bash
17355 user1   9   0  2840 2416  2332 S     0.0  1.9   0:00 pine
12851 user2   9   0  2500 2004  2004 S     0.0  1.5   0:00 sshd
12852 user2   9   0  1776 1436  1436 S     0.0  1.1   0:00 bash
13184 user2   9   0  1792 1076  1076 S     0.0  0.8   0:00 vi
13185 user2   9   0   392  316   316 S     0.0  0.2   0:00 grep
22272 user3   9   0  2604 2592  2292 S     0.0  2.0   0:00 sshd
22273 user3   9   0  1724 1724  1396 S     0.0  1.3   0:00 bash
22283 user3  14   0   980  980   660 R    20.4  0.7   0:00 top
19855 user4   9   0  2476 2048  1996 S     0.0  1.6   0:00 sshd
19856 user4   9   0  1700 1392  1392 S     0.0  1.0   0:00 bash
19858 user4   9   0  2780 2488  2352 S     0.0  1.9   0:00 pine
.....
```

a. Berapakah nomer Process Identification dari program "top" tersebut?

b. Siapakah yang mengeksekusi program "top" tersebut?

c. Sekitar jam berapakah, program tersebut dieksekusi?

d. Sudah berapa lama sistem GNU/Linux tersebut hidup/menyala?

e. Berapa pengguna yang sedang berada pada sistem tersebut?

f. Apakah yang dimaksud dengan "load average"?

g. Apakah yang dimaksud dengan proses "zombie" ?

2. Lima proses tiba secara bersamaan pada saat "t_0" (awal) dengan urutan P_1, P_2, P_3, P_4, dan P_5. Bandingkan (rata-rata) *turn-around time* dan *waiting time* dari ke lima proses tersebut di atas; jika mengimplementasikan algoritma penjadualan seperti FCFS (*First Come First Served*), SJF (*Shortest Job First*), dan RR (*Round Robin*) dengan kuantum 2 (dua) satuan waktu. Waktu *context switch* diabaikan.

   a. Burst time kelima proses tersebut berturut-turut (10, 8, 6, 4, 2) satuan waktu.

   b. Burst time kelima proses tersebut berturut-turut (2, 4, 6, 8, 10) satuan waktu.

3. Diketahui lima (5) PROSES dengan nama berturut-turut:

   - P1 (0, 9)
   - P2 (2, 7)
   - P3 (4, 1)
   - P4 (6, 3)
   - P5 (8, 2)

   Angka dalam kurung menunjukkan: ("*arrival time*", "*burst time*"). Setiap peralihan proses, selalu akan diperlukan waktu-alih (*switch time*) sebesar satu (1) satuan waktu (*unit time*).

   a. Berapakah rata-rata *turnaround time* dan *waiting time* dari kelima proses tersebut, jika diimplementasikan dengan algoritma penjadualan FCFS (*First Come, First Served*)?

   b. Bandingkan *turnaround time* dan *waiting time* tersebut, dengan sebuah algoritma penjadualan dengan ketentuan sebagai berikut:

      - *Pre-emptive*: pergantian proses dapat dilakukan kapan saja, jika ada proses lain yang memenuhi syarat. Namun durasi setiap proses dijamin minimum dua (2) satuan waktu,

sebelum boleh diganti.

- Waktu alih (*switch-time*) sama dengan diatas, yaitu sebesar satu (1) satuan waktu (*unit time*).
- Jika proses telah menunggu >= 15 satuan waktu:
  - dahulukan proses yang telah menunggu paling lama
  - lainnya: dahulukan proses yang menunggu paling sebentar.
- Jika kriteria yang terjadi seri: dahulukan proses dengan nomor urut yang lebih kecil (umpama: P1 akan didahulukan dari P2).

4. Apakah keuntungan menggunakan time quantum size di level yang berbeda dari sebuah antrian sistem multilevel?

   Pertanyaan nomor 4 sampai dengan 5 dibawah menggunakan soal berikut:

   Misal diberikan beberapa proses dibawah ini dengan panjang CPU burst (dalam milidetik)

   Semua proses diasumsikan datang pada saat t=0

5. Gambarkan 4 diagram Chart yang mengilustrasikan eksekusi dari proses-proses tersebut menggunakan FCFS, SJF, prioritas nonpreemptive dan round robin.
6. Hitung waktu tunggu dari setiap proses untuk setiap algoritma penjadualan.
7. Jelaskan perbedaan algoritma penjadualan berikut:

   - FCFS
   - Round Robin
   - Antrian Multilevel feedback

8. Penjadualan CPU mendefinisikan suatu urutan eksekusi dari proses terjadual. Diberikan n buah proses yang akan dijadualkan dalam satu prosesor, berapa banyak kemungkinan penjadualan yang berbeda? berikan formula dari n.
9. Tentukan perbedaan antara penjadualan preemptive dan nonpreemptive (cooperative). Nyatakan kenapa nonpreemptive scheduling tidak dapat digunakan pada suatu komputer center. Di sistem komputer nonpreemptive, penjadualan yang lebih baik digunakan.
10. Jelaskan bagaimana Java RMI dapat bekerja

15.9. Rujukan

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000. John Wiley & Sons, Inc.

William Stallings: Operating Systems -- Fourth Edition, Prentice Hall, 2001.

Bab 16. Algoritma Penjadualan II

Penjadualan pada prosesor jamak jelas lebih kompleks, karena kemungkinan masalah yang timbul jauh lebih banyak daripada prosesor tunggal.

16.1. Prioritas

Prioritas adalah suatu istilah yang digunakan untuk menentukan tingkat urutan atau hirarki suatu proses yang sedang masuk dalam *ready queue*.

16.2. Prosesor Jamak

Mengacu Silberschatz dkk., sistem dengan prosesor jamak yang dimaksud adalah suatu sistem dimana prosesor-prosesornya identik. Dalam hal ini berarti tiap proses dapat masuk antrian mana pun dari prosesor-prosesor yang ada. Yang patut diperhatikan, tiap prosesor dapat memilih proses apa saja yang ingin dijalankan dari *ready queue*. Dengan kata lain, prioritas proses ditentukan secara independen oleh masing-masing prosesor. Jadi salah satu prosesor dapat saja *idle* ketika ada proses yang sedang ditunda. Oleh karena itu, tiap prosesor harus di *synchronize* lebih dulu agar tidak ada dua prosesor atau lebih yang berebut mengeksekusi proses yang sama dan mengubah *shared data*. Sistem seperti ini dikenal juga dengan sebutan *synchronous*. Selain *synchronous*, ada juga sistem lain yang disebut *asynchronous*, yang juga dikenal dengan struktur "*master-slave*" dimana salah satu prosesor dialokasikan khusus untuk mengatur penjadualan. Sedangkan prosesor yang lain ditujukan untuk mengkomputasikan proses yang telah dijadualkan sebelumnya oleh *master* prosesor. Peningkatan dari sistem ini adalah mengalokasikan penjadualan, pemrosesan M/K, dan kegiatan sistem lainnya kepada satu prosesor tertentu kepada *master*. Sedangkan prosesor yang lain hanya bertugas mengeksekusi *user code*.

16.3. Sistem Waktu Nyata

Pada sub bab ini, kami akan mencoba sedikit menggambarkan fasilitas penjadualan yang dibutuhkan untuk mendukung komputasi waktu nyata dengan bantuan sistem komputer.

Suatu sistem komputasi dinamakan waktu nyata jika sistem tersebut dapat mendukung eksekusi program/aplikasi dengan waktu yang memiliki batasan. Dengan kata lain, sistem waktu nyata harus memenuhi kondisi berikut:

- Batasan waktu: memenuhi *deadline*, artinya bahwa aplikasi harus menyelesaikan tugasnya dalam waktu yang telah dibatasi.
- Dapat diprediksi: artinya bahwa sistem harus bereaksi terhadap semua kemungkinan kejadian selama kejadian tersebut dapat diprediksi.
- Proses bersamaan: artinya jika ada beberapa proses yang terjadi bersamaan, maka semua *deadline*-nya harus terpenuhi.
- Komputasi waktu nyata (*real-time*) ada dua jenis, yaitu sistem *Hard Real-time* dan sistem *Soft Real-time*.

16.4. Sistem *Hard Real-Time*

Sistem **hard real-time** dibutuhkan untuk menyelesaikan *critical task* dengan jaminan waktu tertentu. Jika kebutuhan waktu tidak terpenuhi, maka aplikasi akan gagal. Dalam definisi lain disebutkan bahwa kontrol sistem hard real-time dapat mentoleransi keterlambatan tidak lebih dari 100 mikro detik. Secara umum, sebuah proses di kirim dengan sebuah pernyataan jumlah waktu dimana dibutuhkan untuk menyelesaikan atau menjalankan M/K. Kemudian penjadual dapat

menjamin proses untuk selesai atau menolak permintaan karena tidak mungkin dilakukan. Mekanisme ini dikenal dengan **resource reservation**. Oleh karena itu setiap operasi harus dijamin dengan waktu maksimum. Pemberian jaminan seperti ini tidak dapat dilakukan dalam sistem dengan *secondary storage* atau *virtual memory*, karena sistem seperti ini tidak dapat meramalkan waktu yang dibutuhkan untuk mengeksekusi suatu proses.

Contoh dalam kehidupan sehari-hari adalah pada sistem pengontrol pesawat terbang. Dalam hal ini, keterlambatan sama sekali tidak boleh terjadi, karena dapat berakibat tidak terkontrolnya pesawat terbang. Nyawa penumpang yang ada dalam pesawat tergantung dari sistem ini, karena jika sistem pengontrol tidak dapat merespon tepat waktu, maka dapat menyebabkan kecelakaan yang merenggut korban jiwa.

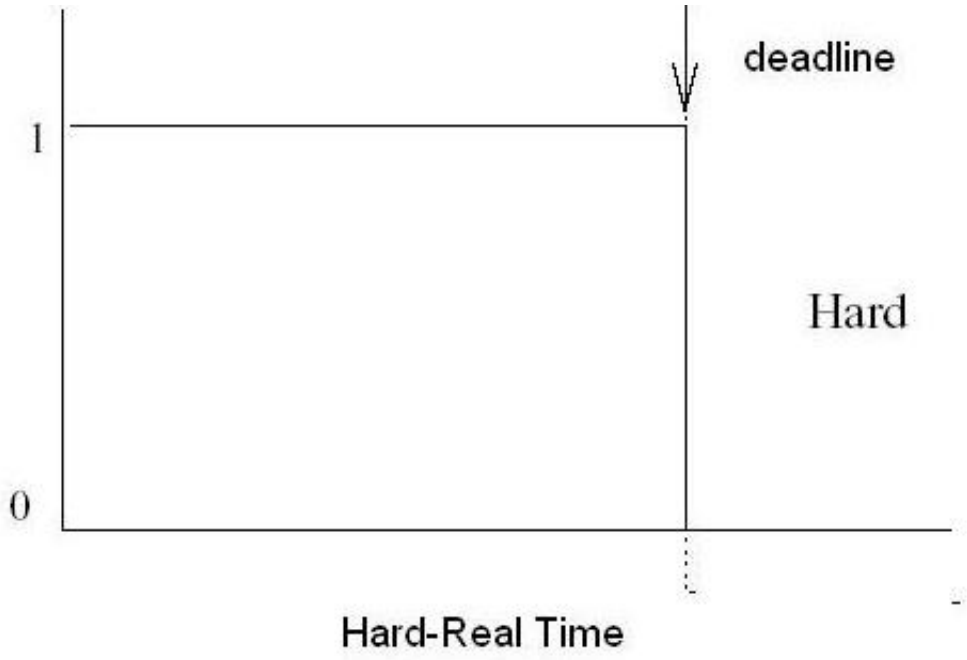
16.5. Sistem *Soft Real-Time*

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

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

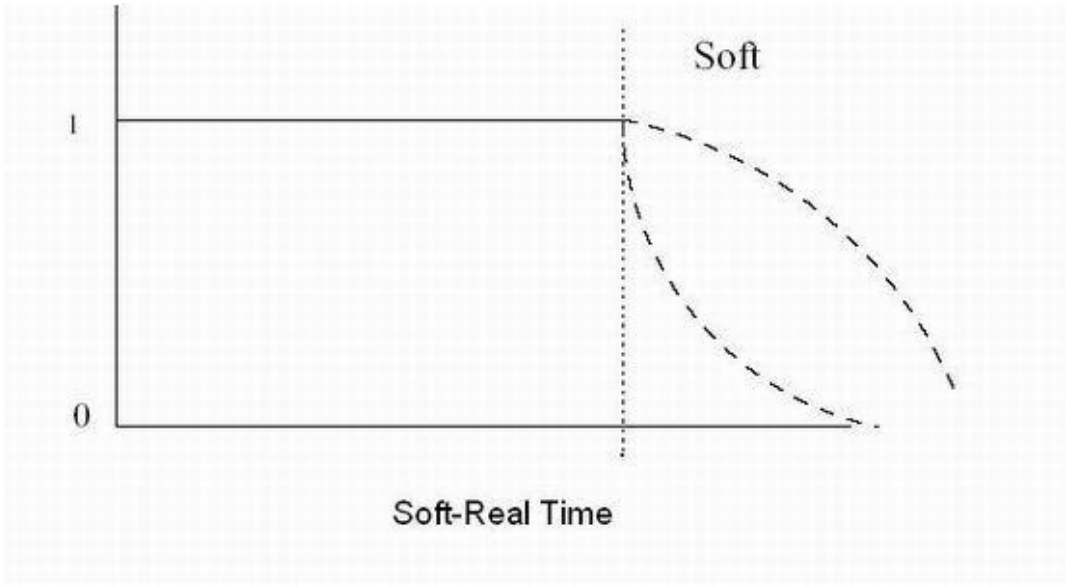
Untuk lebih memahami tentang perbedaan kedua sistem ini dapat diperhatikan dari diagram dibawah ini.

Gambar 16.1. Grafik *Hard Real-Time*



Sumber: http://www.ncst.ernet.in/ education/ pgdst/ coosfac/ slides/ rtos.pdf per Desember 2003.

Gambar 16.2. Grafik *Soft Real-Time*



Sumber: http://www.ncst.ernet.in/ education/ pgdst/ coosfac/ slides/ rtos.pdf; per Desember 2003.

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

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

Untuk menjaga *dispatch* tetap rendah, kita butuh agar *system call* untuk *preemptible*. Ada beberapa cara untuk mencapai tujuan ini. Pertama adalah dengan memasukkan *preemption points* di durasi *system call* yang lama, yang memeriksa apakah prioritas utama butuh untuk dieksekusi. Jika sudah, maka *contex switch* mengambil alih, ketika *high priority* proses selesai, proses yang diinterupsi meneruskan dengan *system call*. Points *premption* dapat diganti hanya di lokasi yang aman di kernel dimana kernel struktur tidak dapat dimodifikasi.

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

16.6. Penjadualan Thread

Seperti yang sudah diketahui pada awal bab ini, thread adalah sebuah unit program yang dieksekusi secara independent dari bagian lain pada program, atau dengan kata lain thread adalah unit dasar dari penggunaan CPU. Penjadualan thread berarti membuat suatu pengaturan atau pembagian jatah CPU untuk menjalankan thread pada proses yang multithreading sehingga masing-masing thread tersebut dapat berjalan secara konkurent dan sinkron. Dengan men-switch CPU diantara thread, sehingga dapat memaksimalkan kerja CPU dan menghasilkan output seperti yang diinginkan. Agar thread dapat dijadualkan, haruslah proses tersebut multithreading, sebab jika bukan multithreading, untuk apa thread tersebut dijadualkan?

Ada beberapa variabel yang mempengaruhi penjadualan thread, yaitu:

1. priority
2. scheduler and queue position
3. scheduling policy
4. contention scope
5. processor affinity
6. default
7. process level control
8. thread level control

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

Untuk linux, terdapat 3 pilihan kebijakan penjadualan thread, yaitu:

1. SCHED_OTHER
2. SCHED_FIFO
3. SCHED_RR: Round Robin Scheduling

SCHED_OTHER

SCHED_OTHER adalah penjadualan standar linux, dan kebijakan penjadualan default yang digunakan pada kebanyakan proses. Proses yang berjalan dipilih dari daftar prioritas static 0 yang ditentukan berdasarkan pada prioritas dinamik, dan prioritas dinamik ditentukan di dalam daftar prioritas. Prioritas dinamik ditentukan berdasarkan pada level nice (diset oleh sistem call nice atau setpriority), dan bertambah setiap kali proses siap dijalankan tetapi dibatalkan oleh scheduler. Ini menjamin kemajuan yang fair diantara semua proses SCHED_OTHER. Pada linux yang standar, ini adalah satu-satunya scheduler yang user biasa dapat jalankan. Standar linux memerlukan keistimewaan superuser untuk menjalankan proses dengan SCHED_FIFO atau SCHED_RR.

SCHED_FIFO: First In First Out scheduling

Ketika proses SCHED_FIFO dapat dijalankan, ia akan segera mem-preempt semua proses SCHED_OTHER yang sedang berjalan normal. SCHED_FIFO adalah algoritma scheduling sederhana tanpa time slicing. Untuk proses yang berdasarkan pada kebijakan SCHED_FIFO, aturan berikut diberlakukan: Sebuah proses SCHED_FIFO yang telahdi-preempted oleh proses lain akan menempati urutan atas untuk prioritasnya dan akan memulai kembali exekusinya segera setelah

proses yang prioritasnya lebih tinggi diblock. Ketika proses SCHED_FIFO dapat dijalankan, ia akan dimasukkan pada urutan akhir dari daftar untuk prioritasnya. Sebuah call untuk sched_setscheduler atau sched_param akan membuat proses SCHED_FIFO diidentifikasi oleh pid pada akhir dari list jika ia runnable. Sebuah proses memanggil sched_yield akan ditaruh diakhir dari list. Tidak ada event yang dapat memindahkan proses yang dijadualkan berdasarkan kebijakan SCHED_FIFO di daftar tunggu runnable process dengan prioritas statik yang sama. Proses SCHED_FIFO berjalan sampai diblok oleh permintaan I/O, di-preempt oleh proses yang berprioritas lebih tinggi, memanggil sched_yield, atau proses tersebut sudah selesai.

SCHED_RR: Round Robin Scheduling

SCHED_RR adalah peningkatan sederhana dari SCHED_FIFO. Semua aturan yang dijelaskan pada SCHED_FIFO juga digunakan pada SCHED_RR, kecuali proses yang hanya diizinkan berjalan untuk sebuah maksimum time quantum. Jika proses telah berjalan selama waktunya atau bahkan lebih lama dari waktu kuantumnya, ia akan ditaruh di bagian akhir dari daftar prioritas. Panjang kuantum time dapat dipulihkan kembali dengan sched_rr_get_interval.

16.7. Penjadualan Java

Java merupakan bahasa pemrograman yang telah mendukung konsep thread. Java sendiri telah mengatur penjadualan thread-thread tersebut secara internal. Secara umum hanya terdapat sebuah thread tunggal dalam java dan bila kita membuat thread-thread baru maka thread tersebut akan menjadi thread anakan dari thread utama tadi.

Java sudah menyediakan method-method untuk menjalankan suatu thread, membuat suatu thread untuk menunggu, agar thread "berhenti" sejenak, dan menghentikan thread yang sedang berjalan. Semuanya sudah disediakan dalam Java API.

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

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

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

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

16.8. Kinerja

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

Masalah pertama yang dihadapi adalah bagaimana mendefinisikan kriteria yang akan digunakan untuk memilih algoritma yang paling baik kinerjanya. Kriteria-kriteria tersebut biasanya didefinisikan sebagai utilisasi CPU, response time atau throughput.

Untuk memilih algoritma dengan kinerja paling baik, pertama kali kita harus mendefinisikan dulu kepentingan-kepentingan relatif yang ingin dicapai dari pengukuran kriteria tersebut. Kriteria-kriteria tadi dapat mencakup beberapa pengukuran seperti:

- Maksimalisasi utilisasi CPU di bawah batas bahwa response time maksimal adalah 1 detik.
- Maksimalisasi throughput sehingga rata-rata turnaround time menjadi proporsional secara linear terhadap total waktu eksekusi.

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

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

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

16.9. Rangkuman

Sistem hard real-time biasa digunakan menyelesaikan critical task dengan jaminan waktu tertentu. Jika kebutuhan waktu tidak terpenuhi, maka aplikasi akan gagal. Sistem operasi menggunakan mekanisme resource reservation untuk menerima atau menolak suatu permintaan proses.

Sistem soft real-time memiliki kelonggaran waktu dibandingkan sistem hard real-time. Sistem soft real-time dapat mengakibatkan delay yang lebih lama atau dapat menyebabkan terjadinya starvation.

16.10. Latihan

1. Apa yang dimaksud dengan:
    a. Prioritas
    b. Master Slave
    c. Komputasi Real Time
    d. Resource Reservation
    e. Priority inversion
2. Apa perbedaan antara Hard Realtime dan Soft Realtime?
3. Apa perbedaan antara Synchronous dan Asynchronous?
4. Buat contoh kasus dalam kehidupan sehari-hari pada prinsip real time dan Soft real time?
5. Sebutkan tiga kondisi yang memenuhi system waktu nyata!

16.11. Rujukan

- charm.cs.uiuc.edu/manuals/html/ converse/ 3_3Thread_Scheduling_Hooks.html

- linserver.cs.tamu.edu/~ravolz/456/Chapter-3.pdf

- www.unet.univie.ac.at/aix/ aixprggd/genprogc/threads_sched.htm

- www.particle.kth.se/~fmi/kurs/PhysicsSimulation/ Lectures/10A/ schedulePriority.html

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000. John Wiley & Sons, Inc.

William Stallings: Operating Systems -- Fourth Edition, Prentice Hall, 2001.

Bab 17. Managemen Proses Linux

17.1. Pendahuluan

Setiap aplikasi yang dijalankan di linux mempunyai pengenal yang disebut sebagai process identification number (PID). Hingga kernel versi 2.4, PID disimpan dalam angka 16 bit dengan kisaran dari 0-32767 untuk menjamin kompatibilitas dengan unix. Dari nomor PID inilah linux dapat mengawasi dan mengatur proses-proses yang terjadi didalam system. Proses yang dijalankan ataupun yang baru dibuat mempunyai struktur data yang disimpan di `task_struct`.

Linux mengatur semua proses di dalam sistem melalui pemeriksaan dan perubahan terhadap setiap struktur data task_struct yang dimiliki setiap proses. Sebuah daftar pointer ke semua struktur data `task_struct` disimpan dalam task vector. Jumlah maksimum proses dalam sistem dibatasi oleh ukuran dari task vector. Linux umumnya memiliki task vector dengan ukuran 512 entries. Saat proses dibuat, task_struct baru dialokasikan dari memori sistem dan ditambahkan ke task vector. Linux juga mendukung proses secara real time. Proses semacam ini harus bereaksi sangat cepat terhadap event eksternal dan diperlakukan berbeda dari proses biasa lainnya oleh penjadual.

Proses akan berakhir ketika ia memanggil `exit()`. Kernel akan menentukan waktu pelepasan sumber daya yang dimiliki oleh proses yang telah selesai tersebut. Fungsi `do_exit()` akan dipanggil saat terminasi yang kemudian memanggil `__exit_mm/files/fs/sighand()` yang akan membebaskan sumber daya. Fungsi `exit_notify()` akan memperbarui hubungan antara proses induk dan proses anak, semua proses anak yang induknya berakhir akan menjadi anak dari proses init. Terakhir akan dipanggil scheduler untuk menjalankan proses baru.

17.2. Deskriptor Proses

Contoh 17.1. Isi Deskriptor Proses

```
struct task_struct{
    volatile long state; /* -1 unrunnable,
                             0 runnable,
                            >0 stopped        */
    unsigned long flags;
                /* 1 untuk setiap flag proses */
    mm_segment_t_addr_limit;
                /* ruang alamat untuk thread  */
    struct exec_domain *exec_domain;
    long need_resched;
    long counter;
    long priority; /* SMP and runqueue state  */
    struct task_struct *next_task, *prev_task;
    struct task_struct *next_run, *prev_run;
    ...
    /* task state                              */
    /* limits                                  */
    /* file system info                        */
    /* ipc stuff                               */
    /* tss for this task                       */
    /* filesystem information                  */
    /* open file information                   */
    /* memory management info                  */
    /* signal handlers                         */
    ...
};
```

Guna keperluan managemen proses, kernel memelihara informasi tentang setiap proses di sebuah deskriptor proses dengan tipe task_struct. Setiap deskriptor proses mengandung informasi antara lain status proses, ruang alamat, daftar berkas yang dibuka, prioritas proses, dan sebagainya. Berikut gambaran isinya:

Setiap proses di Linux memiliki status. Status proses merupakan array dari flag yang mutually exclusive. Setiap proses memiliki tepat satu keadaan (status) pada suatu waktu. Status tersebut adalah:

- TASK_RUNNING

  Pada status ini, proses sedang atau pun siap dieksekusi oleh CPU.

- TASK_INTERRUPTIBLE

  Pada status ini, proses sedang menunggu sebuah kondisi. Interupsi, sinyal, atau pun pelepasan sumber daya akan membangunkan proses.

- TASK_UNINTERRUPTIBLE

  Pada status ini, proses sedang tidur dan tidak dapat dibangunkan oleh suatu sinyal.

- TASK_STOPPED

  Pada status ini proses sedang dihentikan, misalnya oleh sebuah debugger.

- TASK_ZOMBIE

  Pada status ini proses telah berhenti, namun masih memiliki struktur data task_struct di task vector dan masih memegang sumber daya yang sudah tidak digunakan lagi.

Setiap proses atau pun eksekusi yang terjadwal secara independen memiliki deskriptor prosesnya sendiri. Alamat dari deskriptor proses digunakan untuk mengindentifikasi proses. Selain itu, nomor ID proses (PIDs) juga digunakan untuk keperluan tersebut. PIDs adalah 16-bit bilangan yang mengidentifikasikan setiap proses dengan unik. Linux membatasi PIDs berkisar 0-32767 untuk menjamin kompatibilitas dengan sistem UNIX tradisional.

Karena proses merupakan sesuatu yang dinamis, maka deskriptor proses disimpan dalam memori yang dinamis pula. Untuk itu dialokasikan juga memori sebesar 8KB untuk setiap proses untuk menyimpan proses deskriptornya dan stack proses dari modus kernel. Keuntungan dari dal ini adalah pointer dari deskriptor proses dari proses yang sedang berjalan (running) dapat diakses dengan cepat menggunakan stack pointer. Selain itu, 8KB (EXTRA_TASK_STRUCT) dari memori akan di-cache untuk mem-bypass pengalokasi memori kernel ketika sebuah proses dihapus dan sebuah proses baru dibuat. Kedua perintah free_task_struct() dan alloc_task_struct() akan digunakan untuk melepaskan atau mengalokasikan memori seukuran 8KB sebagai cache.

Deskriptor proses juga membangun sebuah daftar proses dari semua proses yang ada di sistem. Daftar proses tersebut merupakan sebuah doubly-linked list yang dibangun oleh bagian next_task dan prev_task dari deskriptor proses. Deskriptor init_task(mis:swapper) berada di awal daftar tersebut dengan prev_task-nya menunjuk ke deskriptor proses yang paling akhir masuk dalam daftar. Sedangkan makro for_each_task() digunakan untuk memindai seluruh daftar.

Proses yang dijadwalkan untuk dieksekusi dari doubly-linked list dari proses dengan status TASK_RUNNING disebut runqueue. Bagian prev_run dan next_run dari deskriptor proses digunakan untuk membangun runqueue, dengan init_task mengawali daftar tersebut. Sedangkan untuk memanipulasi daftar di deskriptor proses tersebut, digunakan fungsi-fungsi: add_to_runqueue(), del_from_runqueue(), move_first_runqueue(), move_last_runqueue(). Makro NR_RUNNING digunakan untuk menyimpan jumlah proses yang dapat dijalankan, sedangkan fungsi wake_up_process membuat sebuah proses menjadi dapat dijalankan.

Untuk menjamin akurasinya, array task akan diperbarui setiap kali ada proses baru dibuat atau pun dihapus. Sebuah daftar terpisah akan melacak elemen bebas dalam array task itu. Ketika suatu

proses dihapus, entrinya ditambahkan di bagian awal dari daftar tersebut.

Proses dengan status task_interruptible dibagi ke dalam kelas-kelas yang terkait dengan suatu event tertentu. Event yang dimaksud misalnya: waktu kadaluarsa, ketersediaan sumber daya. Untuk setiap event atau pun kelas terdapat antrian tunggu yang terpisah. Proses akan diberi sinyal bangun ketika event yang ditunggunya terjadi. Berikut contoh dari antrian tunggu tersebut:

Contoh 17.2. Antrian Tunggu

```
void sleep_on(struct wait_queue **wqptr) {
    struct wait_queue wait;
    current_state=TASK_UNINTERRUPTIBLE;
    wait.task=current;
    add_wait_queue(wqptr, &wait);
    schedule();
    remove_wait_queue(wqptr, &wait);
}
```

Fungsi sleep_on() akan memasukkan suatu proses ke dalam antrian tunggu yang diinginkan dan memulai penjadwal. Ketika proses itu mendapat sinyal untuk bangun, maka proses tersebut akan dihapus dari antrian tunggu.

Bagian lain konteks eksekusi proses adalah konteks perangkat keras, misalnya: isi register. Konteks dari perangkat keras akan disimpan oleh task state segment dan stack modus kernel. Secara khusus tss akan menyimpan konteks yang tidak secara otomatis disimpan oleh perangkat keras tersebut. Perpindahan antar proses melibatkan penyimpanan konteks dari proses yang sebelumnya dan proses berikutnya. Hal ini harus dapat dilakukan dengan cepat untuk mencegah terbuangnya waktu CPU. Versi baru dari Linux mengganti perpindahan konteks perangkat keras ini menggunakan piranti lunak yang mengimplementasikan sederetan instruksi mov untuk menjamin validasi data yang disimpan serta potensi untuk melakukan optimasi.

Untuk mengubah konteks proses digunakan makro switch_to(). Makro tersebut akan mengganti proses dari proses yang ditunjuk oleh prev_task menjadi next_task. Makro switch_to() dijalankan oleh schedule() dan merupakan salah satu rutin kernel yang sangat tergantung pada perangkat keras (hardware-dependent). Lebih jelas dapat dilihat pada kernel/sched.c dan include/asm-*/system.h.

17.3. Proses dan *Thread*

Dewasa ini (2005), banyak sistem operasi yang telah mendukung proses *multithreading*. Setiap sistem operasi memiliki konsep tersendiri dalam mengimplementasikannya ke dalam sistem.

Linux menggunakan representasi yang sama untuk proses dan thread. Secara sederhana thread dapat dikatakan sebuah proses baru yang berbagi alamat yang sama dengan induknya. Perbedaannnya terletak pada saat pembuatannya. Thread baru dibuat dengan system call `clone` yang membuat proses baru dengan identitas sendiri, namun diizinkan untuk berbagi struktur data dengan induknya.

Secara tradisional, sumber daya yang dimiliki oleh proses induk akan diduplikasi ketika membuat proses anak. Penyalinan ruang alamat ini berjalan lambat, sehingga untuk mengatasinya, salinan hanya dibuat ketika salah satu dari mereka hendak menulis di alamat tersebut. Selain itu, ketika mereka akan berbagi alamat tersebut ketika mereka hanya membaca. Inilah proses ringan yang dikenal juga dengan thread.

Thread dibuat dengan `__clone()`. `__clone()` merupakan rutin dari library system call `clone()`. `__clone` memiliki empat buah argumen yaitu:

1. `fn`

   fungsi yang akan dieksekusi oleh thread baru

2. `arg`

   pointer ke data yang dibawa oleh `fn`

3. `flags`

   sinyal yang dikirim ke induk ketika anak berakhir dan pembagian sumber daya antara anak dan induk.

4. `child_stack`

   pointer stack untuk proses anak.

`clone()` mengambil argumen `flags` dan `child_stack` yang dimiliki oleh `__clone` kemudian menentukan id dari proses anak yang akan mengeksekusi `fn` dengan argumen `arg`.

Pembuatan anak proses dapat dilakukan dengan fungsi `fork()` dan `vfork()`. Implementasi `fork()` sama seperti system call `clone()` dengan sighandler `SIGCHLD` di-set, semua bendera clone di-clear yang berarti tidak ada sharing dan `child_stack` dibuat 0 yang berarti kernel akan membuat stack untuk anak saat hendak menulis. Sedangkan `vfork()` sama seperti `fork()` dengan tambahan bendera `CLONE_VM` dan `CLONE_VFORK` di-set. Dengan `vfork()`, induk dan anak akan berbagi alamat, dan induk akan di-block hingga anak selesai.

Untuk memulai pembuatan proses baru, `clone()` akan memanggil fungsi `do_fork()`. Hal yang dilakukan oleh `do_fork()` antara lain:

- memanggil `alloc_task_struct()` yang akan menyediakan tempat di memori dengan ukuran 8KB untuk deskriptor proses dan stack modus kernel.
- memeriksa ketersediaan sumber daya untuk membuat proses baru.
- `find_empty_procees()` memanggil `get_free_taskslot()` untuk mencari sebuah slot di array task untuk pointer ke deskriptor proses yang baru.
- memanggil `copy_files/fm/sighand/mm()` untuk menyalin sumber daya untuk anak, berdasarkan nilai flags yang ditentukan `clone()`.
- `copy_thread()` akan menginisialisasi stack kernel dari proses anak.
- mendapatkan PID baru untuk anak yang akan diberikan kembali ke induknya ketika `do_fork()` selesai.

Beberapa proses sistem hanya berjalan dalam modus kernel di belakang layar. Untuk proses semacam ini dapat digunakan thread kernel. Thread kernel hanya akan mengeksekusi fungsi kernel, yaitu fungsi yang biasanya dipanggil oleh proses normal melalui system calls. Thread kernel juga hanya dieksekusi dalam modus kernel, berbeda dengan proses biasa. Alamat linier yang digunakan oleh thread kernel lebih besar dari PAGE_OFFSET proses normal yang dapat berukuran hingga 4GB. Thread kernel dibuat sebagai berikut:

Contoh 17.3. XXX

```
int kernel_thread(int (*fn) (void *), void *arg, unsigned long flags);
flags=CLONE_SIGHAND, CLONE_FILES, etc
```

Kernel Linux mulai menggunakan *thread* pada versi 2.2. *Thread* dalam Linux dianggap sebagai *task*, seperti halnya proses. Kebanyakan sistem operasi yang mengimplementasikan *multithreading* menjalankan sebuah *thread* terpisah dari proses. Linus Torvalds mendefinisikan bahwa sebuah thread adalah *Context of Execution* (COE), yang berarti bahwa hanya ada sebuah *Process Control Block* (PCB) dan sebuah penjadual yang diperlukan. Linux tidak mendukung *multithreading*, struktur data yang terpisah, atau pun rutin *kernel*.

Linux menyediakan dua macam *system call*, yaitu *fork* dan *clone*. *fork* memiliki fungsi untuk menduplikasi proses dimana proses anak yang dihasilkan bersifat *independent*. *clone* memiliki sifat yang mirip dengan *fork* yaitu sama-sama membuat duplikat dari proses induk. Namun demikian, selain membuat proses baru yang terpisah dari proses induk, *clone* juga mengizinkan terjadinya proses berbagi ruang alamat antara proses anak dengan proses induk, sehingga proses anak yang dihasilkan akan sama persis dengan proses induknya.

Setiap proses memiliki struktur data yang unik. Namun demikian, proses-proses di Linux hanya menyimpan *pointer-pointer* ke struktur data lainnya dimana instruksi disimpan, sehingga tidak harus menyimpan instruksi ke setiap struktur data yang ada. Hal ini menyebabkan *context switch* antar proses di Linux menjadi lebih cepat.

Ketika *fork* dieksekusi, sebuah proses baru dibuat bersamaan dengan proses penyalinan struktur data dari proses induk. Ketika *clone* dieksekusi, sebuah proses baru juga dibuat, namun proses tersebut tidak menyalin struktur data dari proses induknya. Proses baru tersebut hanya menyimpan *pointer* ke struktur data proses induk. Oleh karena itu, proses anak dapat berbagi ruang alamat dan sumber daya dengan proses induknya. Satu set *flag* digunakan untuk mengindikasikan seberapa banyak kedua proses tersebut dapat berbagi. Jika tidak ada *flag* yang ditandai, maka tidak ada *sharing*, sehingga *clone* berlaku sebagai *fork*. Jika kelima *flag* ditandai, maka proses induk harus berbagi semuanya dengan proses anak.

Tabel 17.1. Tabel Flag dan Fungsinya

Flag	Keterangan
CLONE_VM	Berbagi data dan Stack
CLONE_FS	Berbagi informasi sistem berkas
CLONE_FILES	Berbagi berkas
CLONE_SIGHAND	Berbagi sinyal
CLONE_PID	Berbagi PID dengan proses induk

17.4. Penjadualan

Penjadual adalah suatu pekerjaan yang dilakukan untuk mengalokasikan CPU time untuk tasks yang berbeda-beda dalam sistem operasi. Pada umumnya, kita berfikir penjadualan sebagai menjalankan dan menginterupsi suatu proses, untuk linux ada aspek lain yang penting dalam penjadualan: seperti menjalankan dengan berbagai kernel tasks. Kernel tasks meliputi task yang diminta oleh proses yang sedang dijalankan dan tasks yand dieksekusi internal menyangkut device driver yang berkepentingan.

Ketika kernel telah mencapai titik penjadualan ulang, entah karena terjadi interupsi penjadualan ulang mau pun karena proses kernel yang sedang berjalan telah diblokir untuk menunggu beberapa signal bangun, harus memutuskan proses selanjutnya yang akan dijalankan. Linux telah memiliki dua algoritma penjadualan proses yang terpisah satu sama lain. Algoritma yang pertama adalah algoritma time-sharing untuk penjadualan preemptive yang adil diantara sekian banyak proses. Sedangkan algoritma yang kedua didesain untuk tugas real-time dimana proritas mutlak lebih utama daripada keadilan mendapatkan suatu pelayanan.

Bagian dari tiap identitas proses adalah kelas penjadualan, yang akan menentukan algoritma yang digunakan untuk tiap proses. Kelas penjadualan yang digunakan oleh Linux, terdapat dalam standar perluasan POSIX untuk sistem komputer waktu nyata.

Untuk proses time-sharing, Linux menggunakan teknik prioritas, sebuah algoritma yang berdasarkan pada kupon. Tiap proses memiliki sejumlah kupon penjadualan; dimana ketika ada kesempatan untuk menjalankan sebuah tugas, maka proses dengan kupon terbanyaklah yang mendapat giliran. Setiap kali terjadi interupsi waktu, proses yang sedang berjalan akan kehilangan satu kupon; dan ketika kupon yang dimiliki sudah habis maka proses itu akan ditunda dan proses yang lain akan diberikan kesempatan untuk masuk.

Jika proses yang sedang berjalan tidak meiliki kupon sama sekali, linux akan melakukan operasi pemberian kupon, memberikan kupon kepada tiap proses dalam sistem, dengan aturan main:

```
kupon = kupon / 2 + prioritas
```

Algoritma ini cenderung untuk menggabungkan dua faktor yang ada: sejarah proses dan prioritas dari proses itu sendiri. Satu setengah dari kupon yang dimiliki sejak operasi pembagian kupon terakhir akan tetap dijaga setelah algoritma telah dijalankan, menjaga beberapa sejarah sikap proses. Proses yang berjalan sepanjang waktu akan cenderung untuk menghabiskan kupon yang dimilikinya dengan cepat, tapi proses yang lebih banyak menunggu dapat mengakumulasi kuponnya dari. Sistem pembagian kupon ini, akan secara otomatis memberikan proritas yang tinggi ke proses M/K bound atau pun interaktif, dimana respon yang cepat sangat diperlukan.

Kegunaan dari proses pemberian prioritas dalam menghitung kupon baru, membuat prioritas dari suatu proses dapat ditingkatkan. Pekerjaan background batch dapat diberikan prioritas yang rendah; proses tersebut akan secara otomatis menerima kupon yang lebih sedikit dibandingkan dengan pekerjaan yang interaktif, dan juga akan menerima persentase waktu CPU yang lebih sedikit dibandingan dengan tugas yang sama dengan prioritas yang lebih tinggi. Linux menggunakan sistem prioritas ini untuk menerapkan mekanisme standar pembagian prioritas proses yang lebih baik.

Penjadualan waktu nyata Linux masih tetap lebih sederhana. Linux, menerapkan dua kelas penjadualan waktu nyata yang dibutuhkan oleh POSIX 1.b: First In First Out dan round-robin. Pada keduanya, tiap proses memiliki prioritas sebagai tambahan kelas penjadualannya. Dalam penjadualan time-sharing, bagaimana pun juga proses dengan prioritas yang berbeda dapat bersaing dengan beberapa pelebaran; dalam penjadualan waktu nyata, si pembuat jadual selalu menjalankan proses dengan prioritas yang tinggi. Diantara proses dengan prioritas yang sama, maka proses yang sudah menunggu lama, akan dijalankan. Perbedaan satu - satunya antara penjadualan FIFO dan round-robin adalah proses FIFO akan melanjutkan prosesnya sampai keluar atau pun diblokir, sedangkan proses round-robin akan di-preemptive-kan setelah beberapa saat dan akan dipindahkan ke akhir antrian, jadi proses round-robin dengan prioritas yang sama akan secara otomatis membagi waktu jalan antar mereka sendiri.

Perlu diingat bahwa penjadualan waktu nyata di Linux memiliki sifat yang lunak. Pembuat jadual Linux menawarkan jaminan yang tegas mengenai prioritas relatif dari proses waktu nyata, tapi kernel tidak menjamin seberapa cepat penjadualan proses waktu-nyata akan dijalankan pada saat proses siap dijalankan. Ingat bahwa kode kernel Linux tidak akan pernah bisa dipreemptive oleh kode mode pengguna. Apabila terjadi interupsi yang membangunkan proses waktu nyata, sementara kernel siap untuk mengeksekusi sebuah sistem call sebagai bagian proses lain, proses waktu nyata harus menunggu sampai sistem call yang sedang dijalankan selesai atau diblokir.

17.5. *Symmetric Multiprocessing*

Kernel Linux 2.0 adalah kernel Linux pertama yang stabil untuk mendukung perangkat keras symmetric multiprocessor (SMP). Proses mau pun thread yang berbeda dapat dieksekusi secara paralel dengan processor yang berbeda. Tapi bagaimana pun juga untuk menjaga kelangsungan kebutuhan sinkronisasi yang tidak dapat di-preemptive dari kernel, penerapan SMP ini menerapkan aturan dimana hanya satu processor yang dapat dieksekusi dengan kode mode kernel pada suatu saat. SMP menggunakan kernel spinlock tunggal untuk menjalankan aturan ini. Spinlock ini tidak memunculkan permasalahan untuk pekerjaan yang banyak menghabiskan waktu untuk menunggu proses komputasi, tapi untuk pekerjaan yang melibatkan banyak aktifitas kernel, spinlock dapat menjadi sangat mengkhawatirkan.

Sebuah proyek yang besar dalam pengembangan kernel Linux 2.1 adalah untuk menciptakan penerapan SMP yang lebih masuk akal, dengan membagi kernel spinlock tunggal menjadi banyak kunci yang masing-masing melindungi terhadap masuknya kembali sebagian kecil data struktur kernel. Dengan menggunakan teknik ini, pengembangan kernel yang terbaru mengizinkan banyak processor untuk dieksekusi oleh kode mode kernel secara bersamaan.

17.6. Rangkuman

Kernel Linux mengawasi dan mengatur proses berdasarkan nomor process identification number (PID), sedangkan informasi tentang setiap proses disimpan pada deskriptor proses. Deskriptor proses merupakan struktur data yang memiliki beberapa status, yaitu: TASK_RUNNING, TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_STOPPED, dan TASK_ZOMBIE.

Linux menyediakan dua system call, yaitu fork dan clone. Fork memiliki fungsi untuk menduplikasi proses di mana proses anak yang dihasilkan bersifat independen. Sedangkan fungsi clone, selain menduplikasi dari proses induk, juga mengizinkan terjadinya proses berbagi ruang alamat antara proses anak dengan proses induk.

Sinkronisasi kernel Linux dilakukan dengan dua solusi, yaitu:

- Dengan membuat normal kernel yang bersifat code nonpreemptible. Akan muncul salah satu dari tiga aksi, yaitu: interupsi, page fault, atau kernel code memanggil penjadualannya sendiri.
- Dengan meniadakan interupsi pada saat critical section muncul.

Penjadualan proses Linux dilakukan dengan menggunakan teknik prioritas, sebuah alogritma yang berdasarkan pada kupon, dengan aturan bahwa setiap proses akan memiliki kupon, di mana pada suatu saat kupon akan habis dan mengakibatkan tertundanya proses tersebut.

17.7. Latihan

Diketahui sebuah keluaran dari perintah "top b n 1" pada sebuah sistem GNU/Linux yaitu: "bunga.mhs.cs.ui.ac.id", yang diambil dari http://rmsui.vlsm.org/:

1. Apakah yang dimaksud dengan PID?
2. Berapakah PID dari perintah top yang dijalankan?
3. Apakah yang dimaksud dengan "zombie" dan "nice"?
4. Apakah yang dimaksud dengan "PRI " dan "NI"?
5. Apakah yang dimaksud dengan "load average"? Jelaskan ketiga angka di samping tulisan load average tersebut!

6. Jelaskan hubungan dari CPU states idle dari sistem tersebut yang tinggi dengan kondisi dari load average yang terlihat!

Gambar 17.1. XXX

```
 15:34:14 up 28 days, 14:40, 53 users,  load average: 0.28, 0.31, 0.26
265 processes: 264 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:   5.9% user,   1.8% system,   0.1% nice,  92.2% idle
Mem:    126624K total,   113548K used,    13076K free,      680K buffers
Swap:   263160K total,    58136K used,   205024K free,    41220K cached

  PID USER  PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
    1 root    8   0   460  420   408 S     0.0  0.3   0:56 init
    2 root    9   0     0    0     0 SW    0.0  0.0   0:02 keventd
    3 root   19  19     0    0     0 SWN   0.0  0.0   0:02 ksoftirqd_CPU0
    .....
17353 user1   9   0  2500 2004  2004 S     0.0  1.5   0:00 sshd
17354 user1   9   0  1716 1392  1392 S     0.0  1.0   0:00 bash
17355 user1   9   0  2840 2416  2332 S     0.0  1.9   0:00 pine
12851 user2   9   0  2500 2004  2004 S     0.0  1.5   0:00 sshd
12852 user2   9   0  1776 1436  1436 S     0.0  1.1   0:00 bash
13184 user2   9   0  1792 1076  1076 S     0.0  0.8   0:00 vi
13185 user2   9   0   392  316   316 S     0.0  0.2   0:00 grep
22272 user3   9   0  2604 2592  2292 S     0.0  2.0   0:00 sshd
22273 user3   9   0  1724 1724  1396 S     0.0  1.3   0:00 bash
22283 user3  14   0   980  980   660 R    20.4  0.7   0:00 top
19855 user4   9   0  2476 2048  1996 S     0.0  1.6   0:00 sshd
19856 user4   9   0  1700 1392  1392 S     0.0  1.0   0:00 bash
19858 user4   9   0  2780 2488  2352 S     0.0  1.9   0:00 pine
```

sumber: http://rmsui.vlsm.org/

17.8. Rujukan

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000. John Wiley & Sons, Inc.

William Stallings: Operating Systems -- Fourth Edition, Prentice Hall, 2001.

http://people.cs.uchicago.edu/~mark/51081/LabFAQ/lab5/IPC.html

Website Kuliah Sistem Terdistribusi Fasilkom UI, http://telaga.cs.ui.ac.id/WebKuliah/sisdis2003/

http://linas.org/linux/threads-faq.html

http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads.html

Bagian IV. Proses dan Sinkronisasi

Proses, Penjadualan, dan Sinkronisasi merupakan trio yang saling berhubungan, sehingga seharusnya tidak dipisahkan. Bagian yang lalu telah membahas Proses dan Penjadualannya, sehingga bagian ini akan membahas Proses dan Sinkronisasinya.

Bab 18. Konsep Interaksi

Proses yang dijalankan pada suatu sistem operasi dapat bekerja secara bersama-sama atau pun sendiri saja. Bagian sebelum ini telah menjelaskan mengenai konsep proses dan bagaimana penjadualan proses itu. Disini kita akan melihat bagaimana hubungan antara proses-proses itu.

18.1. Proses yang Kooperatif

Proses yang bersifat simultan (*concurrent*) dijalankan pada sistem operasi dapat dibedakan menjadi yaitu proses independen dan proses kooperatif. Suatu proses dikatakan independen apabila proses tersebut tidak dapat terpengaruh atau dipengaruhi oleh proses lain yang sedang dijalankan pada sistem. Berarti, semua proses yang tidak membagi data apa pun (baik sementara/tetap) dengan proses lain adalah independent. Sedangkan proses kooperatif adalah proses yang dapat dipengaruhi atau pun terpengaruhi oleh proses lain yang sedang dijalankan dalam sistem. Dengan kata lain, proses dikatakan kooperatif bila proses dapat membagi datanya dengan proses lain.

Ada empat alasan untuk penyediaan sebuah lingkungan yang memperbolehkan terjadinya proses kooperatif:

1. Pembagian informasi: apabila beberapa pengguna dapat tertarik pada bagian informasi yang sama (sebagai contoh, sebuah berkas bersama), kita harus menyediakan sebuah lingkungan yang mengizinkan akses secara terus menerus ke tipe dari sumber-sumber tersebut.

2. Kecepatan penghitungan/komputasi: jika kita menginginkan sebuah tugas khusus untuk menjalankan lebih cepat, kita harus membagi hal tersebut ke dalam subtask, setiap bagian dari subtask akan dijalankan secara parallel dengan yang lainnya. Peningkatan kecepatan dapat dilakukan hanya jika komputer tersebut memiliki elemen-elemen pemrosesan ganda (seperti CPU atau jalur M/K).

3. Modularitas: kita mungkin ingin untuk membangun sebuah sistem pada sebuah model modular-modular, membagi fungsi sistem menjadi beberapa proses atau *thread*.

4. Kenyamanan: bahkan seorang pengguna individu mungkin memiliki banyak tugas untuk dikerjakan secara bersamaan pada satu waktu. Sebagai contoh, seorang pengguna dapat mengedit, mencetak, dan meng-compile secara parallel.

Proses yang bersifat *concurrent* bekerja sama dengan proses lain. Proses itu kooperatif jika mereka dapat saling mempengaruhi. Kerja sama antar proses itu penting karena beberapa alasan:

- Pembagian informasi: Beberapa proses dapat mengakses beberapa data yang sama.

- Kecepatan komputasi: Tugas yang dijalankan dapat berjalan dengan lebih cepat jika tugas tersebut dipecah-pecah menjadi beberapa sub bagian dan dieksekusi secara paralel dengan sub bagian yang lain. Peningkatan kecepatan ini dapat dilakukan jika komputer tersebut mempunyai beberapa elemen pemrosesan, seperti CPU atau jalur M/K.

- Modularitas: Akan lebih mudah untuk mengatur tugas yang kompleks jika tugas tersebut dipecah menjadi beberapa sub bagian, kemudian mempunyai proses atau *thread* yang berbeda untuk menjalankan setiap sub bagian.

- Kenyamanan: *User* dapat dengan mudah mengerjakan sesuatu yang berbeda dalam waktu yang sama. Contohnya satu *user* dapat mengetik, mengedit, dan mencetak suatu halaman tertentu secara bersamaan.

Kerja sama antar proses membutuhkan suatu mekanisme yang memperbolehkan proses-proses untuk mengkomunikasikan data dengan yang lain dan meng-*synchronize* kerja mereka sehingga tidak ada yang saling menghalangi. Salah satu cara proses dapat saling berkomunikasi adalah *Interprocess*

Communication (IPC) yang akan dijelaskan lebih lanjut di bagian berikut.

18.2. Hubungan Antara Proses

Sebelumnya kita telah ketahui seluk beluk dari suatu proses mulai dari pengertiannya, cara kerjanya, sampai operasi-operasinya seperti proses pembentukannya dan proses pemberhentiannya setelah selesai melakukan eksekusi. Kali ini kita akan mengulas bagaimana hubungan antar proses dapat berlangsung, misal bagaimana beberapa proses dapat saling berkomunikasi dan bekerja-sama.

Contoh 18.1. *Bounded Buffer*

```
import java.util.*;

public class BoundedBuffer {
    public BoundedBuffer() {
        // buffer diinisialisasikan kosong
        count   = 0;
        in      = 0;
        out     = 0;

        buffer = new Object[BUFFER_SIZE];
     }

     // produser memanggil method ini
     public void enter( Object item ) {
        while ( count == BUFFER_SIZE )
            ; // do nothing
        // menambahkan suatu item ke dalam buffer
        ++count;
        buffer[in]  = item;
        in  = ( in + 1 ) % BUFFER_SIZE;
        if ( count == BUFFER_SIZE )
            System.out.println( "Producer Entered " +
                item + " Buffer FULL" );
        else
            System.out.println( "Producer Entered " +
                item + " Buffer Size = " + count );
    }

    // consumer memanggil method ini
    public Object remove() {
        Object item ;
        while ( count == 0 )
            ; // do nothing
        // menyingkirkan suatu item dari buffer
        --count;
        item  = buffer[out];
        out   = ( out + 1 ) % BUFFER_SIZE;
        if ( count == 0 )
            System.out.println( "Consumer consumed " +
              item + " Buffer EMPTY" );
        else
            System.out.println( "Consumer consumed " +
              item + " Buffer Size = " +count );
        return item;
    }
    public static final int NAP_TIME = 5;
    private static final int BUFFER_SIZE = 5;
    private volatile int count;
    private int in;     // arahkan ke posisi kosong selanjutnya
    private int out;    // arahkan ke posisi penuh  selanjutnya
    private Object[] buffer;
}
```

Kalau pada sub-bab sebelumnya kita banyak membahas mengenai buffer, dan lingkungan yang berbagi memori. Pada bagian ini kita lebih banyak membahas teknik komunikasi antara proses melalui kirim (*send*) dan terima (*receive*) yang biasa dikenal sebagai IPC.

Selain itu pada bagian ini kita akan menyingung sedikit mengenai client/server proses. Beberapa topik yang akan dibahas adalah Java *Remote Method Invocation* (RMI) dan *Remote Procedure Call* (RPC). Yang keduanya juga menggunakan mekanisme komunikasi IPC, namun menggunakan sistem yang terdistribusi yang melibatkan jaringan. Pada bagian ini juga akan dibahas mengenai infrastruktur dasar jaringan yaitu *socket*.

Sebuah produsen proses membentuk informasi yang dapat digunakan oleh konsumen proses. Sebagai contoh sebuah cetakan program yang membuat banyak karakter yang diterima oleh *driver* pencetak. Untuk memperbolehkan produser dan konsumer proses agar dapat berjalan secara terus menerus, kita harus menyediakan sebuah item buffer yang dapat diisi dengan proses produser dan dikosongkan oleh proses konsumer. Proses produser dapat memproduksi sebuah item ketika konsumer sedang mengkonsumsi item yang lain. Produser dan konsumer harus dapat selaras. Konsumer harus menunggu hingga sebuah item diproduksi.

18.3. Komunikasi Proses Dalam Sistem

Cara lain untuk meningkatkan efek yang sama adalah untuk sistem operasi yaitu untuk menyediakan alat-alat proses kooperatif untuk berkomunikasi dengan yang lain lewat sebuah komunikasi dalam proses *Inter-Process Communication* (IPC). IPC menyediakan sebuah mekanisme untuk megizinkan proses-proses untuk berkomunikasi dan menyelaraskan aksi-aksi mereka tanpa berbagi ruang alamat yang sama. IPC adalah khusus digunakan dalam sebuah lingkungan yang terdistribusi dimana proses komunikasi tersebut mungkin saja tetap ada dalam komputer-komputer yang berbeda yang tersambung dalam sebuah jaringan. IPC adalah penyedia layanan terbaik dengan menggnakan sebuah sistem penyampaian pesan, dan sistem-sistem pesan dapat diberikan dalam banyak cara.

Fungsi dari sebuah sistem pesan adalah untuk memperbolehkan komunikasi satu dengan yang lain tanpa perlu menggunakan pembagian data. Sebuah fasilitas IPC menyediakan paling sedikit dua operasi yaitu kirim (pesan) dan terima (pesan). Pesan dikirim dengan sebuah proses yang dapat dilakukan pada ukuran pasti atau variabel. Jika hanya pesan dengan ukuran pasti dapat dikirimkan, level sistem implementasi adalah sistem yang sederhana. Pesan berukuran variabel menyediakan sistem implementasi level yang lebih kompleks.

Jika dua buah proses ingin berkomonikasi, misalnya proses P dan proses Q, mereka harus mengirim pesan atau menerima pesan dari satu ke yang lainnya. Jalur ini dapat diimplementasikan dengan banyak cara, namun kita hanya akan memfokuskan pada implementasi logiknya saja, bukan implementasi fisik (seperti *shared memory*, *hardware bus*, atau jaringan). Berikut ini ada beberapa metode untuk mengimplementasikan sebuah jaringan dan operasi pengiriman/penerimaan secara logika:

- Komunikasi langsung atau tidak langsung.
- Komunikasi secara simetris/asimetris.
- Buffer otomatis atau eksplisit.
- Pengiriman berdasarkan salinan atau referensi.
- Pesan berukuran pasti dan variabel.

18.4. Komunikasi Langsung

Proses-proses yang ingin dikomunikasikan harus memiliki sebuah cara untuk memilih satu dengan yang lain. Mereka dapat menggunakan komunikasi langsung/tidak langsung.

Setiap proses yang ingin berkomunikasi harus memiliki nama yang bersifat eksplisit baik penerimaan atau pengirim dari komunikasi tersebut. Dalam konteks ini, pengiriman dan penerimaan pesan secara primitive dapat dijabarkan sebagai:

- Send (P, message) - mengirim sebuah pesan ke proses P.
- Receive (Q, message) - menerima sebuah pesan dari proses Q.

Sebuah jaringan komunikasi pada bahasan ini memiliki beberapa sifat, yaitu:

- Sebuah jaringan yang didirikan secara otomatis diantara setiap pasang dari proses yang ingin dikomunikasikan. Proses tersebut harus mengetahui identitas dari semua yang ingin dikomunikasikan.
- Sebuah jaringan adalah terdiri dari penggabungan dua proses.
- Diantara setiap pesan dari proses terdapat tepat sebuah jaringan.

Pembahasan ini memperlihatkan sebuah cara simetris dalam pemberian alamat. Oleh karena itu, baik keduanya yaitu pengirim dan penerima proses harus memberi nama bagi yang lain untuk berkomunikasi, hanya pengirim yang memberikan nama bagi penerima sedangkan penerima tidak menyediakan nama bagi pengirim. Dalam konteks ini, pengirim dan penerima secara sederhana dapat dijabarkan sebagai:

- Send (P, message) - mengirim sebuah pesan kepada proses P.
- Receive (ID, message) - menerima sebuah pesan dari semua proses. Variabel ID diatur sebagai nama dari proses dengan komunikasi.

Kerugian dari kedua cara yang disebutkan diatas adalah adanya keterbatasan modularitas, merubah nama proses mungkin mengharuskan kita untuk merubah semua definisi proses yang lain. Semua referensi kepada nama yang lama harus ditemukan.

18.5. Komunikasi Tidak Langsung

Dengan komunikasi tidak langsung, pesan akan dikirimkan pada dan diterima dari/melalui *mailbox* (Kotak Surat) atau terminal-terminal, sebuah *mailbox* dapat dilihat secara abstrak sebagai sebuah obyek didalam setiap pesan yang dapat ditempatkan dari proses dan dari setiap pesan yang bias dipindahkan. Setiap kotak surat memiliki sebuah identifikasi (identitas) yang unik, sebuah proses dapat berkomunikasi dengan beberapa proses lain melalui sebuah nomor dari *mailbox* yang berbeda. Dua proses dapat saling berkomunikasi apabila kedua proses tersebut sharing *mailbox*. Pengirim dan penerima dapat dijabarkan sebagai:

- Send (A, *message*) - mengirim pesan ke *mailbox* A.

- Receive (A, *message*) - menerima pesan dari *mailbox* A.

Dalam masalah ini, link komunikasi mempunyai sifat sebagai berikut:

- Sebuah link dibangun diantara sepasang proses dimana kedua proses tersebut membagi *mailbox*.
- Sebuah link mungkin dapat berasosiasi dengan lebih dari dua proses.
- Diantara setiap pasang proses komunikasi, mungkin terdapat link yang berbeda-beda, dimana setiap link berhubungan pada satu *mailbox*.

Misalkan terdapat proses P1, P2 dan P3 yang semuanya *share mailbox*. Proses P1 mengirim pesan ke A, ketika P2 dan P3 masing-masing mengeksekusi sebuah kiriman dari A. Proses mana yang akan menerima pesan yang dikirim P1? Jawabannya tergantung dari jalur yang kita pilih:

- Mengizinkan sebuah link berasosiasi dengan paling banyak dua proses.
- Mengizinkan paling banyak satu proses pada suatu waktu untuk mengeksekusi hasil kiriman (*receive operation*).
- Mengizinkan sistem untuk memilih secara mutlak proses mana yang akan menerima pesan (apakah itu P2 atau P3 tetapi tidak keduanya, tidak akan menerima pesan). Sistem mungkin mengidentifikasi penerima kepada pengirim.

Mailbox mungkin dapat dimiliki oleh sebuah proses atau sistem operasi. Jika *mailbox* dimiliki oleh proses, maka kita mendefinisikan antara pemilik (yang hanya dapat menerima pesan melalui *mailbox*) dan pengguna dari *mailbox* (yang hanya dapat mengirim pesan ke *mailbox*). Selama setiap *mailbox* mempunyai kepemilikan yang unik, maka tidak akan ada kebingungan tentang siapa yang harus menerima pesan dari *mailbox*. Ketika proses yang memiliki *mailbox* tersebut diterminasi, *mailbox* akan hilang. Semua proses yang mengirim pesan ke *mailbox* ini diberi pesan bahwa *mailbox* tersebut tidak lagi ada.

Dengan kata lain, mempunyai *mailbox* sendiri yang independent, dan tidak melibatkan proses yang lain. Maka sistem operasi harus memiliki mekanisme yang mengizinkan proses untuk melakukan hal-hal dibawah ini:

- Membuat *mailbox* baru.
- Mengirim dan menerima pesan melalui *mailbox*.
- Menghapus *mailbox*.

Proses yang membuat *mailbox* pertama kali secara default akan memiliki *mailbox* tersebut. Untuk pertama kali, pemilik adalah satu-satunya proses yang dapat menerima pesan melalui *mailbox* ini. Bagaimana pun, kepemilikan dan hak menerima pesan mungkin dapat dialihkan ke proses lain melalui sistem pemanggilan.

18.6. Sinkronisasi

Komunikasi antara proses membutuhkan *place by calls* untuk mengirim dan menerima data *primitive*. Terdapat design yang berbeda-beda dalam implementasi setiap primitive. Pengiriman pesan mungkin dapat diblok (*blocking*) atau tidak dapat dibloking (*nonblocking*) - juga dikenal dengan nama sinkron atau asinkron.

- Pengiriman yang diblok: Proses pengiriman di blok sampai pesan diterima oleh proses penerima (*receiving process*) atau oleh *mailbox*.
- Pengiriman yang tidak diblok: Proses pengiriman pesan dan mengkalkulasi operasi.
- Penerimaan yang diblok: Penerima memblok samapai pesan tersedia.
- Penerimaan yang tidak diblok: Penerima mengembalikan pesan valid atau null.

18.7. Buffering

Baik komunikasi itu langsung atau tak langsung, penukaran pesan oleh proses memerlukan antrian sementara. Pada dasarnya, terdapat tiga jalan dimana antrian tersebut diimplementasikan:

- Kapasitas nol: antrian mempunyai panjang maksimum 0, maka link tidak dapat mempunyai penungguan pesan (*message waiting*). Dalam kasus ini, pengirim harus memblok sampai penerima menerima pesan.
- Kapasitas terbatas: antrian mempunyai panjang yang telah ditentukan, paling banyak n pesan dapat dimasukkan. Jika antrian tidak penuh ketika pesan dikirimkan, pesan yang baru akan menimpa, dan pengirim pengirim dapat melanjutkan eksekusi tanpa menunggu. Link mempunyai kapasitas terbatas. Jika link penuh, pengirim harus memblok sampai terdapat ruang pada antrian.
- Kapasitas tak terbatas: antrian mempunyai panjang yang tak terhingga, maka, semua pesan dapat menunggu disini. Pengirim tidak akan pernah di blok.

18.8. *Mailbox*

Contoh 18.2. *Mailbox*

```
import java.util.*;
public class MessageQueue {
    private Vector q;
    public MessageQueue() { q = new Vector(); }
    // Mengimplementasikan pengiriman nonblocking
    public void send( Object item ) {
        q.addElement( item );
    }
    // Mengimplementasikan penerimaan nonblocking
    public Object receive() {
        Object item;
        if ( q.size() == 0 ) { return null; }
        else {
            item = q.firstElement();
            q.removeElementAt(0);
            return item;
        }
    }
}
```

1. Menunggu sampai batas waktu yang tidak dapat ditentukan sampai terdapat ruang kosong pada *mailbox*.

2. Menunggu paling banyak n milidetik.

3. Tidak menunggu, tetapi kembali (*return*) secepatnya.

4. Satu pesan dapat diberikan kepada sistem operasi untuk disimpan, walaupun *mailbox* yang dituju penuh. Ketika pesan dapat disimpan pada *mailbox*, pesan akan dikembalikan kepada pengirim (*sender*). Hanya satu pesan kepada *mailbox* yang penuh yang dapat diundur (*pending*) pada suatu waktu untuk diberikan kepada *thread* pengirim.

18.9. *Socket Client/Server System*

Dengan makin berkembangnya teknologi jaringan komputer, sekarang ini ada kecenderungan sebuah sistem yang bekerja sama menggunakan jaringan. Dalam topik ini akan kita bahas beberapa metoda komunikasi antar proses yang melibatkan jaringan komputer.

Socket adalah sebuah *endpoint* untuk komunikasi didalam jaringan. Sepasang proses atau *thread* berkomunikasi dengan membangun sepasang socket, yang masing-masing proses memilikinya. Socket dibuat dengan menyambungkan dua buah alamat IP melalui port tertentu. Secara umum socket digunakan dalam *client/server system*, dimana sebuah server akan menunggu client pada port tertentu. Begitu ada client yang mengkontak server maka server akan menyetujui komunikasi dengan client melalui socket yang dibangun.

18.10. *Server* dan *Thread*

Pada umumnya sebuah server melayani client secara konkuren, oleh sebab itu dibutuhkan *thread* yang masing-masing *thread* melayani clientnya masing-masing. Jadi server akan membentuk *thread* baru begitu ada koneksi dari client yang diterima (*accept*).

Server menggunakan *thread* apabila client melakukan koneksi, sehingga server memiliki tingkat reabilitas yang tinggi. Pada sistem yang memiliki banyak pemakai sekaligus *thread* mutlak dibutuhkan, karena setiap pemakai sistem pasti menginginkan respon yang baik dari server.

Java Socket

Java menyediakan dua buah tipe socket yang berbeda dan sebuah socket spesial. Semua soket ini tersedia dalam paket jaringan, yang merupakan paket standar java. Berikut ini soket yang disediakan oleh java:

- Connection-Oriented (TCP) socket, yang diimplementasikan pada kelas *java.net.Socket*
- Connectionless Socket (UDP), yang diimplentasikan oleh kelas *java.net.DatagramSocket*
- Dan yang terakhir adalah *java.net.MulticastSocket*, yang merupakan perluasan (extended) dari Socket UDP. Tipe socket ini memiliki kemampuan untuk mengirim pesan kebanyak client sekaligus (Multicast), sehingga baik digunakan pada sistem yang memiliki jenis layanan yang sama.

Potongan kode diatas memperlihatkan teknik yang digunakan oleh java untuk membuka socket (pada kasus ini server socket). Selanjutnya server dapat berkomunikasi dengan clientnya menggunakan InputStream untuk menerima pesan dan OutputStream untuk mengirim pesan.

Contoh 18.3. ***WebServer***

```
...
public WebServer(int port, String docRoot) throws IOException
{
   this.docRoot =  new File(docRoot);
   if(!this.docRoot.isDirectory())
   {
        throw new IOException(docRoot + " bukan direktori.");
   }
   System.out.println("Menghidupkan Web server ");
   System.out.println("port: " + port);
   System.out.println("docRoot: " + docRoot);
   try
   {
        serverSocket = new ServerSocket(port);
   }
   catch(IOException ioe)
   {
        System.out.println("Port sudah digunakan");
        System.exit(1);
   }
}

public void run()
{
   while(true)
   {
        try{
            System.out.println("Menanti connection ... ");
            Socket socket = serverSocket.accept();
            String alamatClient = socket.getInetAddress().getHostAddress();

            System.out.println("Menangkap connection dari " + alamatClient);
            InputStream inputStream = socket.getInputStream();
            InputStreamReader inputStreamReader = new InputStreamReader(inputStrea
            BufferedReader bufferedReader = new BufferedReader(inputStreamReader);

            OutputStream outputStream = socket.getOutputStream();
            ...
```

Remote Procedure Call

Remote Procedure Call (RPC) adalah sebuah metoda yang memungkinkan kita untuk mengakses sebuah prosedur yang berada di komputer lain. Untuk dapat melakukan ini sebuah komputer (server) harus menyediakan layanan remote prosedur. Pendekatan yang dilakuan adalah, sebuah server membuka socket, menunggu client yang meminta proseduryang disediakan oleh server.

RPC masih menggunakan cara primitive dalam pemrograman, yaitu menggunakan paradigma procedural programming. Hal itu membuat kita sulit ketika menyediakan banyak remote procedure.

RPC menggunakan soket untuk berkomunikasi dengan proses lainnya. Pada sistem seperti SUN, RPC secara default sudah terinstall kedalam sistemnya, biasanya RPC ini digunakan untuk administrasi sistem. Sehingga seorang administrator jaringan dapat mengakses sistemnya dan mengelola sistemnya dari mana saja, selama sistemnya terhubung ke jaringan.

Pembuatan Obyek *Remote*

Pendekatan kedua yang akan kita bahas adalah *Remote Method Invocation* (RMI), sebuah teknik

pemanggilan method remote yang lebih secara umum lebih baik daripada RPC. RMI menggunakan paradigma pemrograman berorientasi obyek (OOP). Dengan RMI memungkinkan kita untuk mengirim obyek sebagai parameter dari *remote method*. Dengan dibolehkannya program java memanggil method pada remote obyek, RMI membuat pengguna dapat mengembangkan aplikasi java yang terdistribusi pada jaringan

Untuk membuat remote method dapat diakses RMI mengimplementasikan *remote object* menggukan stub dan skleton. Stub bertindak sebagai proxy disisi client, yaitu yang menghubungkan client dengan skleton yang berada disisi server. Stub yang ada disisi client bertanggung-jawab untuk membungkus nama method yang akan diakses, dan parameternya, hal ini biasa dikenal dengan marshalling. Stub mengirim paket yang sudah dibungkus ini ke server dan akan di buka (*unmarshalling*) oleh skleton. Skleton akan menerima hasil keluaran yang telah diproses oleh method yang dituju, lalu akan kembali dibungkus (marshal) dan dikirim kembali ke client yang akan diterima oleh stub dan kembali dibuka paketnya (unmarshall).

Untuk membuat remote obyek kita harus mendefinisikan semua method yang akan kita sediakan pada jaringan, setelah itu dapat digunakan RMI compiler untuk membuat stub dan skleton. Setelah itu kita harus mem-binding remote obyek yang kita sediakan kedalam sebuah RMI registry. Setelah itu client dapat mengakses semua remote method yang telah kita sediakan menggunkan stub yang telah dicompile menggunakan RMI compiler terebut.

Akses ke Obyek *Remote*

Sekali obyek didaftarkan ke server, client dapat mengakses remote object dengan menjalankan *Naming.lookup()* method. RMI menyediakan url untuk pengaksesan ke remote obyek yaitu *rmi://host/obyek*, dimana host adalah nama server tempat kita mendaftarkan remote obyek dan obyek adalah parameter yang kita gunakan ketika kita memanggil method *Naming.rebind()*. Client juga harus menginstall RMISecurityManager untuk memastikan keamanan client ketika membuka soket kejaringan.

Java memiliki sistem security yang baik sehingga user dapat lebih nyaman dalam melakukan komunikasi pada jaringan. Selain itu java sudah mendukung pemorograman berorientasi object, sehingga pengembangan software berskala besar sangat dimungkinkan dilakukan oleh java. RMI sendiri merupakan sistem terdistribusi yang dirancang oleh SUN pada platfrom yang spesifik yaitu Java, apabila anda tertarik untuk mengembangkan sistem terdistribusi yang lebih portable dapat digunakan CORBA sebagai solusi alternatifnya.

18.11. Rangkuman

Proses-proses pada sistem dapat dieksekusi secara berkelanjutan. Disini ada beberapa alasan mengapa proses tersebut dapat dieksekusi secara berkelanjutan: pembagian informasi, penambahan kecepatan komputasi, modularitas, dan kenyamanan atau kemudahan. Eksekusi secara berkelanjutan menyediakan sebuah mekanisme bagi proses pembuatan dan penghapusan.

Pengeksekusian proses-proses pada sistem operasi mungkin dapat digolongkan menjadi proses yang mandiri dan kooperasi. Proses kooperasi harus memiliki beberapa alat untuk mendukung komunikasi antara satu dengan yang lainnya. Prinsipnya adalah ada dua rencana komplementer komunikasi: pembagian memori dan sistem pesan. Metode pembagian memori menyediakan proses komunikasi untuk berbagi beberapa variabel. Proses-proses tersebut diharapkan dapat saling melakukan tukar-menukar informasi seputar pengguna variabel yang terbagi ini. Pada sistem pembagian memori, tanggung-jawab bagi penyedia komunikasi terjadi dengan programer aplikasi; sistem operasi harus menyediakan hanya pembagian memori saja. Metode sistem pesan mengijinkan proses-proses untuk tukar-menukar pesan. Tanggung-jawab bagi penyedia komunikasi ini terjadi dengan sistem operasi tersebut.

18.12. Latihan

1. Definisikan perbedaan antara penjadualan short term, medium term dan long term.

2. Jelaskan tindakan yang diambil oleh sebuah kernel ketika context switch antar proses.

3. Informasi apa saja yang disimpan pada tabel proses saat context switch dari satu proses ke proses lain.

4. Di sistem UNIX terdapat banyak status proses yang dapat timbul (transisi) akibat event (eksternal) OS dan proses tersebut itu sendiri. Transisi state apa sajakah yang dapat ditimbulkan oleh proses itu sendiri. Sebutkan!

5. Apa keuntungan dan kekurangan dari:

   - Komunikasi Simetrik dan asimetrik
   - Automatic dan explicit buffering
   - Send by copy dan send by reference
   - Fixed-size dan variable sized messages

6. Jelaskan perbedaan short-term, medium-term dan long-term ?

7. Jelaskan apa yang akan dilakukan oleh kernel kepada context switch ketika proses sedang berlangsung ?

8. Beberapa single-user mikrokomputer sistem operasi seperti MS-DOS menyediakan sedikit atau tidak sama sekali arti dari pemrosesan yang konkuren. Diskusikan dampak yang paling mungkin ketika pemrosesan yang konkuren dimasukkan ke dalam suatu sistem operasi ?

9. Perlihatkan semua kemungkinan keadaan dimana suatu proses dapat sedang berjalan, dan gambarkan diagram transisi keadaan yang menjelaskan bagaimana proses bergerak diantara state.

10. Apakah suatu proses memberikan 'issue' ke suatu disk M/K ketika, proses tersebut dalam 'ready' state, jelaskan ?

11. Kernel menjaga suatu rekaman untuk setiap proses, disebut Proses Control Blocks (PCB). Ketika suatu proses sedang tidak berjalan, PCB berisi informasi tentang perlunya melakukan restart suatu proses dalam CPU. Jelaskan 2 informasi yang harus dipunyai PCB.

18.13. Rujukan

Avi Silberschatz, Peter Galvin, Greg Gagne. Applied Operationg System Concepts 1st Ed. 2000. John Wiley & Sons, Inc.

William Stallings: Operating Systems -- Fourth Edition, Prentice Hall, 2001.

Bab 19. Sinkronisasi

19.1. Konsep Sinkronisasi

Apakah sinkronisasi itu sebenarnya? Dan mengapa kita memerlukan sinkronisasi tersebut? Marilah kita pelajari lebih lanjut mengenai sinkronisasi. Seperti yang telah kita ketahui bahwa proses dapat bekerja sendiri (*independent process*) dan juga dapat bekerja bersama proses-proses yang lain (*cooperating process*). Pada umumnya ketika proses saling bekerjasama (*cooperating process*) maka proses-proses tersebut akan saling berbagi data. Pada saat proses-proses berbagi data, ada kemungkinan bahwa data yang dibagi secara bersama itu akan menjadi tidak konsisten dikarenakan adanya kemungkinan proses-proses tersebut melakukan akses secara bersamaan yang menyebabkan data tersebut berubah, hal ini dikenal dengan istilah *Race Condition*.

19.2. *Race Condition*

Untuk lebih jelasnya marilah kita lihat contoh program java berikut yang memperlihatkan timbulnya *Race Condition*.

Contoh 19.1. Produser/Konsumer

```
01. int counter = 0;
02.
03. //Proses yang dilakukan oleh produsen
04. item nextProduced;
05.
06. while (1)
07. {
08.    while (counter == BUFFER_SIZE) { ... do nothing ... }
09.
10.    buffer[in] = nextProduced;
11.    in = (in + 1) % BUFFER_SIZE;
12.    counter++;
13. }
14.
15. //Proses yang dilakukan oleh konsumen
16. item nextConsumed;
17.
18. while (1)
19. {
20.     while (counter == 0) { ... do nothing ... }
21.     nextConsumed = buffer[out] ;
22.     out = (out + 1) % BUFFER_SIZE;
23.     counter--;
24. }
```

Pada program produser/konsumer tersebut dapat kita lihat pada baris 12 dan baris 23 terdapat perintah `counter++` dan `counter--` yang dapat diimplementasikan dengan bahasa mesin sebagai berikut:

Contoh 19.2. *Counter* (1)

```
01.  //counter++(nilai counter bertambah 1 setiap dieksekusi)
02.  register1 = counter
03.  register1 = register1 + 1
04.  counter = register1
05.  //counter--(nilai counter berkurang 1 setiap dieksekusi)
06.  register2 = counter
07.  register2 = register2 - 1
08.  counter = register2
```

Dapat dilihat jika perintah dari `counter++` dan `counter--` dieksekusi secara bersama maka akan sulit untuk mengetahui nilai dari *counter* sebenarnya sehingga nilai dari *counter* itu akan menjadi tidak konsisten. Marilah kita lihat contoh berikut ini:

Contoh 19.3. *Counter* (2)

```
01. //misalkan nilai awal counter adalah 2
02. produsen: register1 = counter (register1 = 2)
03. produsen: register1 = register1 + 1 (register1 = 3)
04. konsumen: register2 = counter (register2 = 2)
05. konsumen: register2 = register2 - 1 (register2 = 1)
06. konsumen: counter = register2 (counter = 1)
07. produsen: counter = register1 (counter = 3)
```

Pada contoh tersebut dapat kita lihat bahwa *counter* memiliki dua buah nilai yaitu bernilai tiga (pada saat `counter++` dieksekusi) dan bernilai satu (pada saat `counter--` dieksekusi). Hal ini menyebabkan nilai dari *counter* tersebut menjadi tidak konsisten. Perhatikan bahwa nilai dari *counter* akan bergantung dari perintah terakhir yang dieksekusi. Oleh karena itu maka kita membutuhkan sinkronisasi yang merupakan suatu upaya yang dilakukan agar proses-proses yang saling bekerja bersama-sama dieksekusi secara beraturan demi mencegah timbulnya suatu keadaan yang disebut dengan *Race Condition*.

19.3. Problem *Critical Section*

Pada sub pokok bahasan sebelumnya, kita telah mengenal *race condition* sebagai masalah yang dapat terjadi pada beberapa proses yang memanipulasi suatu data secara konkruen, sehingga data tersebut tidak sinkron lagi. Nilai akhirnya akan tergantung pada proses mana yang terakhir dieksekusi.

Maka bagaimana cara menghindari *race condition* ini serta situasi-situasi lain yang melibatkan memori bersama, berkas bersama atau sumber daya yang digunakan bersama-sama? Kuncinya adalah menemukan jalan untuk mencegah lebih dari satu proses melakukan proses tulis atau baca kepada data atau berkas pada saat yang bersamaan. Dengan kata lain, kita membutuhkan *Mutual Exclusion*. *Mutual Exclusion* adalah suatu cara yang menjamin jika ada sebuah proses yang menggunakan variabel atau berkas yang sama (digunakan juga oleh proses lain), maka proses lain akan dikeluarkan dari pekerjaan yang sama.

Sekarang kita akan membahas masalah *race condition* ini dari sisi teknis programming. Biasanya sebuah proses akan sibuk melakukan perhitungan internal dan hal-hal lainnya tanpa ada bahaya

yang menuju ke *race condition* pada sebagian besar waktu. Akan tetapi, beberapa proses memiliki suatu segmen kode dimana jika segmen itu dieksekusi, maka proses-proses itu dapat saling mengubah variabel, mengupdate suatu tabel, menulis ke suatu file, dan lain sebagainya, dan hal ini dapat membawa proses tersebut ke dalam bahaya *race condition*. Segmen kode yang seperti inilah yang disebut *Critical Section*.

19.4. Persyaratan

Solusi untuk memecahkan masalah *critical section* adalah dengan mendesain sebuah protokol di mana proses-proses dapat menggunakannya secara bersama-sama. Setiap proses harus 'meminta izin' untuk memasuki *critical section*-nya. Bagian dari kode yang mengimplementasikan izin ini disebut *entry section*. Akhir dari *critical section* itu disebut *exit section*. Bagian kode selanjutnya disebut *remainder section*.

Struktur umum dari proses Pi yang memiliki segmen *critical section* adalah:

Contoh 19.4. *Critical Section (1)*

```
do {
   entry section
     <emphasis role="strong">critical section</emphasis>
   exit section
   remainder section
} while (1);
```

Solusi dari masalah *critical section* harus memenuhi tiga syarat berikut [Silbeschatz 2004]:

1. *Mutual Exclusion*.

   Jika suatu proses sedang menjalankan *critical section*-nya, maka proses-proses lain tidak dapat menjalankan *critical section mereka*. Dengan kata lain, tidak ada dua proses yang berada di *critical section* pada saat yang bersamaan.

2. Terjadi kemajuan (*progress*).

   Jika tidak ada proses yang sedang menjalankan *critical section*-nya dan ada proses-proses lain yang ingin masuk ke *critical section*, maka hanya proses-proses yang yang sedang berada dalam *entry section* saja yang dapat berkompetisi untuk mengerjakan critical section.

3. Ada batas waktu tunggu (*bounded waiting*).

   Jika seandainya ada proses yang sedang menjalankan *critical section*, maka proses lain memiliki waktu tunggu yang ada batasnya untuk menjalankan *critical section*-nya, sehingga dapat dipastikan bahwa proses tersebut dapat mengakses *critical section*-nya (tidak mengalami *starvation*: proses seolah-olah berhenti, menunggu request akses ke *critical section* diperbolehkan).

19.5. Rangkuman

Critical section adalah suatu segmen kode yang mengakses data yang digunakan secara bersama-sama. Problema *critical section* yaitu bagaimana menjamin bahwa jika suatu proses sedang menjalankan *critical section*, maka proses lain tidak boleh masuk ke dalam *critical section* tersebut.

19.6. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:

   - *"Preemptive Shortest Job First" vs. "Non-preemptive Shortest Job First"*.
   - *Inter Process Communication: "Direct Communication" vs. "Indirect Communication"'*.
   - *Process Synchronization: "Monitor" vs. "Semaphore"*.
   - *"Deadlock Avoidance" vs. "Deadlock Detection"*.

2. Sebutkan dan jelaskan tiga syarat untuk mengatasi problema *critical section*!

   Jawab:

   a. Mutual Exclusion. Jika proses `Pi` sedang menjalankan *critical section* (dari proses `Pi`), maka tidak ada proses-proses lain yang dapat menjalankan *critical section* dari proses-proses tersebut. Dengan kata lain, tidak ada dua proses yang berada di *critical section* pada saat yang bersamaan.

   b. Terjadi kemajuan (progress). Jika tidak ada proses yang sedang menjalankan *critical section*nya dan jika terdapat lebih dari satu proses lain yang ingin masuk ke *critical section*, maka hanya proses-proses yang tidak sedang menjalankan *remainder section*nya yang dapat berpartisipasi dalam memutuskan siapa yang berikutnya yang akan masuk ke *critical section*, dan pemilihan siapa yang berhak masuk ke *critical section* ini tidak dapat ditunda secara tak terbatas (sehingga tidak terjadi *deadlock*).

   c. Ada batas waktu tunggu (*bounded waiting*). Jika seandainya ada proses yang sedang menjalankan *critical section*, maka terdapat batasan waktu berapa lama suatu proses lain harus menunggu giliran untuk mengakses *critical section*. Dengan adanya batas waktu tunggu akan menjamin proses dapat mengakses ke *critical section* (tidak mengalami *starvation*: proses seolah-olah berhenti, menunggu request akses ke *critical section* diperbolehkan).

19.7. Rujukan

FIXME

Bibliografi

[Silberschatz2000] Avi Silberschatz, Peter Galvin, dan Grag Gagne. Hak Cipta © 2000. *Applied Operating Systems*. First Edition. Edisi Pertama. John Wiley & Sons.

[KennethRosen1999] Kenneth H. Rosen. Hak Cipta © 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Stallings2001] William Stallings. Hak Cipta © 2001. *Operating Systems*. Prentice Hall.

[Tanenbaum1992] Andrew S. Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. Prentice-Hall Inc..

Bab 20. Pemecahan Masalah *Critical Section*

20.1. Solusi Untuk Dua Proses

Ada dua jenis solusi masalah *critical section*, yaitu:

1. Solusi perangkat lunak.

   Dengan menggunakan algoritma-alogoritma yang nilai kebenarannya tidak tergantung pada asumsi-asumsi lain, selain bahwa setiap proses berjalan pada kecepatan yang bukan nol.

2. Solusi perangkat keras.

   Tergantung pada beberapa instruksi mesin tertentu, misalnya dengan me-non-aktifkan interupsi atau dengan mengunci suatu variabel tertentu (Lihat: Bab 25, *Bounded Buffer*).

Selanjutnya akan dibahas sebuah algoritma sebagai solusi masalah dari *critical section* yang memenuhi tiga syarat seperti yang telah disebutkan di atas. Solusi ini tidak tergantung pada asumsi mengenai instruksi-instruksi perangkat keras atau jumlah prosesor yang dapat didukung oleh perangkat keras. Namun, kita mengasumsikan bahwa insruksi bahasa mesin yang dasar (instruksi-instruksi primitif seperti `load, store,` dan `test`) dieksekusi secara atomik. Artinya, jika dua instruksi tersebut dieksekusi secara konkuren, hasilnya ekuivalen dengan eksekusi instruksi tersebut secara sekuensial dalam urutan tertentu. Jadi, jika `load` dan `store` dieksekusi secara konkuren, `load` akan mendapatkan salah satu dari nilai yang lama atau nilai yang baru, tetapi tidak kombinasi dari keduanya.

Contoh 20.1. ***Critical Section (2)***

```
 /**
   * Program implementasi CriticalSectionAlgoritma
   * dengan menggunakan elemen flag dan elemen kunci
   * Author: V A Pragantha( vap20@mhs.cs.ui.ac.id)
   *
   **/

public class CriticalSectionAlgoritma
{
   public static void main(String args[])
   {
      int kunci=0;

      Pengguna user1;
      Pengguna user2 ;

      user1 = new Pengguna(kunci,0);
      user2 = new Pengguna(kunci,1);

      user1.setUser(user2);
      user2.setUser(user1);

      user1.start();
      user2.start();
   }
}
```

Contoh 20.2. ***Critical Section (2)***

```
class Pengguna extends Thread
{
   public Pengguna(int elemenKunci,int noStatus)
   {
      noPengguna = noStatus;
      kunci = elemenKunci;
   }
   public void setBagianKritis(int t)
   {
      int others = 1-t;
      kunci = others;

      if(t==noPengguna)
      {
         butuh = true;

         System.out.println("\nuser "+noPengguna +
            " mempersilahkan user lain  memakai"+
            "  dan memberikan kunci");
         while( (lain.getFlag() == true)
            &amp;&amp; (lain.getKunci()== others) )
         {
            System.out.println("\nuser lain"+
               " mengambil alih kendali");
            Thread.yield();
         }
         System.out.println("\ntampaknya user lain"+
            " tidak membutuhkan\n");
      }
      else
      {
         lain.setFlag(true);

         while( (butuh == true) && (kunci == others)
            lain.yield();
      }
  }

  public void keluarBagianKritis(int t)
  {
      if( t == noPengguna )
      {
          butuh = false;
      }
      else
      {
          lain.setFlag(false);
      }
  }
```

Contoh 20.3. ***Critical Section (3)***

```
    public void run()
    {
        while(true)
        {
           try
           {
               setBagianKritis(noPengguna);

               System.out.println(" user "+
                   noPengguna+" sedang menggunakan"+
                   ", kunci dipegang oleh user "+
                   kunci);

               keluarBagianKritis(noPengguna);
               System.out.println("user "+ noPengguna+
                   " selesai memakai");
               Thread.sleep(10);
           }
           catch(Exception e)
           {
               System.out.println(e);
               System.exit(0);
           }
        }
    }

    public boolean getFlag()
    {
        return butuh;
    }

    public int getKunci()
    {
        return kunci;
    }

    public void setFlag( boolean flag)
    {
        butuh = flag;
    }

    public void setUser( Pengguna p1)
    {
        lain = p1;
    }

    private boolean butuh;
    private int kunci;
    private int noPengguna;
    private Pengguna lain;
}
```

Untuk mengilustrasikan proses-proses yang akan masuk ke *critical section*, kita mengimplementasikan thread dengan menggunakan `class` Pengguna. `Class` Algoritma123 akan digunakan untuk menjalankan ketiga algoritma tersebut.

Pada algoritma di atas, *thread-thread* digambarkan sebagai pengguna atau *user* dan kami membatasi sebanyak 2 pengguna saja. Kedua pengguna ini pada awalnya belum memiliki kunci tetapi hanya nomor pengguna yang berbeda. Ketika salah satu *user* ingin memasuki *critical section*, maka ia akan

memberikan kuncinya kepada pengguna lain terlebih dahulu sedangkan pengguna lain akan medapatkan giliran dari nomor statusnya beserta kuncinya. Oleh karena itu ia akan diizinkan untuk memasuki *critical section*-nya. Pada program di atas, yang akan mengatur masuknya suatu thread ke dalam *critical section*-nya adalah metoda `setBagianKritis`. Jika salah satu pengguna selesai menjalankan *critical section*-nya, ia tidak akan lagi membutuhkan kuncinya, sehingga butuh akan diset menjadi `false`. Pada program di atas, bagian yang mengatur keluarnya suatu thread dari *critical section*nya adalah `keluarBagianKritis`.

20.2. Algoritma I

Nama algoritma ini adalah algoritma I, idenya dengan memberikan giliran kepada setiap proses untuk memproses critical section-nya secara bergantian. Asumsi yang digunakan disini setiap proses secara bergantian memasuki critical section-nya.

Misal ada n buah proses yang berjalan dalam sebuah komputer dengan PID yang unik. Proses yang berjalan saat itu adalah proses dengan PID 4 maka akan berprilaku dengan pola seperti gambar dibawah ini.

Gambar 20.1. Algoritma 1

```
do
{
    // menunggu giliran
    while(turn != 4);

    // critical section

    // turn diberikan untuk
    // proses selanjutnya

    // remaninder s
```

Statement while(turn != 4) akan memeriksa apakah pada saat itu proses 4 mendapatkan turn, jika tidak maka proses 4 akan busy waiting(lihat kembali bahwa printah while diakhiri dengan ";"). Jika ternyata pada saat itu merupakan giliran proses 4 maka proses 4 akan mengerjakan critical section-nya. Sampai sini jelas terlihat bahwa mutex terpenuhi! Proses yang tidak mendapatkan turn tidak akan dapat mengerjakan critical section-nya dan turn hanya akan diberikan pada satu proses saja.

Setelah proses 4 selesai mengerjakan critical section maka turn diberikan pada proses lainnya (turn = j, j merupakan proses selanjutnya yang dapat mengerjakan critical section). Setelah turn-nya diberikan kepada proses lain, proses 4 akan mengerjakan remainder section. Disini jelas terlihat bahwa syarat bounded waiting jelas terpenuhi. Ingat asumsi yang digunakan dalam algoritma ini adalah setiap proses secar bergantian memasuki critical section-nya, jika pada saat itu proses 4 ternyata belum mau mengrjakan critical section-nya maka proses ke-j tidak akan mendapatkan kesempatan untuk mengerjakan critical section walau saat itu sebenarnya proses ke-j akan memasuki critical section. Artinya syarat progress tidak terpenuhi pada algoritma ini.

20.3. Algoritma 2

Masalah yang terjadi pada algoritma 1 ialah ketika di entry section terdapat sebuah proses yang ingin masuk ke critical section, sementara di critical section sendiri tidak ada proses yang sedang berjalan, tetapi proses yang ada di entry section tadi tidak bisa masuk ke critical section. Hal ini terjadi karena giliran untuk memasuki critical section adalah giliran proses yg lain sementara proses tersebut masih berada di remainder section.

Untuk mengatasi masalah ini maka dapat diatasi dengan merubah variabel trun pada algoritma pertama dengan array

Boolean flag [2];

Elemen array diinisialisasi false. Jika flag[i] true, nilai tersebut menandakan bahwa Pi ready untuk memasuki critical section.

Pada algoritma ini. hal pertama yang dilakukan ialah mengeset proses Pi dengan nilai True, ini menandakan bahwa Pi ready untuk masuk ke critical section. kemudian, Pi memeriksa apakah Pj tidak ready untuk memasukui critical section. Jika Pj ready, maka Pi menunggu sampai Pj keluar dari critical section (flag[j] bernilai false). Ketika keluar dari critcal section, Pi harus merubah nilai flag[i] menjadi false agar prores lain dapat memasuki critical section.

Contoh 20.4. XXX

```
do {
   Flag [i] = true;
   While (flag[j]);
       Critical section
   Flag [i]= false;
       Remainder section
} while (1);
```

Pada algoritma ini, kriteria Mutual-exclusion terpenuhi, tetapi sayang sekali tidak memenuhi kriteria progress. Ilustrasinya seperti di bawah ini.

T0 : Po set flag [0] = true

T1 : Po set flag [1] = true

Dari ilustrasi diatas terlihat bahwa algoritma ini memungkinkan terjadinya nilai true untuk kedua proses, akibatnya tidak ada proses yangakan berhasil memasuki critical section.

Jadi untuk algoritma 2 masih terdapat kelemahan, serti yang terjadi di atas.

20.4. Algoritma 3

Idenya berasal dari algoritma 1 dan 2. Algoritma 3 mengatasi kelemahan pada algoritma 1 dan 2 sehingga progres yang diperlukan untuk mengatasi critical section terpenuhi. Berikut ini code dari algoritma 3:

Contoh 20.5. XXX

```
public class Algorithm_3 implements MutualExclusion {
```

```
   private volatile boolean flag0;
   private volatile boolean flag1;
   private volatile int turn;
   public Algorithm_3() {
      flag0 = false;
      flag1 = false;
      turn = TURN_0;
   }

   public void enteringCriticalSection(int t) {
      int other = 1 - t;
      turn = other;
      if (t == 0) {
         flag0 = true;
         while(flag1 == true && turn == other)
             Thread.yield();
      } else {
         flag1 = true;
         while (flag0 == true && turn == other)
             Thread.yield();
      }
   }

   public void leavingCriticalSection(int t) {
      if (t == 0)
         flag0 = false;
      else
         flag1 = false;
  }
}
```

20.5. Algoritma Tukang Roti

Algoritma ini didasarkan pada algoritma penjadualan yang biasanya digunakan oleh tukang roti, di mana urutan pelayanan ditentukan dalam situasi yang sangat sibuk.

Algoritma ini dapat digunakan untuk memecahkan masalah *critical section* untuk n buah proses, yang diilustrasikan dengan n buah pelanggan. Ketika memasuki toko, setiap pelanggan menerima sebuah nomor. Sayangnya, algoritma tukang roti ini tidak dapat menjamin bahwa dua proses (dua pelanggan) tidak akan menerima nomor yang sama. Dalam kasus di mana dua proses menerima nomor yang sama, maka proses dengan nomor ID terkecil yang akan dilayani dahulu. Jadi, jika `Pi` dan `Pj` menerima nomor yang sama dan `i < j,` maka `Pi` dilayani dahulu. Karena setiap nama proses adalah unik dan berurut, maka algoritma ini dapat digunakan untuk memecahkan masalah *critical section* untuk n buah proses.

Struktur data umum algoritma ini adalah

```
boolean choosing[n];
int number [n];
```

Awalnya, struktur data ini diinisialisasi masing-masing ke `false` dan `0,` dan menggunakan notasi berikut:

- (a, b) < (c, d) jika a < c atau jika a= c dan b < d

- max(a0, ..., an-1) adalah sebuah bilangan k, sedemikian sehingga k >= ai untuk setiap i= 0, ..., n - 1

Contoh 20.6. ***Algoritma Tukang Roti***

```
do {
   choosing[i] = true;
   number[i] = max(number[0], number [1], ..., number [n+1])+1;
   choosing[i] = false;
   for (j=0; j < n; j++) {
      while (choosing[j]);
      while ((number[j]!=0) && ((number[j],j) < number[i],i)));
   }
      <foreignphrase>critical section</foreignphrase>
   number[i] = 0;
      <foreignphrase>remainder section</foreignphrase>
} while (1);
```

20.6. Rangkuman

Solusi dari *critical section* harus memenuhi tiga syarat, yaitu:

1. *mutual exclusion*
2. terjadi kemajuan (*progress*)
3. ada batas waktu tunggu (*bounded waiting*)

Solusi dari *critical section* dibagi menjadi dua jenis, yaitu solusi perangkat lunak dan solusi perangkat keras. Solusi dengan perangkat lunak yaitu dengan menggunakan algoritma 1, algoritma 2 dan algoritma 3 seperti yang telah dijelaskan. Dari ketiga algoritma itu, hanya algoritma 3 yang memenuhi ketiga syarat solusi *critical section*. Untuk menyelesaikan masalah *critical section* untuk lebih dari dua proses, maka dapat digunakan algoritma tukang roti.

20.7. Latihan

1. FIXME

20.8. Rujukan

FIXME

Bibliografi

[Silberschatz2000] Avi Silberschatz, Peter Galvin, dan Grag Gagne. Hak Cipta © 2000. *Applied Operating Systems*. First Edition. Edisi Pertama. John Wiley & Sons.

[KennethRosen1999] Kenneth H. Rosen. Hak Cipta © 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Stallings2001] William Stallings. Hak Cipta © 2001. *Operating Systems*. Prentice Hall.

[Tanenbaum1992] Andrew S. Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. Prentice-Hall Inc..

Bab 21. Perangkat Sinkronisasi I

21.1. Peranan Perangkat Keras

Seperti yang telah kita ketahui bahwa untuk tercapainya sinkronisasi, salah satu syaratnya adalah harus tercipta suatu kondisi yang *mutual exclusive*, yaitu suatu kondisi dimana hanya ada sebuah proses yang sedang dieksekusi. Pada pendekatan perangkat keras ini ditekankan bagaimana caranya agar kondisi *mutual exclusive* itu tercapai. Pendekatan dari sisi perangkat keras dapat dibagi menjadi dua:

1. *Processor Synchronous*
2. *Memory Synchronous*

Processor Synchronous

Central Processing Unit (CPU) mempunyai suatu mekanisme yang dinamakan interupsi. Di dalam sistem operasi, mekanisme ini digunakan secara intensif, atau dengan kata lain, banyak Konsep sistem operasi yang menggunakan mekanisme ini. Sebagai contoh: *system call*, *process scheduling*, dsb.

Berbicara mengenai sinkronisasi berarti kita mengasumsikan bahwa akan ada dua atau lebih proses yang sedang berjalan di komputer secara *concurrent*, atau dengan kata lain konsep *time-shared* sudah diimplementasikan di sistem operasi.

Sistem *time-shared* yang sering diimplementasikan dengan algoritma RR (*Round Robin*), memanfaatkan mekanisme interupsi di CPU. Jadi di dalam RR ada suatu satuan waktu yg dinamakan kuantum yang mana setiap kuantum dibatasi oleh satu interupsi perangkat lunak.

Teknisnya, akan ada suatu interupsi -- yang biasanya adalah timer interupsi -- yang secara berkala akan menginterupsi sistem. Pada saat interupsi dilakukan sistem operasi akan segera melakukan proses pergantian dari proses yang satu ke proses yang lainnya sesuai dengan algoritma.

Seperti yang telah diketahui bahwa untuk menghentikan instruksi tersebut kita memerlukan suatu mekanisme yang terdapat pada sistem operasi (baca mengenai *process scheduling*). Dan mekanisme tersebut sangat bergantung kepada mekanisme interupsi dari perangkat keras. Sehingga, jika kita dapat menon-aktifkan interupsi pada saat sebuah proses berada di dalam *critical section* maka permasalahan dari sinkronisasi dapat diselesaikan.

Contoh 21.1. *Critical Section*

```
00  mainModul:
01          CLI           ' masuk ke Critical Section dengan cara
02                        ' men-disable interupsi
03          ADD r1,r2     ' Critical Section
04          ....          ' Critical Section
05          SBI           ' pergi dari Critical Section dengan cara
06                        ' meng-enable interupsi
07          ....          ' Remainder Section
```

Ternyata para perancang komputer melihat celah ini, sehingga saat ini hampir semua komputer yang ada telah mengimplementasikan instruksi mesin yang akan menon-aktifkan serfis interupsi, dan terdapat instruksi mesin lain yang kemudian akan mengaktifkan interupsi tersebut.

Sebagai contoh sederhana, kita akan melihat contoh program dari prosesor Atmel ARM [tm] (contoh

ini diambil karena prosesor ini mudah didapatkan dan harganya tidak terlalu mahal, serta memiliki dev-kit, silahkan merujuk ke http://www.atmel.com [http://www.atmel.com]).

Pada baris ke 01, prosesor akan menon-aktifkan interupsi, yang menyebabkan instruksi-instruksi berikutnya tidak akan terganggu oleh interupsi. Kemudian setelah setelah baris 03 dieksekusi maka proses akan keluar dari *critical section*, yang menyebabkan prosesor mengaktifkan kembali interupsi dan mekanisme *scheduling* di sistem operasi dapat berjalan kembali.

Terlihat bahwa dengan mekanisme ini kita sudah cukup mengatasi isu yang ada. Tetapi ternyata mekanisme ini tidak dapat diterapkan dengan baik di lingkungan *multiprocessor*. Hal ini disebabkan jika kita menon-aktifkan interupsi, maka yang akan dinon-aktifkan hanyalah **satu** prosesor saja, sehingga dapat mengakibatkan terjadinya hal-hal yang tidak diinginkan.

Memory Synchronous

Dilihat dari nama mekanismenya, maka kita sudah dapat memprediksi bahwa mekanisme ini menggunakan jasa dari memori. Hal tersebut benar adanya, mekanisme *memory synchronous* memakai suatu nilai yang disimpan di dalam memori, dan jika suatu proses berhasil mengubah nilai ini, maka proses tersebut akan meneruskan ke instruksi selanjutnya. Tetapi jika tidak, maka proses ini akan berusaha terus untuk mengubah nilai tersebut.

Jika dilihat dari paragraf di atas, mekanisme ini lebih cocok dikategorikan sebagai pendekatan dari perangkat lunak. Tetapi, jika kita perhatikan lebih lanjut, ternyata mekanisme ini memerlukan jasa dari perangkat keras. Mekanisme ini memiliki suatu syarat yang harus dipenuhi agar dapat berjalan sesuai dengan yang diinginkan yaitu perlunya perangkat keras mempunyai kemampuan untuk membuat suatu instruksi dijalankan secara **atomik**. Pengertian dari instruksi atomik adalah satu atau sekelompok instruksi yang tidak dapat diberhentikan sampai instruksi tsb selesai. Detil mengenai hal ini akan dibicarakan di bagian-bagian selanjutnya.

Sebagai contoh, kita dapat memperhatikan contoh program Java[tm] yang ada di bawah ini:

Contoh 21.2. *testANDset*

```
00 boolean testAndSet( boolean  variable[] )
01      {
02          boolean t = variable[0];
03           variable[0] = true;
04          return t;
05       }
         .....
56       while (testAndSet(lock)) { /* do nothing  */ }
57         // Critical Section
58       Lock[0] = false;
59
// Remainder Section
```

Metoda testAndSet haruslah bersifat atomik, sehingga method ini dianggap sebagai satu instruksi mesin. Perhatikan pada baris 56 dimana method ini dipakai. Pada baris ini proses berusaha untuk mengubah nilai dari variable reference lock. Jikalau ia tidak berhasil maka akan terus mencoba, tapi jika berhasil maka proses akan masuk ke bagian kritis dan setelah ini proses akan mengubah nilai dari lock sehingga memberikan kemungkinan proses lain untuk masuk.

Janganlah bingung dengan lock, boolean[], yang terkesan aneh. Hal ini bukanlah bagian dari sinkronisasi tetapi hanyalah suatu bagian dari konsep *pass-by-reference* dan *pass-by-value* dari Java[tm], untuk lebih lanjut mengenai konsep ini dapat dibaca buku-buku programming Java[tm]. Satu catatan di sini adalah, contoh ini hanyalah sebuah ilustrasi dan tidak dapat dicompile dan dijalankan, karena Java[tm] konsep instruksi atomik di Java[tm] bersifat transparan dari sisi programer (akan dijelaskan pada bagian-bagian selanjutnya).

Keunggulan dari *memory synchronous* adalah pada lingkungan *multiprocessor*, semua processor

akan terkena dampak ini. Jadi semua proses yang berada di processor, yang ingin mengakses *critical section*, meski pun berada di processor yang berbeda-beda, akan berusaha untuk mengubah nilai yang dimaksud. Sehingga semua processor akan tersinkronisasi.

21.2. Instruksi Atomik

Seperti yang telah dijelaskan pada bagian sebelumnya, instruksi atomik adalah satu atau sekelompok instruksi yang tidak dapat diberhentikan sampai instruksi tersebut selesai. Kita telah memakai instruksi ini pada method testAndSet.

Instruksi yang dimaksud di sini adalah instruksi-instruksi pada high-level programming, bukanlah pada tingkat instruksi mesin yang memang sudah bersifat atomik. Sebagai contoh: i++ pada suatu bahasa pemrograman akan diinterpertasikan beberapa instruksi mesin yang bersifat atomik sebagai berikut.

Contoh 21.3. XXX

```
00  Load  R1,i  ' load nilai i ke register 1
01  Inc   R1    ' tambahkan nilai register 1 dengan angka 1
02  Store i,R1  ' simpan nilai register 1 ke i
```

Instruksi baris 00-02 bersifat atomik, tetapi i++ tidak bersifat atomik, mengapa? Sebagai contoh kasus, katakanlah sekarang processor baru menyelesaikan baris 01, dan ternyata pada saat tersebut interupsi datang, dan menyebabkan processor melayani interupsi terlebih dahulu. Hal ini menyebabkan terhentinya instruksi i++ sebelum instruksi ini selesai. Jikalau instruksi ini (i++) bersifat atomik, maka ketiga instruksi mesin tsb tidak akan diganggu dengan interupsi.

Perlu diketahui bahwa instruksi ini bukanlah seperti pada *processor synchronous* yang mana akan mematikan interupsi terlebih dahulu, tetapi instruksi ini sudah build-in di processor.

Designer processor dapat mengimplementasi konsep ini dengan dua cara yaitu:

1. mengimplementasi instruksi yang *build-in*
2. mengimplementasi processor mampu membuat suatu instruksi menjadi atomik.

Intel Pentium ternyata memakai cara yang kedua, yaitu dengan adanya suatu perintah LOCK-Assert. Dengan perintah ini maka semua instruksi dapat dijadikan atomik. Sedangkan SPARC dan IBM mengimplementasikan suatu rutin yang bersifat atomik seperti swap dan compareAndSwap.

21.3. Semafor

Telah dikatakan di atas bahwa pada awalnya orang-orang memakai konsep-konsep sinkronisasi yang sederhana yang didukung oleh perangkat keras, seperti pemakaian interupsi atau pemakaian rutin-rutin yang mungkin telah diimplementasi oleh perangkat keras.

Pada tahun 1967, Djikstra mengajukan suatu konsep dimana kita memakai suatu variable integer untuk menghitung banyaknya proses yang sedang aktif atau yang sedang tidur. Jenis variabel ini disebut semafor.

Tahun-tahun berikutnya, semafor banyak dipakai sebagai primitif dari mekanisme sinkronisasi yang lebih tinggi dan kompleks lagi. Sebagai contoh: monitor dari Java[tm]. Selain untuk hal tersebut, kebanyakkan semafor juga digunakan untuk sinkronisasi dalam komunikasi antar device perangkat keras.

Konsep semafor yang diajukan oleh Djikstra terdiri dari dua subrutin yang bernama P dan V. Nama

P dan V berasal dari bahasa Belanda yang berarti Naik dan Turun atau Wait dan Signal. Untuk pembahasan kali ini, kita akan memakai Wait dan Signal.

Sub-rutin wait akan memeriksa apakah nilai dari semafor tersebut di atas 0. Jika ya, maka nilainya akan dikurangi dan akan melanjutkan operasi berikutnya. Jika tidak maka proses yang menjalankan wait akan menunggu sampai ada proses lain yang menjalankan subrutin signal.

Satu hal yang perlu diingat adalah subrutin wait dan signal haruslah bersifat atomik. Di sini kita lihat betapa besarnya dukungan perangkat keras dalam proses sinkronisasi.

Nilai awal dari semaphore tersebut menunjukkan berapa banyak proses yang boleh memasuki *critical section* dari suatu program. Biasanya untuk mendukung sifat *mutual exclusive*, nilai ini diberi 1.

Perlu ditekankan di sini, bahwa semafor bukan digunakan untuk menyelesaikan masalah *critical section* saja, melainkan untuk menyelesaikan permasalahan sinkronisasi secara umum.

21.4. *Wait* dan *Signal*

Seperti yang telah dikatakan di atas, bahwa di dalam subrutin ini, proses akan memeriksa harga dari semafor, apabila harganya 0 atau kurang maka proses akan menunggu, sebaliknya jika lebih dari 0, maka proses akan mengurangi nilai dari semaphore tersebut dan menjalankan operasi yang lain.

Arti dari harga semafor dalam kasus ini adalah hanya boleh satu proses yang dapat melewati subrutin wait pada suatu waktu tertentu, sampai ada salah satu atau proses itu sendiri yang akan memanggil signal.

Bila kita perhatikan lebih kritis lagi, pernyataan "menunggu" sebenarnya masih abstrak. Bagaimanakah cara proses tersebut menunggu, adalah hal yang menarik. Cara proses menunggu dapat dibagi menjadi dua:

1. *spinlock waiting*
2. *non-spinlock waiting*

Spinlock waiting berarti proses tersebut menunggu dengan cara menjalankan perintah-perintah yang tidak ada artinya. Dengan kata lain proses masih *running state* di dalam *spinlock waiting*. Keuntungan *spinlock* pada lingkungan multiprocessor adalah, tidak diperlukan *context switch*. Tetapi *spinlock* yang biasanya disebut *busy waiting* ini menghabiskan *cpu cycle* karena, daripada proses tersebut melakukan perintah-perintah yang tidak ada gunanya, sebaiknya dialihkan ke proses lain yang mungkin lebih membutuhkan untuk mengeksekusi perintah-perintah yang berguna.

Berbeda dengan *spinlock waiting*, *non-spinlock waiting*, memanfaatkan fasilitas sistem operasi. Proses yang melakukan *non-spinlock waiting* akan memblock dirinya sendiri dan secara otomatis akan membawa proses tersebut ke dalam *waiting queue*. Di dalam *waiting queue* ini proses tidak aktif dan menunggu sampai ada proses lain yang membangunkan dia sehingga membawanya ke *ready queue*.

Maka marilah kita lihat listing subrutin dari kedua versi wait. Perbedaan dari kedua subrutin ini adalah terletak pada aksi dari kondisi nilai semafor kurang atau sama dengan dari 0 (nol). Untuk *spinlock*, disini kita dapat melihat bahwa proses akan berputar-putar di while baris 02 (maka itu disebut *spinlock* atau menunggu dengan berputar). Sedangkan pada *non-spinlock*, proses dengan mudah memanggil perintah wait, setelah itu sistem operasi akan mengurus mekanisme selanjutnya.

Jangan bingung dengan kata *synchronized* pada baris 10. Kata ini ada karena memang konsep dari Java[tm], apabila sebuah proses ingin menunggu, maka proses tersebut harus menunggu di suatu obyek. Pembahasan mengenai hal ini sudah diluar dari konteks buku ini, jadi untuk lebih lanjut silahkan merujuk kepada buku Java[tm] pegangan anda.

Contoh 21.4. ***waitSpinLock***

```
00 void waitSpinLock(int semaphore[])
01 {
02    while(semaphore[0] &lt;= 0)
03    { .. Do nothing .. } // spinlock
03    semaphore[0]--;
04 }
05 void synchronized waitNonSpinLock( int semaphore [])
06 {
07    while(semaphore[0]  &lt;= 0)
08    {
09       wait(); // blocks thread
10    }
11    semaphore[0]--;
12 }
```

Karena subrutin wait memiliki dua versi maka hal ini juga berpengaruh terhadap subrutin signal. Subrutin signal akan terdiri dari dua versi sesuai dengan yang ada pada subrutin wait.

Marilah kita lihat listing programnya

Contoh 21.5. ***signalSpinLock***

```
00 void signalSpinLock( int  semaphore [])
01 {
02    semaphore[0]++;
03 }
04
05 void synchronizedsignalNonSpinLock( int semaphore [])
06 {
07    semaphore[0]++;
08    notifyAll(); // membawa waiting thread
09                 // ke ready queue
10 }
```

Letak perbedaan dari kedua subrutin di atas adalah pada notifyAll. NotifyAll berarti membangunkan semua proses yang sedang berada di waiting queue dan menunggu semaphore yang disignal.

Perlu diketahui di sini bahwa setelah semaphore disignal, proses-proses yang sedang menunggu, apakah itu spinlock waiting ataukah *non-spinlock* waiting, akan **berkompetisi** mendapatkan akses semafor tersebut. Jadi memanggil signal bukan berarti membangunkan salah satu proses tetapi memberikan kesempatan proses-proses untuk berkompetisi.

21.5. Jenis Semafor

Ada 2 macam semafor yang cukup umum, yaitu:

1. *Binary semaphore*
2. *Counting semaphore*

Binary semaphore adalah semafor yang bernilai hanya 1 dan 0. Sedangkan *Counting semaphore* adalah semafor yang dapat bernilai 1 dan 0 dan nilai integer yang lainnya.

Banyak sistem operasi yang hanya mengimplementasi *binary semaphore* sebagai primitif, sedangkan *counting semaphore* dibuat dengan memakai primitif ini. Untuk lebih rinci mengenai cara pembuatan *counting semaphore* dapat dilihat pada bagian berikutnya.

Ada beberapa jenis *counting semaphore*, yaitu semafor yang dapat mencapai nilai negatif dan semafor yang tidak dapat mencapai nilai negatif (seperti yang telah dicontohkan pada bagian sebelumnya).

21.6. Solusi Masalah *Critical Section* Dengan Semafor

Seperti yang telah dikatakan di atas bahwa semafor tidak hanya digunakan untuk menyelesaikan masalah *critical section* saja, tetapi menyelesaikan masalah sinkronisasi yang lainnya. Bahkan tidak jarang semafor dijadikan primitif untuk membuat solusi dari masalah sinkronisasi yang lebih kompleks.

Kita telah lihat bagaimana penggunaan semafor untuk menyelesaikan masalah sinkronisasi dengan memakai contoh pada masalah *critical section*. Pada bagian ini, kita akan melihat lebih dekat lagi apa dan seberapa besar sebenarnya peran dari semafor itu sendiri sebagai solusi dalam memecahkan masalah *critical section*.

Lihatlah pada kode-kode di bagian demo. Telitilah, bagian manakah yang harus dieksekusi secara *mutual exclusive*, dan bagian manakah yang tidak. Jika diperhatikan lebih lanjut anda akan menyadari bahwa akan selalu ada **satu pasang** instruksi wait dan signal dari suatu semafor.

Perintah wait digunakan sebagai pintu masuk *critical section* dan perintah signal sebagai pintu keluarnya. Mengapa semafor dapat dijadikan seperti ini? Hal ini disebabkan dengan semafor ketiga syarat utama sinkronisasi dapat dipenuhi.

Seperti yang telah dijelaskan pada bagian sebelumnya, agar *critical section* dapat terselesaikan ada tiga syarat yaitu:

1. *Mutual exclusive*
2. *Make progress*
3. *Bounded waiting*

Sekarang marilah melihat lagi listing program yg ada di bagian sebelumnya mengenai wait dan signal. Jika nilai awal dari semafor diberikan 1, maka artinya adalah hanya ada satu proses yang akan dapat melewati pasangan wait-signal. Proses-proses yang lainnya akan menunggu. Dengan kata lain, mekanisme semaphore dengan *policy* nilai diberikan 1, dapat menjamin syarat yang pertama, yaitu *mutual exclusive*.

Bagaimana dengan syarat yang kedua, *make progress*? Sebenarnya pada waktu proses yang sedang berada di dalam *critical section* keluar dari bagian tersebut dengan memanggil signal, proses tersebut **tidak** memberikan akses ke *critical section* kepada proses tertentu yang sedang menunggu tetapi, **membuka** kesempatan bagi proses lain untuk berkompetisi untuk mendapatkannya. Lalu bagaimana jika ada 2 proses yang sedang menunggu dan saling mengalah? mekanisme semafor memungkinkan salah satu pasti ada yang masuk, yaitu yang pertama kali yang berhasil mengurangi nilai semaphore menjadi 0. Jadi di sini semafor juga berperan dalam memenuhi syarat kedua.

Untuk syarat yang ketiga, jelas tampak bahwa semafor didefinisikan sebagai pasangan wait-signal. Dengan kata lain, setelah wait, pasti ada signal. Jadi proses yang sedang menunggu pasti akan mendapat giliran, yaitu pada saat proses sedang berada di *critical section* memanggil signal.

Contoh suatu potongan program critical section yang memakai semaphore dapat dilihat di bawah ini. Catatan bahwa program di bawah ini hanyalah *pseudo code*.

```
00 wait(semaphoreVar)
01 // critical section
02 signal(semaphoreVar)
```

Baris 00 dan 02 menjamin adanya *mutual exclusive*. Sedangkan mekanisme semaphore menjamin kedua syarat yang lainnya.

21.7. Solusi Masalah Sinkronisasi Antar Proses Dengan Semafor

Kadangkala kita ingin membuat suatu proses untuk menunggu proses yang lain untuk menjalankan suatu perintah. Isu yang ada di sini adalah bagaimana caranya suatu proses mengetahui bahwa proses yang lain telah menyelesaikan instruksi tertentu. Oleh karena itu digunakanlah semafor karena semafor adalah solusi yang cukup baik dan mudah untuk mengatasi hal tersebut.

Nilai semaphore diset menjadi 0

```
Proses 1                          Proses 2
56 print "satu"                17  wait(semaphoreVar)
57 signal(semaphoreVar)        18  print "dua"
```

siapa pun yang berjalan lebih cepat, maka keluarannya pasti "satu" kemudian diikuti oleh "dua". Hal ini disebabkan karena jika proses 2 berjalan terlebih dahulu, maka proses tersebut akan menunggu (nilai semafor = 0) sampai proses 1 memanggil signal. Sebaliknya jika proses 1 berjalan terlebih dahulu, maka proses tersebut akan memanggil signal untuk memberikan jalan terlebih dahulu kepada proses 2.

21.8. *Counting Semaphore* Dari *Binary Semaphore*

Pembuatan *counting semaphore* banyak dilakukan para programer untuk memenuhi alat sinkronisasi yang sesuai dengannya. Seperti yang telah dibahas di atas, bahwa *counting semaphore* ada beberapa macam. Pada bagian ini, akan dibahas *counting semaphore* yang memperbolehkan harga negatif.

Listing program di bawah ini diambil dari buku Silberschatz.

```
00 binary-semaphore S1,S2;
01 int C;
```

Subrutin waitC dapat dilihat di bawah ini:

```
02 wait (S1);
03 C--;
04 if ( C < 0 ) {
05     signal (S1);
06     wait (S2);
07 }
08 signal (S1);
```

subrutin signalC dapat dilihat di bawah ini:

```
09 wait (S1);
10 C++;
11 if (C <= 0)
12   signal (S2);
13 else
14    signal (S1);
```

Kita memerlukan dua *binary semaphore* pada kasus ini, maka pada baris 00 didefinisikan dua *binary semaphore*. Baris 01 mendefinisikan nilai dari semafor tersebut. Perlu diketahui di sini bahwa waitC adalah wait untuk *counting semaphore*, sedangkan wait adalah untuk *binary semaphore*.

Jika diperhatikan pada subrutin waitC dan signalC di awal dan akhir diberikan pasangan wait dan signal dari *binary semaphore*. Fungsi dari *binary semaphore* ini adalah untuk menjamin *critical section* (instruksi wait dan signal dari semafor bersifat atomik, maka begitu pula untuk waitC dan signalC, jadi kegunaan lain semafor adalah untuk membuat suatu subrutin bersifat atomik).

Binary semaphore S2 sendiri digunakan sebagai tempat menunggu giliran proses-proses. Proses-proses tersebut menunggu dengan cara *spinlock* atau *non-spinlock* tergantung dari implementasi *binary semaphore* yang ada.

Perhatikan baris 03 dan 04. Baris ini berbeda dengan apa yang sudah dijabarkan pada bagian sebelumnya. Karena baris ini maka memungkinkan nilai semafor untuk menjadi negatif. Lalu apa artinya bagi kita? Ternyata nilai negatif mengandung informasi tambahan yang cukup berarti bagi kita yaitu bila nilai semafor negatif, maka absolut dari nilai tersebut menunjukkan banyaknya proses yang sedang menunggu atau wait. Jadi arti baris 11 menyatakan bahwa bila ada proses yang menunggu maka semua proses dibangunkan untuk berkompetisi.

Mengapa pada baris 05 dilakukan signal untuk S1? Alasannya karena seperti yang telah kita ketahui bahwa semaphore menjamin ketiga sifat dari *critical section*. Tetapi adalah tidak relevan bila pada saat waktu menunggu, waitC masih mempertahankan mutual exclusivenya. Bila hal ini terjadi, proses lain tidak akan dapat masuk, sedangkan proses yang berada di dalam menunggu proses yang lain untuk signal. Dengan kata lain *deadlock* terjadi. Jadi, baris 05 perlu dilakukan untuk menghilangkan sifat *mutual exclusive* pada saat suatu proses menunggu.

Pada baris 12 hanya menyatakan signal untuk S2 saja. Hal ini bukanlah merupakan suatu masalah, karena jika signal S2 dipanggil, maka pasti ada proses yang menunggu akan masuk dan meneruskan ke instruksi 07 kemudian ke instruksi 08 di mana proses ini akan memanggil signal S1 yang akan mewakili kebutuhan di baris 12.

21.9. Pemrograman *Windows*

Win32API (Windows 32 bit *Application Programming Interface*), menyediakan fungsi-fungsi yang berkaitan dengan semafor. Fungsi-fungsi yang ada antara lain adalah membuat semaphore dan menambahkan semafor.

Hal yg menarik dari semaphore yang terdapat di Windows [tm] adalah tersedianya dua jenis semafor yaitu, *Binary semaphore* dan *counting semaphore*. Pada Windows[tm] selain kita dapat menentukan nilai awal dari semafor, kita juga dapat menentukan nilai maksimal dari semafor. Setiap thread yang menunggu di semafor pada Windows[tm] menggunakan metode antrian FIFO (*First In First Out*.)

21.10. Rangkuman

Hardware merupakan faktor pendukung yang sangat berperan dalam proses sinkronisasi. Banyak

dari para perancang prosesor yang membuat fasilitas *atomic* instruction dalam produknya. Ada 2 metode dalam sinkronisasi hardware, yaitu: *Processor Synchronous* dan *Memory Synchronous*. Semafor merupakan konsep yang dibuat oleh Djikstra dengan mengandalkan sebuah variable integer dan fasilitas atomic instruction dari prosesor. Semafor merupakan primitif dalam pembuatan alat sinkronisasi yang lebih tinggi lagi. Semafor dapat menyelesaikan permasalahan seperti: *Critical section*, sinkronisasi baris, *counting semaphore*, *Dining philosopher*, *readers-writers*, dan *producer-consumer*. Semafor banyak dipakai oleh para programer, sebagai contoh dapat dilihat di pemrograman Win32API. Tetapi ternyata Java[tm] tidak menggunakan semaphore secara explisit namun memakai konsep monitor yang dibangun dari semafor ini.

21.11. Latihan

1. Tunjukkan, bahwa jika operasi wait dan signal tidak dieksekusi secara atomik, maka terjadi pelanggaran terhadap mutual exclusion.

2. Apakah arti dari busy waiting? Apakah ada jenis waiting yang lain dalam sistem operasi? Dapatkah busy waiting dihindari? Jelaskan jawaban anda

3. Jelaskan mengapa spinlock tidak sesuai untuk sistem uniprocessor tetapi sesuai untuk sistem multiprocessor.

4. Masalah Cigarette-Smoker

   Ada proses tiga perokok dan proses satu agen. Setiap perokok terus menerus menggulung rokok dan menghisapnya. Tapi untuk menggulung dan menghisap rokok, perokok membutuhkan tiga bahan: tembakau, kertas, dan korek api.

   Salah satu proses perokok mempunyai kertas, perokok lain mempunyai tembakau, dan perokok ketiga mempunyai korek api. Agen mempunyai stok tidak terbatas dari ketiga bahan tsb. Agen menempatkan dua dari bahan-bahan tsb di atas meja. Perokok yang mempunyai bahan yang tidak ada di atas meja akan membuat dan menghisap rokok, memberi tahu agen bila sudah selesai. Agen lalu menempatkan dua bahan dari ketiga bahan lagi, dan siklusnya berulang. Tulis sebuah program untuk mensinkronisasikan agen dan para perokok tsb.

21.12. Rujukan

FIXME

Bibliografi

[Silberschatz2000] Avi Silberschatz, Peter Galvin, dan Grag Gagne. Hak Cipta © 2000. *Applied Operating Systems*. First Edition. Edisi Pertama. John Wiley & Sons.

[KennethRosen1999] Kenneth H. Rosen. Hak Cipta © 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Stallings2001] William Stallings. Hak Cipta © 2001. *Operating Systems*. Prentice Hall.

[Tanenbaum1992] Andrew S. Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. Prentice-Hall Inc..

Bab 22. Perangkat Sinkronisasi II

22.1. Latar Belakang

Sejauh ini, kita telah mengenal semafor sebagai perangkat keras sinkronisasi yang ampuh, lalu mengapa kita membutuhkan bahasa pemrograman untuk melakukan sinkronisasi? Alasan yang sederhana adalah karena ternyata implementasi semafor memiliki beberapa kelemahan. Kelemahan yang pertama adalah kenyataan bahwa semafor memerlukan implementasi di tingkat rendah, sesuatu hal yang kompleks untuk dilakukan kebanyakan orang. Bahasa pemrograman lebih mudah untuk dipelajari dan diimplementasikan. Kode semafor terdistribusi dalam seluruh program sehingga menyulitkan pemeliharaan. Hal ini merupakan kelemahan lain dari semafor. Sehingga jelas tampaknya, bahwa kita memerlukan konstruksi tingkat tinggi yang dapat mengatasi atau paling tidak mengurangi kelemahan-kelemahan yang terdapat dalam semafor.

22.2. Transaksi Atomik

Yang dimaksud dengan transaksi atomik adalah suatu transaksi yang dilakukan secara keseluruhan atau tidak dilakukan sama sekali. Sebagai ilustrasi adalah ketika dilakukan transfer dari rekening A ke rekening B terjadi kegagalan listrik, maka lebih baik tidak dilakukan perubahan terhadap balance setiap rekening. Disinilah digunakan instruksi atomik.

Keatomikan (atomicity) merupakan komponen penting dalam menghindari bahaya race condition. Operasi atomik dijamin hanya ada dua kemungkinan keluaran (contohnya berhasil atau gagal) dan ketika banyak proses berusaha melakukan operasi atomik dapat dipastikan hanya satu yang akan berhasil (meskipun semuanya dapat gagal).

Secara tipikal, keatomikan diimplementasikan dengan menyediakan mekanisme yang mencatat transaksi mana yang telah dimulai dan selesai atau dengan membuat salinan data sebelum dilakukan perubahan. Sebagai contoh banyak database mendukung mekanisme commit-rollback dalam penerapan transaksi atomik, dimana bila transaksi berhasil maka dilakukan commit tetapi bila transaksi gagal akan dilakukan rollback ke kondisi awal. Metode ini biasanya menyimpan perubahan dalam sebuah log. Bila sebuah perubahan berhasil dilakukan maka akan disimpan dalam log. Bila terjadi kegagalan maka hal tersebut tidak disimpan dalam log. Bila diperlukan kondisi akan diubah ke kondisi terakhir dari transaksi yang berhasil.

Pada tingkat hardware diperlukan instruksi seperti TestAndSet dan operasi increment/decrement. Bila diperlukan maka dapat dilakukan pencegahan pelayanan interupsi yang terjadi ketika transaksi dijalankan dimana transaksi tersebut harus selesai dijalankan barulah interupsi dilayani.

22.3. *Critical Region*

Critical region adalah bagian dari program dan diamanatkan untuk selalu berada dalam keadaan mutual exclusion. Perbedaan critical region ini dengan mutual exclusion biasa yang dibahas sebelumnya adalah critical region diimplementasikan oleh compiler. Keuntungan menggunakan ini adalah programer tidak perlu lagi mengimplementasikan algoritma yang rumit untuk mendapatkan mutual exclusion.

Pada critical region memiliki sebuah komponen boolean yang mentest apakah bagian dari program boleh masuk kedalam state critical region atau tidak. Jika nilai boolean ini true maka proses boleh masuk ke critical region. Jika boolean ini bernilai false bagian yang ini akan dimasukan kedalam sebuah antrian sampai nilai boolean ini bernilai true.

Dalam critical region dikenal ada 2 antrian: main queue dan event queue. Main queue berfungsi untuk menampung proses yang akan memasuki critical region hanya saja critical region masih digunakan oleh proses lain. Event queue berguna untuk menampung proses yang tidak dapat memasuki critical region karena nilai boolennya bernilai false.

22.4. Monitor

Konsep monitor diperkenalkan pertama kali oleh Hoare (1974) dan Brinch Hansen (1975) untuk mengatasi beberapa masalah yang timbul ketika memakai semafor.

Monitor merupakan kumpulan dari prosedur, variabel, dan struktur data dalam satu modul. monitor hanya dapat diakses dengan menjalankan fungsinya. Kita tidak dapat mengambil variabel dari monitor tanpa melalui prosedurnya. Hal ini dilakukan untuk melindungi variabel dari akses yang tidak sah dan juga mengurangi terjadinya error.

Monitor mungkin bisa dianalogikan dengan sebuah sekretariat di dalam sebuah fakultas. Dengan sekretariat sebagai suatu monitor dan mahasiswa, dosen sebagai proses. Dan informasi akademik (jadual, nilai, jadual dosen etc) sebagai variabel. Bila seorang mahasiswa ingin mengambil transkip nilainya dia akan meminta kepada petugas sekretariat mengambilnya sendiri dan mencarinya sendiri, berapa besar kemungkinan kerusakan yang bisa ditimbulkan dengan mengambilnya secara langsung? Jika meminta kepada petugas sekretariat maka petugas akan melakukan berbagi kegiatan untuk memberikan transkip. Dan kerusakan terhadap terhadap dokumen lain bisa dihindari.

Karena monitor berkaitan dengan konsep bahasa pemrograman sehingga kompiler bertanggung-jawab untuk mengkondisikan monitor sebagai mutual eksklusif. Namun, pada kenyataannya tidak semua kompiler dapat menerapkan peraturan mutual eksklusif seperti yang dibutuhkan oleh Monitor tersebut. Mungkin bahasa pemrograman yang sama juga tidak memiliki semafor, tetapi menambahkan semafor jauh lebih mudah.

Jika dianalogikan lagi dengan sekretariat akademik fakultas. Ambil contoh kasus penjadwalan ulang ujian untuk seorang mahasiswa. Dalam proses ini sekretariat tidak mampu langsung memberikan jadwal ulang tetapi perlu bantuan mahasiswa tersebut untuk menghubungi dosen yang bersangkutan, dosen perlu untuk membuat soal yang baru, mahasiswa mungkin perlu mengurus izin ke pembantu dekan 1, dan seterusnya.

22.5. Pemrograman Java^{tm}

Seperti yang telah kita ketahui bahwa satu-satunya alat sinkronisasi yang disediakan oleh Java^{tm} untuk programer adalah kata kunci *synchronized*. Sebenarnya kata kunci ini diilhami dari konsep monitor.

Sedikit mengulang, monitor, konsep sinkronisasi yang sudah sangat kompleks, adalah konsep di mana potongan program ikut di dalamnya. Dalam konsep ini biasanya menggunakan semafor sebagai primitif.

Jadi dengan kata lain, secara implisit Java^{tm} telah menyediakan semafor bagi kita, namun seperti layaknya *Thread Event Dispatcher* di Java^{tm} yang bersifat transparan bagi programer maupun *end-user*, semaphore juga tidak dapat diraba dan diketahui oleh kita di Java^{tm} ini. Hanya pengetahuan mengenai semafor dan monitorlah yang dapat menyimpulkan bahwa Java^{tm} sebenarnya memakai semafor untuk alat sinkronisasinya.

22.6. Masalah Umum Sinkronisasi

Secara garis besar ada tiga masalah umum yang berkaitan dengan sinkronisasi yang dapat diselesaikan dengan menggunakan semafor, ketiga masalah itu adalah:

1. Masalah *Bounded Buffer* (*Producer/Consumer*)
2. Masalah *Readers/Writers*
3. Masalah *Dining Philosophers*

Latar belakang dan solusi dari ketiga permasalahan di atas akan kita pahami lebih lanjut di bab-bab berikutnya.

22.7. Sinkronisasi Kernel Linux

Cara penjadualan kernel pada operasinya secara mendasar berbeda dengan cara penjadualan suatu proses. Terdapat dua cara agar sebuah permintaan akan eksekusi kernel-mode dapat terjadi. Sebuah program yang berjalan dapat meminta service sistem operasi, dari system call atau pun secara implisit (untuk contoh:ketika page fault terjadi). Sebagai alternatif, device driver dapat mengirim interupsi perangkat keras yang menyebabkan CPU memulai eksekusi kernel-define handler untuk suatu interupsi.

Problem untuk kernel muncul karena berbagai tasksmungkin mencoba untuk mengakses struktur data internal yang sama. Jika hanya satu kernel task ditengah pengaksesan struktur data ketika interupsi service routine dieksekusi, maka service routine tidak dapat mengakses atau merubah data yang sama tanpa resiko mendapatkan data yang rusak. Fakta ini berkaitan dengan ide dari *critical section* (Bab 24, *Diagram Graf*).

Sehagai hasilnya, sinkronisasi kernel melibatkan lebih banyak dari hanya penjadualan proses saja. sebuah framework dibutuhkan untuk memperbolehkan kernel's critical sections berjalan tanpa diinterupsi oleh critical section yang lain.

Solusi pertama yang diberikan oleh linux adalah membuat normal kernel code nonpreemptible (Bab 13, *Konsep Penjadualan*). Biasanya, ketika sebuah timer interrupt diterima oleh kernel, membuat penjadualan proses, kemungkinan besar akan menunda eksekusi proses yang sedang berjalan pada saat itu dan melanjutkan menjalankan proses yang lain. Biar bagaimana pun, ketika timer interrupt diterima ketika sebuah proses mengeksekusi kernel-system service routine, penjadualan ulang tidak dilakukan secara mendadak; cukup, kernel need_resched flag terset untuk memberitahu kernel untuk menjalankan penjadualan kembali setelah system call selesai dan control dikembalikan ke user mode.

Sepotong kernel code mulai dijalankan, akan terjamin bahwa itu adalah satu-satunya kernel code yang dijalankan sampai salah satu dari aksi dibawah ini muncul:

- interupsi
- *page fault*
- kernel code memanggil fungsi penjadualan sendiri

Interupsi adalah suatu masalah bila mengandung critical section-nya sendiri. Timer interrupt tidak secara langsung menyebabkan terjadinya penjadualan ulang suatu proses; hanya meminta suatu jadual untuk dilakukan kemudian, jadi kedatangan suatu interupsi tidak mempengaruhi urutan eksekusi dari noninterrupt kernel code. Sekali interrupt serviceselesai, eksekusi akan menjadi lebih simpel untuk kembali ke kernel code yang sedang dijalankan ketika interupsi mengambil alih.

Page faults adalah suatu masalah yang potensial; jika sebuah kernel routine mencoba untuk membaca atau menulis ke user memory, akan menyebabkan terjadinya page fault yang membutuhkan M/K diskuntuk selesai, dan proses yang berjalan akan di tunda sampai M/K selesai. Pada kasus yang hampir sama, jika system call service routine memanggil penjadualan ketika sedang berada di mode kernel, mungkin secara eksplisit dengan membuat direct call pada code penjadualan atau secara implisit dengan memanggil sebuah fungsi untuk menunggu M/K selesai, setelah itu proses akan menunggu dan penjadualan ulang akan muncul. Ketika proses jalan kembali, proses tersebut akan melanjutkan untuk mengeksekusi dengan mode kernel, melanjutkan intruksi setelah call (pemanggilan) ke penjadualan.

Kernel code dapat terus berasumsi bahwa ia tidak akan diganggu (pre-empted) oleh proses lainnya dan tidak ada tindakan khusus dilakukan untuk melindungi critical section. Yang diperlukan adalah critical section tidak mengandung referensi ke user memory atau menunggu M/K selesai.

Teknik kedua yang di pakai Linux untuk critical section yang muncul pada saat interrupt service routines. Alat dasarnya adalah perangkat keras interrupt-control pada processor. Dengan meniadakan interupsi pada saat critical section, maka kernel menjamin bahwa ia dapat melakukan proses tanpa resiko terjadinya ketidak-cocokan akses dari struktur data yang di share.

Untuk meniadakan interupsi terdapat sebuah pinalti. Pada arsitektur perangkat keras kebanyakan, pengadaan dan peniadaan suatu interupsi adalah sesuatu yang mahal. Pada prakteknya, saat interupsi ditiadakan, semua M/K ditunda, dan device yang menunggu untuk dilayani akan menunggu sampai interupsi diadakan kembali, sehingga kinerja meningkat. Kernel Linux menggunakan synchronization architecture yang mengizinkan critical section yang panjang dijalankan untuk seluruh durasinya tanpa mendapatkan peniadaan interupsi. Kemampuan secara spesial berguna pada networking code: Sebuah interupsi pada network device driver dapat memberikan sinyal kedatangan dari keseluruhan paket network, dimana akan menghasilkan code yang baik dieksekusi untuk disassemble, route, dan forward paket ditengah interrupt service routine.

Linux mengimplementasikan arsitektur ini dengan memisahkan interrupt service routine menjadi dua seksi: the top half dan the bottom half. The top half adalah interupsi yang normal, dan berjalan dengan rekursive interupt ditiadakan (interupsi dengan prioritas yang lebih tinggi dapat menginterupsi routine, tetapi interupsi dengan prioritas yang sama atau lebih rendah ditiadakan). The bottom half service routine berjalan dengan semua interupsi diadakan, oleh miniatur penjadualan yang menjamin bahwa bottom halves tidak akan menginterupsi dirinya sendiri. The bottom half scheduler dilakukan secara otomatis pada saat interupt service routine ada.

Pemisahan itu berarti bahwa kegiatan proses yang komplek dan harus selesai diberi tanggapan untuk suatu interupsi dapat diselesaikan oleh kernel tanpa kecemasan tentang diinterupsi oleh interupsi itu sendiri. Jika interupsi lain muncul ketika bottom half dieksekusi, maka interupsi dapat meminta kepada bottom half yang sama untuk dieksekusi, tetapi eksekusinya akan dilakukan setelah proses yang sedang berjalan selesai. Setiap eksekusi dari bottom half dapat di interupsi oleh top half tetapi tidak dapat diinterupsi dengan bottom half yang mirip.

Arsitektur Top-half bottom-half komplit dengan mekanisme untuk meniadakan bottom halver yang dipilih ketika dieksekusi secara normal, foreground kernel code. Kernel dapat meng-codekan critical section secara mudah dengan mengunakan sistem ini: penanganan interupsi dapat mengkodekan *critical section*-nya sebagai bottom halves, dan ketika foreground kernel ingin masuk ke critical section, setiap bottom halves ditiadakan untuk mencegah critical section yang lain diinterupsi. Pada akhir dari critical section, kernel dapat kembali mengadakan bottom halves dan menjalankan bottom half tasks yang telah di masukkan kedalam queue oleh top half interrupt service routine pada saat critical section.

22.8. Rangkuman

Critical Region merupakan bagian kode yang selalu dilaksanakan dalam kondisi mutual eksklusif. Perbedaannya adalah bahwa yang mengkondisikan mutual eksklusif adalah kompiler dan bukan programer sehingga mengurangi resiko kesalahan programer. Monitor merupakan kumpulan dari prosedur, variabel, dan struktur data dalam satu modul. Dengan mempergunakan monitor, sebuah proses dapat memanggil prosedur di dalam monitor, tetapi tidak dapat mengakses struktur data (termasuk variabel- variabel) internal dalam monitor. Dengan karakteristik demikian, monitor dapat mengatasi manipulasi yang tidak sah terhadap variabel yang diakses bersama-sama karena variabel lokal hanya dapat diakses oleh prosedur lokal.

22.9. Latihan

1. Sebutkan keterbatasan penggunaan Monitor!

   Jawab: Tidak semua kompiler dapat menerapkan aturan mutual eksklusif dan tidak dapat diterapkan pada sistem terdistribusi.

22.10. Rujukan

http://www-ist.massey.ac.nz/csnotes/355/lectures/monitors.pdf

Bibliografi

[Silberschatz2000] Avi Silberschatz, Peter Galvin, dan Grag Gagne. Hak Cipta © 2000. *Applied Operating Systems*. First Edition. Edisi Pertama. John Wiley & Sons.

[KennethRosen1999] Kenneth H. Rosen. Hak Cipta © 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Stallings2001] William Stallings. Hak Cipta © 2001. *Operating Systems*. Prentice Hall.

[Tanenbaum1992] Andrew S. Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. Prentice-Hall Inc..

Bab 23. *Deadlock*

23.1. Prinsip dari *Deadlock*

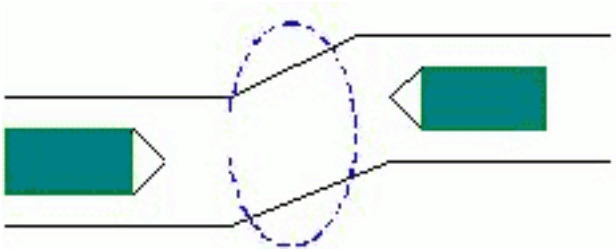
Deadlock dalam arti sebenarnya adalah kebuntuan. Kebuntuan yang dimaksud dalam sistem operasi adalah kebuntuan proses. Jadi *Deadlock* ialah suatu kondisi dimana proses tidak berjalan lagi atau pun tidak ada komunikasi lagi antar proses. *Deadlock* disebabkan karena proses yang satu menunggu sumber daya yang sedang dipegang oleh proses lain yang sedang menunggu sumber daya yang dipegang oleh proses tersebut. Dengan kata lain setiap proses dalam set menunggu untuk sumber yang hanya dapat dikerjakan oleh proses lain dalam set yang sedang menunggu. Contoh sederhananya ialah pada gambar berikut ini.

Contoh 23.1. XXX

```
Proses P1          Proses P2
.....              .....
.....              .....
Receive (P2);      Receive (P1);
.....              .....
.....              .....
Send (P2, M1);     Send (P1, M2);
```

Proses tersebut dapat direpresentasikan dengan gambar sebagai berikut.

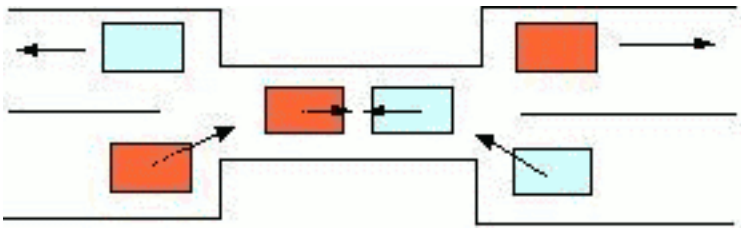
Gambar 23.1. Contoh Deadlock pada rel kereta



Sumber www.tvcc.cc.or.us/ staff/ fuller/ cs160/ chap3/ chap3.html

Dari gambar tersebut bisa dilihat bahwa kedua kereta tersebut tidak dapat berjalan. Karena kedua kereta tersebut saling menunggu kereta yang lain untuk lewat dulu agar keretanya dapat berjalan. Sehingga terjadilah *Deadlock.*

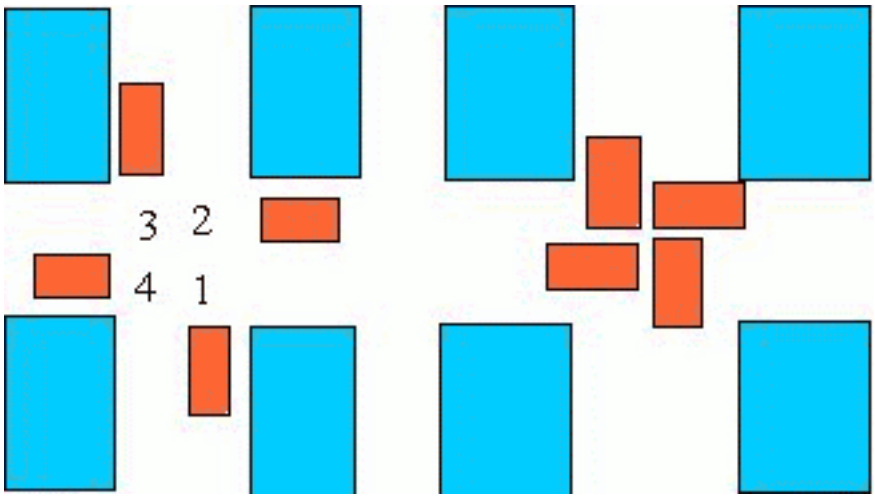
Gambar 23.2. Contoh Deadlock di Jembatan



Disadur dari [Tanenbaum 1992]

Contoh lain yang dapat merepresentasikan *Deadlock* ialah jembatan gantung sebagai berikut. Sehingga orang yang ada di sebelah kiri jembatan tidak dapat melaju sebab terjadi *Deadlock* di tengah jembatan (bagian yang dilingkari). Contoh lain ialah di persimpangan jalan berikut ini:

Gambar 23.3. Contoh Deadlock di Persimpangan Jalan



disadur dari buku Stallings, William, "*Operating Systems -- Fourth Edition*", Prentice Hall, 2001

Dalam kasus ini setiap mobil bergerak sesuai nomor yang ditentukan, tetapi tanpa pengaturan yang benar, maka setiap mobil akan bertemu pada satu titik yang permanen (yang dilingkari) atau dapat dikatakan bahwa setiap mobil tidak dapat meanjutkan perjalanan lagi atau dengan kata lain terjadi *Deadlock*. Contoh lain pada proses yang secara umum terdiri dari tiga tahap, yaitu untuk meminta, memakai, dan melepaskan sumber daya yang di mintanya. Contoh kode-nya:

Contoh 23.2. Lalulintas

```
public class Proses {
    public synchronized void getA() {
        //proses untuk mendapat sumber daya a
    }
    public synchronized void getB(){
        //proses untuk mendapat sumber daya b
    }
    public void releaseA(){
        //proses untuk melepaskan sumber daya a
    }
    public void releaseB(){
        //proses untuk melepaskan sumber daya b
    }
}
public class Coba {
    public static void main(String [] args) {
        Proses P = new Proses();
        Proses Q = new Proses();
        P.getA();
        Q.getB();
        P.getB();
        Q.getA();
    }
}
```

Tanpa adanya perintah untuk mereleased artinya saat P mendapatkan A dan Q mendapatkan B,

tetapi tidak dilepaskan, maka saat P minta B dan Q minta A, maka keduanya akan saling menunggu hingga salah satu melepaskan sumber dayanya, sedangkan kebutuhan P ada pada Q dan Q ada pada P, sehingga terjadi *Deadlock*. Secara umum kejadian ini dapat mudah terjadi dalam pemrograman multi-thread. Sebab ada kemungkinan lebih besar untuk menggunakan sumber daya bersama.

23.2. Sumber Daya yang Bisa Dipakai Berulang-Ulang

Kejadian *Deadlock* selalu tidak lepas dari sumber daya, seperti kita lihat dari contoh-contoh diatas, bahwa hampir seluruhnya merupakan masalah sumber daya yang digunakan bersama-sama. Oleh karena itu, kita juga perlu tahu tentang jenis sumber daya, yaitu: sumber daya dapat digunakan lagi berulang-ulang dan sumber daya yang dapat digunakan dan habis dipakai atau dapat dikatakan sumber daya sekali pakai.

Sumber daya ini tidak habis dipakai oleh proses mana pun.Tetapi setelah proses berakhir, sumber daya ini dikembalikan untuk dipakai oleh proses lain yang sebelumnya tidak kebagian sumber daya ini. Contohnya prosesor, kanal M/K, disk, semafor. Contoh peran sumber daya jenis ini pada terjadinya *Deadlock* ialah misalnya sebuah proses memakai disk A dan B, maka akan terjadi *Deadlock* jika setiap proses sudah memiliki salah satu disk dan meminta disk yang lain. Masalah ini tidak hanya dirasakan oleh pemrogram tetapi oleh seorang yang merancang sebuah sistem operasi. Cara yang digunakan pada umumnya dengan cara memperhitungkan dahulu sumber daya yang digunakan oleh proses-proses yang akan menggunakan sumber daya tersebut. Contoh lain yang menyebabkan *Deadlock* dari sumber yang dapat dipakai berulang-ulang ialah berkaitan dengan jumlah proses yang memakai memori utama. Contohnya dapat dilihat dari kode berikut ini:

Contoh 23.3. P-Q

```
//dari kelas proses kita tambahkan method yaitu meminta
public void meminta (int banyakA) {
    //meminta dari sumber daya a
    if ( banyakA &lt; banyak )
         banyak = banyak - banyakA;
    else
        wait();
}

//mengubah kode pada mainnya sebagai berikut
public static void main ( String [] args ) {
    Proses P = new Proses();
    Proses Q = new Proses();
    P.meminta(80);
    Q.meminta(70);
    P.meminta(60);
    Q.meminta(80);
}

private int banyak = 200;
private int banyakA;
```

Setelah proses P dan Q telah melakukan fungsi meminta untuk pertama kali, maka sumber daya yang tersedia dalam banyak ialah 50 (200-70-80). Maka saat P menjalankan fungsi meminta lagi sebanyak 60, maka P tidak akan menemukan sumber daya dari banyak sebanyak 60, maka P akan menunggu hingga sumber daya yang diminta dipenuhi. Demikian juga dengan Q, akan menunggu hingga permintaannya dipenuhi, akhirnya terjadi *Deadlock*. Cara mengatasinya dengan menggunakan memori maya.

23.3. Sumber Daya Sekali Pakai

Dalam kondisi biasa tidak ada batasan untuk memakai sumber daya apa pun, selain itu dengan tidak terbatasnya produksi akan membuat banyak sumber daya yang tersedia.Tetapi dalam kondisi ini juga dapat terjadi *Deadlock*. Contohnya:

Contoh 23.4. Deadlock

```
//menambahkan method receive dan send
public void receive( Proses p ){
    //method untuk menerima sumber daya
}

public void send ( Proses p ){
    //method untuk memberi sumber daya
}
```

dari kedua fungsi tersebut ada yang bertindak untuk menerima dan memberi sumber daya, tetapi ada kalanya proses tidak mendapat sumber daya yang dibuat sehingga terjadi blok, karena itu terjadi *Deadlock*. Tentu saja hal ini sangat jarang terjadi mengingat tidak ada batasan untuk memproduksi dan mengkonsumsi, tetapi ada suatu keadaan seperti ini yang mengakibatkan *Deadlock*. Hal ini mengakibatkan *Deadlock* jenis ini sulit untuk dideteksi. Selain itu *Deadlock* ini dihasilkan oleh beberapa kombinasi yang sangat jarang terjadi.

23.4. Kondisi untuk Terjadinya *Deadlock*

Menurut Coffman (1971) ada empat kondisi yang dapat mengakibatkan terjadinya *Deadlock*, yaitu:

1. Mutual Eksklusif: hanya ada satu proses yang boleh memakai sumber daya, dan proses lain yang ingin memakai sumber daya tersebut harus menunggu hingga sumber daya tadi dilepaskan atau tidak ada proses yang memakai sumber daya tersebut.

2. Memegang dan menunggu: proses yang sedang memakai sumber daya boleh meminta sumber daya lagi maksudnya menunggu hingga benar-benar sumber daya yang diminta tidak dipakai oleh proses lain, hal ini dapat menyebabkan kelaparan sumber daya sebab dapat saja sebuah proses tidak mendapat sumber daya dalam waktu yang lama

3. Tidak ada *Preemption*: sumber daya yang ada pada sebuah proses tidak boleh diambil begitu saja oleh proses lainnya. Untuk mendapatkan sumber daya tersebut, maka harus dilepaskan terlebih dahulu oleh proses yang memegangnya, selain itu seluruh proses menunggu dan mempersilahkan hanya proses yang memiliki sumber daya yang boleh berjalan

4. *Circular Wait*: kondisi seperti rantai, yaitu sebuah proses membutuhkan sumber daya yang dipegang proses berikutnya

Banyak cara untuk menanggulangi *Deadlock*:

1. Mengabaikan masalah *Deadlock*.

2. Mendeteksi dan memperbaiki

3. Penghindaran yang terus menerus dan pengalokasian yang baik dengan menggunakan protokol

untuk memastikan sistem tidak pernah memasuki keadaan *Deadlock*. Yaitu dengan *Deadlock* avoidance sistem untuk mendata informasi tambahan tentang proses mana yang akan meminta dan menggunakan sumber daya.

4. Pencegahan yang secara struktur bertentangan dengan empat kondisi terjadinya *Deadlock* dengan *Deadlock* prevention sistem untuk memasti- kan bahwa salah satu kondisi yang penting tidak dapat menunggu.

23.5. Mengabaikan Masalah *Deadlock*

Metode ini lebih dikenal dengan Algoritma Ostrich. Dalam algoritma ini dikatakan bahwa untuk menghadapi *Deadlock* ialah dengan berpura-pura bahwa tidak ada masalah apa pun. Hal ini seakan-akan melakukan suatu hal yang fatal, tetapi sistem operasi Unix menanggulangi *Deadlock* dengan cara ini dengan tidak mendeteksi *Deadlock* dan membiarkannya secara otomatis mematikan program sehingga seakan-akan tidak terjadi apa pun. Jadi jika terjadi *Deadlock*, maka tabel akan penuh, sehingga proses yang menjalankan proses melalui operator harus menunggu pada waktu tertantu dan mencoba lagi.

23.6. Mendeteksi dan Memperbaiki

Caranya ialah dengan cara mendeteksi jika terjadi *Deadlock* pada suatu proses maka dideteksi sistem mana yang terlibat di dalamnya. Setelah diketahui sistem mana saja yang terlibat maka diadakan proses untuk memperbaiki dan menjadikan sistem berjalan kembali.

Hal-hal yang terjadi dalam mendeteksi adanya *Deadlock* adalah:

1. Permintaan sumber daya dikabulkan selama memungkinkan.
2. Sistem operasi memeriksa adakah kondisi *circular wait* secara periodik.
3. Pemeriksaan adanya *Deadlock* dapat dilakukan setiap ada sumber daya yang hendak digunakan oleh sebuah proses.
4. Memeriksa dengan algoritma tertentu.

Ada beberapa jalan untuk kembali dari *Deadlock*:

Lewat *Preemption*

Dengan cara untuk sementara waktu menjauhkan sumber daya dari pemakainya, dan memberikannya pada proses yang lain. Ide untuk memberi pada proses lain tanpa diketahui oleh pemilik dari sumber daya tersebut tergantung dari sifat sumber daya itu sendiri. Perbaikan dengan cara ini sangat sulit atau dapat dikatakan tidak mungkin. Cara ini dapat dilakukan dengan memilih korban yang akan dikorbankan atau diambil sumber dayanya utuk sementara, tentu saja harus dengan perhitungan yang cukup agar waktu yang dikorbankan seminimal mungkin. Setelah kita melakukan preemption dilakukan pengkondisian proses tersebut dalam kondisi aman. Setelah itu proses dilakukan lagi dalam kondisi aman tersebut.

Lewat Melacak Kembali

Setelah melakukan beberapa langkah *preemption*, maka proses utama yang diambil sumber dayanya akan berhenti dan tidak dapat melanjutkan kegiatannya, oleh karena itu dibutuhkan langkah untuk kembali pada keadaan aman dimana proses masih berjalan dan memulai proses lagi dari situ. Tetapi untuk beberapa keadaan sangat sulit menentukan kondisi aman tersebut, oleh karena itu umumnya dilakukan cara mematikan program tersebut lalu memulai kembali proses. Meski pun sebenarnya lebih efektif jika hanya mundur beberapa langkah saja sampai *Deadlock* tidak terjadi lagi. Untuk beberapa sistem mencoba dengan cara mengadakan pengecekan beberapa kali secara periodik dan

menandai tempat terakhir kali menulis ke disk, sehingga saat terjadi *Deadlock* dapat mulai dari tempat terakhir penandaannya berada.

Lewat membunuh proses yang menyebabkan *Deadlock*

Cara yang paling umum ialah membunuh semua proses yang mengalami *Deadlock*. Cara ini paling umum dilakukan dan dilakukan oleh hampir semua sistem operasi. Namun, untuk beberapa sistem, kita juga dapat membunuh beberapa proses saja dalam siklus *Deadlock* untuk menghindari *Deadlock* dan mempersilahkan proses lainnya kembali berjalan. Atau dipilih salah satu korban untuk melepaskan sumber dayanya, dengan cara ini maka masalah pemilihan korban menjadi lebih selektif, sebab telah diperhitungkan beberapa kemungkinan jika si proses harus melepaskan sumber dayanya.

Kriteria seleksi korban ialah:

1. Yang paling jarang memakai prosesor
2. Yang paling sedikit hasil programnya
3. Yang paling banyak memakai sumber daya sampai saat ini
4. Yang alokasi sumber daya totalnya tersedkit
5. Yang memiliki prioritas terkecil

23.7. Menghindari *Deadlock*

Pada sistem kebanyakan permintaan terhadap sumber daya dilakukan sebanyak sekali saja. Sistem sudah harus dapat mengenali bahwa sumber daya itu aman atau tidak(dalam arti tidak terkena *Deadlock*), setelah itu baru dialokasikan. Ada dua cara yaitu:

1. Jangan memulai proses apa pun jika proses tersebut akan membawa kita pada kondisi *Deadlock*, sehingga tidak mungkin terjadi *Deadlock* karena ketika akan menuju *Deadlock* sudah dicegah.
2. Jangan memberi kesempatan pada suatu proses untuk meminta sumber daya lagi jika penambahan ini akan membawa kita pada suatu keadaan *Deadlock*

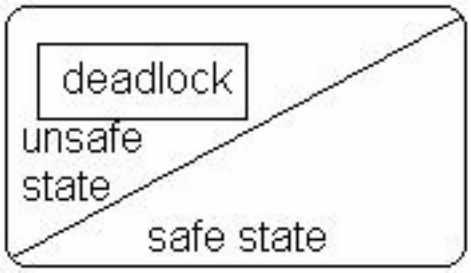
Jadi diadakan dua kali penjagaan, yaitu saat pengalokasian awal, dijaga agar tidak *Deadlock* dan ditambah dengan penjagaan kedua saat suatu proses meminta sumber daya, dijaga agar jangan sampai terjadi *Deadlock*. Pada *Deadlock* avoidance sistem dilakukan dengan cara memastikan bahwa program memiliki maksimum permintaan. Dengan kata lain cara sistem ini memastikan terlebih dahulu bahwa sistem akan selalu dalam kondisi aman. Baik mengadakan permintaan awal atau pun saat meminta permintaan sumber daya tambahan, sistem harus selalu berada dalam kondisi aman.

Kondisi Aman

Saat kondisi aman, maka suatu sistem dapat mengalokasikan sumber daya pada setiap proses (sampai pada batas maksimumnya) dengan urutan tertentu. Dengan gambar sebagai berikut:

Dengan mengenal arti dari kondisi aman ini, kita dapat membuat algoritma untuk menghindari *Deadlock*. Idenya ialah dengan memastikan bahwa sistem selalu berada dalam kondisi aman. Dengan asumsi bahwa dalam kondisi tidak aman terkandung *Deadlock*. Contoh penerapan algoritmanya ialah algoritma bankir.

Gambar 23.4. Kondisi Deadlock Dilihat dari Safe State



disadur dari buku Silberschatz, dkk, *Applied Operating System Concepts*, 2000.

Algoritma Bankir

Menurut Dijstra (1965) algoritma penjadualan dapat menghindari *Deadlock* dan algoritma penjadualan itu lebih dikenal dengan sebutan algoritma bankir. Algoritma ini dapat digambarkan sebagai seorang bankir dikota kecil yang berurusan dengan kelompok orang yang meminta pinjaman. Jadi kepada siapa dia dapat memberikan pinjamannya. Dan setiap pelanggan memberikan batas pinjaman maksimum kepada setiap peminjam dana.

Tentu saja si bankir tahu bahwa si peminjam tidak akan meminjam dana maksimum yang mereka butuhkan dalam waktu yang singkat melainkan bertahap. Jadi dana yang ia punya lebih sedikit dari batas maksimum yang dipinjamkan. Lalu ia memprioritaskan yang meminta dana lebih banyak, sedangkan yang lain disuruh menunggu hingga peminta dana yang lebih besar itu mengembalikan pinjaman berikut bunganya, baru setelah itu ia meminjamkan pada peminjam yang menunggu.

Jadi algoritma bankir ini mempertimbangkan apakah permintaan mereka itu sesuai dengan jumlah dana yang ia miliki, sekaligus memperkirakan jumlah dana yang mungkin diminta lagi. Jangan sampai ia sampai pada kondisi dimana dananya habis dantidak dapat meminjamkan uang lagi. Jika demikian maka akan terjadi kondisi *Deadlock*. Agar kondisi aman, maka asumsi setiap pinjaman harus dikembalikan waktu yang tepat.

Secara umum algoritma bankir dapat dibagi menjadi 4 struktur data:

1. Tersedia: jumlah sumber daya/dana yang tersedia
2. Maksimum: jumlah sumber daya maksimum yang diminta oleh setiap proses
3. Alokasi: jumlah sumber daya yang dibutuhkan oleh setiap proses
4. Kebutuhan: sumber daya yang sedang dibutuhkan oleh setiap proses

23.8. Pencegahan *Deadlock*

Jika pada awal bab ini kita membahas tentang ke-empat hal yang menyebabkan terjadinya *Deadlock*. Maka pada bagian ini, kita akan membahas cara menanggulangi keempat penyebab *Deadlock* itu, sehingga dengan kata lain kita mengadakan pencegahan terhadap *Deadlock*.

Penanggulangannya ialah sebagai berikut:

1. Masalah Mutual Eksklusif Kondisi ini tidak dapat dilarang, jika aksesnya perlu bersifat spesial untuk satu proses, maka hal ini harus di dukung oleh kemampuan sistem operasi. Jadi diusahakan agar tidak mempergunakan kondisi spesial tersebut sehingga sedapat mungkin

Deadlock dapat dihindari.

2. Masalah Kondisi Menunggu dan Memegang Penanggulangan *Deadlock* dari kondisi ini lebih baik dan menjanjikan, asalkan kita dapat menahan proses yang memegang sumber daya untuk tidak menunggu sumber daya laun, kita dapat mencegah *Deadlock*. Caranya ialah dengan meminta semua sumber daya yang ia butuhkan sebelum proses berjalan. Tetapi masalahnya sebagian proses tidak mengetahui keperluannya sebelum ia berjalan. Jadi untuk mengatasi hal ini, kita dapat menggunakan algoritma bankir. Yang mengatur hal ini dapat sistem operasi atau pun sebuah protokol. Hasil yang dapat terjadi ialah sumber daya lebih di-spesifikasi dan kelaparan sumber daya, atau proses yang membutuhkan sumber daya yang banyak harus menunggu sekian lama untuk mendapat sumber daya yang dibutuhkan.

3. Masalah tidak ada *Preemption* Hal ketiga ialah jangan sampai ada *preemption* pada sumber daya yang telah dialokasikan. Untuk memastikan hal ini, kita dapat menggunakan protokol. Jadi jika sebuah proses meminta sumber daya yang tidak dapat dipenuhi saat itu juga, maka proses mengalami preempted. Atau dengan kata lain ada sumber daya dilepaskan dan diberikan ke proses yang menunggu, dan proses itu akan menunggu sampai kebutuhan sumber dayanya dipenuhi.

   Atau kita harus mencek sumber daya yang dimaui oleh proses di cek dahulu apakah tersedia. Jika ya maka kita langsung alokasikan, sedangkan jika tidak tersedia maka kita melihat apakah ada proses lain yang menunggu sumber daya juga. Jika ya, maka kita ambil sumber daya dari proses yang menunggu tersebut dan memberikan pada proses yang meminta tersebut. Jika tidak tersedia juga, maka proses itu harus menunggu. Dalam menunggu, beberapa dari sumber dayanya dapat saja di preempted, tetapi jika ada proses yang memintanya. Cara ini efektif untuk proses yang menyimpan dalam memory atau register.

4. Masalah Circular Wait Masalah ini dapat ditangani oleh sebuah protokol yang menjaga agar sebuah proses tidak membuat lingkaran siklus yang dapat mengakibatkan *Deadlock*

23.9. Rangkuman

Deadlock ialah suatu kondisi permanen dimana proses tidak berjalan lagi atau pun tidak berkomunikasi lagi antar proses. Perebutan sumber daya itu dapat dibagi dua yaitu sumber daya yang dapat dipakai berulang-ulang dan sumber daya yang sekali dibuat dan langsung dipakai.

Sebenarnya *deadlock* dapat disebabkan oleh empat hal yaitu:

1. Proses *Mutual Exclusion*
2. Proses memegang dan menunggu
3. Proses *Preemption*
4. Proses Menunggu dengan siklus *deadlock* tertentu

Penanganan *deadlock* Banyak cara untuk menanggulangi *deadlock*:

1. mengabaikan masalah *deadlock*.
2. mendeteksi dan memperbaiki
3. penghindaran yang terus menerus dan pengalokasian yang baik.
4. pencegahan yang secara struktur bertentangan dengan empat kondisi terjadinya *deadlock*.

23.10. Latihan

1. Proses dapat meminta berbagai kombinasi dari sumber daya dibawah ini: *CDROM, soundcard*, dan *floppy*. Jelaskan tiga macam pencegahan deadlock skema yang meniadakan:

   - *Hold and Wait*
   - *Circular Wait*
   - *No Preemption*

2. Buatlah implementasi dengan menggunakan monitor dari pemecahan Readers/Writers dengan solusi thread pembaca dan penulis mendapatkan prioritas saling bergantian.

3. Jelaskan tentang keempat hal yang menyebabkan *deadlock*?

4. Bagaimana cara mengatasi keempat masalah tersebut?

5. Jelaskan tentang algoritma bankir!

6. Telah dibahas mengenai program dari counting semafor. lihatlah potongan program di bawah ini.

```
Subrutin Wait
02 wait (S1);
03 C--;
04 if ( C < 0 ) {
05     signal (S1);
06     wait (S2);
07 }
08 signal (S1);
Subrutin Signal
09 wait (S1);
10 C++;
11 if (C <= 0)
12   signal (S2);
13 else
14    signal (S1);
```

   a. Apakah yang terjadi bila pada baris nomor 11 diubah menjadi lebih besar dan sama dengan ?

   b. Apakah yang terjadi apabila pada baris 4 ditambahkan sama dengan sehingga menjadi <= ?

   Jawab:

   a. Program tidak akan berjalan karena tanda lebih kecil mempunyai arti bahwa ada proses yang sedang wait. Jadi arti dari baris 11 adalah jika ada proses yang sedang wait maka proses yg sekarang akan memanggil signal S2.

   b. Program tidak akan berjalan **dengan benar.** Sebagai contoh jika nilai awal semafor adalah 1, maka jika ada proses yang memanggil wait, seharusnya proses tsb mengunci semafor tersebut, tetapi kenyataannya semafor tsb akan terhenti seakan - akan ada proses lain yang sudah mengunci (padahal tidak ada).

7. Pernyataan manakah yang benar mengenai deadlock:

   i. Pencegahan deadlock lebih sulit dilakukan (implementasi) daripada menghindari deadlock.

ii. Deteksi deadlock dipilih karena utilisasi dari resources dapat lebih optimal.

iii. Salah satu prasyarat untuk melakukan deteksi deadlock adalah: hold and wait.

iv. Algoritma Banker's (Djikstra) tidak dapat menghindari terjadinya deadlock.

v. Suatu sistem jika berada dalam keadaan tidak aman: "*unsafe*", berarti telah terjadi deadlock.

23.11. Rujukan

FIXME

Bibliografi

[Silberschatz2000] Avi Silberschatz, Peter Galvin, dan Grag Gagne. Hak Cipta © 2000. *Applied Operating Systems*. First Edition. Edisi Pertama. John Wiley & Sons.

[KennethRosen1999] Kenneth H. Rosen. Hak Cipta © 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Stallings2001] William Stallings. Hak Cipta © 2001. *Operating Systems*. Prentice Hall.

[Tanenbaum1992] Andrew S. Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. Prentice-Hall Inc..

Bab 24. Diagram Graf

Sebuah sistem komputer terdiri dari berbagai macam sumber-daya (*resources*), seperti:

1. Fisik (Perangkat, Memori)
2. Logik (Lock, Database record)
3. Sistem Operasi (PCB Slots)
4. Aplikasi (Berkas)

Diantara sumber-daya tersebut ada yang pre-emptable dan ada juga yang tidak. Sumber-daya ini akan digunakan oleh proses-proses yang membutuhkannya. Mekanisme hubungan dari proses-proses dan sumber-daya yang dibutuhkan/digunakan dapat di diwakilkan dengan dengan graf.

Graf adalah suatu struktur diskrit yang terdiri dari vertex dan sisi, dimana sisi menghubungkan vertex-vertex yang ada. Berdasarkan tingkat kompleksitasnya, graf dibagi menjadi dua bagian, yaitu simple graf dan multigraf. Simpel graf tidak mengandung sisi paralel (lebih dari satu sisi yang menghubungkan dua vertex yang sama). Berdasarkan arahnya graf dapat dibagi menjadi dua bagian yaitu graf berarah dan graf tidak berarah. Graf berarah memperhatikan arah sisi yang menghubungkan dua vertex, sedangkan graf tidak berarah tidak memperhatikan arah sisi yang menghubungkan dua vertex.

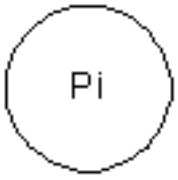
Dalam hal ini akan dibahas mengenai implementasi graf dalam sistem operasi. Salah satunya dalah graf alokasi sumber daya. Graf alokasi sumber daya merupakan graf sederhana dan graf berarah. Graf alokasi sumber daya adalah bentuk visualisasi dalam mendeteksi maupun menyelesaikan masalah. *deadlock*.

24.1. Komponen Graf Alokasi Sumber Daya

Pada dasarnya graf G=(V, E) terdiri dari 2 komponen yaitu vertex dan sisi.

Untuk graf alokasi sumber daya, vertex maupun sisinya dibedakan menjadi beberapa bagian.

Gambar 24.1. Proses Pi



Sumber: Silberschatz, "*Operating System Concepts -- Fourth Edition*", John Wiley & Sons, 2003

Vertex terdiri dari dua jenis, yaitu:

1. Proses P= {P0, P1, P2, P3, , Pi, , Pm}. Terdiri dari semua proses yang ada di sistem. Untuk proses, vertexnya digambarkan sebagai lingkaran dengan nama prosesnya.
2. Sumber daya R= {R0, R1, R2, R3, , Rj, , Rn}. Terdiri dari semua sumber daya yang ada di sistem. Untuk sumber daya, vertexnya digambarkan sebagai segi empat dengan instans yang dapat dialokasikan serta nama sumber dayanya.

Dalam hal ini jumlah proses dan sumber daya tidak selalu sama.

Gambar 24.2. Sumber daya Rj dengan dua instans

Sumber: Silberschatz, "*Operating System Concepts -- Fourth Edition*", John Wiley & Sons, 2003

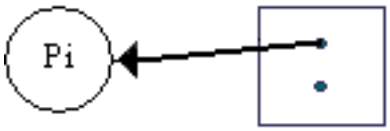
Sisi, E={Pi-> Rj, , Rj-> Pi, } terdiri dari dua jenis, yaitu:

Gambar 24.3. Proses Pi meminta sumber daya Rj

Sumber: Silberschatz, "*Operating System Concepts -- Fourth Edition*", John Wiley & Sons, 2003

1. Sisi permintaan: Pi -> Rj Sisi permintaan menggambarkan adanya suatu proses Pi yang meminta sumber daya Rj.
2. Sisi alokasi: Rj -> Pi. Sisi alokasi menggambarkan adanya suatu sumber daya Rj yang mengalokasikan salah satu instansnya pada proses Pi.

Gambar 24.4. Sumber daya Rj yang mengalokasikan salah satu instansnya pada proses Pi



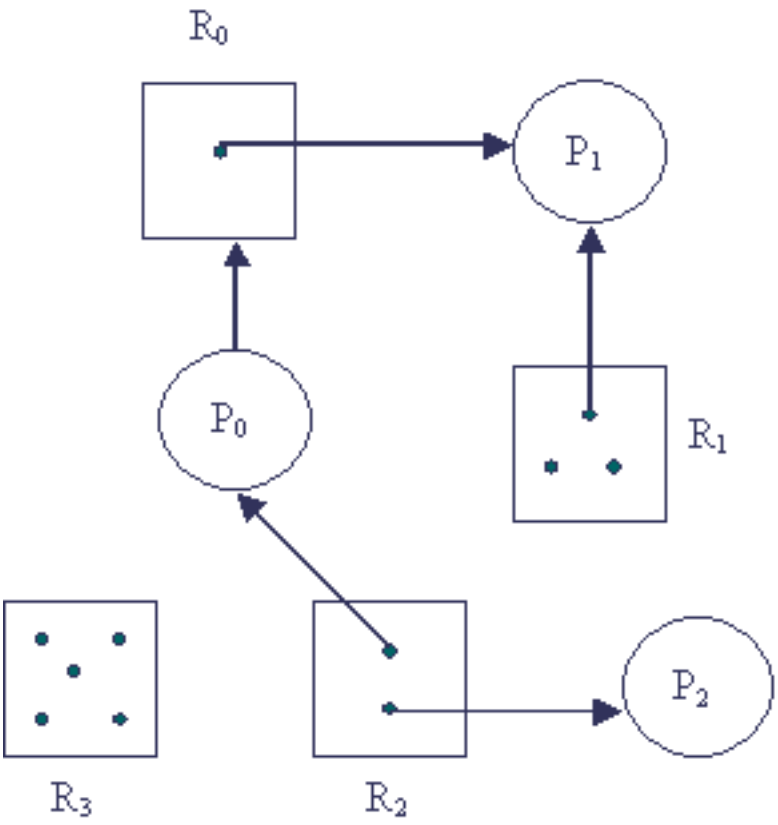
Sumber: Silberschatz, "*Operating System Concepts -- Fourth Edition*", John Wiley & Sons, 2003

Pada graf di atas terdiri dari 7 vertex, V={P0, P1, P2, P3, R0, R1, R3} dan 5 sisi, E= {P0->R0, R0->P1, R1->P1, R2->P0, R2->P2}. Gambar 24.5, “Graf Alokasi Sumber Daya” menunjukkan beberapa hal:

1. P0 meminta sumber daya dari R0.
2. R0 memberikan sumber dayanya kepada P1.
3. R1 memberikan salah satu instans sumber dayanya kepada P1.
4. R2 memberikan salah satu instans sumber dayanya kepada P0.

5. R2 memberikan salah satu instans sumber dayanya kepada P2.

Gambar 24.5. Graf Alokasi Sumber Daya



Setelah suatu proses telah mendapatkan semua sumber daya yang diperlukan maka sumber daya tersebut dilepas dan dapat digunakan oleh proses lain.

24.2. Deteksi *Deadlock* Dengan Graf Alokasi

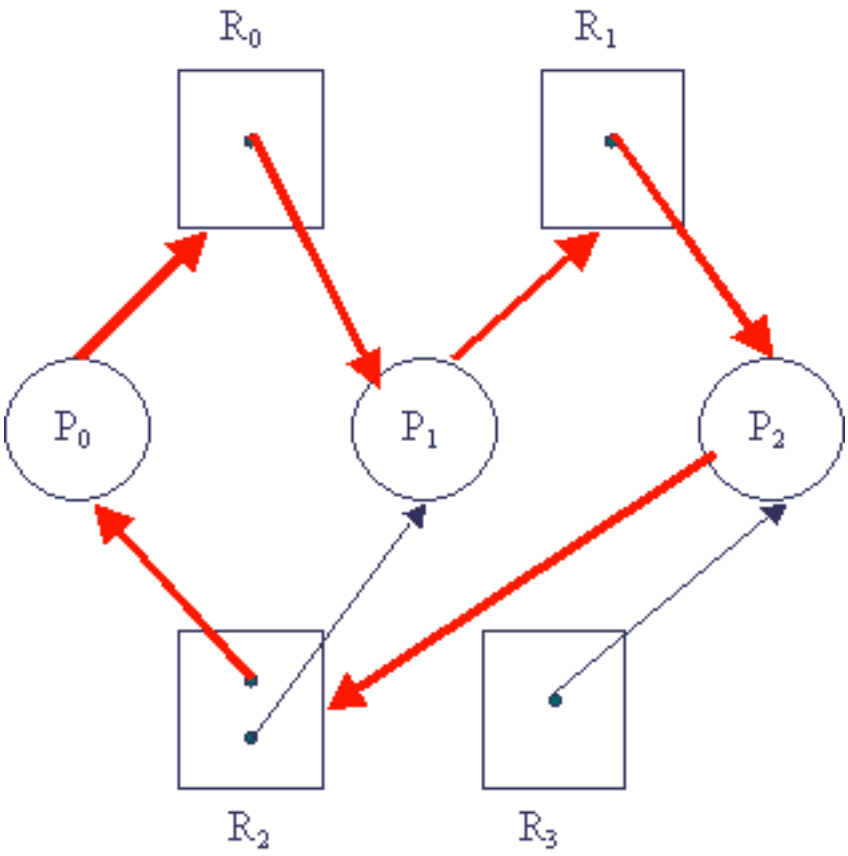
Penghindaran dan pencegahan *deadlock* dalam dilihat pada subbab sebelumnnya. yang telah menjelaskan secara cukup lengkap langkah-langkah untuk menghindari terjadinya *deadlock.*

Untuk mengetahui ada atau tidaknya *deadlock* dalam suatu graf dapat dilihat dari perputaran dan resource yang dimilikinya.

1. Jika tidak ada perputaran berarti tidak *deadlock*.
2. Jika ada perputaran, ada potensi terjadi *deadlock.*
3. Resource dengan instan tunggal DAN perputaran mengakibatkan *deadlock.*

Pada bagian berikut ini akan ditunjukkan bahwa perputaran tidak selalu mengakibatkan *deadlock*. Pada Gambar 24.6, “Graf dengan *deadlock* ” graf memiliki perputaran dan *deadlock* terjadi sedangkan pada Gambar 24.7, “ Tanpa *deadlock* ” graf memiliki perputaran tetapi tidak terjadi *deadlock.*

Gambar 24.6. Graf dengan *deadlock*



Sumber: Silberschatz, "*Operating System Concepts -- Fourth Edition*", John Wiley & Sons, 2003

Gambar 24.6, “Graf dengan *deadlock* ” Terlihat bahwa ada perputaran yang memungkinkan tejadinya *deadlock* dan semua sumber daya memiliki satu instans kecuali sumber daya R2.

Graf di atas memiliki minimal dua perputaran:

1. R2 -> P0 -> R0 -> P1 -> R1 -> P2 -> R2
2. R2 -> P1 -> R1 -> P2 -> R2

Gambar di atas menunjukkan beberapa hal sebagai berikut:

1. P0 meminta sumber daya R0.
2. R0 mengalokasikan sumber dayanya pada P1.
3. P1 meminta sumber daya R1.
4. R1 mengalokasikan sumber dayanya pada P2.
5. P2 meminta sumber daya R2.

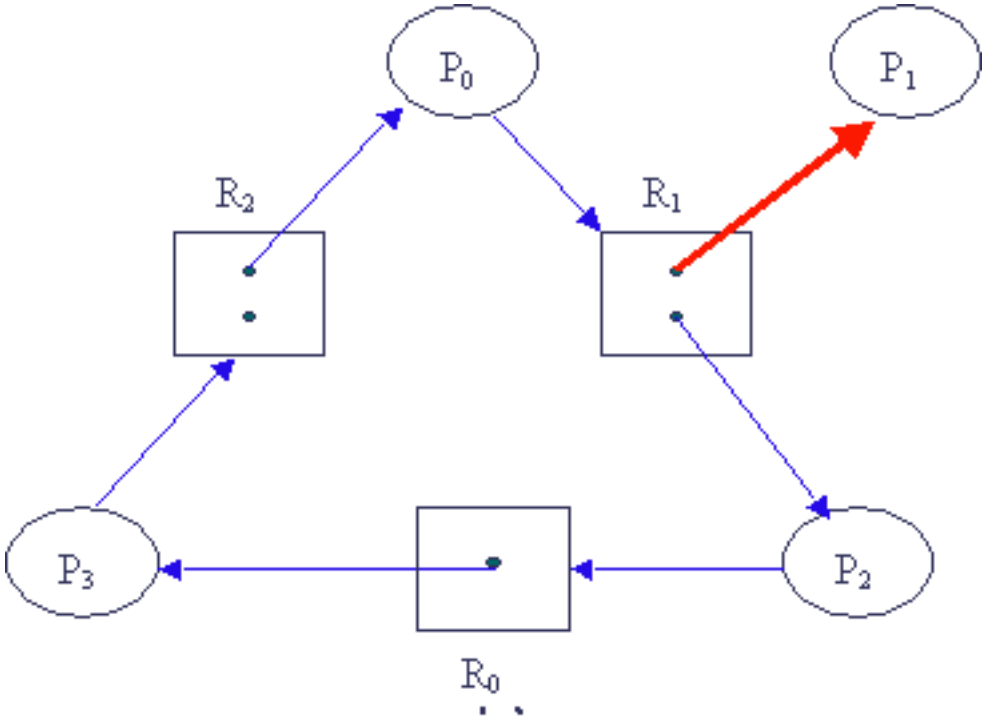
6. R2 mengalokasikan sumber dayanya pada P0 dan P1.

7. R3 mengalokasikan sumber dayanya pada P2.

Hal-hal tersebut dapat mengakibatkan *deadlock* sebab P0 memerlukan sumber daya R0 untuk menyelesaikan prosesnya, sedangkan R0 dialokasikan untuk P1. Di lain pihak P1 memerlukan sumber daya R1 sedangkan R1 dialokasikan untuk P2. P2 memerlukan sumber daya R2 akan tetapi R2 mengalokasikan sumber dayanya pada R3.

Dengan kata lain, tidak ada satu pun dari proses-proses tersebut yang dapat menyelesaikan tugasnya sebab sumber daya yang diperlukan sedang digunakan oleh proses lain. Sedangkan proses lain juga memerlukan sumber daya lain. Semua sumber daya yang diperlukan oleh suatu proses tidak dapat dpenuhi sehingga proses tersebut tidak dapat melepaskan sumber daya yang telah dialokasikan kepadanya. Dan terjadi proses tunggu-menunggu antarproses yang tidak dapat berakhir. Inilah yang dinamakan *deadlock*.

Gambar 24.7. Tanpa *deadlock*



Sumber: Silberschatz, "*Operating System Concepts -- Fourth Edition*", John Wiley & Sons, 2003

Gambar 24.7, " Tanpa *deadlock* " memiliki perputaran tetapi *deadlock* tidak terjadi. Pada gambar di atas, graf memiliki 1 perputaran yaitu: **P0 -> R1 -> P2 -> R0 -> P3 - > R2 -> P0**

Graf di atas menunjukkan beberapa hal:

1. P0 meminta sumber daya R1.

2. R1 mengalokasikan sumber dayanya pada P2.

3. P2 meminta sumber daya R0.

4. R0 mengalokasikan sumber dayanya pada P3.

5. P3 meminta sumber daya R2.

6. R0 mengalokasikan sumber dayanya pada P3.

7. R1 mengalokasikan sumber dayanya pada P1.

Hal ini tidak menyebabkan *deadlock* walaupun ada perputaran sebab semua sumber daya yand diperlukan P1 dapat terpenuhi sehingga P1 dapat melepaskan semua sumber dayanya dan sumber daya tersebut dapat digunakan oleh proses lain.

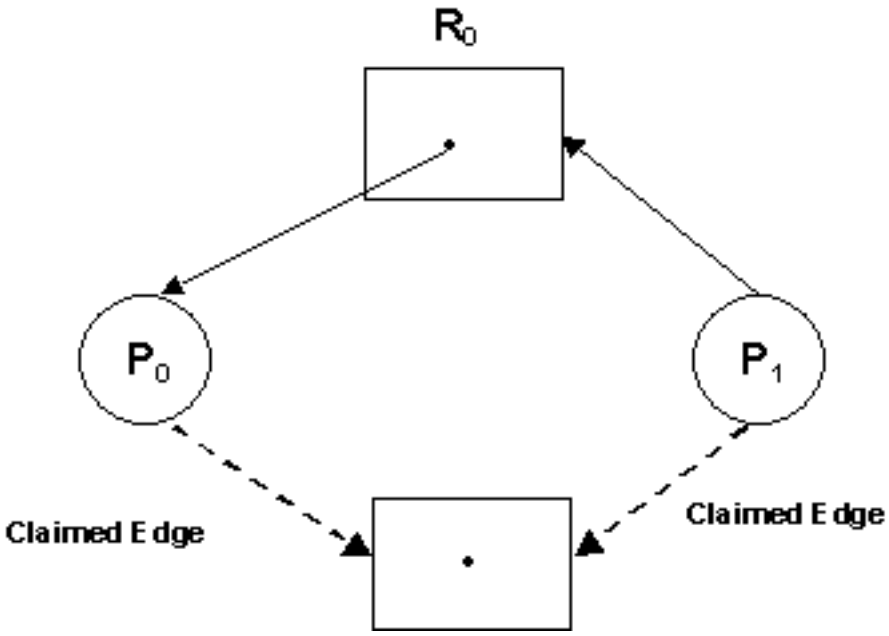
24.3. Algoritma Graf Alokasi Sumber Daya untuk Mencegah *Deadlock*

Algoritma ini dapat dipakai untuk mencegah *deadlock* jika sumber daya hanya memiliki satu instans. Pada algoritma ini ada komponen tambahan pada sisi yaitu claimed edge. Sama halnya dengan sisi yang lain, claimed edge menghubungkan antara sumber daya dan vertex.

Claimed edge Pi -> Rj berarti bahwa proses Pi akan meminta sumber daya Rj pada suatu waktu. *Claimed edge* sebenarnya merupakan sisi permintaan yang digamabarkan sebagai garis putus-putus. Ketika proses Pi memerlukan sumber daya Rj, *claimed edge* diubah menjadi sisi permintaan. Dan setelah proses Pi selesai menggunakan Rj, sisi alokasi diubah kembali menjadi *claimed edge.*

Dengan algoritma ini bentuk perputaran pada graf tidak dapat terjadi. Sebab untuk setiap perubahan yang terjadi akan diperiksa dengan algoritma deteksi perputaran. Algoritma ini memerlukan waktu n^2 dalam mendeteksi perputaran dimana n adalah jumlah proses dalam sistem. Jika tidak ada perputaran dalam graf, maka sistem berada dalam status aman. Tetapi jika perputaran ditemukan maka sistem berada dalam status tidak aman. Pada saat status tidak aman ini, proses Pi harus menunggu sampai permintaan sumber dayanya dipenuhi.

Gambar 24.8. Graf alokasi sumber daya dalam status aman

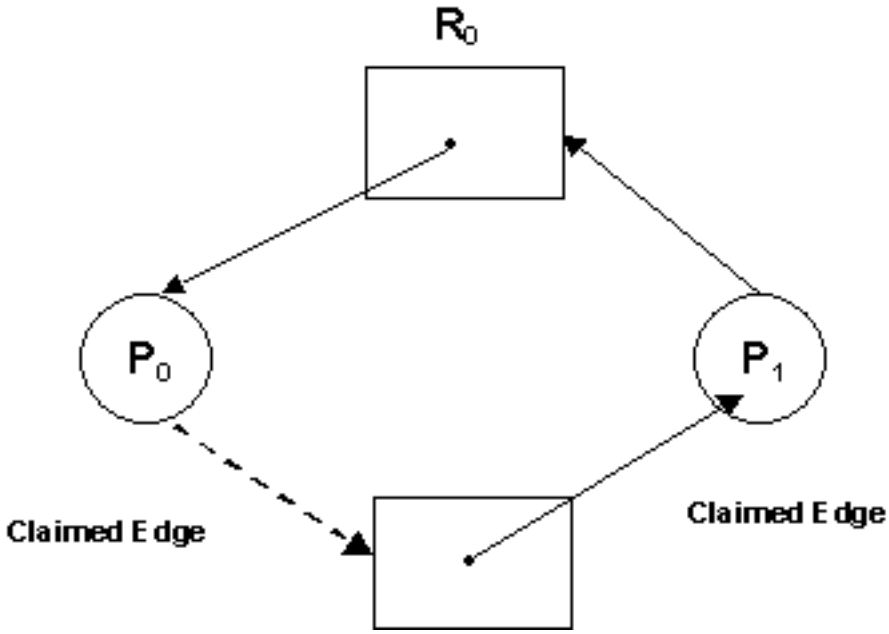


Sumber: Silberschatz, "*Operating System Concepts -- Fourth Edition*", John Wiley & Sons, 2003

Pada saat ini R1 sedang tidak mengalokasikan sumber dayanya, sehingga P1 dapat memperoleh sumber daya R1. Namun, jika *claimed edge* diubah menjadi sisi permintaan dan kemudian diubah menjadi sisi alokasi, hal ini dapat menyebabkan terjadinya perputaran (Gambar 24.9, “Graf alokasi

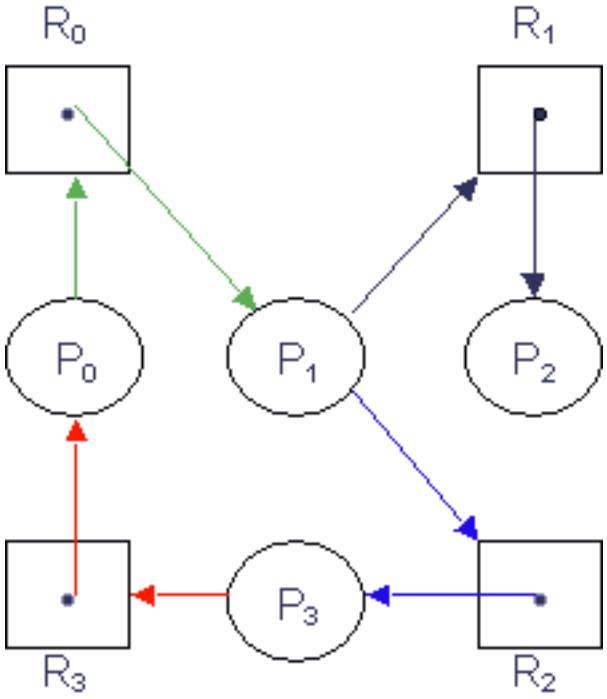
sumber daya dalam status tidak aman").

Gambar 24.9. Graf alokasi sumber daya dalam status tidak aman



24.4. Deteksi *Deadlock* dengan Graf Tunggu

Gambar 24.10. Graf alokasi sumber daya

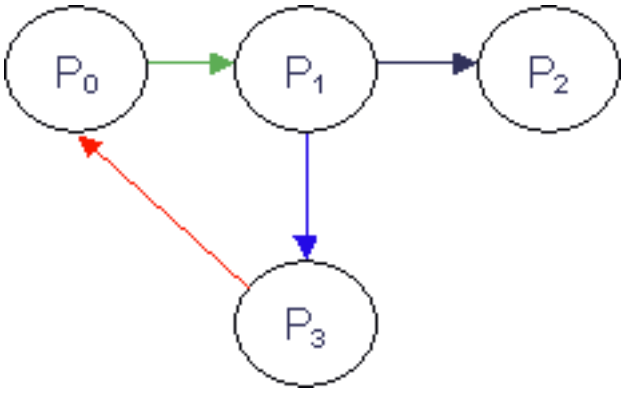


Jika semua sumber daya hanya memiliki satu instans, *deadlock* dapat dideteksi dengan mengubah

graf alokasi sumber daya menjadi graf tunggu. Ada pun caranya sebagai berikut:

1. Cari sumber daya Rm yang memberikan instansnya pada Pi dan Pj yang meminta sumber daya pada Rm.
2. Hilangkan sumber daya Rm dan hubungkan sisi Pi dan Pj dengan arah yang bersesuaian yaitu Pj->Pi.
3. Lihat apakah terdapat perputaran pada graf tunggu? ***Deadlock* terjadi jika dan hanya jika pada graf tunggu terdapat perputaran.**

Gambar 24.11. Graf tunggu



sumber: Silberschatz, "*Operating System Concepts -- Fourth Edition*", John Wiley & Sons, 2003

Untuk mendeteksi *deadlock*, sistem perlu membuat graf tunggu dan secara berkala memeriksa apakah ada perputaran atau tidak. Untuk mendeteksi adanya perputaran diperlukan operasi sebanyak n^2, dimana n adalah jumlah vertex dalam graf alokasi sumber daya.

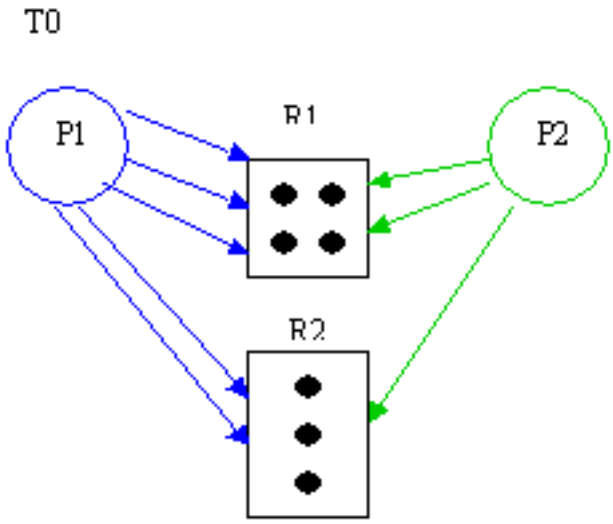
24.5. Rangkuman

Untuk mendeteksi deadlock dan menyelesaikannya dapat digunakan graf sebagai visualisasinya. Jika tidak ada *cycle*, berarti tidak ada *deadlock*. Jika ada *cycle*, ada potensi terjadi *deadlock*. Resource dengan satu instans dan *cycle* mengakibatkan *deadlock*.

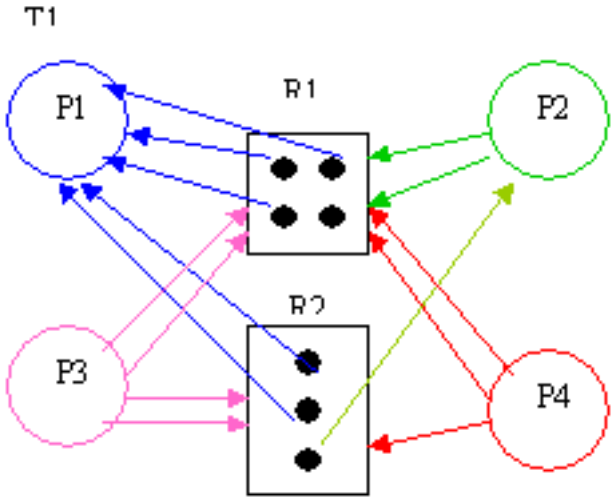
24.6. Latihan

1. Dalam sebuah sistem terdapat 4 proses yang akan siap di ready queue. Proses(Waktu Datang, Permintaan R1, Permintaan R2) P1(0, 3, 2) P2(0, 2, 1) P3(1, 2, 2) P4(1, 2, 1) Jumlah sumber daya R1 = 4, R2 = 3 Pemberian sumber daya berdasarkan aturan berikut:
    1. Jika ada dua proses yang sedang meminta sumber daya dan sumber daya yang tersedia hanya mencukupi salah satu proses, maka proses dengan ID terkecil didahulukan. Jika sumber daya dapat memenuhi semua proses yang meminta, maka sumber daya yang tersedia diberikan kepada semua proses yang membutuhkan.
    2. Jika sumber daya yang dibutuhkan proses telah terpenuhi semuanya pada Tn, maka pada

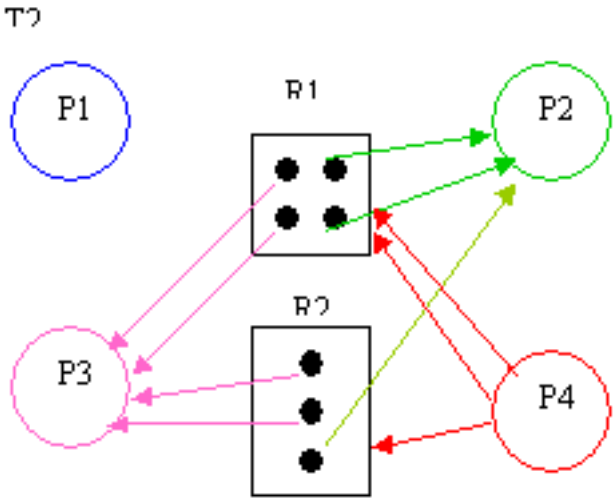
Tn+1 sumber daya dilepas dan dapat dipakai oleh proses lain pada Tn+1
Jawab:



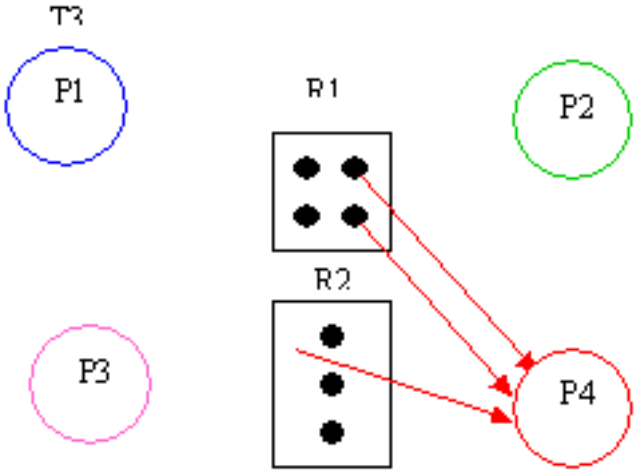
Graf alokasi sumber daya saat T0



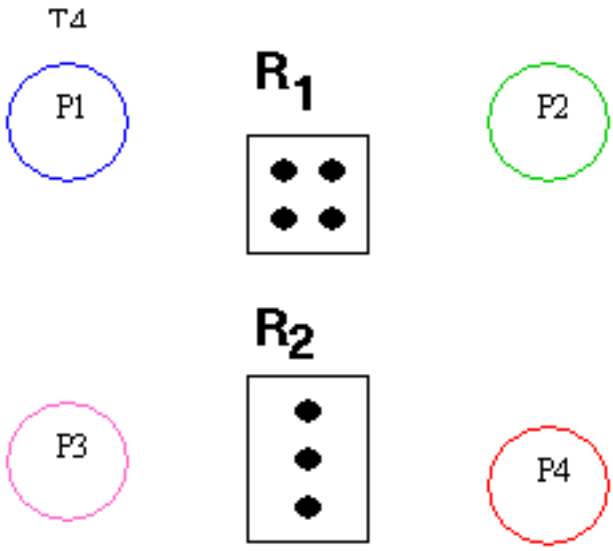
Graf alokasi sumber daya saat T1



Graf alokasi sumber daya saat T2



Graf alokasi sumber daya saat T3



Graf alokasi sumber daya saat T4

2. Diasumsikan proses P0 memegang sumber daya R2 dan R3, meminta sumber daya R4; P1 menggunakan R4 dan meminta R1; P2 menggunakan R1 dan meminta R3. Gambarkan *Wait-for Graph*. Apakah sistem terjebak dalam *deadlock*? Jika ya, tunjukkan proses mana yang menyebabkan *deadlock*. Jika tidak, tunjukkan urutan proses untuk selesai.

3. User x telah menggunakan 7 printer dan harus menggunakan 10 printer. User y telah menggunakan 1 printer dan akan memerlukan paling banyak 4 printer. User z telah menggunakan 2 printer dan akan menggunakan paling banyak 4 printer. Setiap user pada saat ini meminta 1 printer. Kepada siapakah OS akan memberikan grant printer tersebut dan tunjukkan "safe sequence" yang ada sehingga tidak terjadi deadlock.

4. Diketahui:

   a. set P yang terdiri dari dua (2) proses; P = { P1, P2 }.

   b. set R yang terdiri dari dua (2) sumber-daya (resources); dengan berturut-turut lima (5) dan dua (2) instances; R = { R1, R2 } = { {r11, r12, r13, r14, r15 }, {r21, r22 } }.

   c. Plafon (jatah maksimum) sumber-daya untuk masing-masing proses ialah:

	r1	r2
p1	5	1
p2	3	1

   d. Pencegahan deadlock dilakukan dengan Banker's Algorithm.

   e. Alokasi sumber-daya yang memenuhi kriteria Banker's Algorithm di atas, akan diprioritaskan pada proses dengan indeks yang lebih kecil.

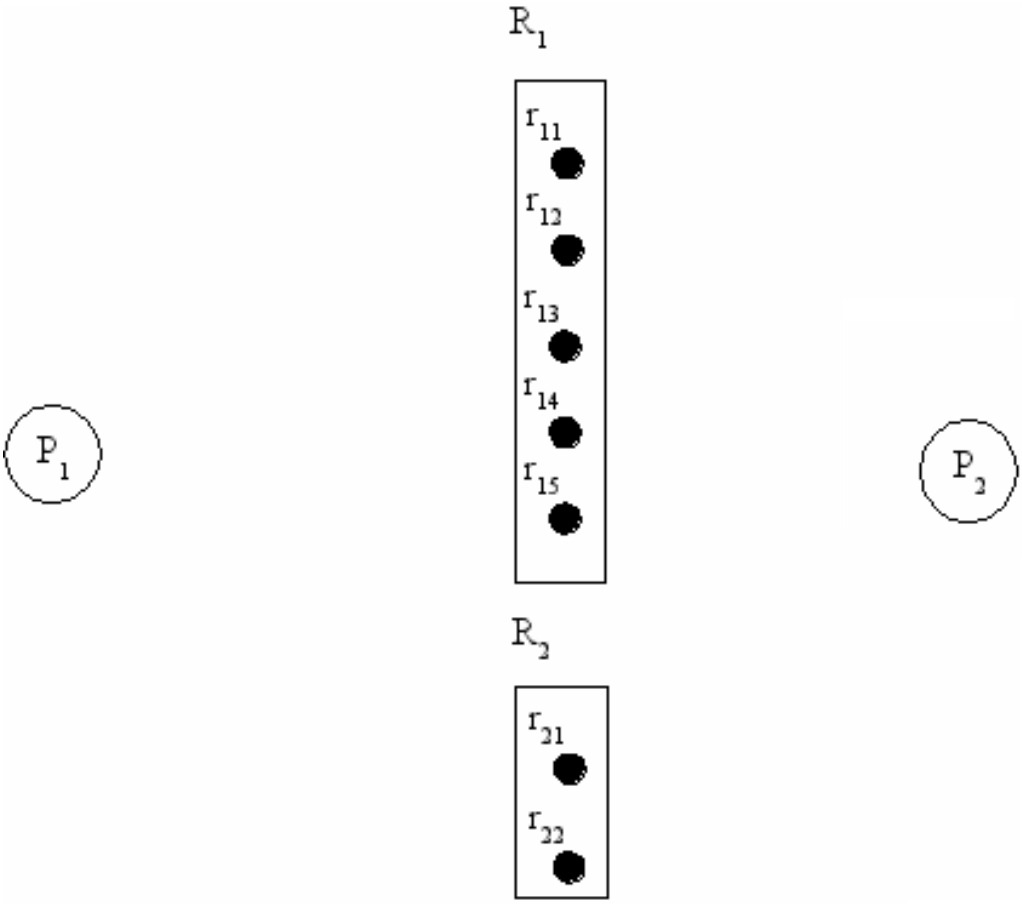
   f. Setelah mendapatkan semua sumber-daya yang diminta, proses akan mengembalikan SELURUH sumber-daya tersebut.

   g. Pada saat T0, "Teralokasi" serta "Permintaan" sumber-daya proses ditentukan sebagai berikut:

teralokasi	permintaan
|R1|R2|	|R1|R2|
p1| 2| 0|	|2 | 1|
p2| 2| 0|	|1 | 1|

Gambarkan graph pada urutan T0, T1,... dan seterusnya, hingga semua permintaan sumber-daya terpenuhi dan dikembalikan. Sebutkan, jika terjadi kondisi "unsafe"!

Gambar 24.12. ***Deadlock I***



5. Diketahui:

   a. set P yang terdiri dari tiga (3) proses; P = { P1, P2, P3 }.

   b. set R yang terdiri dari tiga (3) resources; masing-masing terdiri dari dua (2) instances; R = { R1, R2, R3 } = { {r11, r12 }, {r21, r22 }, {r31, r32 } }.

   c. Prioritas alokasi sumber daya (resource) akan diberikan pada proses dengan indeks yang lebih kecil.

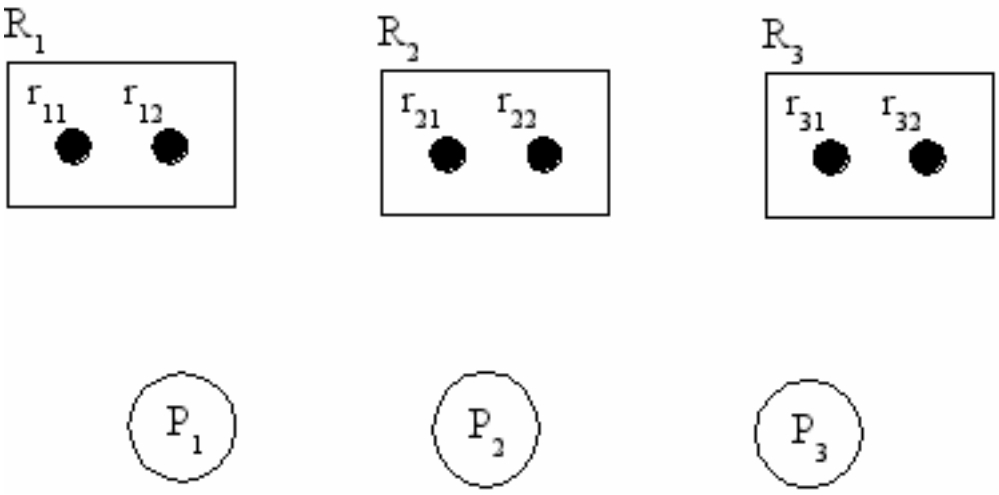
   d. Jika tersedia: permintaan alokasi sumber daya pada TN akan dipenuhi pada urutan berikutnya (TN + 1).

   e. Proses yang telah dipenuhi semua permintaan sumber daya (resources) pada TM ; akan melepaskan semua sumber daya tersebut pada urutan berikutnya (TM + 1).

f. Pencegahan deadlock dilakukan dengan menghindari circular wait.

g. Pada saat T0 , set E0 = { } (atau kosong), sehingga gambar graph-nya sebagai berikut:

Gambar 24.13. ***Deadlock II***



Jika set E pada saat T1 menjadi: E1 = { P1 --> R1, P1 --> R2, P2 --> R1, P2 --> R2, P3 --> R1, P3 --> R2, P3 --> R3 }, gambarkan graph pada urutan T1, T2,... serta (E2, E3, ...) berikutnya hingga semua permintaan sumberdaya terpenuhi dan dikembalikan.

24.7. Rujukan

FIXME

Bibliografi

[Silberschatz2000] Avi Silberschatz, Peter Galvin, dan Grag Gagne. Hak Cipta © 2000. *Applied Operating Systems*. First Edition. Edisi Pertama. John Wiley & Sons.

[KennethRosen1999] Kenneth H. Rosen. Hak Cipta © 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Stallings2001] William Stallings. Hak Cipta © 2001. *Operating Systems*. Prentice Hall.

[Tanenbaum1992] Andrew S. Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. Prentice-Hall Inc..

Bab 25. *Bounded Buffer*

25.1. Gambaran Umum

Masalah bounded buffer merupakan salah satu masalah yang menerangkan sinkronisasi antara proses-proses yang berjalan secara konkuren untuk mengakses data yang sama. Oleh karena itu kita banyak mengulang konsep sinkronisasi dalam membahas masalah bounded buffer ini. Yang dimaksud dengan bounded buffer adalah suatu struktur data untuk menampung (buffer) suatu nilai dimana kapasitasnya tertentu/terbatas (bounded). Yang menjadi pokok pembahasan utama dalam masalah bounded buffer adalah bagaimana jika ada dua proses berbeda yang berusaha mengakses buffer tersebut. Salah satu proses akan memberi nilai pada buffer dan mengisi buffer tersebut. Proses yang lain akan membaca nilai dan mengosongkan buffer tersebut. Proses yang pertama biasa disebut produser sedangkan yang kedua disebut konsumer.

Potongan program diatas menunjukkan bahwa produser akan terus mengisi buffer dengan sebuah item dan konsumer akan terus-menerus membaca isi buffer. *Bounded buffer* merupakan suatu struktur data yang mampu untuk menyimpan beberapa nilai dan mengeluarkannya kembali ketika diperlukan. Jika dianalogikan bounded buffer ini akan mirip dengan sebuah tumpukan piring. Kita menaruh piring dan menaruh lagi sebuah piring, ketika ingin mengambil piring maka tumpukan yang paling atas yang akan terambil. Jadi piring terakhir yang dimasukan akan pertama kali diambil.

Pada bagian ini akan dicontohkan suatu produser konsumer. produser akan menghasilkan suatu barang dan konsumer akan mengkonsumsi barang yang dihasilkan oleh produser. produser dan konsumer ini akan mengakses bounded buffer yang sama. produser setelah menghasilkan suatu barang dia akan menaruh barang itu di bounded buffer sebaliknya konsumer ketika membutuhkan suatu barang, dia akan mengambilkannya dari bounded buffer.

Kita memiliki dua aktor di sini, yaitu Produser dan Konsumer. Produser adalah *thread* yang menghasilkan waktu(Date) kemudian menyimpannya ke dalam antrian pesan. Produser juga mencetak waktu tersebut di layer (sebagai umpan balik bagi kita). Konsumer adalah *thread* yang akan mengakses antrian pesan untuk mendapatkan waktu(Date) itu dan tak lupa mencetaknya di layer. Kita menginginkan supaya konsumer itu mendapatkan waktu sesuatu dengan urutan sebagaimana produser menyimpan waktu tersebut. Kita akan menghadapi salah satu dari dua kemungkinan situasi di bawah ini:

25.2. Program Java

Contoh 25.1. Class BoundedBufferServer

```
001 // Authors: Greg Gagne, Peter Galvin, Avi Silberschatz
002 // Slightly Modified by: Rahmat M. Samik-Ibrahim
003 // Copyright (c) 2000 by Greg Gagne, Peter Galvin, Avi Silberschatz
004 // Applied Operating Systems Concepts - John Wiley and Sons, Inc.
005 //
006 // Class "Date":
007 //     Allocates a Date object and initializes it so that it represents
008 //     the time at which it was allocated,
009 //     (E.g.): "Wed Apr 09 11:12:34 JAVT 2003"
010 // Class "Object"/ method "notify":
011 //     Wakes up a single thread that is waiting on this object's monitor
012 // Class "Thread"/ method "start":
013 //     Begins the thread execution and calls the run method of the threa
014 // Class "Thread"/ method "run":
015 //     The Runnable object's run method is called.
016
017 import java.util.*;
018 // main ******************************************************
```

```
019 public class BoundedBufferServer
020 {
021     public static void main(String args[])
022     {
023         BoundedBuffer server         = new BoundedBuffer();
024         Producer      producerThread = new Producer(server);
025         Consumer      consumerThread = new Consumer(server);
026         producerThread.start();
027         consumerThread.start();
028     }
029 }
030
```

Contoh 25.2. Class Producer

```
031 // Producer ******************************************************
032 class Producer extends Thread
033 {
034     public Producer(BoundedBuffer b)
035     {
036          buffer = b;
037     }
038
039     public void run()
040     {
041         Date message;
042         while (true)
043      {
044             BoundedBuffer.napping();
045
046             message = new Date();
047             System.out.println("P: PRODUCE  " + message);
048             buffer.enter(message);
049         }
050     }
051     private  BoundedBuffer buffer;
052 }
053
```

Contoh 25.3. Class Consumer

```
054 // Consumer ******************************************************
055 class Consumer extends Thread
056 {
057    public Consumer(BoundedBuffer b)
058    {
059       buffer = b;
060    }
061    public void run()
062    {
063          Date message;
064          while (true)
065          {
066              BoundedBuffer.napping();
```

```
067              System.out.println("C: CONSUME  START");
068              message = (Date)buffer.remove();
069           }
070     }
071     private  BoundedBuffer buffer;
072 }
073
```

Contoh 25.4. Class BoundedBuffer

```
074 // BoundedBuffer.java ****************************************
075 class BoundedBuffer
076 {
077    public BoundedBuffer()
078    {
079         count  = 0;
080         in     = 0;
081         out    = 0;
082         buffer = new Object[BUFFER_SIZE];
083         mutex  = new Semaphore(1);
084         empty  = new Semaphore(BUFFER_SIZE);
085         full   = new Semaphore(0);
086    }
087    public static void napping()
088    {
089         int sleepTime = (int) (NAP_TIME * Math.random() );
090         try { Thread.sleep(sleepTime*1000); }
091         catch(InterruptedException e) { }
092    }
093    public void enter(Object item)
094    {
095         empty.P();
096         mutex.P();
097         ++count;
098         buffer[in] = item;
099         in = (in + 1) % BUFFER_SIZE;
100         System.out.println("P: ENTER    " + item);
101         mutex.V();
102         full.V();
103    }
104    public Object remove()
105    {
106         Object item;
107         full.P();
108         mutex.P();
109         --count;
110         item = buffer[out];
111         out = (out + 1) % BUFFER_SIZE;
112         System.out.println("C: CONSUMED " + item);
113         mutex.V();
114         empty.V();
115         return item;
116     }
117     public  static final int NAP_TIME    = 5;
118     private static final int BUFFER_SIZE = 3;
119     private Semaphore        mutex;
120     private Semaphore        empty;
121     private Semaphore        full;
122     private int              count, in, out;
123     private Object[]         buffer;
124 }
125
```

Contoh 25.5. Class Semaphore

```
126 // Semaphore.java ************************************************
127
128 final class Semaphore
129 {
130     public Semaphore()
131     {
132         value = 0;
133     }
134     public Semaphore(int v)
135     {
136        value = v;
137     }
138     public synchronized void P()
139     {
140         while (value <= 0)
141         {
142             try { wait(); }
143             catch (InterruptedException e) { }
144         }
145         value --;
146     }
147     public synchronized void V()
148     {
149         ++value;
150         notify();
151     }
152     private int value;
153 }
```

Contoh 25.6. Class Semaphore

```
C: CONSUME  START
P: PRODUCE  Mon Aug 22 11:19:19 WIT 2005
P: ENTER    Mon Aug 22 11:19:19 WIT 2005
C: CONSUMED Mon Aug 22 11:19:19 WIT 2005
C: CONSUME  START
P: PRODUCE  Mon Aug 22 11:19:20 WIT 2005
P: ENTER    Mon Aug 22 11:19:20 WIT 2005
C: CONSUMED Mon Aug 22 11:19:20 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:21 WIT 2005
P: ENTER    Mon Aug 22 11:19:21 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:22 WIT 2005
P: ENTER    Mon Aug 22 11:19:22 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:23 WIT 2005
P: ENTER    Mon Aug 22 11:19:23 WIT 2005
C: CONSUME  START
C: CONSUMED Mon Aug 22 11:19:21 WIT 2005
C: CONSUME  START
C: CONSUMED Mon Aug 22 11:19:22 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:27 WIT 2005
P: ENTER    Mon Aug 22 11:19:27 WIT 2005
```

```
C: CONSUME  START
C: CONSUMED Mon Aug 22 11:19:23 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:29 WIT 2005
P: ENTER    Mon Aug 22 11:19:29 WIT 2005
C: CONSUME  START
C: CONSUMED Mon Aug 22 11:19:27 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:32 WIT 2005
P: ENTER    Mon Aug 22 11:19:32 WIT 2005
P: PRODUCE  Mon Aug 22 11:19:33 WIT 2005
P: ENTER    Mon Aug 22 11:19:33 WIT 2005
```

Dari ilustrasi diatas apa yang terjadi bila ketika produser ingin mengisi sebuah item ke buffer ternyata buffer sudah penuh (karena kapasitasnya terbatas)? Apakah produser tetap mengisi buffer tersebut dengan item yang baru sehingga item lama yang belum dibaca konsumer akan hilang? Sebaliknya bila konsumer ingin membaca item dari buffer tetapi produser belum mengisi item yang baru ke buffer apakah konsumer tetap membaca item yang lama dua kali (atau mungkin berkali-kali)?

Kita dapat menerapkan konsep semaphore untuk menyelesaikan masalah tersebut. Disini kita menggunakan tiga buah semaphore yaitu mutex, full dan empty. Mutex digunakan untuk menjamin hanya boleh satu proses yang berjalan mengakses buffer pada suatu waktu, awalnya dinisialisasi sebesar satu (1). Full digunakan untuk menghitung jumlah buffer yang berisi, yang pada awalnya diinisialisasi sebesar nol (0). Sedangkan empty digunakan untuk menghitung jumlah buffer yang kosong, yang awalnya dinisialisasi sebesar ukuran buffer.

Produser membuat suatu informasi yang dapat dibagi dengan proses lainnya. Konsumer menghabiskan data yang dibuat oleh produser. Misalnya program cetak memproduksi karakter yang dipakai oleh *printer*.

Masalah yang biasanya dihadapi oleh produser dan konsumer adalah bagaimana caranya mensinkronisasikan kerja mereka sehingga tidak ada yang saling mengganggu. Salah satu contoh bagaimana masalah ini dapat terjadi adalah *Bounded Buffer Problem*.

Penjelasan program diatas adalah sebagai berikut. Pada kelas Semaphore kita membuat suatu variabel value yang akan dipakai oleh setiap objek Semaphore. Variabel value itulah yang akan dicek dalam method kunci pada baris tiga belas (analogi dengan method wait dalam penjelasan proses produser/konsumer) yang mana bila nilai value sama dengan nol maka objek Semaphore yang memanggil method kunci tersebut akan memanggil method wait() pada baris lima belas yang membuat thread yang sedang berjalan tidak melakukan apa-apa. Namun bila nilai value tidak sama dengan nol maka objek Semaphore tersebut akan mengurangi nilai value.

Bila suatu objek Semaphore memanggil method buka (pada baris ke-24) maka nilai value dari Semaphore tersebut akan ditambah satu kemudian dipanggil method notifyAll() pada baris ke-26 untuk membangunkan thread lain yang sedang dalam method wait().

Dalam kelas BoundedBuffer kita membuat suatu array buffer dengan kapasitas tiga (baris 41). Disini juga dibuat tiga objek Semaphore yaitu mutex dengan nilai value satu (baris 43), empty dengan nilai value tiga (baris 44), dan full dengan nilai value nol (baris 45). Juga terdapat dua variabel integer penunjuk index buffer yaitu in (untuk produser) dan out (untuk konsumer) yang masing-masing diberi nilai nol (baris 38-39).

Hal yang harus diperhatikan dalam contoh program ini bahwa:

- Bounded buffer memiliki batas banyaknya data yang dimasukan.
- Barang yang dikonsumsi oleh konsumer terbatas.
- Jika bounded buffer telah penuh produser tidak mampu menaruh lagi dan akan menunggu sampai ada tempat yang kosong.

- Jika bounded buffer kosong maka konsumer harus menunggu sampai ada barang yang ditauh oleh produser.

25.3. Rangkuman

Jadi dapat disimpulkan bahwa pokok permasalahan bounded buffer adalah bagaimana mengatur sinkronisasi dari beberapa proses yang secara konkuren ingin mengakses buffer (mengisi dan mengosongkan buffer). Pengaturan itu dilakukan dengan menerapkan konsep semaphore yang menjamin hanya ada satu proses dalam suatu waktu yang boleh mengakses buffer sehingga tidak terjadi race condition. Bounded buffer ini merupakan salah satu contoh dari masalah sinkronisasi dimana ada beberapa proses ingin bersama-sama mengakses critical section.

25.4. Latihan

1. Perhatikan berkas "BoundedBufferServer.java".
   a. Berapakah ukuran penyangga (*buffer*)?
   b. Modifikasi program (sebutkan nomor barisnya) agar ukuran penyangga menjadi 6.
   c. Tuliskan/perkirakan keluaran (*output*) 10 baris pertama, jika menjalankan program ini.
   d. Jelaskan fungsi dari ketiga semaphore (*mutex*, *full*, *empty*) pada program tersebut.
   e. Tambahkan (sebutkan nomor barisnya) sebuah thread dari class Supervisor yang berfungsi:
      i. pada awal dijalankan, melaporkan ukuran penyangga (buffer).
      ii. secara berkala (acak), melaporkan jumlah pesan (*message*) yang berada dalam penyangga.
   f. Semaphore mana yang paling relevan untuk modifikas butir e di atas?

Rujukan (FM)

[Deitel2005] Harvey M Deitel dan Paul J Deitel . 2005. *Java How To Program*. Sixth Edition. Prentice Hall.

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

Bab 26. *Readers/Writers*

26.1. Gambaran Umum

Salah satu dari sekian banyak permasalahan sinkronisasi yang sering terjadi di dunia nyata, yaitu ketika ada dua jenis proses -readers dan writers- yang bisa saling berbagi (shared) data dan mengakses database secara paralel. proses yang pertama (reader) bertugas untuk membaca data, sedangkan proses kedua bertugas untuk menulis data baru/mengupdate nilai dalam data (writer). Solusinya adalah menjadikan proses dapat dijalankan dalam keadaan terpisah/terisolasi dari proses lain. untuk menghindari gangguan dalam perubahan data, writer harus mempunyai kemampuan mengakses data secara eksklusif.

Ada tiga hal yang jadi inti dari solusi ini, yaitu:

- readers yang baru tiba mendapat prioritas yang lebih tinggi daripada writers yang sedang menunggu.
- writer yang baru tiba atau sedang menunggu, bisa mengakses file hanya jika tidak ada readers dalan system
- ketika writers dijalankan oleh system, semua readers yang akan menunggu mempunyai prioritas lebih tinggi untuk dijalankan. daripada semua writers yang sedang mengantri

Akan tetapi, dari solusi ini masih timbul permasalahan baru. yaitu ketika readers terus menerus datang, writers tidak akan mendapatkan giliran untuk mengakses data (starvation).

26.2. Program Java

Contoh 26.1. Class ReaderWriterServer

```
001 // Gabungan ReaderWriterServer.java Reader.java Writer.java
002 //          Semaphore.java Database.java
003 // (c) 2000 Gagne, Galvin, Silberschatz
004
005 public class ReaderWriterServer {
006    public static void main(String args[]) {
007       Database server = new Database();
008       Reader[] readerArray = new Reader[NUM_OF_READERS];
009       Writer[] writerArray = new Writer[NUM_OF_WRITERS];
010       for (int i = 0; i < NUM_OF_READERS; i++) {
011          readerArray[i] = new Reader(i, server);
012          readerArray[i].start();
013       }
014       for (int i = 0; i < NUM_OF_WRITERS; i++) {
015          writerArray[i] = new Writer(i, server);
016          writerArray[i].start();
017       }
018    }
019    private static final int NUM_OF_READERS = 3;
020    private static final int NUM_OF_WRITERS = 2;
021 }
022
```

Contoh 26.2. Class Reader

```
023 class Reader extends Thread {
024    public Reader(int r, Database db) {
025       readerNum = r;
026       server = db;
027    }
028    public void run() {
029      int c;
030      while (true) {
031        Database.napping();
032        System.out.println("reader " + readerNum + " wants to read.");
033        c = server.startRead();
034        System.out.println("reader " + readerNum +
035             " is reading. Reader Count = " + c);
036        Database.napping();
037        System.out.print("reader " + readerNum + " is done reading. ");
038        c = server.endRead();
039       }
040    }
041    private Database  server;
042    private int       readerNum;
043 }
044
```

Contoh 26.3. Class Writer

```
045 class Writer extends Thread {
046    public Writer(int w, Database db) {
047       writerNum = w;
048       server = db;
049    }
050    public void run() {
051      while (true) {
052        System.out.println("writer " + writerNum + " is sleeping.");
053        Database.napping();
054        System.out.println("writer " + writerNum + " wants to write.");
055        server.startWrite();
056        System.out.println("writer " + writerNum + " is writing.");
057        Database.napping();
058        System.out.println("writer " + writerNum + " is done writing.");
059        server.endWrite();
060       }
061    }
062    private Database  server;
063    private int       writerNum;
064 }
065
```

Contoh 26.4. Class Semaphore

```
066 final class Semaphore {
067    public Semaphore() {
068       value = 0;
069    }
070    public Semaphore(int v) {
071       value = v;
072    }
073    public synchronized void P() {
074       while (value <= 0) {
075          try { wait(); }
076          catch (InterruptedException e) { }
077       }
078       value--;
079    }
080    public synchronized void V() {
081       ++value;
082       notify();
083    }
084    private int value;
085 }
086
```

Contoh 26.5. Class Database

```
087 class Database {
088    public Database() {
089       readerCount = 0;
090       mutex = new Semaphore(1);
091       db = new Semaphore(1);
092    }
093    public static void napping() {
094      int sleepTime = (int) (NAP_TIME * Math.random() );
095      try { Thread.sleep(sleepTime*1000); }
096      catch(InterruptedException e) {}
097    }
098    public int startRead() {
099       mutex.P();
100       ++readerCount;
101       if (readerCount == 1) {
102          db.P();
103       }
104       mutex.V();
105       return readerCount;
106    }
107    public int endRead() {
108       mutex.P();
109       --readerCount;
110       if (readerCount == 0) {
111          db.V();;
112       }
113       mutex.V();
114       System.out.println("Reader count = " + readerCount);
115       return readerCount;
116    }
117    public void startWrite() {
118       db.P();
119    }
120    public void endWrite() {
121       db.V();
122    }
123    private int readerCount;
```

```
124    Semaphore mutex;
125    Semaphore db;
126    private static final int NAP_TIME = 15;
127 }
128
129 // The Class java.lang.Thread
130 // When a thread is created, it is not yet active; it begins to run when metho
131 // start is called. Invoking the .tart method causes this thread to begin
132 // execution; by calling the .run. method.
133 // public class Thread implements Runnable {
134 //     ...
135 //     public void run();
136 //     public void start()
137 //        throws IllegalThreadStateException;
138 //     ...
139 // }
```

Contoh 26.6. Keluaran

```
writer 0 is sleeping.
writer 1 is sleeping.
writer 0 wants to write.
writer 0 is writing.
writer 1 wants to write.
reader 1 wants to read.
reader 2 wants to read.
reader 0 wants to read.
writer 0 is done writing.
writer 0 is sleeping.
writer 1 is writing.
writer 0 wants to write.
writer 1 is done writing.
writer 1 is sleeping.
reader 1 is reading. Reader Count = 1
reader 2 is reading. Reader Count = 2
reader 0 is reading. Reader Count = 3
reader 0 is done reading. Reader count = 2
reader 2 is done reading. Reader count = 1
reader 0 wants to read.
reader 0 is reading. Reader Count = 2
writer 1 wants to write.
reader 1 is done reading. Reader count = 1
reader 0 is done reading. Reader count = 0
writer 0 is writing.
reader 0 wants to read.
reader 1 wants to read.
writer 0 is done writing.
writer 0 is sleeping.
writer 1 is writing.
writer 1 is done writing.
writer 1 is sleeping.
reader 0 is reading. Reader Count = 1
reader 1 is reading. Reader Count = 2
reader 2 wants to read.
reader 2 is reading. Reader Count = 3
reader 2 is done reading. Reader count = 2
reader 1 is done reading. Reader count = 1
reader 1 wants to read.
reader 1 is reading. Reader Count = 2
writer 0 wants to write.
```

Penjelasan:

- penulis akan mulai mencoba untuk melewati db.tutup()/membaca ketika mengetahui tiada Reader yang menunggu lagi (banyakReader = 0).
- dari contoh output terlihat jelas pada baris ke-2 dan baris terakhir, bahwa writer baru bisa menulis ketika tiada lagi reader yang ingin membaca.
- pada baris ke-10 menunjukkan bahwa writer 1 ingin menulis, akan tetapi baru bisa menulis pada baris ke-20. setelah semua reader yang ingin membaca selesai dilayani.
- bahkan pada baris ke-15, reader 2 meminta membaca. dan langsung diberikan giliran

Kaidah-kaidah:

1. hanya 1 proses yang di izinkan untuk mengakses database; jika reader sedang membaca, writer tidak boleh menulis jika writer sedang menulis, reader tidak boleh membaca

2. reader yang membaca boleh lebih dari satu

penjelasan:

1. method P() dan V() di Semaphore di ganti dengan tutup() dan buka() tutup() bisa dimisalkan dengan mengambil kunci yang sedang di gantung/menutup pintu; buka() dimisalkan dengan meletakkan kembali kunci di gantungan/membuka pintu.

2. disini ada tiga readers dan dua writers

3. ada dua Semaphore yang digunakan di sini, yaitu:

a. db

digunakan agar hanya ada satu proses yang bisa membaca/menulis. jika reader sedang membaca, writer tidak boleh menulis, jika writer sedang menulis, reader tidak boleh membaca

penjelasan jalannya program --> ketika Reader memanggil method 'mulaiBaca()', maka akan terdapat pemanggilan method 'tutup', sehingga (jika kita lihat di 'tutup()') akan mencegah masuknya writer sebelum Reader memanggil method 'buka()'

b. mutex

digunakan untuk menjaga agar variabel 'banyakReader' mempunyai nilai yang betul/konsisten.

26.3. Rangkuman

Yang dibahas di sini adalah ilustrasi dari beberapa kasus sinkronisasi yang acapkali terjadi di dunia sebenarnya. benar, bahwa sinkronisasi ini memiliki peran yang penting dalam menjaga validitas data. seperti contoh diatas, bagaimana jika kesalahan data itu terjadi di tempat-tempat penting yang sering melakukan transaksi uang dalam jumlah yang besar...bagaimana jika kita sebagai nasabah bank yang menyetor uang, jumlah uang kita di bank malah dikurangi? dan berbagai transaksi tingkat tinggi lainnya... solusi yang ditawarkan ada tiga:

1. Solusi Pembaca diprioritaskan

2. Solusi Penulis diprioritaskan

3. Solusi Prioritas Bergantian

Akan tetapi, dari ketiga solusi tersebut hanya solusi ketiga yang tidak mengakibatkan starvation.

Rujukan

[KennethRosen1999] Kenneth H. Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[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.

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

26.4. Latihan

1. Problem Reader/Writer I

   Perhatikan berkas "ReaderWriterServer.java" berikut ini (source-code terlampir):

   a) Ada berapa object class "Reader" yang terbentuk? Sebutkan nama-namanya!

   b) Ada berapa object class "Writer" yang terbentuk? Sebutkan nama-namanya!

   c) Modifikasi kode program tersebut (cukup baris terkait), sehingga akan terdapat 6 (enam) "Reader" dan 4 (empat) "Writer".

   d) Modifikasi kode program tersebut, dengan menambahkan sebuah (satu!) object thread baru yaitu "janitor". Sang "janitor" berfungsi untuk membersihkan (cleaning). Setelah membersihkan, "janitor" akan tidur (sleeping). Pada saat bangun, "janitor" kembali akan membersihkan. Dan seterusnya... Pada saat "janitor" akan membersihkan, tidak boleh ada "reader" atau "writer" yang aktif. Jika ada, "janitor" harus menunggu. Demikian pula, "reader" atau "writer" harus menunggu "janitor" hingga selesai membersihkan.

2. Problem Reader/Writer II

   Perhatikan berkas ReaderWriterServer.java berikut ini, yang merupakan gabungan ReaderWriterServer.java, Reader.java, Writer.java, Semaphore.java, Database.java, oleh Gagne, Galvin, dan Silberschatz. Terangkan berdasarkan berkas tersebut:

   a. akan terbentuk berapa thread, jika menjalankan program class ReaderWriterServer ini? Apa yang membedakan antara sebuah thread dengan thread lainnya?

   b. mengapa: jika ada Reader yang sedang membaca, tidak ada Writer yang dapat menulis; dan mengapa: jika ada Writer yang sedang menulis, tidak ada Reader yang dapat membaca?

   c. mengapa: jika ada Reader yang sedang membaca, boleh ada Reader lainnya yang turut membaca?

   d. modifikasi kode program tersebut (cukup mengubah baris terkait), sehingga akan terdapat 5 (lima) Reader dan 4 (empat) Writer!

   Modifikasi kode program tersebut (cukup mengubah method terkait), sehingga pada saat RAJA (Reader 0) ingin membaca, tidak boleh ada RAKYAT (Reader lainnya) yang sedang/akan membaca. JANGAN MEMPERSULIT DIRI SENDIRI: jika RAJA sedang membaca, RAKYAT boleh turut membaca.

3. Perhatikan berkas ReaderWriterServer.java diatas. Terangkan berdasarkan berkas tersebut:

   a. akan terbentuk berapa thread, jika menjalankan program class ReaderWriterServer ini? Apa yang membedakan antara sebuah thread dengan thread lainnya?

   b. mengapa: jika ada Reader yang sedang membaca, tidak ada Writer yang dapat menulis; dan mengapa: jika ada Writer yang sedang menulis, tidak ada Reader yang dapat membaca?

   c. mengapa: jika ada Reader yang sedang membaca, boleh ada Reader lainnya yang turut membaca?

   d. modifikasi kode program tersebut (cukup mengubah baris terkait), sehingga akan terdapat 5 (lima) Reader dan 4 (empat) Writer!

   Modifikasi kode program tersebut (cukup mengubah method terkait), sehingga pada saat RAJA (Reader 0) ingin membaca, tidak boleh ada RAKYAT (Reader lainnya) yang sedang/akan membaca. JANGAN MEMPERSULIT DIRI SENDIRI: jika RAJA sedang membaca, RAKYAT boleh turut membaca.

4. Perhatikan berkas "ReaderWriterServer.java" diatas:

   a. Ada berapa object class "Reader" yang terbentuk? Sebutkan nama - namanya!

   b. Ada berapa object class "Writer" yang terbentuk? Sebutkan nama - namanya!

   c. Modifikasi kode program tersebut (cukup baris terkait), sehingga akan terdapat 6 (enam) "Reader" dan 4 (empat) "Writer".

   d. Modifikasi kode program tersebut, dengan menambahkan sebuah (satu!) object thread baru yaitu "janitor". Sang "janitor" berfungsi untuk membersihkan (cleaning). Setelah membersihkan,"janitor" akan tidur (sleeping). Pada saat bangun, "janitor" kembali akan membersihkan. Dan seterusnya... Pada saat "janitor" akan membersihkan, tidak boleh ada "reader" atau "writer" yang aktif. Jika ada, "janitor" harus menunggu. Demikian pula, "reader" atau "writer" harus menunggu "janitor" hingga selesai membersihkan.

5. Perhatikan berkas program java pada halaman berikut ini.

   a. Berapa jumlah thread class Reader yang akan terbentuk?

   b. Berapa jumlah thread class Writer yang akan terbentuk?

   c. Perkirakan bagaimana bentuk keluaran (output) dari program tersebut!

   d. Modifikasi program agar nap rata-rata dari class Reader lebih besar daripada class Writer.

```
================================================================
/*****************************************************************
* Gabungan/Modif: Factory.java Database.java RWLock.java Reader.java
* Semaphore.java SleepUtilities.java Writer.java
* Operating System Concepts with Java - Sixth Edition
* Gagne, Galvin, Silberschatz Copyright John Wiley & Sons - 2003.
*/
public class Factory {
   public static void main(String args[]){
      System.out.println("INIT Thread...");
      Database server = new Database();
      Thread readerX = new Thread(new Reader(server));
      Thread writerX = new Thread(new Writer(server));
      readerX.start();
      writerX.start();
      System.out.println("Wait...");
   }
}
```

```
// Reader // ************************************************************
class Reader implements Runnable {
   public Reader(Database db) { server = db; }
   public void run() {
      while (--readercounter > 0){
         SleepUtilities.nap();
         System.out.println("readerX: wants to read.");
         server.acquireReadLock();
         System.out.println("readerX: is reading.");
         SleepUtilities.nap();
         server.releaseReadLock();
         System.out.println("readerX: done...");
      }
   }
   private Database server;
   private int readercounter = 3;
}
-----------------------------------------------------------------------
// Writer // ************************************************************
class Writer implements Runnable {
   public Writer(Database db) { server = db; }
   public void run() {
      while (writercounter-- > 0){
         SleepUtilities.nap();
         System.out.println("writerX: wants to write.");
         server.acquireWriteLock();
         System.out.println("writerX: is writing.");
         SleepUtilities.nap();
         server.releaseWriteLock();
         System.out.println("writerX: done...");
      }
   }
   private Database server;
   private int writercounter = 3;
}

// Semaphore // **********************************************************
class Semaphore{
   public Semaphore() { value = 0; }
   public Semaphore(int val) { value = val; }
   public synchronized void acquire() {
      while (value == 0) {
         try { wait(); }
         catch (InterruptedException e) { }
      }
      value--;
   }
   public synchronized void release() {
      ++value;
      notifyAll();
   }
   private int value;
}
// SleepUtilities // *****************************************************
class SleepUtilities{
   public static void nap() { nap(NAP_TIME); }
   public static void nap(int duration) {
      int sleeptime = (int) (duration * Math.random() );
      try { Thread.sleep(sleeptime*1000); }
      catch (InterruptedException e) {}
   }
   private static final int NAP_TIME = 3;
}
// Database // ***********************************************************
class Database implements RWLock{
   public Database() { db = new Semaphore(1); }
   public void acquireReadLock() { db.acquire(); }
```

```
   public void releaseReadLock() { db.release(); }
   public void acquireWriteLock() { db.acquire(); }
   public void releaseWriteLock() { db.release(); }
   Semaphore db;
}

// An interface for reader-writer locks. // ************************
interface RWLock{
   public abstract void acquireReadLock();
   public abstract void releaseReadLock();
   public abstract void acquireWriteLock();
   public abstract void releaseWriteLock();
}
```

Bab 27. Sinkronisasi Dua Arah

27.1. Gambaran Umum

Ini merupakan ilustrasi dimana sebuah thread memegang kendali sinkronisasai lainnya. Yang berikut ini terdiri dari beberapa class yaitu SuperProses, SuperP, Proses, dan Semafor.

27.2. Program Java

Contoh 27.1. Class SuperProses

```
000 /************************************************************
001  * SuperProses (c) 2005 Rahmat M. Samik-Ibrahim, GPL-like   */
002
003 // ********* SuperProses *
004 public class SuperProses {
005    public static void main(String args[]) {
006       Semafor[] semafor1 = new Semafor[JUMLAH_PROSES];
007       Semafor[] semafor2 = new Semafor[JUMLAH_PROSES];
008       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
009          semafor1[ii] = new Semafor();
010          semafor2[ii] = new Semafor();
011       }
012
013       Thread superp = new
014                       Thread(new SuperP(semafor1,semafor2,JUMLAH_PROSES));
015       superp.start();
016
017       Thread[] proses= new Thread[JUMLAH_PROSES];
018       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
019          proses[ii]=new Thread(new Proses(semafor1,semafor2,ii));
020       }
021       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
022          proses[ii].start();
023       }
024    }
025
026    private static final int JUMLAH_PROSES = 16;
027 }
028
```

Contoh 27.2. Class SuperP

```
029 // ** SuperP ********************
030 class SuperP implements Runnable {
031    SuperP(Semafor[] sem1, Semafor[] sem2, int jmlh) {
032       semafor1      = sem1;
033       semafor2      = sem2;
034       jumlah_proses = jmlh;
035    }
036
037    public void run() {
038       for (int ii = 0; ii < jumlah_proses; ii++) {
```

```
039          semafor1[ii].kunci();
040       }
041       System.out.println("SUPER PROSES siap...");
042       for (int ii = jumlah_proses-1 ; ii >= 0; ii--) {
043          semafor2[ii].buka();
044          semafor1[ii].kunci();
045       }
046    }
047
048    private Semafor[] semafor1, semafor2;
049    private int       jumlah_proses;
050 }
051
```

Contoh 27.3. Class Proses

```
052 // ** Proses ********************
053 class Proses implements Runnable {
054    Proses(Semafor[] sem1, Semafor[] sem2, int num) {
055       num_proses = num;
056       semafor1   = sem1;
057       semafor2   = sem2;
058    }
059
060    public void run() {
061       try {Thread.sleep((int)(ISTIROHAT * Math.random()));}
062       catch(InterruptedException e) { }
063       System.out.println("Proses " + num_proses + " hadir...");
064       semafor1[num_proses].buka();
065       semafor2[num_proses].kunci();
066       System.out.println("Proses " + num_proses + " siap...");
067       semafor1[num_proses].buka();
068    }
069    private static final int ISTIROHAT = 16;     // (ms)
070    private Semafor[]        semafor1, semafor2;
071    private int              num_proses;
072 }
073
```

Contoh 27.4. Class Semafor

```
074 // ** Semafor *
075 class Semafor {
076    public Semafor()        { value = 0;   }
077    public Semafor(int val) { value = val; }
078
079    public synchronized void kunci() {
080       while (value == 0) {
081          try   { wait(); }
082          catch (InterruptedException e) { }
083       }
084       value--;
085    }
086
```

```
087    public synchronized void buka() {
088       value++;
089       notify();
090    }
091
092    private int value;
093 }
```

Contoh 27.5. Keluaran Program

```
Proses 12 hadir...
Proses 11 hadir...
Proses 7 hadir...
Proses 6 hadir...
Proses 13 hadir...
Proses 5 hadir...
Proses 10 hadir...
Proses 4 hadir...
Proses 3 hadir...
Proses 8 hadir...
Proses 14 hadir...
Proses 15 hadir...
Proses 2 hadir...
Proses 0 hadir...
Proses 1 hadir...
Proses 9 hadir...
SUPER PROSES siap...
Proses 15 siap...
Proses 14 siap...
Proses 13 siap...
Proses 12 siap...
Proses 11 siap...
Proses 10 siap...
Proses 9 siap...
Proses 8 siap...
Proses 7 siap...
Proses 6 siap...
Proses 5 siap...
Proses 4 siap...
Proses 3 siap...
Proses 2 siap...
Proses 1 siap...
Proses 0 siap...
```

27.3. Rangkuman

Ini merupakan ilustrasi dimana sebuah thread memegang kendali sinkronisasai lainnya.

27.4. Latihan

1. Perhatikan berkas program java berikut ini:

```
001 /* Gabungan Berkas:
```

```
002  * FirstSemaphore.java, Runner,java, Semaphore.java, Worker.java.
003  * Copyright (c) 2000 oleh Greg Gagne, Peter Galvin, Avi Silberschatz.
004  * Applied Operating Systems Concepts - John Wiley and Sons, Inc.
005  * Slightly modified by Rahmat M. Samik-Ibrahim.
006  *
007  * Informasi Singkat (RMS46):
008  * Threat.start()    --> memulai thread yang akan memanggil Threat.run().
009  * Threat.sleep(xxx) --> thread akan tidur selama xxx milidetik.
010  * try {...} catch (InterruptedException e) {} --> sarana terminasi program
011  */
012
013 public class FirstSemaphore
014 {
015     public static void main(String args[]) {
016         Semaphore  sem = new Semaphore(1);
017         Worker[]  bees = new Worker[NN];
018         for (int ii = 0; ii < NN; ii++)
019             bees[ii] = new Worker(sem, ii);
020         for (int ii = 0; ii < NN; ii++)
021             bees[ii].start();
022     }
023     private final static int NN=4;
024 }
025
026 // Worker ============================================================
027 class Worker extends Thread
028 {
029     public Worker(Semaphore sss, int nnn) {
030         sem     = sss;
031         wnumber = nnn;
032               wstring = WORKER + (new Integer(nnn)).toString();
033     }
034
035     public void run() {
036         while (true) {
037             System.out.println(wstring + PESAN1);
038             sem.P();
039             System.out.println(wstring + PESAN2);
040             Runner.criticalSection();
041             System.out.println(wstring + PESAN3);
042             sem.V();
043             Runner.nonCriticalSection();
044        }
045     }
046     private Semaphore sem;
047     private String    wstring;
048     private int       wnumber;
049     private final static String PESAN1=" akan masuk ke Critical Section.";
050     private final static String PESAN2=" berada di dalam Critical Section."
051     private final static String PESAN3=" telah keluar dari Critical Section
052     private final static String WORKER="PEKERJA ";
053 }
054
055 // Runner ============================================================
056 class Runner
057 {
058     public static void criticalSection() {
059         try {
060             Thread.sleep( (int) (Math.random() * CS_TIME * 1000) );
061         }
062         catch (InterruptedException e) { }
063     }
064
065     public static void nonCriticalSection() {
066         try {
067             Thread.sleep( (int) (Math.random() * NON_CS_TIME * 1000) );
068         }
069         catch (InterruptedException e) { }
```

```
070     }
071     private final static int     CS_TIME = 2;
072     private final static int NON_CS_TIME = 2;
073 }
074
075 // Semaphore ==========================================================
076 final class Semaphore
077 {
078     public Semaphore() {
079         value = 0;
080     }
081
082     public Semaphore(int v) {
083         value = v;
084     }
085
086     public synchronized void P() {
087         while (value <= 0) {
088             try {
089                 wait();
090             }
091             catch (InterruptedException e) { }
092         }
093         value --;
094     }
095
096     public synchronized void V() {
097         ++value;
098         notify();
099     }
100
101     private int value;
102 }
103
104 // END ================================================================
```

a. Berapakah jumlah object dari "Worker Class" yang akan terbentuk?

b. Sebutkan nama-nama object dari "Worker Class" tersebut!

c. Tuliskan/perkirakan keluaran (output) 10 baris pertama, jika menjalankan program ini!

d. Apakah keluaran pada butir "c" di atas akan berubah, jika parameter CS_TIME diubah menjadi dua kali NON_CS_TIME? Terangkan!

e. Apakah keluaran pada butir "c" di atas akan berubah, jika selain parameter CS_TIME diubah menjadi dua kali NON_CS_TIME, dilakukan modifikasi NN menjadi 10? Terangkan!

Rujukan

[KennethRosen1999] Kenneth H. Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[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.

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

Bagian V. Memori

Daftar Isi

Bab 28. Managemen Memori

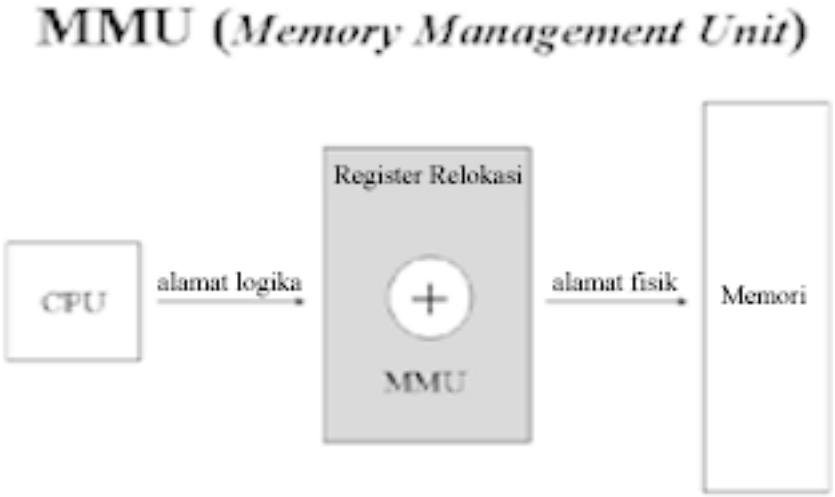
28.1. Pengalamatan

Memori adalah pusat kegiatan pada sebuah komputer, karena setiap proses yang akan dijalankan, harus melalui memori terlebih dahulu. CPU mengambil instruksi dari memori sesuai yang ada pada *Program Counter*. Instruksi dapat berupa menempatkan/menyimpan dari/ke alamat di memori, penambahan, dan sebagainya. Tugas sistem operasi adalah mengatur peletakan banyak proses pada suatu memori. Memori harus dapat digunakan dengan baik, sehingga dapat memuat banyak proses dalam suatu waktu. Dalam managemen memori ini, kita akan membahas bagaimana urutan alamat memori yang dibuat oleh program yang berjalan.

28.2. Ruang Alamat Logika dan Fisik

Alamat Logika adalah alamat yang dibentuk di CPU, disebut juga alamat virtual. Alamat fisik adalah alamat yang telihat oleh memori. Waktu kompilasi dan waktu pemanggilan menghasilkan daerah dimana alamat logika dan alamat fisik sama. Sedangkan pada waktu eksekusi menghasilkan alamat fisik dan logika yang berbeda. Kumpulan alamat logika yang dibuat oleh program adalah ruang alamat logika. Kumpulan alamat fisik yang berkorespondensi dengan alamat logika disebut ruang alamat fisik. Untuk mengubah dari alamat logika ke alamat fisik diperlukan suatu perangkat keras yang bernama *Memory Management Unit* (MMU).

Gambar 28.1. Memory Management Unit



Register utamanya disebut register relokasi. Nilai pada register relokasi bertambah setiap alamat dibuat oleh proses pengguna, pada waktu yang sama alamat ini dikirim ke memori. Program pengguna tidak dapat langsung mengakses memori. Ketika ada program yang menunjuk ke alamat memori, kemudian mengoperasikannya, dan menaruh lagi di memori, akan di lokasikan awal oleh MMU, karena program pengguna hanya berinterkasi dengan alamat logika. Pengubahan dari alamat logika ke alamat fisik adalah pusat dari managemen memori.

28.3. Pemanggilan Dinamis

Telah kita ketahui seluruh proses dan data berada memori fisik ketika dieksekusi. Ukuran dari memori fisik terbatas. Untuk mendapatkan utilisasi ruang memori yang baik, kita melakukan pemanggilan dinamis. Dengan pemanggilan dinamis, sebuah rutin tidak akan dipanggil sampai diperlukan. Semua rutin diletakkan di *disk*, dalam format yang dapat dialokasikan ulang. Program

utama di tempatkan di memori dan dieksekusi. Jika sebuah rutin memanggil rutin lainnya, maka akan dicek dulu apakah rutin yang dipanggil ada di dalam memori atau tidak, jika tidak ada maka *linkage loader* dipanggil untuk menempatkan rutin yang diinginkan ke memori dan memperbaharui tabel alamat program untuk menyesuaikan perubahan. Kemudian kontrol diletakan pada rutin yang baru dipanggil.

Keuntungan dari pemanggilan dinamis adalah rutin yang tidak digunakan tidak pernah dipanggil. Metode ini berguna untuk kode dalam jumlah banyak, ketika muncul kasus-kasus yang tidak lazim, seperti rutin yang salah. Dalam kode yang besar, walaupun ukuran kode besar, tapi yang dipanggil dapat jauh lebih kecil.

Pemanggilan Dinamis tidak memerlukan bantuan sistem operasi. Ini adalah tanggung-jawab para pengguna untuk merancang program yang mengambil keuntungan dari metode ini. Sistem operasi dapat membantu pembuat program dengan menyediakan kumpulan data rutin untuk mengimplementasi pemanggilan dinamis.

28.4. Penghubungan Dinamis dan Perpustakaan Bersama

Pada proses dengan banyak langkah, ditemukan juga penghubungan-penghubungan perpustakaan yang dinamis, dimana menghubungkan semua rutin yang ada di perpustakaan. Beberapa sistem operasi hanya mendukung penghubungan yang statis, dimana seluruh rutin yang ada dihubungkan ke dalam suatu ruang alamat. Setiap program memiliki salinan dari seluruh perpustakaan. Konsep penghubungan dinamis, serupa dengan konsep pemanggilan dinamis. Pemanggilan lebih banyak ditunda selama waktu eksekusi, dari pada lama penundaan oleh penghubungan dinamis. Keistimewaan ini biasanya digunakan dalam sistem kumpulan perpustakaan, seperti perpustakaan bahasa subrutin. Tanpa fasilitas ini, semua program dalam sebuah sistem, harus mempunyai salinan dari pustaka bahasa mereka (atau setidaknya referensi rutin oleh program) termasuk dalam tampilan yang dapat dieksekusi. Kebutuhan ini sangat boros baik untuk *disk*, maupun memori utama. Dengan pemanggilan dinamis, sebuah potongan dimasukkan ke dalam tampilan untuk setiap rujukan perpustakaan subrutin. Potongan ini adalah sebuah bagian kecil dari kode yang menunjukan bagaimana mealokasikan perpustakaan rutin di memori dengan tepat, atau bagaimana menempatkan pustaka jika rutin belum ada.

Ketika potongan ini dieksekusi, dia akan memeriksa dan melihat apakah rutin yang dibutuhkan sudah ada di memori. Jika rutin yang dibutuhkan tidak ada di memori, program akan menempatkannya ke memori. Jika rutin yang dibutuhkan ada di memori, maka potongan akan mengganti dirinya dengan alamat dari rutin, dan mengeksekusi rutin. Demikianlah, berikutnya ketika segmentasi kode dicapai, rutin pada perpustakaan dieksekusi secara langsung, dengan begini tidak ada biaya untuk penghubungan dinamis. Dalam skema ini semua proses yang menggunakan sebuah kumpulan bahasa, mengeksekusi hanya satu dari salinan kode perpustakaan.

Fasilitas ini dapat diperluas menjadi pembaharuan perpustakaan. Sebuah kumpulan data dapat ditempatkan lagi dengan versi yang lebih baru dan semua program yang merujuk ke perpustakaan akan secara otomatis menggunakan versi yang baru. Tanpa pemanggilan dinamis, semua program akan akan membutuhkan pemanggilan kembali, untuk dapat mengakses perpustakaan yang baru. Jadi semua program tidak secara sengaja mengeksekusi yang baru, perubahan versi perpustakaan, informasi versi dapat dimasukkan ke dalam memori, dan setiap program menggunakan informasi versi untuk memutuskan versi mana yang akan digunakan dari salinan perpustakaan. Sedikit perubahan akan tetap meggunakan nomor versi yang sama, sedangkan perubahan besar akan menambah satu versi sebelumnya. Karenanya program yang dikompilasi dengan versi yang baru akan dipengaruhi dengan perubahan yang terdapat di dalamnya. Program lain yang berhubungan sebelum pustaka baru diinstal, akan terus menggunakan pustaka lama. Sistem ini juga dikenal sebagai berbagi pustaka. Jadi seluruh pustaka yang ada dapat digunakan bersama-sama. Sistem seperti ini membutuhkan bantuan sistem operasi.

28.5. *Overlays*

Overlays berguna untuk memasukkan suatu proses yang membutuhkan memori lebih besar dari yang tersedia. Idenya untuk menjaga agar di dalam memori berisi hanya instruksi dan data yang

dibutuhkan dalam satuan waktu. Rutinnya dimasukkan ke memori secara bergantian.

Gambar 28.2. *Two-Pass Assembler*

Sebagai contoh, sebuah *two-pass assembler*. Selama pass1 dibangun sebuah tabel simbol, kemudian selama pass2, akan membuat kode bahasa mesin. Kita dapat mempartisi sebuah *assembler* menjadi kode pass1, kode pass2, dan simbol tabel, dan rutine biasa digunakan untuk kedua pass1 dan pass2.

Untuk menempatkan semuanya sekaligus, kita akan membutuhkan 200K memori. Jika hanya 150K yang tersedia, kita tidak dapat menjalankan proses. Bagaimana pun perhatikan bahwa pass1 dan pass2 tidak harus berada di memori pada saat yang sama. Kita mendefinisikan dua *overlays*. *Overlays* A untuk pass1, tabel simbol dan rutin, *overlays* dua untuk simbol tabel, rutin, dan pass2.

Kita menambahkan sebuah *driver overlays* (10K) dan mulai dengan *overlays* A di memori. Ketika selesai pass1, pindah ke *driver*, dan membaca *overlays* B ke dalam memori, menimpa *overlays* A, dan mengirim kontrol ke pass2. *Overlays* A butuh hanya 120K, dan B membutuhkan 150K memori. Kita sekarang dapat menjalankan *assembler* dalam 150K memori. Pemanggilan akan lebih cepat, karena lebih sedikit data yang ditransfer sebelum eksekusi dimulai. Jalan program akan lebih lambat, karena ekstra M/K dari kode *overlays* B melalui *overlays* A.

Seperti dalam pemanggilan dinamis, *overlays* tidak membutuhkan bantuan dari sistem operasi. Implementasi dapat dilakukan secara lengkap oleh pengguna dengan berkas struktur yang sederhana, membaca dari berkas ke memori, dan pindah dari memori tersebut, dan mengeksekusi instruksi yang baru dibaca. Sistem operasi hanya memperhatikan jika ada lebih banyak M/K dari biasanya.

Di sisi lain pemrogram harus merancang program dengan struktur *overlays* yang layak. Tugas ini membutuhkan pengetahuan yang lengkap tentang struktur dari program, kode dan struktur data.

Pemakaian dari *overlays*, dibatasi oleh komputer mikro, dan sistem lain yang mempunyai batasan jumlah memori fisik, dan kurangnya dukungan perangkat keras, untuk teknik yang lebih maju. Teknik otomatis menjalankan program besar dalam dalam jumlah memori fisik yang terbatas, lebih diutamakan.

28.6. Rangkuman

Memori adalah pusat kegiatan pada sebuah komputer, karena setiap proses yang akan dijalankan harus memlalui memori terlebih dahulu. CPU mengambil instruksi dari memori sesuai dengan yang ada pada *program counter*. Tugas sistem operasi adalah mengatur peletakan banyak proses pada suatu memori.

Sebelum masuk ke memori, suatu proses harus menunggu di sebuah *input queue*, setelah itu barulah mereka akan diberikan alamat pada memori. Pemberian alamat dapat dilakukan pada waktu *compile*, waktu pemanggilan, dan waktu eksekusi. Alamat logika (virtual) adalah alamat yang dibentuk di CPU, sedangkan alamat fisik adalah alamat yang terlihat oleh memori. Seluruh proses dan data berada dalam memori ketika dieksekusi. Berhubung ukuran dari memori fisik terbatas, kita harus melakukan pemanggilan dinamis untuk mendapatkan utilisasi ruang memori yang baik.

28.7. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:
   - *Address Space: "Logical" vs. "Physical"*.
   - *Virtual Memory Allocation Strategy: "Global" vs. "Local Replacement"*.
2. Coba jelaskan tahapan-tahapan agar suatu proses bisa masuk ke dalam memori!
3. Jelaskan apa yang dimaksud dengan alamat logika dan alamat fisik!
4. Jelaskan apa yang dimaksud dengan pemanggilan dinamis beserta kegunaannya!
5. Apakah kegunaan dari *overlays*?

28.8. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts*: 6th ed. John Wiley & Sons

Tanenbaum, Andrew S. Woodhull, Albert S. 1997. *Operating Systems Design and Implementation: Second Edition*. Prentice Hall.

http://css.uni.edu/

http://allan.ultra.nyu.edu/~gottlieb/courses/1999-00-spring/os/lecture-11.html

http://www.cs.wisc.edu/~solomon/cs537/paging.html

http://www.cs.princeton.edu/courses/archive/spring02/cs217/lectures/paging.pdf

http://cwx.prenhall.com/bookbind/pubbooks/tanenbaum/chapter0/deluxe.html

http://www.cs.jhu.edu/~yairamir/cs418/os5/

Bab 29. Alokasi Memori

29.1. *Swap*

Sebuah proses harus berada di dalam memori untuk dapat dieksekusi. Sebuah proses, bagaimana pun juga, dapat ditukar sementara keluar memori ke sebuah penyimpanan sementara, dan kemudian dibawa masuk lagi ke memori untuk melanjutkan pengeksekusian. Sebagai contoh, asumsikan sebuah *multiprogramming environment*, dengan penjadualan algoritma penjadualan CPU *round-robin*. Ketika kuantum habis, pengatur memori akan mulai menukar proses yang telah selesai, dan memasukkan proses yang lain ke dalam memori yang sudah bebas. Sementara di saat yang bersamaan, penjadual CPU akan mengalokasikan waktu untuk proses lain di dalam memori. Ketika waktu kuantum setiap proses sudah habis, proses tersebut akan ditukar dengan proses lain. Idealnya, manajer memori dapat melakukan penukaran proses-proses tersebut dengan cukup cepat sehingga beberapa proses akan selalu berada di dalam memori dan siap untuk dieksekusi saat penjadual CPU hendak menjadual CPU. Lama kuantum pun harus cukup besar sehingga jumlah komputasi yang dilakukan selama terjadi pertukaran cukup masuk akal.

Variasi dari kebijakan swapping ini, digunakan untuk algoritma penjadualan berbasis prioritas. Jika proses dengan prioritas lebih tinggi tiba dan meminta layanan, manajer memori dapat menukar keluar proses-proses yang prioritasnya rendah, sehingga proses-proses yang prioritasnya lebih tinggi tersebut dapat dieksekusi. Setelah proses-proses yang memiliki prioritas lebih tinggi tersebut selesai dieksekusi, proses-proses dengan prioritas rendah dapat ditukar kembali ke dalam memori dan dilanjutkan eksekusinya. Cara ini disebut juga dengan metoda ***roll out, roll in***.

Pada umumnya, proses yang telah ditukar keluar akan ditukar kembali menempati ruang memori yang sama dengan yang ditempatinya sebelum proses tersebut keluar dari memori. Pembatasan ini dinyatakan menurut metoda pemberian alamat. Apabila pemberian alamat dilakukan pada saat waktu pembuatan atau waktu pemanggilan, maka proses tersebut tidak dapat dipindahkan ke lokasi memori lain. Tetapi apabila pemberian alamat dilakukan pada saat waktu eksekusi, maka proses tersebut dapat ditukar kembali ke dalam ruang memori yang berbeda, karena alamat fisiknya dihitung pada saat pengeksekusian.

Penukaran membutuhkan sebuah penyimpanan sementara. Penyimpanan sementara pada umumnya adalah sebuah *fast disk*, dan harus cukup untuk menampung salinan dari seluruh gambaran memori untuk semua pengguna, dan harus mendukung akses langsung terhadap gambaran memori tersebut. Sistem mengatur *ready queue* yang berisikan semua proses yang gambaran memorinya berada di memori dan siap untuk dijalankan. Saat sebuah penjadual CPU ingin menjalankan sebuah proses, ia akan memeriksa apakah proses yang mengantri di *ready queue* tersebut sudah berada di dalam memori tersebut atau belum. Apabila belum, penjadual CPU akan melakukan penukaran keluar terhadap proses-proses yang berada di dalam memori sehingga tersedia tempat untuk memasukkan proses yang hendak dieksekusi tersebut. Setelah itu *register* dikembalikan seperti semula dan proses yang diinginkan akan dieksekusi.

Waktu pergantian isi dalam sebuah sistem yang melakukan penukaran pada umumnya cukup tinggi. Untuk mendapatkan gambaran mengenai waktu pergantian isi, akan diilustrasikan sebuah contoh. Misalkan ada sebuah proses sebesar 1 MB, dan media yang digunakan sebagai penyimpanan sementara adalah sebuah *hard disk* dengan kecepatan transfer 5 MBps. Waktu yang dibutuhkan untuk mentransfer proses 1 MB tersebut dari atau ke dalam memori adalah:

1000 KB/5000 KBps = 1/5 detik = 200 milidetik

Apabila diasumsikan *head seek* tidak dibutuhkan dan rata-rata waktu latensi adalah 8 milidetik, satu proses penukaran memakan waktu 208 milidetik. Karena kita harus melakukan proses penukaran sebanyak 2 kali, (memasukkan dan mengeluarkan dari memori), maka keseluruhan waktu yang dibutuhkan adalah 416 milidetik.

Untuk penggunaan CPU yang efisien, kita menginginkan waktu eksekusi kita relatif panjang apabila dibandingkan dengan waktu penukaran kita. Sehingga, misalnya dalam penjuadualan CPU menggunakan metoda *round robin*, kuantum yang kita tetapkan harus lebih besar dari 416 milidetik.

Bagian utama dari waktu penukaran adalah waktu transfer. Besar waktu transfer berhubungan langsung dengan jumlah memori yang di-tukar. Jika kita mempunyai sebuah komputer dengan memori utama 128 MB dan sistem operasi memakan tempat 5 MB, besar proses pengguna maksimal adalah 123 MB. Bagaimana pun juga, proses pengguna pada kenyataannya dapat berukuran jauh lebih kecil dari angka tersebut. Bahkan terkadang hanya berukuran 1 MB. Proses sebesar 1 MB dapat ditukar hanya dalam waktu 208 milidetik, jauh lebih cepat dibandingkan menukar proses sebesar 123 MB yang akan menghabiskan waktu 24.6 detik. Oleh karena itu, sangatlah berguna apabila kita mengetahui dengan baik berapa besar memori yang dipakai oleh proses pengguna, bukan sekedar dengan perkiraan saja. Setelah itu, kita dapat mengurangi besar waktu penukaran dengan cara hanya menukar proses-proses yang benar-benar membutuhkannya. Agar metoda ini bisa dijalankan dengan efektif, pengguna harus menjaga agar sistem selalu memiliki informasi mengenai perubahan kebutuhan memori. Oleh karena itu, proses yang membutuhkan memori dinamis harus melakukan pemanggilan sistem (permintaan memori dan pelepasan memori) untuk memberikan informasi kepada sistem operasi akan perubahan kebutuhan memori.

Penukaran dipengaruhi oleh banyak faktor. Jika kita hendak menukar suatu proses, kita harus yakin bahwa proses tersebut siap. Hal yang perlu diperhatikan adalah kemungkinan proses tersebut sedang menunggu M/K. Apabila M/K secara asinkron mengakses memori pengguna untuk M/K *buffer*, maka proses tersebut tidak dapat ditukar. Bayangkan apabila sebuah operasi M/K berada dalam antrian karena peralatan M/K-nya sedang sibuk. Kemudian kita hendak mengeluarkan proses P1 dan memasukkan proses P2. Operasi M/K mungkin akan berusaha untuk memakai memori yang sekarang seharusnya akan ditempati oleh P2. Cara untuk mengatasi masalah ini adalah:

1. Hindari menukar proses yang sedang menunggu M/K.

2. Lakukan eksekusi operasi M/K hanya di *buffer* sistem operasi.

Hal tersebut akan menjaga agar transfer antara *buffer* sistem operasi dan proses memori hanya terjadi saat si proses ditukar kedalam.

Pada masa sekarang ini, proses penukaran secara dasar hanya digunakan di sedikit sistem. Hal ini dikarenakan penukaran menghabiskan terlalu banyak waktu tukar dan memberikan waktu eksekusi yang terlalu kecil sebagai solusi dari managemen memori. Akan tetapi, banyak sistem yang menggunakan versi modifikasi dari metoda penukaran ini.

Salah satu sistem operasi yang menggunakan versi modifikasi dari metoda penukaran ini adalah UNIX. Penukaran berada dalam keadaan non-aktif, sampai apabila ada banyak proses yang berjalan yang menggunakan memori yang besar. Penukaran akan berhenti lagi apabila jumlah proses yang berjalan sudah berkurang.

Pada awal pengembangan komputer pribadi, tidak banyak perangkat keras (atau sistem operasi yang memanfaatkan perangkat keras) yang dapat mengimplementasikan memori managemen yang baik, melainkan digunakan untuk menjalankan banyak proses berukuran besar dengan menggunakan versi modifikasi dari metoda penukaran. Salah satu contoh yang baik adalah Microsoft Windows 3.1, yang mendukung eksekusi proses berkesinambungan. Apabila suatu proses baru hendak dijalankan dan tidak terdapat cukup memori, proses yang lama perlu dimasukkan ke dalam *disk*. Sistem operasi ini, bagaimana pun juga, tidak mendukung penukaran secara keseluruhan karena yang lebih berperan menentukan kapan proses penukaran akan dilakukan adalah pengguna dan bukan penjadual CPU. Proses-proses yang sudah dikeluarkan akan tetap berada di luar memori sampai pengguna memilih proses yang hendak dijalankan. Sistem-sistem operasi Microsoft selanjutnya, seperti misalnya Windows NT, memanfaatkan fitur Unit Managemen Memori.

29.2. Proteksi Memori

Proteksi memori adalah sebuah sistem yang mencegah sebuah proses dari pengambilan memori proses lain yang sedang berjalan pada komputer yang sama dan pada saat yang sama pula. Proteksi memori selalu mempekerjakan hardware (Memory Managemen Unit) dan sistem software untuk mengalokasikan memori yang berbeda untuk proses yang berbeda dan untuk mengatasi exception yang muncul ketika sebuah proses mencoba untuk mengakses memori di luar batas. Efektivitas dari

proteksi memori berbeda antara sistem operasi yang satu dengan yang lainnya. Ada beberapa cara yang berbeda untuk mencapai proteksi memori. Segmentasi dan pemberian halaman adalah dua metode yang paling umum digunakan.

Segmentasi adalah skema managemen memori dengan cara membagi memori menjadi segmen-segmen. Dengan demikian, sebuah program dibagi menjadi segmen-segmen. Segmen adalah sebuah unit logis, yaitu unit yang terdiri dari beberapa bagian yang berjenis yang sama. Segmen dapat terbagi jika terdapat elemen di tabel segmen yang berasal dari dua proses yang berbeda yang menunjuk pada alamat fisik yang sama. Saling berbagi ini muncul di level segmen dan pada saat ini terjadi semua informasi dapat turut terbagi. Proteksi dapat terjadi karena ada bit proteksi yang berhubungan dengan setiap elemen dari segmen tabel. Bit-proteksi ini berguna untuk mencegah akses ilegal ke memori. Caranya: menempatkan sebuah array di dalam segmen itu sehingga perangkat keras managemen memori secara otomatis akan mengecek indeks array-nya legal atau tidak.

Pemberian halaman merupakan metode yang paling sering digunakan untuk proteksi memori. Pemberian halaman adalah suatu metoda yang memungkinkan suatu alamat fisik memori yang tersedia dapat tidak berurutan. Proteksi memori di lingkungan halaman bisa dilakukan dengan cara memproteksi bit-bit yang berhubungan dengan setiap frame. Biasanya bit-bit ini disimpan didalam sebuah tabel halaman. Satu bit bisa didefinisikan sebagai baca-tulis atau hanya baca saja. Setiap referensi ke memori menggunakan tabel halaman untuk menemukan nomor frame yang benar. Pada saat alamat fisik sedang dihitung, bit proteksi bisa mengecek bahwa kita tidak bisa menulis ke mode tulis saja. Untuk lebih jelasnya, kedua metode ini akan dijelaskan pada bab berikutnya.

Ketika sebuah program berjalan di luar batas memori, DOS, Windows 3.x, Windows 95/98 dan sistem operasi lain sebelumnya, tidak dapat mengatasi hal tersebut. Sistem operasi seperti Unix, OS/2, Windows NT, 2000 dan XP lebih tahan dan mengizinkan program tersebut untuk diberhentikan tanpa mempengaruhi program lain yang sedang aktif.

29.3. Alokasi Memori Berkesinambungan

Memori utama harus dapat melayani baik sistem operasi maupun proses pengguna. Oleh karena itu kita harus mengalokasikan pembagian memori seefisien mungkin. Salah satunya adalah dengan cara **alokasi memori berkesinambungan**. Alokasi memori berkesinambungan berarti alamat memori diberikan kepada proses secara berurutan dari kecil ke besar. Keuntungan menggunakan alokasi memori berkesinambungan dibandingkan menggunakan alokasi memori tidak berkesinambungan adalah:

1. Sederhana
2. Cepat
3. Mendukung proteksi memori

Sedangkan kerugian dari menggunakan alokasi memori berkesinambungan adalah apabila tidak semua proses dialokasikan di waktu yang sama, akan menjadi sangat tidak efektif sehingga mempercepat habisnya memori.

Alokasi memori berkesinambungan dapat dilakukan baik menggunakan sistem partisi banyak, maupun menggunakan sistem partisi tunggal. Sistem partisi tunggal berarti alamat memori yang akan dialokasikan untuk proses adalah alamat memori pertama setelah pengalokasian sebelumnya. Sedangkan sistem partisi banyak berarti sistem operasi menyimpan informasi tentang semua bagian memori yang tersedia untuk dapat diisi oleh proses-proses (disebut lubang). Sistem partisi banyak kemudian dibagi lagi menjadi sistem partisi banyak tetap, dan sistem partisi banyak dinamis. Hal yang membedakan keduanya adalah untuk sistem partisi banyak tetap, memori dipartisi menjadi blok-blok yang ukurannya tetap yang ditentukan dari awal. Sedangkan sistem partisi banyak dinamis artinya memori dipartisi menjadi bagian-bagian dengan jumlah dan besar yang tidak tentu. Untuk selanjutnya, kita akan memfokuskan pembahasan pada sistem partisi banyak.

Sistem operasi menyimpan sebuah tabel yang menunjukkan bagian mana dari memori yang

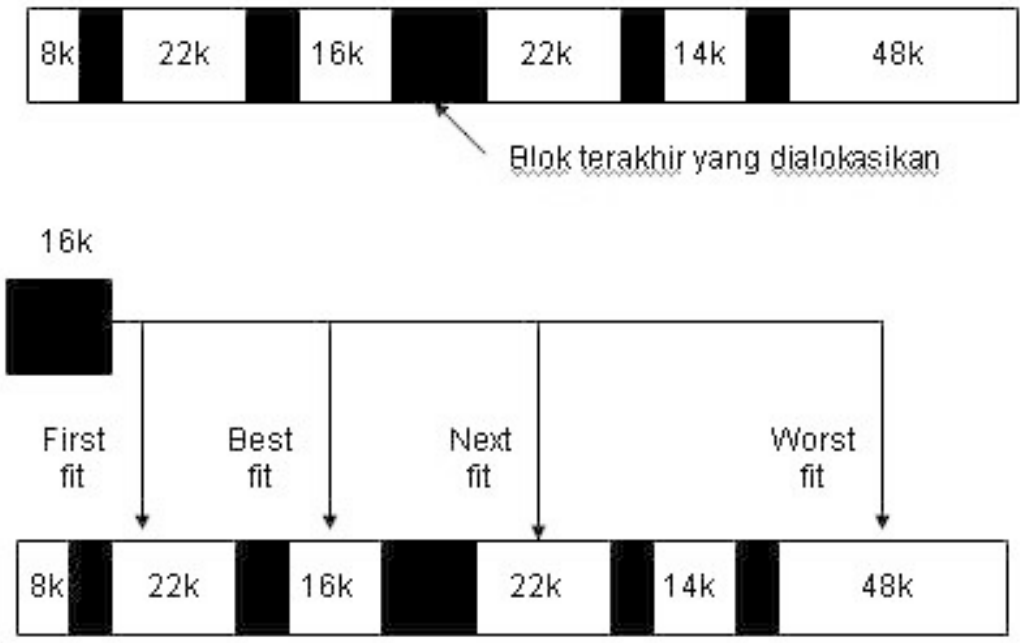
memungkinkan untuk menyimpan proses, dan bagian mana yang sudah diisi. Pada intinya, seluruh memori dapat diisi oleh proses pengguna. Saat sebuah proses datang dan membutuhkan memori, CPU akan mencari lubang yang cukup besar untuk menampung proses tersebut. Setelah menemukannya, CPU akan mengalokasikan memori sebanyak yang dibutuhkan oleh proses tersebut, dan mempersiapkan sisanya untuk menampung proses-proses yang akan datang kemudian (seandainya ada).

Saat proses memasuki sistem, proses akan dimasukkan ke dalam antrian masukan. Sistem operasi akan menyimpan besar memori yang dibutuhkan oleh setiap proses dan jumlah memori kosong yang tersedia, untuk menentukan proses mana yang dapat diberikan alokasi memori. Setelah sebuah proses mendapat alokasi memori, proses tersebut akan dimasukkan ke dalam memori. Setelah proses tersebut dimatikan, proses tersebut akan melepas memori tempat dia berada, yang mana dapat diisi kembali oleh proses lain dari antrian masukan.

Sistem operasi setiap saat selalu memiliki catatan jumlah memori yang tersedia dan antrian masukan. Sistem operasi dapat mengatur antrian masukan berdasarkan algoritma penjadualan yang digunakan. Memori dialokasikan untuk proses sampai akhirnya kebutuhan memori dari proses selanjutnya tidak dapat dipenuhi (tidak ada lubang yang cukup besar untuk menampung proses tersebut). Sistem operasi kemudian dapat menunggu sampai ada blok memori cukup besar yang kosong, atau dapat mencari proses lain di antrian masukan yang kebutuhan memorinya memenuhi jumlah memori yang tersedia.

Pada umumnya, kumpulan lubang-lubang dalam berbagai ukuran tersebar di seluruh memori sepanjang waktu. Apabila ada proses yang datang, sistem operasi akan mencari lubang yang cukup besar untuk menampung memori tersebut. Apabila lubang yang tersedia terlalu besar, akan dipecah menjadi 2. Satu bagian akan dialokasikan untuk menerima proses tersebut, sementara bagian lainnya tidak digunakan dan siap menampung proses lain. Setelah proses selesai, proses tersebut akan melepas memori dan mengembalikannya sebagai lubang-lubang. Apabila ada dua lubang yang kecil yang berdekatan, keduanya akan bergabung untuk membentuk lubang yang lebih besar. Pada saat ini, sistem harus memeriksa apakah ada proses yang menunggu yang dapat dimasukkan ke dalam ruang memori yang baru terbentuk tersebut.

Gambar 29.1. Permasalahan alokasi penyimpanan dinamis



Disadur dari berbagai sumber di internet.

Hal ini disebut **Permasalahan alokasi penyimpanan dinamis**, yakni bagaimana memenuhi permintaan sebesar *n* dari kumpulan lubang-lubang yang tersedia. Ada berbagai solusi untuk mengatasi hal ini, yaitu:

1. *First fit*: Mengalokasikan lubang pertama ditemukan yang besarnya mencukupi. Pencarian dimulai dari awal.
2. *Best fit*: Mengalokasikan lubang dengan besar minimum yang mencukupi permintaan.
3. *Next fit*: Mengalokasikan lubang pertama ditemukan yang besarnya mencukupi. Pencarian dimulai dari akhir pencarian sebelumnya.
4. *Worst fit*: Mengalokasikan lubang terbesar yang ada.

Memilih yang terbaik diantara keempat metoda diatas adalah sepenuhnya tergantung kepada pengguna, karena setiap metoda memiliki kelebihan dan kekurangan masing-masing. Menggunakan *best fit* dan *worst fit* berarti kita harus selalu memulai pencarian lubang dari awal, kecuali apabila lubang sudah disusun berdasarkan ukuran. Metode *worst fit* akan menghasilkan sisa lubang yang terbesar, sementara metoda *best fit* akan menghasilkan sisa lubang yang terkecil.

29.4. Fragmentasi

Fragmentasi adalah munculnya lubang-lubang yang tidak cukup besar untuk menampung permintaan dari proses. Fragmentasi dapat berupa fragmentasi internal maupun fragmentasi eksternal. Fragmentasi ekstern muncul apabila jumlah keseluruhan memori kosong yang tersedia memang mencukupi untuk menampung permintaan tempat dari proses, tetapi letaknya tidak berkesinambungan atau terpecah menjadi beberapa bagian kecil sehingga proses tidak dapat masuk. Sedangkan fragmentasi intern muncul apabila jumlah memori yang diberikan oleh penjadual CPU untuk ditempati proses lebih besar daripada yang diminta proses karena adanya selisih antara permintaan proses dengan alokasi lubang yang sudah ditetapkan.

Algoritma alokasi penyimpanan dinamis mana pun yang digunakan, tetap tidak bisa menutup kemungkinan terjadinya fragmentasi. Bahkan hal ini bisa menjadi fatal. Salah satu kondisi terburuk adalah apabila kita memiliki memori terbuang setiap dua proses. Apabila semua memori terbuang itu digabungkan, bukan tidak mungkin akan cukup untuk menampung sebuah proses. Sebuah contoh statistik menunjukkan bahwa saat menggunakan metoda *first fit*, bahkan setelah dioptimisasi, dari N blok teralokasi, sebanyak 0.5N blok lain akan terbuang karena fragmentasi. Jumlah sebanyak itu berarti kurang lebih setengah dari memori tidak dapat digunakan. Hal ini disebut dengan **aturan 50%**.

Fragmentasi ekstern dapat diatasi dengan beberapa cara, diantaranya adalah:

1. **Pemadatan**, yaitu mengatur kembali isi memori agar memori yang kosong diletakkan bersama di suatu bagian yang besar, sehingga proses dapat masuk ke ruang memori kosong tersebut.
2. **Penghalamanan**.
3. **Segmentasi**.

Fragmentasi intern hampir tidak dapat dihindarkan apabila kita menggunakan sistem partisi banyak berukuran tetap, mengingat besar *hole* yang disediakan selalu tetap.

29.5. Rangkuman

Sebuah proses dapat di-*swap* sementara keluar memori ke sebuah *backing store* untuk kemudian dibawa masuk lagi ke memori untuk melanjutkan pengeksekusian. Salah satu proses yang memanfaatkan metode ini adalah *roll out, roll in*, yang pada intinya adalah proses *swapping*

berdasarkan prioritas.

Agar *main memory* dapat melayani sistem operasi dan proses dengan baik, dibutuhkan pembagian memori seefisien mungkin. Salah satunya adalah dengan *contiguous memory allocation*. Artinya alamat memori diberikan kepada OS secara berurutan dari kecil ke besar. Ruang memori yang masih kosong dan dapat dialokasikan untuk proses disebut *hole*. Metode pencarian hole ada beberapa, diantaranya adalah *first fit, next fit, best fit, worst fit*. Masalah yang sering muncul dalam pengalamatan memori adalah fragmentasi (baik intern maupun ekstern), yaitu munculnya *hole-hole* yang tidak cukup besar untuk menampung permintaan dari proses.

29.6. Latihan

1. Sebutkan faktor-faktor yang mempengaruhi proses swapping!
2. Sebutkan keuntungan menggunakan *contiguous memory allocation* dibandingkan dengan *non-contiguous memory allocation*!
3. Apakah yang dimaksud dengan **permasalahan *storage-allocation* dinamis** , dan sebutkan serta jelaskan solusi untuk mengatasi permasalahan tersebut!
4. Jelaskan perbedaan mengenai fragmentasi intern dengan fragmentasi ekstern!

29.7. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts*: 6th ed. John Wiley & Sons

Tanenbaum,Andrew S. Woodhull, Albert S. 1997. *Operating Systems Design and Implementation: Second Edition*. Prentice Hall.

http://css.uni.edu/

http://allan.ultra.nyu.edu/~gottlieb/courses/1999-00-spring/os/lecture-11.html

http://www.cs.wisc.edu/~solomon/cs537/paging.html

http://www.cs.princeton.edu/courses/archive/spring02/cs217/lectures/paging.pdf

http://cwx.prenhall.com/bookbind/pubbooks/tanenbaum/chapter0/deluxe.html

http://www.cs.jhu.edu/~yairamir/cs418/os5/

http://en.wikipedia.org/wiki/Memory_protection

Bab 30. Pemberian Halaman

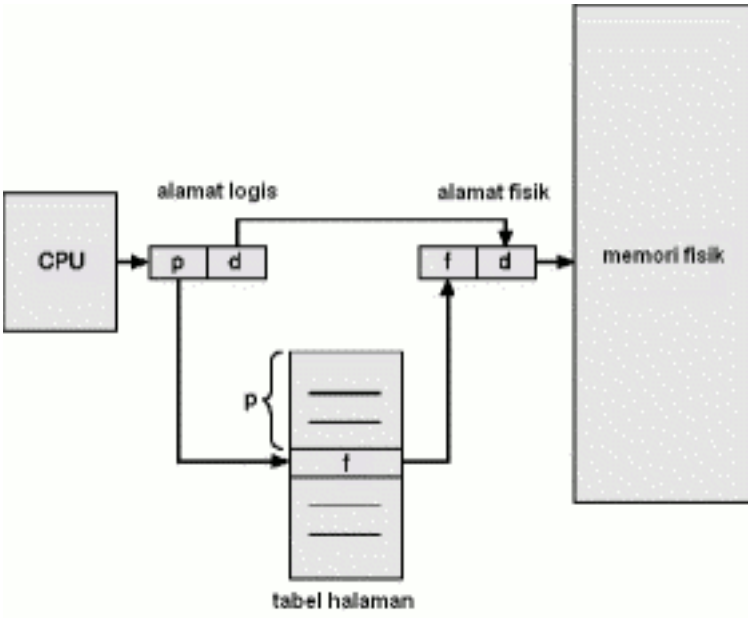
Yang dimaksud dengan pemberian halaman adalah suatu metoda yang memungkinkan suatu alamat fisik memori yang tersedia dapat tidak berurutan. Pemberian halaman bisa menjadi solusi untuk pemecahan masalah luar. Untuk bisa mengimplementasikan solusi ini adalah melalui pengunaan dari skema pemberian halaman. Dengan pemberian halaman bisa mencegah masalah penting dari pengepasan besar ukuran memori yang bervariasi kedalam penyimpanan cadangan. Ketika beberapa pecahan kode dari data yang tersisa di memori utama perlu untuk ditukar keluar, harus ditemukan ruang untuk penyimpanan cadangan. Masalah pemecahan kode didiskusikan dengan kaitan bahwa pengaksesannya lebih lambat. Biasanya bagian yang menunjang untuk pemberian halaman telah ditangani oleh perangkat keras. Bagaimana pun, desain yang ada baru-baru ini telah mengimplementasikan dengan menggabungkan perangkat keras dan sistem operasi, terutama pada prosesormikro 64 bit .

30.1. Metoda Dasar

Jadi metoda dasar yang digunakan adalah dengan memecah memori fisik menjadi blok-blok berukuran tetap yang akan disebut sebagai frame. selanjutnya memori logis juga dipecah menjadi blok-blok dengan ukuran yang sama disebut sebagai halaman. Selanjutnya kita membuat suatu tabel halaman yang akan menterjemahkan memori logis kita kedalam memori fisik. Jika suatu proses ingin dieksekusi maka memori logis akan melihat dimanakah dia akan ditempatkan di memori fisik dengan melihat kedalam tabel halamannya.

Untuk jelasnya bisa dilihat pada Gambar 30.1, “Penerjemahan Halaman”. Kita lihat bahwa setiap alamat yang dihasilkan oleh CPU dibagi-bagi menjadi dua bagian yaitu sebuah nomor halaman (p) dan sebuah offset halaman (d). Nomor halaman ini akan digunakan sebagai indeks untuk tabel halaman. Tabel halaman mengandung basis alamat dari tiap-tiap halaman di memori fisik. Basis ini dikombinasikan dengan offset halaman untuk menentukan alamat memori fisik yang dikirim ke unit memori.

Gambar 30.1. Penerjemahan Halaman



Sumber: Silberschatz et. al.

30.2. Dukungan Perangkat Keras

Setiap sistem operasi mempunyai caranya tersendiri untuk menyimpan tabel halaman. Biasanya sistem operasi mengalokasikan sebuah tabel halaman untuk setiap proses. sebuah penunjuk ke tabel halaman disimpan dengan nilai register yang lain didalam blok pengontrol proses.

Salah satu dukungan perangkat keras adalah dengan menggunakan apa yang dinamakan **TLB (translation look aside-buffer)**. TLB adalah asosiatif, memori berkecepatan tinggi. Setiap bagian di TLB terdiri dari kunci dan nilai. Ketika kita ingin mendapatkan alamat fisik memori maka alamat logikal dari CPU akan dibandingkan dengan nilai yang ada di TLB. jika nilainya ketemu maka dinamakan TLB *hit* dan jika nilainya tidak ketemu dinamakan TLB *miss*. Biasanya ukurannya kecil antara 64 sampai 1024.

Persentasi dari beberapa kali TLB *hit* adalah disebut *hit ratio*. *hit ratio* 80% berarti menemukan nomor halaman yang ingin kita cari didalam TLB sebesar 80%. Jika waktu akses ke TLB memakan waktu 20 nanodetik dan akses ke memori memakan waktu sebesar 100 nanodetik maka total waktu kita memetakan memori adalah 120 nanodetik jika TLB *hit*. dan jika TLB *miss* maka total waktunya adalah 220 nanodetik. Jadi untuk mendapatkan waktu akses memori yang efektif maka kita harus membagi-bagi tiap kasus berdasarkan kemungkinannya:

waktu akses yang efektif = 80% x 120 x 20% x 220
= 140 nanodetik

30.3. Proteksi

Proteksi memori dilingkungan halaman bisa dilakukan dengan cara memproteksi bit-bit yang berhubungan dengan setiap frame. Biasanya bit-bit ini disimpan didalam sebuah tabel halaman. satu bit bisa didefinisikan sebagai baca-tulis atau hanya baca saja. Setiap referensi ke memori menggunakan tabel halaman untuk menemukan nomor frame yang benar. pada saat alamat fisik sedang dihitung, bit proteksi bisa mengecek bahwa kita tidak bisa menulis ke mode tulis saja.

30.4. Keuntungan dan Kerugian Pemberian Halaman

- Jika kita membuat ukuran dari masing-masing halaman menjadi lebih besar.

  Keuntungan: Akses memori akan relatif lebih cepat.

  Kerugian: Kemungkinan terjadinya fragmentasi intern sangat. besar

- Jika kita membuat ukuran dari masing-masing halaman menjadi lebih kecil.

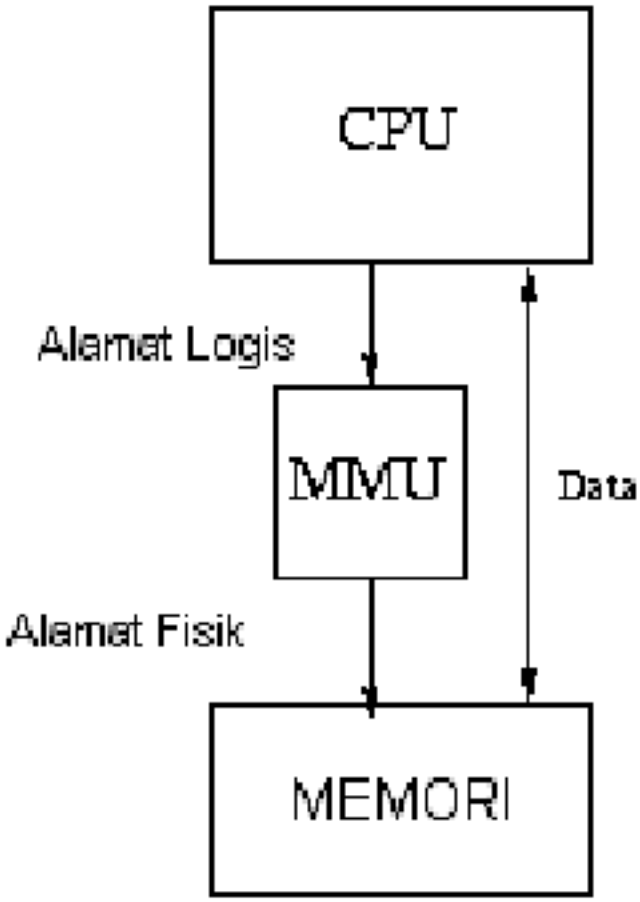
  Keuntungan: Kemungkinan terjadinya internal Framentasi akan menjadi lebih kecil.

  Kerugian: Akses memori akan relatif lebih lambat.

30.5. Tabel Halaman

Sebagian besar komputer modern memiliki perangkat keras istimewa yaitu **unit managemen memori** (MMU). Unit tersebut berada diantara CPU dan unit memori. Jika CPU ingin mengakses memori (misalnya untuk memanggil suatu instruksi atau memanggil dan menyimpan suatu data), maka CPU mengirimkan alamat memori yang bersangkutan ke MMU, yang akan menerjemahkannya ke alamat lain sebelum melanjutkannya ke unit memori. Alamat yang dihasilkan oleh CPU, setelah adanya pemberian indeks atau aritmatik ragam pengalamatan lainnya disebut **alamat logis (*virtual address*)**. Sedangkan alamat yang didapatkan setelah diterjemahkan oleh CPU disebut **alamat fisik (*physical address*)**.

Gambar 30.2. Struktur MMU



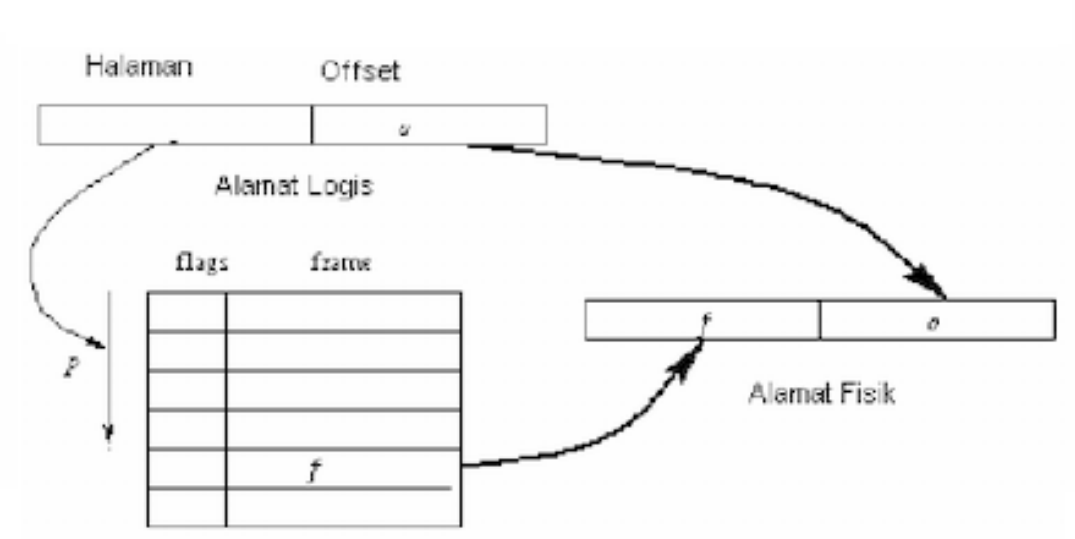
Sumber: Operating System karya Avi Silberschatz, Peter Galvin, dan Greg Gagne, 2000

Biasanya, penterjemahan dilakukan di granularitas dari suatu halaman. Setiap halaman mempunyai pangkat 2 bytes, diantara 1024 dan 8192 bytes. Jika alamat logis ***p*** dipetakan ke alamat fisik ***f*** (dimana ***p*** adalah kelipatan dari ukuran halaman), maka alamat ***p+o*** dipetakan ke alamat fisik ***f+o*** untuk setiap ofset ***o*** kurang dari ukuran halaman. Dengan kata lain, setiap halaman dipetakan ke *contigous region* di alamat fisik yang disebut *frame*.

MMU yang mengizinkan *contigous region* dari alamat logis dipetakan ke *frame* yang tersebar disekitar alamat fisik membuat sistem operasi lebih mudah pekerjaannya saat mengalokasikan memori. Lebih penting lagi, MMU juga mengizinkan halaman yang tidak sering digunakan bisa disimpan di *disk*. Cara kerjanya adalah sbb: Tabel yang digunakan oleh MMU mempunyai bit sahih untuk setiap halaman di bagian alamat logis. Jika bit tersebut di set, maka penterjemahan oleh

alamat logis di halaman itu berjalan normal. Akan tetapi jika dihapus, adanya usaha dari CPU untuk mengakses suatu alamat di halaman tersebut menghasilkan suatu interupsi yang disebut *page fault trap*. Sistem operasi telah mempunyai *interrupt handler* untuk kesalahan halaman, juga bisa digunakan untuk mengatasi interupsi jenis yang lain. *Handler* inilah yang akan bekerja untuk mendapatkan halaman yang diminta ke memori.

Gambar 30.3. Skema Tabel Halaman Dua tingkat



Sumber: http://www.cs.wisc.edu/~solomon/cs537/paging.html

Untuk lebih jelasnya, saat kesalahan halaman dihasilkan untuk halaman ***p1***, ***interrupt handler*** melakukan hal-hal berikut ini:

- Mencari dimana isi dari halaman ***p1*** disimpan di *disk*. Sistem operasi menyimpan informasi ini di dalam tabel. Ada kemungkinan bahwa halaman tersebut tidak ada dimana-mana, misalnya pada kasus saat referensi memori adalah *bug*. Pada kasus tersebut , sistem operasi mengambil beberapa langkah kerja seperti mematikan prosesnya. Dan jika diasumsikan halamannya berada dalam *disk*:
- Mencari halaman lain yaitu ***p2*** yang dipetakan ke *frame* lain ***f*** dari alamat fisik yang tidak banyak dipergunakan.
- Menyalin isi dari *frame* ***f*** keluar dari *disk*.
- Menghapus bit sahih dari halaman ***p2*** sehingga sebagian referensi dari halaman ***p2*** akan menyebabkan kesalahan halaman.
- Menyalin data halaman ***p1*** dari *disk* ke *frame* ***f***.
- *Update* tabel MMU sehingga halaman ***p1*** dipetakan ke *frame* ***f***.
- Kembali dari interupsi dan mengizinkan CPU mengulang instruksi yang menyebabkan interupsi tersebut.

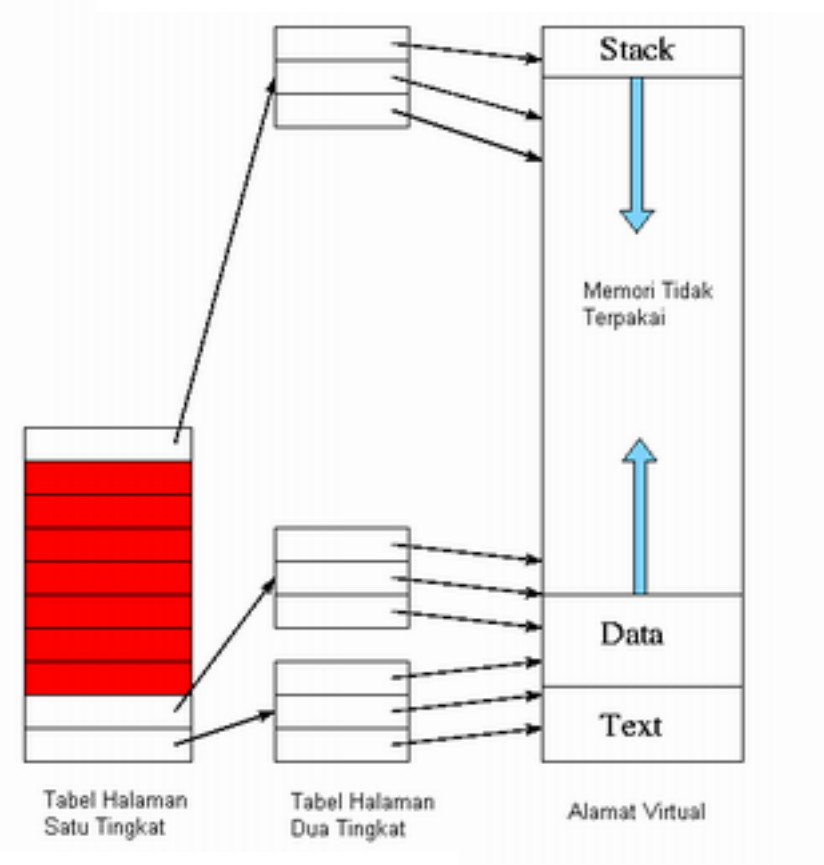
Pada dasarnya MMU terdiri dari tabel halaman yang merupakan sebuah rangkaian *array* dari masukan-masukan (*entries*) yang mempunyai indeks berupa nomor halaman (*p*). Setiap masukan terdiri dari *flags* (contohnya bit sahih dan nomor *frame*). Alamat fisik dibentuk dengan menggabungkan nomor *frame* dengan ofset, yaitu bit paling rendah dari alamat logis.

Setiap sistem operasi mempunyai metodanya sendiri untuk menyimpan tabel halaman. Sebagian

besar mengalokasikan tabel halaman untuk setiap proses. Penunjuk ke tabel halaman disimpan dengan nilai register yang lain (seperti pencacah instruksi) di blok kontrol proses. Ketika pelaksana *dispatcher* mengatakan untuk memulai proses, maka harus disimpan kembali register-register pengguna dan mendefinisikan nilai tabel halaman perangkat keras yang benar dari tempat penyimpanan tabel halaman dari pengguna.

30.6. Pemberian Page Secara *Multilevel*

Gambar 30.4. Tabel Halaman secara *Multilevel*



Sumber: allan.ultra.nyu.edu/~gottlieb/courses/1999-00-spring/ os/lecture-11.html

Idenya adalah dengan menambahkan tingkatan secara tidak langsung dan memiliki tabel halaman yang terdiri dari pointer-pointer ke tabel halaman.

- Bayangkan suatu tabel halaman yang besar.
- Panggil tabel halaman dua tingkat dan potong menjadi bagian-bagian untuk setiap ukuran dari halaman tersebut.
- Sebagai catatan bahwa anda bisa mendapatkan banyak PTE-PTE dalam satu halaman maka anda akan mempunyai jauh lebih sedikit dari halaman tersebut daripada yang dimiliki oleh PTE.
- Sekarang buatlah tabel halaman satu tingkat yang terdiri dari PTE-PTE yang memiliki *pointer* ke halaman tersebut.
- Tabel halaman satu tingkat ini cukup kecil untuk disimpan di memori.

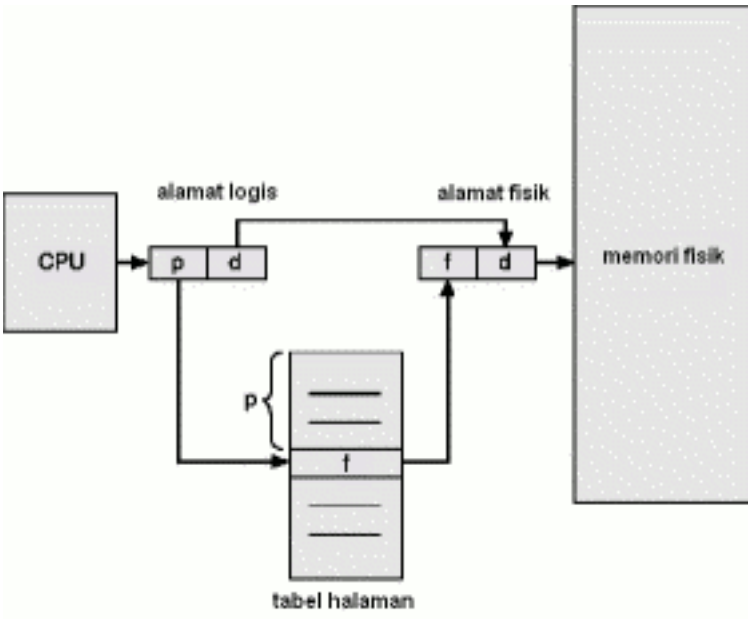
- Jika kita telah memiliki tabel halaman dua tingkat maka pekerjaan akan jauh lebih mudah.
- Jangan menyimpan semua *reference* PTE ke memori yang tidak terpakai dalam memori tabel halaman dua tingkat. Hal tersebut menggunakan permintaan halaman pada tabel halaman dua tingkat.
- Untuk tabel halaman dua tingkat, alamat logis dibagi menjadi tiga bagian yaitu **P#1, P#2**, dan **Ofset**
- P#1 menunjukkan indeks menuju tabel halaman satu tingkat.
- Ikuti penunjuk pada PTE yang berkaitan untuk meraih *frame* yang terdiri dari tabel halaman dua tingkat yang relevan.
- P#2 menunjukkan indeks menuju tabel halaman dua tingkat.
- Ikuti *pointer* pada PTE yang berkaitan untuk meraih *frame* yang terdiri dari *frame* asli yang diminta.
- Ofset menunjukkan ofset dari *frame* dimana terdapat lokasi adanya permintaan kata.

Banyak sistem komputer modern mendukung ruang alamat logis yang sangat luas (2 pangkat 32 sampai 2 pangkat 64). Pada lingkungan seperti itu tabel halamannya sendiri menjadi besar sekali. Untuk contoh, misalkan suatu sistem dengan ruang alamat logis 32-bit. Jika ukuran halaman di sistem seperti itu adalah 4K byte (2 pangkat 12), maka tabel halaman mungkin berisi sampai 1 juta masukan ((2^32)/(2^12)). Karena masing-masing masukan terdiri atas 4 byte, tiap-tiap proses mungkin perlu ruang alamat fisik sampai 4 megabyte hanya untuk tabel halamannya saja. Jelasnya, kita tidak akan mau mengalokasi tabel halaman secara berdekatan di dalam memori. Satu solusi sederhananya adalah dengan membagi tabel halaman menjadi potongan-potongan yang lebih kecil lagi. Ada beberapa cara yang berbeda untuk menyelesaikan ini.

30.7. Tabel Halaman secara *Inverted*

Biasanya, setiap proses mempunyai tabel halaman yang diasosiasikan dengannya. Tabel halaman hanya punya satu masukan untuk setiap halaman proses tersebut sedang digunakan (atau satu slot untuk setiap alamat maya, tanpa meperhatikan validitas terakhir). Semenjak halaman referensi proses melalui alamat maya halaman, maka representasi tabel ini adalah alami. Sistem operasi harus menterjemahkan referensi ini ke alamat memori fisik. Semenjak tabel diurutkan berdasarkan alamat maya, sistem operasi dapat menghitung dimana pada tabel yang diasosiasikan dengan masukan alamat fisik, dan untuk menggunakan nilai tersebut secara langsung. Satu kekurangan dari skema ini adalah masing-masing halaman mungkin mengandung jutaan masukan. Tabel ini mungkin memakan memori fisik dalam jumlah yang besar, yang mana dibutukan untuk tetap menjaga bagaimana memori fisik lain sedang digunakan.

Gambar 30.5. Tabel Halaman secara *Inverted*



Sumber: Operating System karya Avi Silberschatz, Peter Galvin, dan Greg Gagne, 2000.

30.8. Berbagi Halaman

Keuntungan lain dari pemberian halaman adalah kemungkinannya untuk berbagi kode yang sama. Pertimbangan ini terutama sekali penting pada lingkungan yang berbagi waktu. Pertimbangkan sebuah sistem yang mendukung 40 pengguna, yang masing-masing menjalankan aplikasi pengedit teks. Jika editor teks tadi terdiri atas 150K kode dan 50K ruang data, kita akan membutuhkan 8000K untuk mendukung 40 pengguna. Jika kodenya dimasukan ulang, bagaimana pun juga dapat dibagi-bagi, seperti pada gambar. Disini kita lihat bahwa tiga halaman editor (masing-masing berukuran 50K; halaman ukuran besar digunakan untuk menyederhanakan gambar) sedang dibagi-bagi diantara tiga proses. Masing-masing proses mempunyai halaman datanya sendiri.

30.9. Rangkuman

Paging adalah suatu metoda yang memungkinkan suatu alamat fisik memori yang tersedia dapat tidak berurutan. Prinsipnya adalah memecah memori fisik dan memori logika menjadi blok-blok dengan ukuran sama (disebut *page*). Setelah itu kita membuat *page table* yang akan menerjemahkan memori logika menjadi memori fisik dengan perantara *Memory Management Unit* (MMU), dan pengeksekusian proses akan mencari memori berdasarkan tabel tersebut.

30.10. Latihan

1. Apakah yang dimaksud dengan metoda "Pemberian Halaman"? Jelaskan pula keuntungan dan kerugian penggunaan metoda ini?

2. Jelaskan cara kerja unit managemen memori (MMU)!

3. Apakah fungsi dari skema bit *valid-invalid* ?

4. Bagaimanakah konsep dasar dari *page replacement*? Jelaskan secara singkat!

5. Apabila diberikan *reference string* 3, 1, 5, 1, 1, 2, 4, 3, 5, 3, 1 dan dilakukan *page replacement* algoritma optimal dengan 3 frame. Berapakah *page-fault* yang terjadi?

6. Memori I

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!

7. Memori II

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!

8. Multilevel Paging Memory

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 alrorithma "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

30.11. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts*: 6th ed. John Wiley & Sons

Tanenbaum, Andrew S., Woodhull, Albert S. 1997. Operating Systems Design and Implementation: Second Edition. Prentice Hall.

http://css.uni.edu/

http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html

http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html

http://www.cs.princeton.edu/ courses/ archive/ spring02/ cs217/ lectures/ paging.pdf

http://cwx.prenhall.com/ bookbind/ pubbooks/ tanenbaum/ chapter0/ deluxe.html

http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/

Bab 31. Segmentasi

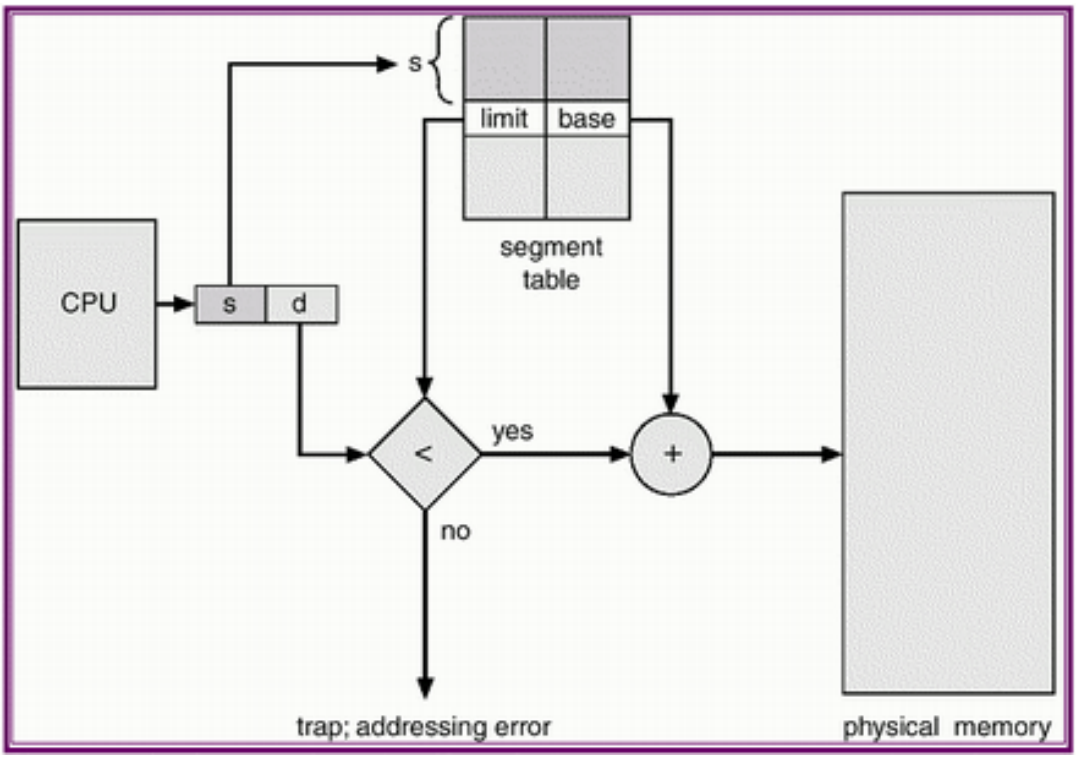
Segmentasi adalah skema managemen memori dengan cara membagi memori menjadi segmen-segmen. Dengan demikian, sebuah program dibagi menjadi segmen-segmen. *Segmen* adalah sebuah unit logis, yaitu unit yang terdiri dari beberapa bagian yang berjenis yang sama. Contoh: program utama, variabel lokal, *procedure* dan sebagainya. Berbeda dengan halaman, ukuran tiap segmen tidak harus sama dan memiliki 'ciri' tertentu. Ciri tertentu itu adalah nama segmen dan panjang segmen. Nama segmen dirujuk oleh nomor segmen sedangkan panjang segmen ditentukan oleh *offset*.

31.1. Arsitektur Segmentasi

Ukuran tiap segmen tidak harus sama. Saat sebuah program atau proses dimasukkan ke CPU, segmen yang berbeda dapat ditempatkan dimana saja di dalam memori utama (dapat menggunakan cara *first-fit* atau *best-fit*).

Alamat logis dari sebuah segmen adalah alamat dua dimensi, sedangkan alamat fisik memori adalah alamat satu dimensi. Oleh karena itu, agar implementasinya menjadi mudah (dari alamat logis ke alamat fisik) diperlukan Tabel Segmen yang yang terdiri dari *base* dan *limit*. *Base* menunjukkan alamat awal segmen (dari alamat fisik) dan *limit* menunjukkan panjang segmen.

Gambar 31.1. Arsitektur Segmentasi



Alamat logisnya: *s* dan *d*, *s* adalah nomor segmen/index di dalam tabel segmen *d* adalah *offset*. Jika *offset* kurang dari nol dan tidak lebih besar dari besarnya *limit* maka *base* akan dijumlahkan dengan *d* (*offset*), yang dijumlahkan itu adalah alamat fisik dari segmen tersebut.

31.2. Saling Berbagi dan Proteksi

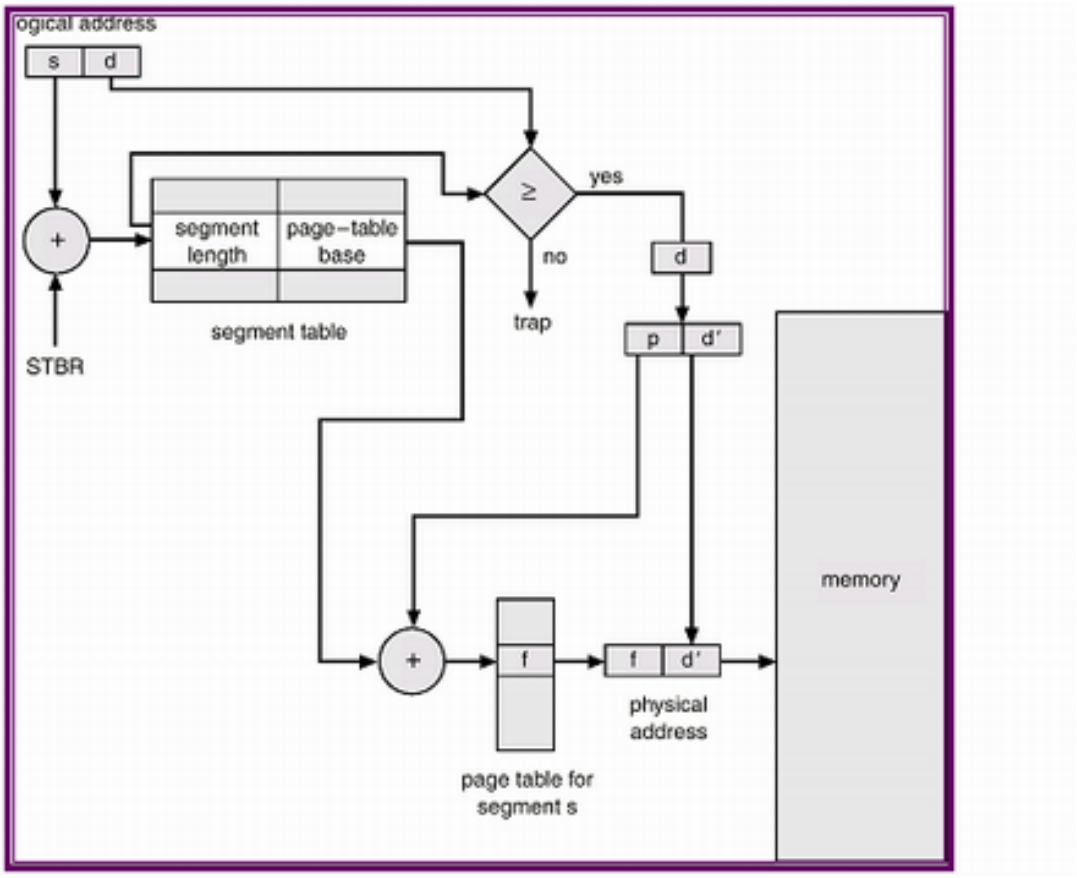
Segmen dapat terbagi jika terdapat elemen di tabel segmen yang berasal dari dua proses yang berbeda yang menunjuk pada alamat fisik yang sama. Saling berbagi ini muncul di level segmen dan pada saat ini terjadi semua informasi dapat turut terbagi. Proteksi dapat terjadi karena ada bit-proteksi yang berhubungan dengan setiap elemen dari segmen tabel. Bit-proteksi ini berguna untuk mencegah akses ilegal ke memori. Caranya: menempatkan sebuah *array* di dalam segmen itu sehingga perangkat keras managemen memori secara otomatis akan mengecek indeks *array*-nya legal atau tidak.

31.3. Segmentasi dengan Pemberian Halaman

Kelebihan Pemberian Halaman: tidak ada fragmentasi luar-alokasinya cepat.

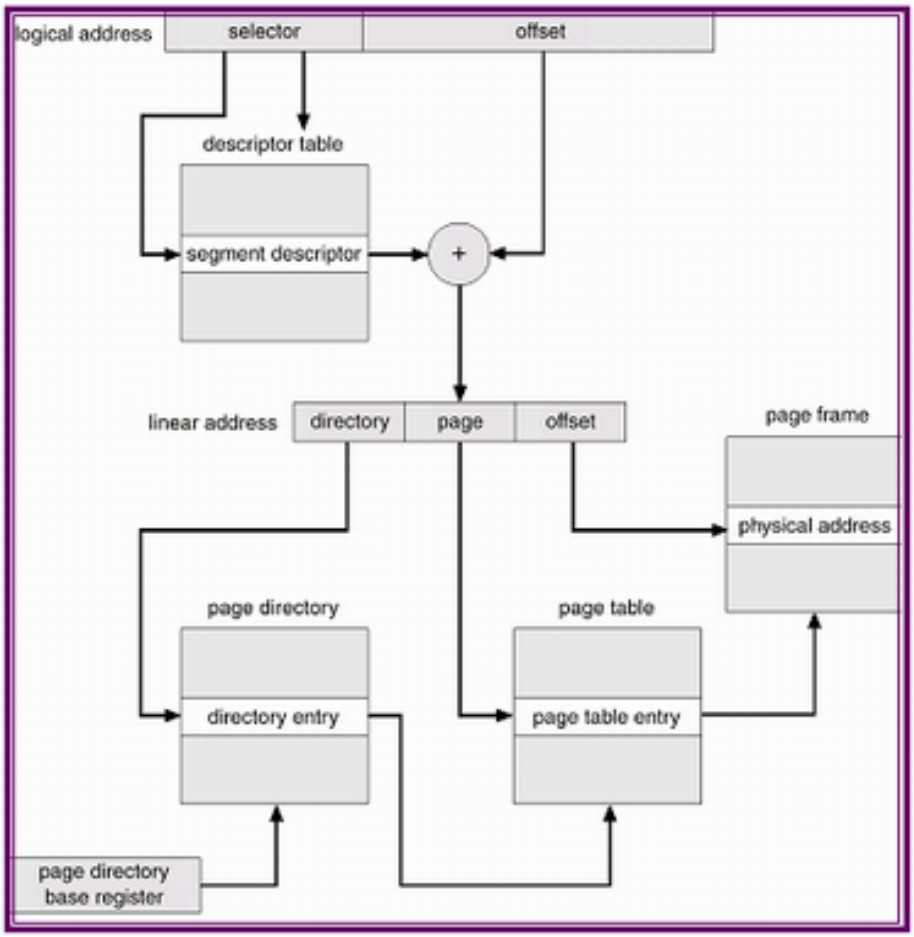
Kelebihan Segmentasi: saling berbagi-proteksi.

Gambar 31.2. Segmentasi dengan Pemberian Halaman



31.4. Penggunaan Segmentasi INTEL

Gambar 31.3. Penggunaan Segmentasi dengan Pemberian Halaman pada INTEL 30386



31.5. Masalah Dalam Segmentasi

- Segmen dapat Membesar.
- Muncul Fragmentasi Luar.
- Bila Ada Proses yang Besar.

31.6. Rangkuman

Segmentasi adalah skema managemen memori dengan cara membagi memori menjadi segmen-segmen. Berbeda dengan *page*, ukuran tiap segmen tidak harus sama dan memiliki ciri tertentu, yaitu nama dan panjang segmen.

31.7. Latihan

1. Apakah yang dimaksud dengan Segmentasi? Jelaskan pula bagaimana arsitektur segmentasi!
2. Sebutkan persamaan dan/atau perbedaan *demand paging* dan *demand segmentation*!
3. Sebutkan persamaan dan/atau perbedaan algoritma LFU dan algoritma MFU!

31.8. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts*: 6th ed. John Wiley & Sons

Tanenbaum,Andrew S. Woodhull, Albert S. 1997. *Operating Systems Design and Implementation: Second Edition*. Prentice Hall.

http://css.uni.edu/

http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html

http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html

http://www.cs.princeton.edu/ courses/ archive/ spring02/ cs217/ lectures/ paging.pdf

http://cwx.prenhall.com/ bookbind/ pubbooks/ tanenbaum/ chapter0/ deluxe.html

http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/

Bab 32. Memori Virtual

32.1. Pengertian

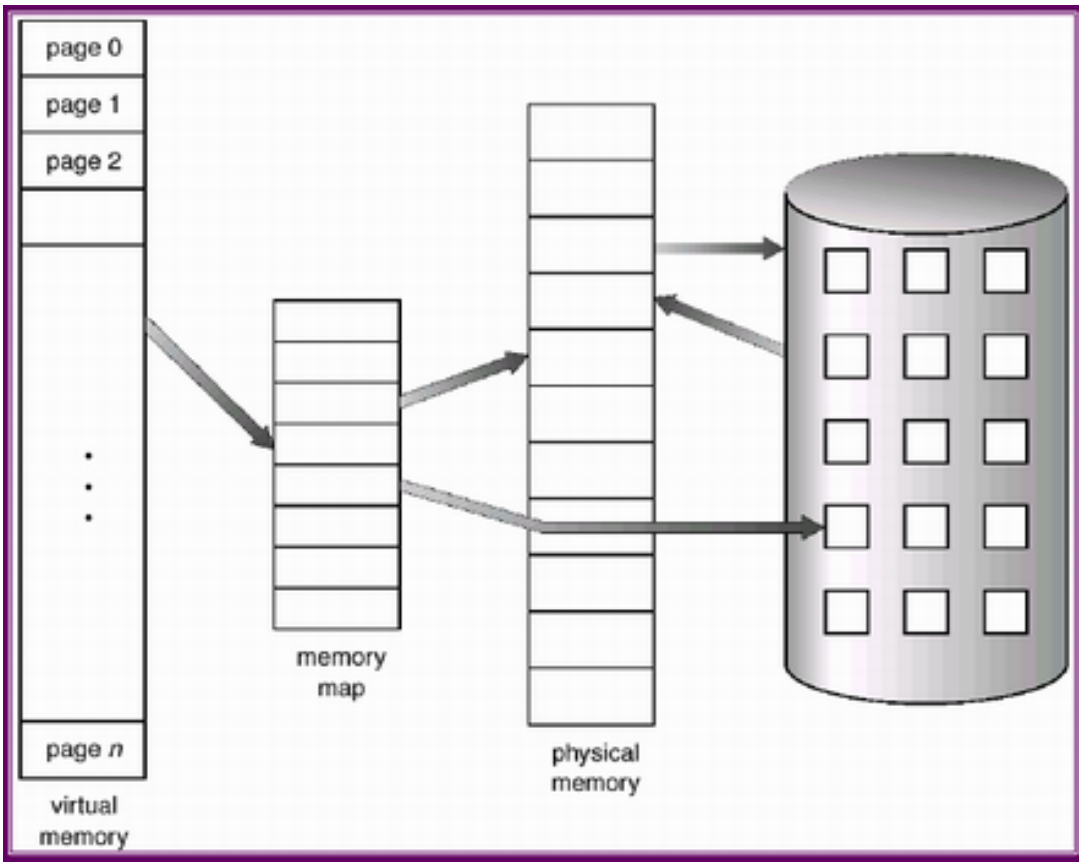
Managemen memori pada intinya adalah menempatkan semua bagian proses yang akan dijalankan kedalam memori sebelum proses itu dijalankan. Untuk itu, semua bagian proses itu harus memiliki tempat sendiri di dalam memori fisik.

Tetapi tidak semua bagian dari proses itu akan dijalankan, misalnya:

- Pernyataan atau pilihan yang hanya akan dieksekusi pada kondisi tertentu. Contohnya adalah: pesan-pesan *error* yang hanya muncul bila terjadi kesalahan saat program dijalankan.
- Fungsi-fungsi yang jarang digunakan.
- Pengalokasian memori yang lebih besar dari yang dibutuhkan. Contoh: *array*, *list* dan tabel.

Pada memori berkapasitas besar, hal-hal ini tidak akan menjadi masalah. Akan tetapi, pada memori yang sangat terbatas, hal ini akan menurunkan optimalisasi utilitas dari ruang memori fisik. Sebagai solusi dari masalah-masalah ini digunakanlah konsep memori virtual.

Gambar 32.1. Memori Virtual



Silberschatz, Galvin and Gagne @ 2002

Memori virtual adalah suatu teknik yang memisahkan antara memori logis dan memori fisiknya.

Teknik ini menyembunyikan aspek-aspek fisik memori dari pengguna dengan menjadikan memori sebagai lokasi alamat virtual berupa *byte* yang tidak terbatas dan menaruh beberapa bagian dari memori virtual yang berada di memori logis.

Berbeda dengan keterbatasan yang dimiliki oleh memori fisik, memori virtual dapat menampung program dalam skala besar, melebihi daya tampung dari memori fisik yang tersedia.

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

Konsep memori virtual pertama kali dikemukakan Fotheringham pada tahun 1961 pada sistem komputer Atlas di Universitas Manchester, Inggris (Hariyanto, Bambang: 2001).

Sebagaimana dikatakan di atas bahwa hanya sebagian dari program yang diletakkan di memori fisik. Hal ini memberikan keuntungan:

- Berkurangnya proses M/K yang dibutuhkan (lalu lintas M/K menjadi rendah). Misalnya untuk program butuh membaca dari *disk* dan memasukkan dalam memory setiap kali diakses.
- Ruang menjadi lebih leluasa karena berkurangnya memori fisik yang digunakan. Contoh, untuk program 10 MB tidak seluruh bagian dimasukkan dalam memori fisik. Pesan-pesan error hanya dimasukkan jika terjadi error.
- Meningkatnya respon, karena menurunnya beban M/K dan memori.
- Bertambahnya jumlah pengguna yang dapat dilayani. Ruang memori yang masih tersedia luas memungkinkan komputer untuk menerima lebih banyak permintaan dari pengguna.

Gagasan utama dari memori virtual adalah ukuran gabungan program, data dan stack melampaui jumlah memori fisik yang tersedia. Sistem operasi menyimpan bagian-bagian proses yang sedang digunakan di memori fisik (memori utama) dan sisanya diletakkan di disk. Begitu bagian yang berada di disk diperlukan, maka bagian di memori yang tidak diperlukan akan dikeluarkan dari memori fisik (*swap-out*) dan diganti (*swap-in*) oleh bagian disk yang diperlukan itu.

Memori virtual diimplementasikan dalam sistem *multiprogramming*. Misalnya: 10 program dengan ukuran 2 Mb dapat berjalan di memori berkapasitas 4 Mb. Tiap program dialokasikan 256 KByte dan bagian-bagian proses *swap in*) masuk ke dalam memori fisik begitu diperlukan dan akan keluar (*swap out*) jika sedang tidak diperlukan. Dengan demikian, sistem *multiprogramming* menjadi lebih efisien.

Memori virtual dapat dilakukan melalui dua cara:

- Permintaan pemberian halaman (*demand paging*).
- Permintaan segmentasi (*demand segmentation*). Contoh: IBM OS/2. Algoritma dari permintaan segmentasi lebih kompleks, karena itu jarang diimplementasikan.

32.2. *Demand Paging*

Demand paging atau permintaan pemberian halaman adalah salah satu implementasi dari memori virtual yang paling umum digunakan. *Demand paging* pada prinsipnya hampir sama dengan permintaan halaman (*paging*) hanya saja halaman (*page*) tidak akan dibawa ke ke dalam memori fisik sampai ia benar-benar diperlukan. Untuk itu diperlukan bantuan perangkat keras untuk mengetahui lokasi dari halaman saat ia diperlukan.

Karena *demand paging* merupakan implementasi dari memori virtual, maka keuntungannya sama dengan keuntungan memori virtual, yaitu:

- Sedikit M/K yang dibutuhkan.
- Sedikit memori yang dibutuhkan.
- Respon yang lebih cepat.
- Dapat melayani lebih banyak pengguna.

Ada tiga kemungkinan kasus yang dapat terjadi pada saat dilakukan pengecekan pada halaman yang dibutuhkan, yaitu: halaman ada dan sudah berada di memori. Statusnya valid ("1"). Halaman ada tetapi masih berada di disk belum diberada di memori (harus menunggu sampai dimasukkan). Statusnya tidak valid ("0"). Halaman tidak ada, baik di memori maupun di *disk* (*invalid reference --> abort*).

Saat terjadi kasus kedua dan ketiga, maka proses dinyatakan mengalami kesalahan halaman. Perangkat keras akan menjebaknya ke dalam sistem operasi.

32.3. Skema Bit Valid - Tidak Valid

Dengan meminjam konsep yang sudah pernah dijelaskan dalam Bab 31, *Segmentasi*, maka dapat ditentukan halaman mana yang ada di dalam memori dan mana yang tidak ada di dalam memori.

Konsep itu adalah skema bit valid-tidak valid, di mana di sini pengertian valid berarti bahwa halaman legal dan berada dalam memori (kasus 1), sedangkan tidak valid berarti halaman tidak ada (kasus 3) atau halaman ada tapi tidak ditemui di memori (kasus 2).

Pengesetan bit:

- Bit 1 --> halaman berada di memori
- Bit 0 --> halaman tidak berada di memori.

(Dengan inisialisasi: semua bit di-set 0)

Apabila ternyata hasil dari translasi, bit halaman = 0, berarti kesalahan halaman terjadi.

32.4. Penanganan Kesalahan Halaman

Prosedur penanganan kesalahan halaman sebagaimana tertulis dalam buku *Operating System Concept 5th Ed*. halaman 294 adalah sebagai berikut:

- Memeriksa tabel internal yang dilengkapi dengan PCB untuk menentukan valid atau tidaknya bit.
- Apabila tidak valid, program akan di terminasi (interupsi oleh *illegal address trap*). Jika valid tapi proses belum dibawa ke halaman, maka kita halaman kan sekarang.
- Memilih *frame* kosong (*free-frame*), misalnya dari *free-frame list*. Jika tidak ditemui ada *frame* yang kosong, maka dilakukan *swap-out* dari memori. *Frame* mana yang harus di-*swap-out* akan ditentukan oleh algoritma (lihat sub bab penggantian halaman).
- Menjadualkan operasi disk untuk membaca halaman yang diinginkan ke *frame* yang baru dialokasikan.
- Ketika pembacaan komplit, ubah bit validasi menjadi "1" yang berarti halaman sudah diidentifikasi ada di memori.
- Mengulang instruksi yang tadi telah sempat diinterupsi. Jika tadi kesalahan halaman terjadi saat

instruksi di-*ambil*, maka akan dilakukan pengambilan lagi. Jika terjadi saat operan sedang di-*ambil*, maka harus dilakukan *pengambilan* ulang, dekode, dan pengambilan operan lagi.

32.5. Apa yang terjadi pada saat kesalahan?

Kesalahan halaman menyebabkan urutan kejadian berikut:

- Ditangkap oleh Sistem Operasi.
- Menyimpan *register pengguna* dan proses.
- Tetapkan bahwa interupsi merupakan kesalahan halaman.
- Periksa bahwa referensi halaman adalah legal dan tentukan lokasi halaman pada disk.
- Kembangkan pembacaan disk ke *frame* kosong.
- Selama menunggu, alokasikan CPU ke pengguna lain dengan menggunakan penjadualan CPU.
- Terjadi interupsi dari disk bahwa M/K selesai.
- Simpan register dan status proses untuk pengguna yang lain.
- Tentukan bahwa interupsi berasal dari disk.
- Betulkan tabel halaman dan tabel yang lain bahwa halaman telah berada di memory.
- Tunggu CPU untuk untuk dialokasikan ke proses tersebut
- Kembalikan register pengguna, status proses, tabel halaman, dan meneruskan instruksi interupsi.

Pada berbagai kasus, ada tiga komponen yang kita hadapi pada saat melayani kesalahan halaman:

- Melayani interupsi kesalahan halaman
- Membaca halaman
- Mengulang kembali proses

32.6. Kinerja *Demand Paging*

Menggunakan *Effective Access Time* (EAT), dengan rumus:

EAT = (1-p) x *ma* + *p* x waktu kesalahan halaman

- *p*: kemungkinan terjadinya kesalahan halaman ($0 < p < 1$)
- $p = 0$; tidak ada kesalahan halaman
- $p = 1$; semuanya mengalami kesalahan halaman
- *ma*: waktu pengaksesan memory (memory access time)

Untuk menghitung EAT, kita harus tahu berapa banyak waktu dalam pengerjaan kesalahan halaman.

Contoh penggunaan *Effective Access Time*

Diketahui: Waktu pengaksesan memory = *ma* = 100 nanodetik Waktu kesalahan halaman = 20 milidetik

Maka, EAT = (1-p) x 100 + p (20 milidetik) = 100 - 100p + 20.000.000p = 100 + 19.999.900p (milidetik)

Pada sistem *demand paging*, sebisa mungkin kita jaga agar tingkat kesalahan halamannya rendah. Karena bila *Effective Access Time* meningkat, maka proses akan berjalan lebih lambat.

32.7. Permasalahan Lain *Demand Paging*

Sebagaimana dilihat di atas, bahwa ternyata penanganan kesalahan halaman menimbulkan masalah-masalah baru pada proses *restart instruction* yang berhubungan dengan arsitektur komputer.

Masalah yang terjadi, antara lain mencakup:

- Bagaimana mengulang instruksi yang memiliki beberapa lokasi yang berbeda?
- Bagaimana pengalamatan dengan menggunakan ragam pengalamatan spesial, termasuk *autoincrement* dan *autodecrement mode*?
- Bagaimana jika instruksi yang dieksekusi panjang (contoh: *block move*)?

Masalah pertama dapat diatasi dengan dua cara yang berbeda.

- komputasi *microcode* dan berusaha untuk mengakses kedua ujung dari blok, agar tidak ada modifikasi halaman yang sempat terjadi.
- memanfaatkan register sementara (*temporary register*) untuk menyimpan nilai yang sempat tertimpa/ termodifikasi oleh nilai lain.

Masalah kedua diatasi dengan menciptakan suatu status register spesial baru yang berfungsi menyimpan nomor register dan banyak perubahan yang terjadi sepanjang eksekusi instruksi. Sedangkan masalah ketiga diatasi dengan men-*set* bit FPD (*first phase done*) sehingga *restart instruction* tidak akan dimulai dari awal program, melainkan dari tempat program terakhir dieksekusi.

32.8. Persyaratan Perangkat Keras

Pemberian nomor halaman melibatkan dukungan perangkat keras, sehingga ada persyaratan perangkat keras yang harus dipenuhi. Perangkat-perangkat keras tersebut sama dengan yang digunakan untuk *paging* dan *swapping*, yaitu:

- tabel halaman "bit valid-tidak valid"
    - Valid ("1") artinya halaman sudah berada di memori
    - Tidak valid ("0") artinya halaman masih berada di disk.
- Memori sekunder, digunakan untuk menyimpan proses yang belum berada di memori.

Lebih lanjut, sebagai konsekuensi dari persyaratan ini, akan diperlukan pula perangkat lunak yang dapat mendukung terciptanya pemberian nomor halaman.

32.9. Rangkuman

Memori virtual adalah suatu teknik yang memisahkan antara memori logika dan memori fisiknya. Keuntungannya adalah memori virtual dapat menampung program dalam skala besar, menurunnya lalu lintas M/K, ruang memori menjadi lebih leluasa, meningkatnya respon, dan bertambahnya jumlah *user* yang dapat dilayani. *Demand paging* adalah salah satu implementasi memori virtual yang umum digunakan, yakni mengatur agar *page* tidak akan dibawa ke memori fisik sampai benar-benar dibutuhkan. Mengukur kinerja *demand paging* adalah dengan mengukur *Effective Access Time*-nya.

32.10. Latihan

1. Bagaimanakah perbedaan dan/atau persamaan Windows NT, Solaris 2, dan Linux dalam mengimplementasikan *virtual memory*?

32.11. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts*: 6th ed. John Wiley & Sons

Tanenbaum,Andrew S. Woodhull, Albert S. 1997. *Operating Systems Design and Implementation: Second Edition*. Prentice Hall.

http://css.uni.edu/

http://allan.ultra.nyu.edu/~gottlieb/courses/1999-00-spring/os/lecture-11.html

http://www.cs.wisc.edu/~solomon/cs537/paging.html

http://www.cs.princeton.edu/courses/archive/spring02/cs217/lectures/paging.pdf

http://cwx.prenhall.com/bookbind/pubbooks/tanenbaum/chapter0/deluxe.html

http://www.cs.jhu.edu/~yairamir/cs418/os5/

Bab 33. Permintaan Halaman Pembuatan Proses

Sistem permintaan halaman dan memori virtual memberikan banyak keuntungan selama pembuatan proses berlangsung. Pada sub-bab ini, akan dibahas mengenai dua teknik yang disediakan oleh memori virtual untuk meningkatkan kinerja pembuatan dan pengeksekusian suatu proses.

33.1. *Copy-On-Write*

Dengan memanggil sistem pemanggilan **fork()**, sistem operasi akan membuat proses anak sebagai salinan dari proses induk. Sistem pemanggilan *fork()* bekerja dengan membuat salinan alamat proses induk untuk proses anak, lalu membuat salinan halaman milik proses induk tersebut.Tapi, karena setelah pembuatan proses anak selesai, proses anak langsung memanggil sistem pemanggilan *exec()* yang menyalin alamat proses induk yang kemungkinan tidak dibutuhkan.

Oleh karena itu, lebih baik kita menggunakan teknik lain dalam pembuatan proses yang disebut sistem ***copy-on-write***. Teknik ini bekerja dengan memperbolehkan proses anak untuk menginisialisasi penggunaan halaman yang sama secara bersamaan. halaman yang digunakan bersamaan itu, disebut dengan "halaman *copy-on-write*", yang berarti jika salah satu dari proses anak atau proses induk melakukan penulisan pada halaman tersebut, maka akan dibuat juga sebuah salinan dari halaman itu.

Sebagai contoh, sebuah proses anak hendak memodifikasi sebuah halaman yang berisi sebagian dari *stack*. Sistem operasi akan mengenali hal ini sebagai *copy-on-write*, lalu akan membuat salinan dari halaman ini dan memetakannya ke alamat memori dari proses anak, sehingga proses anak akan memodifikasi halaman salinan tersebut, dan bukan halaman milik proses induk. Dengan tekik *copy-on-write* ini, halaman yang akan disalin adalah halaman yang dimodifikasi oleh proses anak atau proses induk. Halaman-halaman yang tidak dimodifikasi akan bisa dibagi untuk proses anak dan proses induk.

Saat suatu halaman akan disalin menggunakan teknik *copy-on-write*, digunakan teknik ***zero-fill-on-demand*** untuk mengalokasikan halaman kosong sebagai tempat meletakkan hasil duplikat. Halaman kosong tersebut dialokasikan saat *stack* atau *heap* suatu proses akan diperbesar atau untuk mengatur halaman *copy-on-write*. Halaman *Zero-fill-on-demand* akan dibuat kosong sebelum dialokasikan, yaitu dengan menghapus isi awal dari halaman. Karena itu, dengan *copy-on-write*, halaman yang sedang disalin akan disalin ke sebuah halaman *zero-fill-on*.

Teknik *copy-on-write* digunakan oleh beberapa sistem operasi seperti Windows 2000, Linux, dan Solaris2.

33.2. *Memory-Mapped Files*

Kita dapat menganggap berkas M/K sebagai akses memori *rutin* pada teknik memori virtual. Cara ini disebut dengan "pemetaan memori" sebuah berkas yang mengizinkan sebuah bagian dari alamat virtual dihubungkan dengan sebuah berkas. Dengan teknik pemetaan memori sebuah blok *disk* dapat dipetakan ke ke sebuah halaman pada memori.

Proses membaca dan menulis sebuah berkas ditangani oleh akses memori *rutin* agar memudahkan mengakses dan menggunakan sebuah berkas yaitu dengan mengizinkan manipulasi berkas melalui memori dibandingkan memanggil dengan sistem pemanggilan *read()* dan *write()*.

Beberapa sistem operasi menyediakan pemetaan memori hanya melalui sistem pemanggilan yang khusus dan menjaga semua berkas M/K yang lain dengan menggunakan sistem pemanggilan yang biasa.

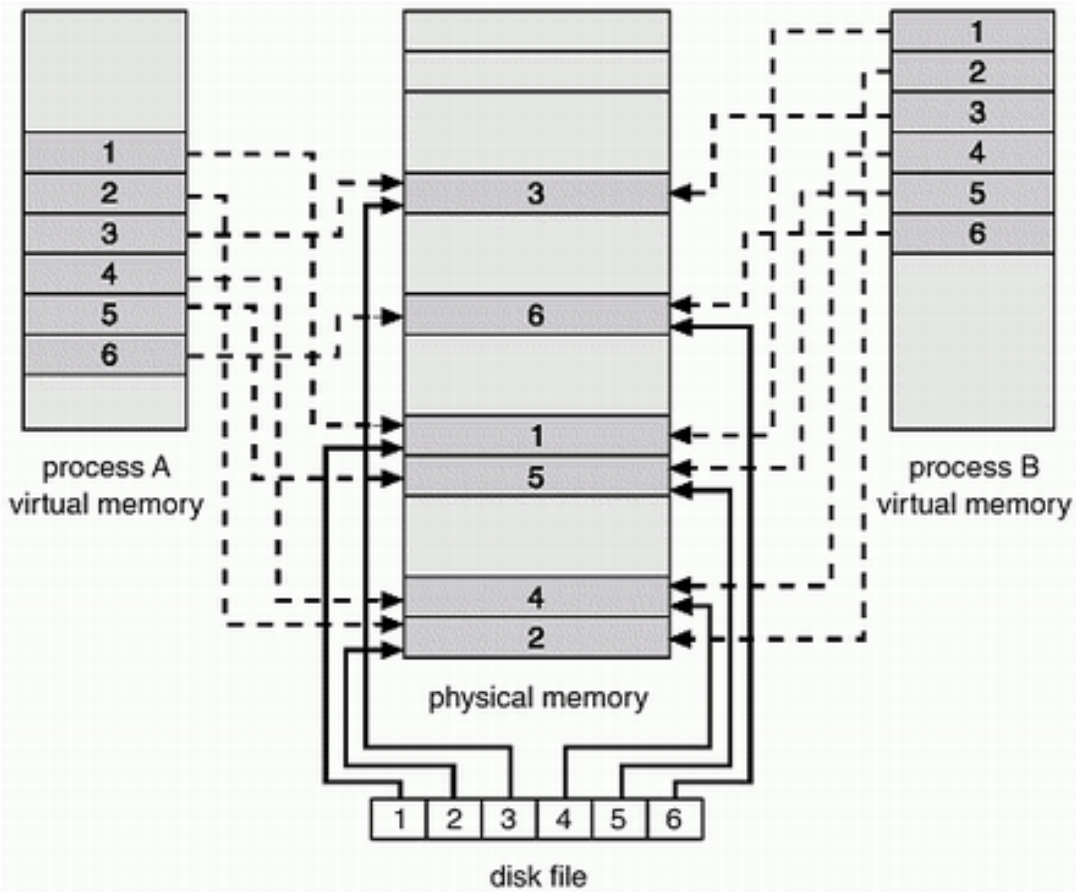
Proses diperbolehkan untuk memetakan berkas yang sama ke dalam memori virtual dari masing-masing berkas, agar data dapat digunakan secara bersamaan. Memori virtual memetakan tiap proses ke dalam halaman yang sama pada memori virtual, yaitu halaman yang menyimpan

salinan dari blok *disk*.

Dengan sistem pemetaan memori, sistem pemangggilan dapat juga mendukung fungsi *copy-on-write*, yang mengizinkan proses untuk menggunakan sebuah berkas secara bersamaan pada keadaan *read only*, tapi tetap memiliki salinan dari data yang diubah.

Berikut ini merupakan bagan dari proses ***memory-mapped files***.

Gambar 33.1. Bagan proses *memory-mapped files*



Gambar ini diambil dari buku *Applied Operating System*, Silberchatz, Galvin, Gagne, edisi VI tahun 2002.

33.3. Rangkuman

Beberapa teknik yang disediakan memori virtual untuk meningkatkan kinerja pembuatan dan pengeksekusian suatu proses antara lain adalah *copy-on-write* dan *memory-mapped files*.

33.4. Latihan

1. 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.

2. Apakah yang dimaksud dengan *copy-on-write*?

33.5. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts*: 6th ed. John Wiley & Sons

Tanenbaum,Andrew S. Woodhull, Albert S. 1997. *Operating Systems Design and Implementation: Second Edition*. Prentice Hall.

http://css.uni.edu/

http://allan.ultra.nyu.edu/~gottlieb/courses/1999-00-spring/os/lecture-11.html

http://www.cs.wisc.edu/~solomon/cs537/paging.html

http://www.cs.princeton.edu/courses/archive/spring02/cs217/lectures/paging.pdf

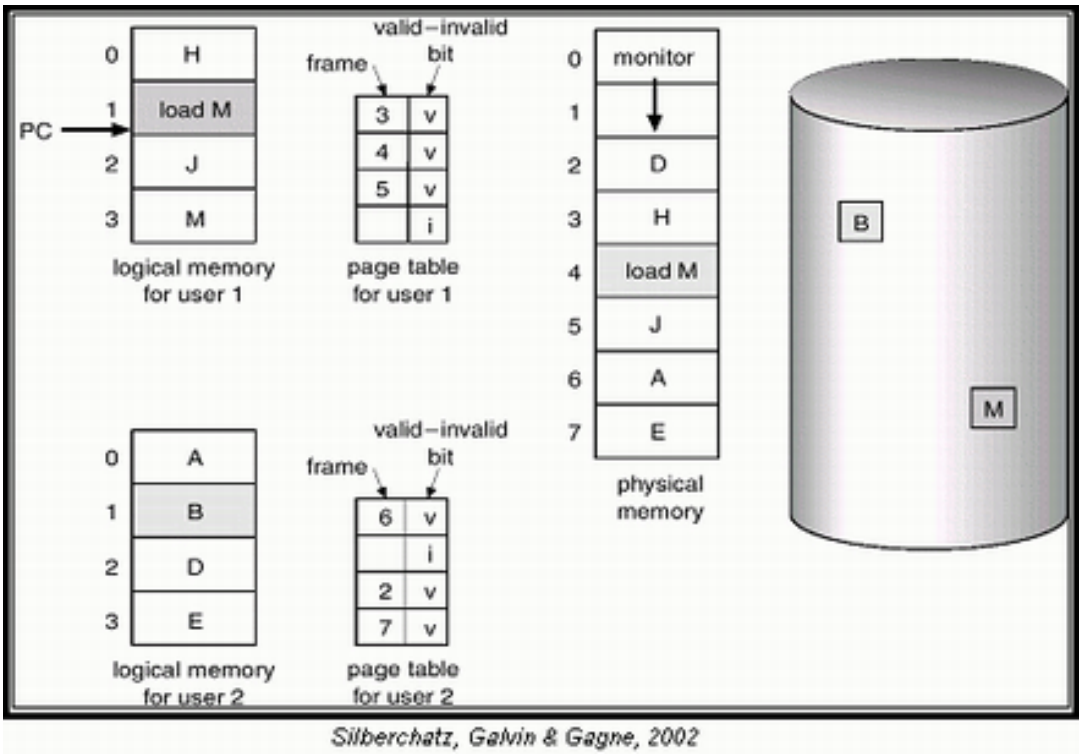
http://cwx.prenhall.com/bookbind/pubbooks/tanenbaum/chapter0/deluxe.html

http://www.cs.jhu.edu/~yairamir/cs418/os5/

Bab 34. Algoritma Pergantian Halaman

Masalah kesalahan halaman pasti akan dialami oleh setiap halaman minimal satu kali. Akan tetapi, sebenarnya sebuah proses yang memiliki N buah halaman hanya akan menggunakan N/2 diantaranya. Kemudian permintaan halaman akan menyimpan M/K yang dibutuhkan untuk mengisi N/2 halaman sisanya. Dengan demikian utilisasi CPU dan *throughput* dapat ditingkatkan.

Gambar 34.1. Kondisi yang memerlukan Pemindahan Halaman



Gambar di atas diambil dari buku *Applied Operating System*, Silberchatz, Galvin, Gagne, edisi VI tahun 2002.

Upaya yang dilakukan oleh permintaan halaman dalam mengatasi kesalahan halaman didasari oleh pemindahan halaman. Sebuah konsep yang akan kita bahas lebih lanjut dalam sub bab ini.

Pendekatan untuk pemindahan halaman: "Jika tidak ada *frame* yang kosong, cari *frame* yang tidak sedang digunakan atau yang tidak akan digunakan dalam jangka waktu yang lama, lalu kosongkan dengan memindahkan isinya ke dalam ruang pertukaran dan ubah semua tabelnya sebagai indikasi bahwa halaman tersebut tidak akan lama berada di dalam memori."

Dalam pemindahan halaman, *frame* kosong seperti tersebut di atas, akan digunakan sebagai tempat penyimpanan dari halaman yang salah.

Rutinitas yang dilakukan dalam pemindahan halaman antara lain:

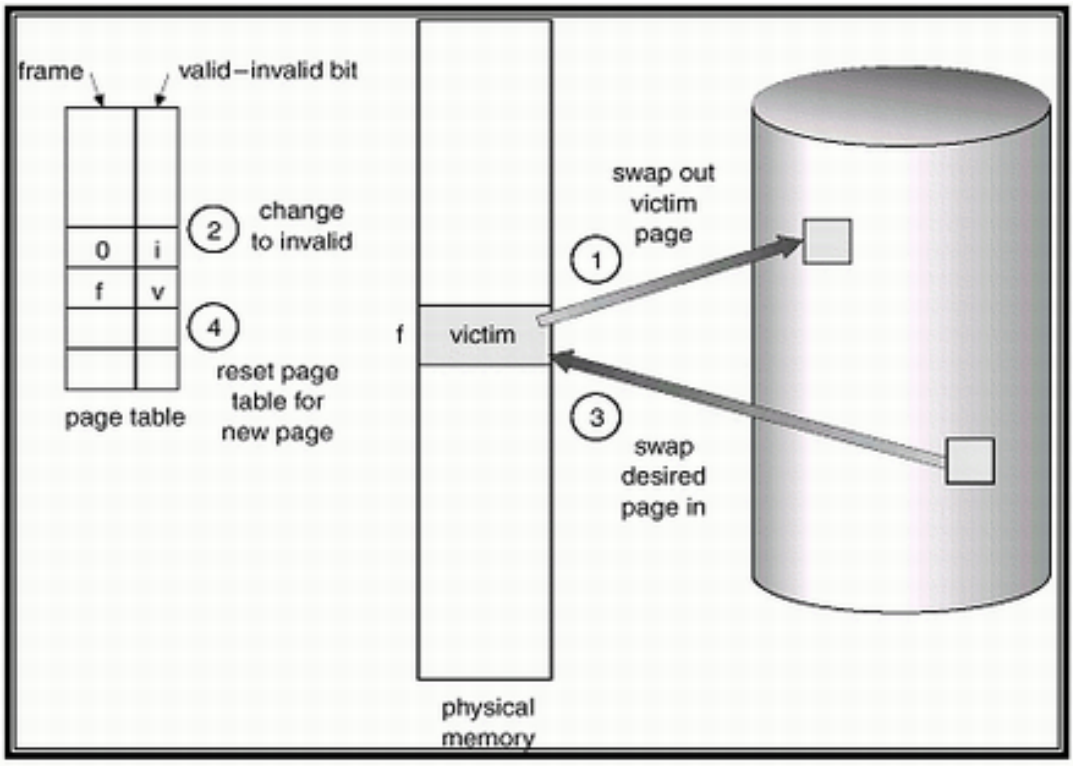
- Mencari lokasi dari halaman yang diinginkan pada *disk*.
- Mencari *frame* yang kosong:

- Jika ada, maka gunakan *frame* tersebut.
- Jika tidak ada, maka tentukan *frame* yang tidak sedang dipakai atau yang tidak akan digunakan dalam jangka waktu lama, lalu kosongkan *frame* tersebut. Gunakan algoritma pemindahan halaman untuk menentukan *frame* yang akan dikosongkan.

  Usahakan agar tidak menggunakan frame yang akan digunakan dalam waktu dekat. Jika terpaksa, maka sebaiknya segera masukkan kembali frame tersebut agar tidak terjadi *overhead*.
- Tulis halaman yang dipilih ke *disk*, ubah tabel halaman dan tabel *frame*.

- Membaca halaman yang diinginkan ke dalam *frame* kosong yang baru.
- Mengulangi proses pengguna dari awal.

Gambar 34.2. Pemindahan halaman



Gambar di atas diambil dari buku *Applied Operating System*, Silberchatz, Galvin, Gagne, edisi VI tahun 2002.

Rutinitas di atas belum tentu berhasil. Jika kita tidak dapat menemukan *frame* yang kosong atau akan dikosongkan, maka sebagai jalan keluarnya kita dapat melakukan pentransferan dua halaman (satu masuk, satu keluar). Cara ini akan menambah waktu pelayanan kesalahan halaman dan waktu akses efektif. Olehkarena itu, perlu digunakan bit tambahan untuk masing-masing halaman dan *frame* yang diasosiasikan dalam perangkat keras.

Sebagai dasar dari permintaan halaman, pemindahan halaman merupakan "jembatan pemisah" antara memori logis dan memori fisik. Mekanisme yang dimilikinya memungkinkan memori virtual berukuran sangat besar dapat disediakan untuk pemrogram dalam bentuk memori fisik yang

berukuran lebih kecil.

Dalam permintaan halaman, jika kita memiliki banyak proses dalam memori, kita harus menentukan jumlah *frame* yang akan dialokasikan ke masing-masing proses. Ketika pemindahan halaman diperlukan, kita harus memilih *frame* yang akan dipindahkan(dikosongkan). Masalah ini dapat diselesaikan dengan menggunakan algoritma pemindahan halaman.

Ada beberapa macam algoritma pemindahan halaman yang dapat digunakan. Algoritma yang terbaik adalah yang memiliki tingkat kesalahan halaman terendah. Selama jumlah *frame* meningkat, jumlah kesalahan halaman akan menurun. Peningkatan jumlah *frame* dapat terjadi jika memori fisik diperbesar.

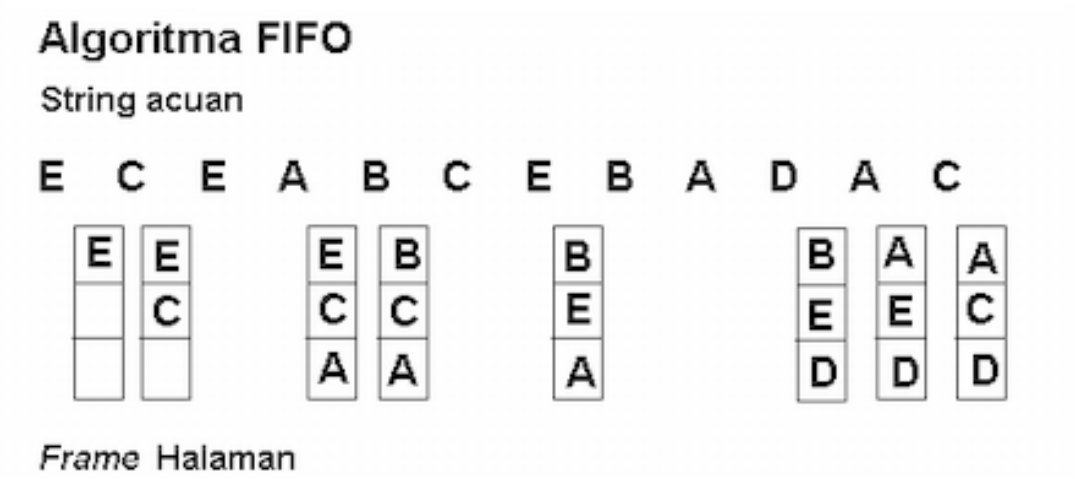
Evaluasi algoritma pemindahan halaman dapat dilakukan dengan menjalankan sejumlah string acuan di memori dan menghitung jumlah kesalahan halaman yang terjadi. Sebagai contoh, suatu proses memiliki urutan alamat: 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105; per 100 *bytes*-nya dapat kita turunkan menjadi string acuan: 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.

Pemindahan halaman diimplementasikan dalam algoritma yang bertujuan untuk menghasilkan tingkat kesalahan halaman terendah. Ada beberapa algoritma pemindahan halaman yang berbeda. Pemilihan halaman yang akan diganti dalam penggunaan algoritma-algoritma tersebut bisa dilakukan dengan berbagai cara, seperti dengan memilih secara acak, memilih dengan berdasarkan pada penggunaan, umur halaman, dan lain sebagainya. Pemilihan algoritma yang kurang tepat dapat menyebabkan peningkatan tingkat kesalahan halaman sehingga proses akan berjalan lebih lambat.

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

Prinsip yang digunakan dalam algoritma FIFO yaitu halaman yang diganti adalah halaman yang paling lama berada di memori. Algoritma ini adalah algoritma pemindahan halaman yang paling mudah diimplementasikan, akan tetapi paling jarang digunakan dalam bentuk aslinya, biasanya dikombinasikan dengan algoritma lain.

Gambar 34.3. Contoh Algoritma FIFO



Gambar di atas terinspirasi dari buku *Applied Operating System*, Silberchatz, Galvin, Gagne, edisi VI tahun 2002.

Pengimplementasian algoritma FIFO dilakukan dengan menggunakan antrian untuk menandakan halaman yang sedang berada di dalam memori. Setiap halaman baru yang diakses diletakkan di bagian belakang (ekor) dari antrian. Apabila antrian telah penuh dan ada halaman yang baru diakses maka halaman yang berada di bagian depan (kepala) dari antrian akan diganti.

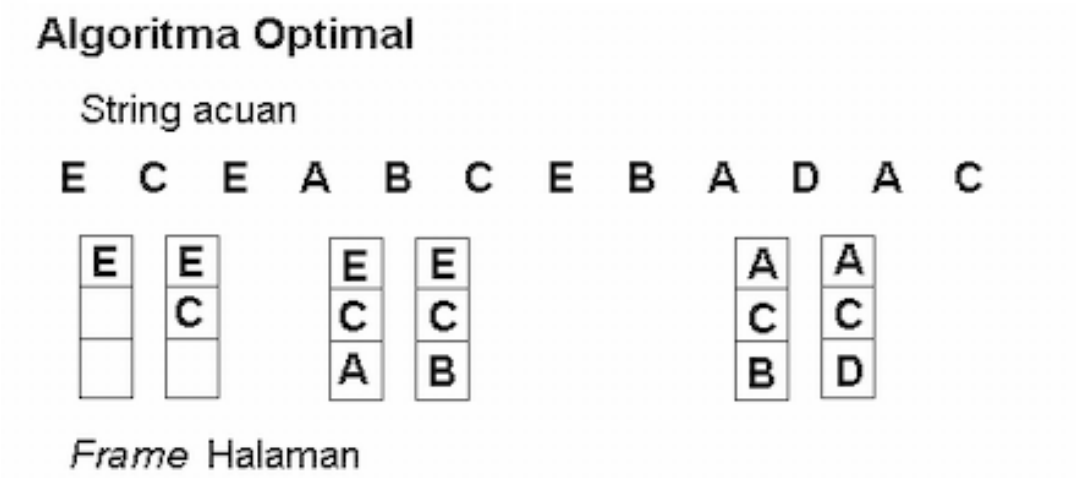
Kelemahan dari algoritma FIFO adalah kinerjanya yang tidak selalu baik. Hal ini disebabkan karena ada kemungkinan halaman yang baru saja keluar dari memori ternyata dibutuhkan kembali. Di samping itu dalam beberapa kasus, tingkat kesalahan halaman justru bertambah seiring dengan meningkatnya jumlah *frame*, yang dikenal dengan nama anomali Belady.

34.2. Algoritma Optimal

Algoritma optimal pada prinsipnya akan mengganti halaman yang tidak akan digunakan untuk jangka waktu yang paling lama. Kelebihannya antara lain dapat menghindari terjadinya anomali Belady dan juga memiliki tingkat kesalahan halaman yang terendah diantara algoritma-algoritma pemindahan halaman yang lain.

Meski pun tampaknya mudah untuk dijelaskan, tetapi algoritma ini sulit atau hampir tidak mungkin untuk diimplementasikan karena sistem operasi harus dapat mengetahui halaman-halaman mana saja yang akan diakses berikutnya, padahal sistem operasi tidak dapat mengetahui halaman yang muncul di waktu yang akan datang.

Gambar 34.4. Contoh Algoritma Optimal



Gambar di atas terinspirasi dari buku *Applied Operating System*, Silberchatz, Galvin, Gagne, edisi VI tahun 2002.

34.3. Algoritma *Least Recently Used* (LRU)

Algoritma LRU akan mengganti halaman yang telah tidak digunakan dalam jangka waktu terlama. Pertimbangannya yaitu berdasarkan observasi bahwa halaman yang telah sering diakses kemungkinan besar akan diakses kembali. Kelebihan dari algoritma LRU sama halnya seperti pada algoritma optimal, yaitu tidak akan mengalami anomali Belady.

Pengimplementasiannya dapat dilakukan dengan dua cara, yaitu dengan menggunakan:

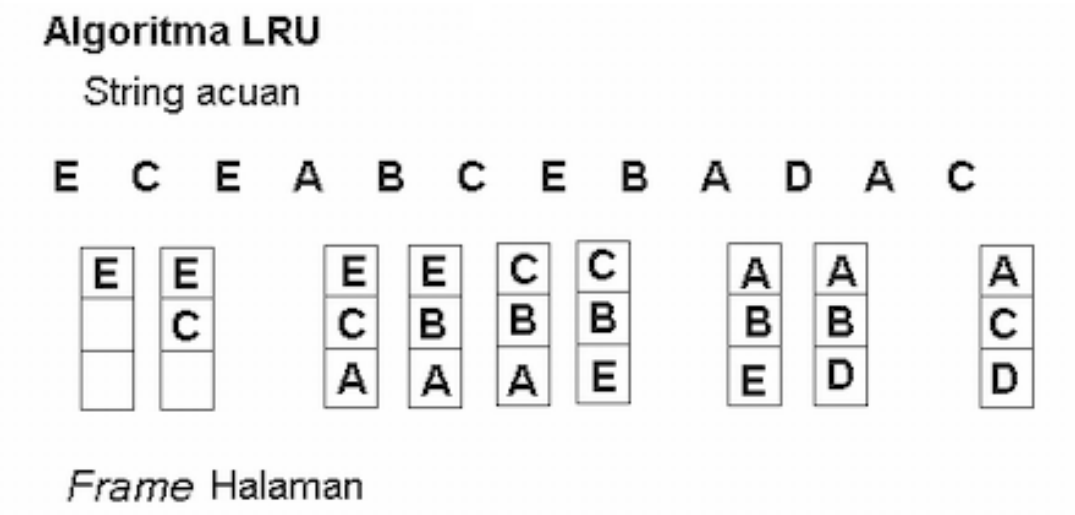
- **Pencacah**

  Cara ini dilakukan dengan menggunakan *clock*. Setiap halaman memiliki nilai yang pada awalnya diinisialisasi dengan 0. Ketika mengakses ke suatu halaman baru, nilai pada *clock* di halaman tersebut akan ditambah 1. Halaman yang diganti adalah halaman yang memiliki nilai terkecil.

- ***Stack***

  Penggunaan implementasi 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. Akan tetapi cara ini lebih mahal implementasinya dibandingkan dengan menggunakan pencacah.

Gambar 34.5. Contoh Algoritma LRU



Gambar di atas terinspirasi dari buku *Applied Operating System*, Silberchatz, Galvin, Gagne, edisi VI tahun 2002.

34.4. Algoritma Perkiraan LRU

Pada dasarnya algoritma perkiraan LRU memiliki prinsip yang sama dengan algoritma LRU, yaitu halaman yang diganti adalah halaman yang tidak digunakan dalam jangka waktu terlama, hanya saja dilakukan modifikasi pada algoritma ini untuk mendapatkan hasil yang lebih baik. Perbedaannya dengan algoritma LRU terletak pada penggunaan bit acuan. Setiap halaman yang berbeda memiliki bit acuan. Pada awalnya bit acuan diinisialisasikan oleh perangkat keras dengan nilai 0. Nilainya akan berubah menjadi 1 bila dilakukan akses ke halaman tersebut.

Ada beberapa cara untuk mengimplementasikan algoritma ini, yaitu:

- **Algoritma NFU (*Not Frequently Used*)**

  Setiap halaman akan memiliki bit acuan yang terdiri dari 8 bit byte sebagai penanda. Pada awalnya semua bit nilainya 0, contohnya: 00000000. Setiap selang beberapa waktu, pencatat waktu melakukan interupsi kepada sistem operasi, kemudian sistem operasi menggeser 1 bit ke kanan dengan bit yang paling kiri adalah nilai dari bit acuan, yaitu bila halaman tersebut diakses nilainya 1 dan bila tidak diakses nilainya 0. Jadi halaman yang selalu digunakan pada setiap periode akan memiliki nilai 11111111. Halaman yang diganti adalah halaman yang memiliki nilai terkecil.

- **Algoritma *Second-Chance***

  Pengimplementasian algoritma ini dilakukan dengan menyimpan halaman pada sebuah *linked-list* dan halaman-halaman tersebut diurutkan berdasarkan waktu ketika halaman tersebut tiba di memori yang berarti menggunakan juga prinsip algoritma FIFO disamping menggunakan algoritma LRU. Apabila nilai bit acuan-nya 0, halaman dapat diganti. Dan apabila nilai bit acuan-nya 1, halaman tidak diganti tetapi bit acuan diubah menjadi 0 dan dilakukan pencarian kembali.

- **Algoritma *Clock***

  Meski pun algoritma *second-chance* sudah cukup baik, namun pada kenyataannya penggunaan algortima tersebut tidak efisien. Algoritma *clock* adalah penyempurnaan dari algoritma tersebut. Pada prinsipnya kedua algoritma tersebut sama, hanya terletak perbedaan pada pengimplementasiannya saja. Algortima ini menggunakan antrian melingkar yang berbentuk seperti jam dengan sebuah penunjuk yang akan berjalan melingkar mencari halaman untuk diganti.

- **Algoritma NRU (*Not Recently Used*)**

  Algoritma NRU mudah untuk dimengerti, efisien, dan memiliki kinerja yang cukup baik. Algoritma ini mempertimbangkan dua hal sekaligus, yaitu bit acuan dan bit modifikasi. Ketika terjadi kesalahan halaman, sistem operasi memeriksa semua halaman dan membagi halaman-halaman tersebut ke dalam empat kelas:

  - Kelas 1: Tidak digunakan dan tidak dimodifikasi, bit terbaik untuk dipindahkan.
  - Kelas 2: Tidak digunakan tapi dimodifikasi, tidak terlalu baik untuk dipindahkan karena halaman ini perlu ditulis sebelum dipindahkan.
  - Kelas 3: Digunakan tapi tidak dimodifikasi, ada kemungkinan halaman ini akan segera digunakan lagi.
  - Kelas 4: Digunakan dan dimodifikasi, halaman ini mungkin akan segera digunakan lagi dan halaman ini perlu ditulis ke disk sebelum dipindahkan.

34.5. Algoritma *Counting*

Dilakukan dengan menyimpan pencacah dari nomor acuan yang sudah dibuat untuk masing-masing halaman. Penggunaan algoritma ini memiliki kekurangan yaitu lebih mahal. Algoritma *Counting* dapat dikembangkan menjadi dua skema dibawah ini:

- **Algoritma *Least Frequently Used* (LFU)**

  Halaman yang diganti adalah halaman yang paling sedikit dipakai (nilai pencacah terkecil) dengan alasan bahwa halaman yang digunakan secara aktif akan memiliki nilai acuan yang besar.

- **Algoritma *Most Frequently Used* (MFU)**

  Halaman yang diganti adalah halaman yang paling sering dipakai (nilai pencacah terbesar) dengan alasan bahwa halaman dengan nilai terkecil mungkin baru saja dimasukkan dan baru digunakan

34.6. Algoritma *Page Buffering*

Sistem menyimpan *pool* dari *frame* yang kosong. Prosedur ini memungkinkan suatu proses mengulang dari awal secepat mungkin, tanpa perlu menunggu halaman yang akan dipindahkan

untuk ditulis ke disk karena *frame*-nya telah ditambahkan ke dalam *pool frame* kosong. Teknik seperti ini digunakan dalam sistem VAX/VMS.

34.7. Rangkuman

Konsep *page replacement* adalah jika tidak ada *frame* yang kosong, carilah *frame* yang tidak sedang digunakan, lalu dikosongkan dengan *swapping* dan ubah semua tabelnya sebagai indikasi bahwa *page* tersebut tidak akan lama berada di dalam memori. Beberapa algoritma *page replacement*: *First-in-First-Out* (FIFO), Optimal, *Counting*, LRU, dan Perkiraan *LRU*.

34.8. Latihan

1. Sebutkan perbedaan dan/atau persamaan alokasi global dan alokasi lokal!

34.9. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts*: 6th ed. John Wiley & Sons

Tanenbaum,Andrew S. Woodhull, Albert S. 1997. *Operating Systems Design and Implementation: Second Edition*. Prentice Hall.

http://css.uni.edu/

http://allan.ultra.nyu.edu/~gottlieb/courses/1999-00-spring/os/lecture-11.html

http://www.cs.wisc.edu/~solomon/cs537/paging.html

http://www.cs.princeton.edu/courses/archive/spring02/cs217/lectures/paging.pdf

http://cwx.prenhall.com/bookbind/pubbooks/tanenbaum/chapter0/deluxe.html

http://www.cs.jhu.edu/~yairamir/cs418/os5/

Bab 35. Strategi Alokasi Frame

35.1. Alokasi *Frame*

Hal yang mendasari strategi alokasi *frame* yang menyangkut memori virtual adalah bagaimana membagi memori yang bebas untuk beberapa proses yang sedang dikerjakan. Dapat kita ambil satu contoh yang mudah pada sistem satu pemakai. Misalnya sebuah sistem mempunyai seratus *frame* bebas untuk proses *user*. Untuk permintaan halaman murni, keseratus *frame* tersebut akan ditaruh pada daftar *frame* bebas. Ketika sebuah *user* mulai dijalankan, akan terjadi sederetan kesalahan halaman. Sebanyak seratus kesalahan halaman pertama akan mendapatkan *frame* dari daftar *frame* bebas. Saat *frame* bebas sudah habis, sebuah algoritma pergantian halaman akan digunakan untuk memilih salah satu dari seratus halaman di memori yang diganti dengan yang ke seratus satu, dan seterusnya. Ketika proses selesai atau diterminasi, seratus *frame* tersebut akan disimpan lagi pada daftar *frame* bebas.

Ada beberapa variasi untuk strategi sederhana ini antara lain kita bisa meminta sistem operasi untuk mengalokasikan seluruh *buffer* dan ruang tabelnya dari daftar *frame* bebas. Saat ruang ini tidak digunakan oleh sistem operasi, ruang ini bisa digunakan untuk mendukung permintaan halaman dari *user*. Kita juga dapat menyimpan tiga *frame* bebas dari daftar *frame* bebas, sehingga ketika terjadi kesalahan halaman, ada *frame* bebas yang dapat digunakan untuk permintaan halaman. Saat penggantian halaman terjadi, penggantinya dapat dipilih, kemudian ditulis ke *disk*, sementara proses *user* tetap berjalan.

Pada dasarnya yaitu proses pengguna diberikan *frame* bebas yang mana saja. Masalah yang muncul ketika penggantian halaman dikombinasikan dengan *multiprogramming*. Hal ini terjadi karena *multiprogramming* menaruh dua (atau lebih) proses di memori pada waktu yang bersamaan.

Jumlah *Frame* Minimum

Ada suatu batasan dalam mengalokasikan *frame*, kita tidak dapat mengalokasikan *frame* lebih dari jumlah *frame* yang ada. Hal yang utama adalah berapa minimum *frame* yang harus dialokasikan agar jika sebuah instruksi dijalankan, semua informasinya ada dalam memori. Jika terjadi kesalahan halaman sebelum eksekusi selesai, instruksi tersebut harus diulang. Sehingga kita harus mempunyai jumlah *frame* yang cukup untuk menampung semua halaman yang dibutuhkan oleh sebuah instruksi.

Jumlah *frame* mimimum yang bisa dialokasikan ditentukan oleh arsitektur komputer. Sebagai contoh, instruksi *move* pada PDP-11 adalah lebih dari satu kata untuk beberapa modus pengalamatan, sehingga instruksi tersebut bisa membutuhkan dua halaman. Sebagai tambahan, tiap operannya mungkin merujuk tidak langsung, sehingga total ada enam *frame*. Kasus terburuk untuk IBM 370 adalah instruksi MVC. Karena instruksi tersebut adalah instruksi perpindahan dari penyimpanan ke penyimpanan, instruksi ini butuh 6 bit dan dapat memakai dua halaman. Satu blok karakter yang akan dipindahkan dan daerah tujuan perpindahan juga dapat memakai dua halaman, sehingga situasi ini membutuhkan enam *frame*.

Algoritma Alokasi

Algoritma pertama yaitu *equal allocation*. Algoritma ini memberikan bagian yang sama, sebanyak m/n frame untuk tiap proses. Sebagai contoh ada 100 *frame* tersisa dan lima proses, maka tiap proses akan mendapatkan 20 *frame*. *Frame* yang tersisa, sebanyak 3 buah dapat digunakan sebagai *frame* bebas cadangan.

Sebuah alternatif yaitu pengertian bahwa berbagai proses akan membutuhkan jumlah memori yang berbeda. Jika ada sebuah proses sebesar 10K dan sebuah proses basis data 127K dan hanya kedua proses ini yang berjalan pada sistem, maka ketika ada 62 *frame* bebas, tidak masuk akal jika kita memberikan masing-masing proses 31 *frame*. Proses pertama hanya butuh 10 *frame*, 21 *frame* lain akan terbuang percuma. Untuk menyelesaikan masalah ini, kita menggunakan algoritma kedua yaitu *proportional allocation*. Kita mengalokasikan memori yang tersedia kepada setiap proses tergantung pada ukurannya.

Ukuran memori virtual untuk proses pi = si, dan S = si.

Lalu, jika jumlah total dari *frame* yang tersedia adalah m, kita mengalokasikan proses ai ke proses pi, dimana ai mendekati:

ai = si / S x m

Dalam kedua strategi ini, tentu saja, alokasi untuk setiap proses bisa bervariasi berdasarkan *multiprogramming* level-nya. Jika *multiprogramming* level-nya meningkat, setiap proses akan kehilangan beberapa *frame* guna menyediakan memori yang dibutuhkan untuk proses yang baru. Di sisi lain, jika *multiprogramming* level-nya menurun, *frame* yang sudah dialokasikan pada bagian proses sekarang bisa disebar ke proses-proses yang masih tersisa.

Mengingat hal itu, dengan *equal allocation* atau pun *proportional allocation*, proses yang berprioritas tinggi diperlakukan sama dengan proses yang berprioritas rendah. Berdasarkan definisi tersebut, bagaimana pun juga, kita ingin memberi memori yang lebih pada proses yang berprioritas tinggi untuk mempercepat eksekusi-nya.

Satu pendekatan adalah menggunakan skema *proportional allocation* dimana perbandingan *frame*-nya tidak tergantung pada ukuran relatif dari proses, melainkan lebih pada prioritas proses, atau tergantung kombinasi dari ukuran dan prioritas. Algoritma ini dinamakan alokasi prioritas.

Alokasi Global lawan Lokal

Hal penting lainnya dalam pengalokasian *frame* adalah pergantian halaman. Proses-proses bersaing mendapatkan *frame*, maka dari itu kita dapat mengklasifikasikan algoritma penggantian halaman kedalam dua kategori; Penggantian Global dan Penggantian Lokal. Pergantian global memperbolehkan sebuah proses mencari *frame* pengganti dari semua *frame-frame* yang ada, walaupun *frame* tersebut sedang dialokasikan untuk proses yang lain. Hal ini memang efisien. tetapi ada kemungkinan proses lain tidak mendapatkan *frame* karena framenya terambil oleh proses lain. Penggantian lokal memberi aturan bahwa setiap proses hanya boleh memilih *frame* pengganti dari *frame-frame* yang memang dialokasikan untuk proses itu sendiri.

Sebagai contoh, misalkan ada sebuah skema alokasi yang memperbolehkan proses berprioritas tinggi untuk mencari *frame* pengganti dari proses yang berprioritas rendah. Proses berprioritas tinggi ini dapat mancari *frame* pengganti dari *frame-frame* yang telah dialokasikan untuknya atau dari *frame-frame* yang dialokasikan untuk proses berprioritas lebih rendah.

Dalam penggantian lokal, jumlah *frame* yang teralokasi tidak berubah. Dengan Penggantian Global, ada kemungkinan sebuah proses hanya menyeleksi frame-frame yang teralokasi pada proses lain, sehingga meningkatkan jumlah frame yang teralokasi pada proses itu sendiri (asumsi bahwa proses lain tidak memilih *frame* proses tersebut untuk penggantian.

Masalah pada algoritma Penggantian Global adalah bahwa sebuah proses tidak bisa mengontrol kasalahan halaman-nya sendiri. Halaman-halaman dalam memori untuk sebuah proses tergantung tidak hanya pada prilaku penghalamanan dari proses tersebut, tetapi juga pada prilaku penghalamanan dari proses lain. Karena itu, proses yang sama dapat tampil berbeda (memerlukan 0,5 detik untuk satu eksekusi dan 10,3 detik untuk eksekusi berikutnya). Dalam Penggantian Lokal, halaman-halaman dalam memori untuk sebuah proses hanya dipengaruhi prilaku penghalamanan proses itu sendiri. Penggantian Lokal dapat menyembunyikan sebuah proses dengan membuatnya tidak tersedia bagi proses lain, menggunakan halaman yang lebih sedikit pada memori. Jadi, secara umum Penggantian Global menghasilkan sistem *throughput* yang lebih bagus, maka itu artinya metode yang paling sering digunakan.

35.2. *Thrashing*

Thrashing adalah keadaan dimana terdapat aktifitas yang tinggi dari penghalamanan. Aktifitas penghalamanan yang tinggi ini maksudnya adalah pada saat sistem sibuk melakukan *swap-in* dan *swap-out* dikarenakan banyak kasalahan halaman yang terjadi. Suatu proses dapat mengurangi jumlah *frame* yang digunakan dengan alokasi yang minimum. Tetapi jika sebuah proses tidak memiliki *frame* yang cukup, tetap ada halaman dalam jumlah besar yang memiliki kondisi aktif

digunakan. Maka hal ini mengakibatkan kasalahan halaman. Untuk seterusnya sistem harus mengganti beberapa halaman menjadi halaman yang akan dibutuhkan. Karena semua halamannya aktif digunakan, maka halaman yang diganti adalah halaman yang dalam waktu dekat berkemungkinan akan digunakan kembali. Hal ini mengakibatkan kesalahan halaman yang terus-menerus

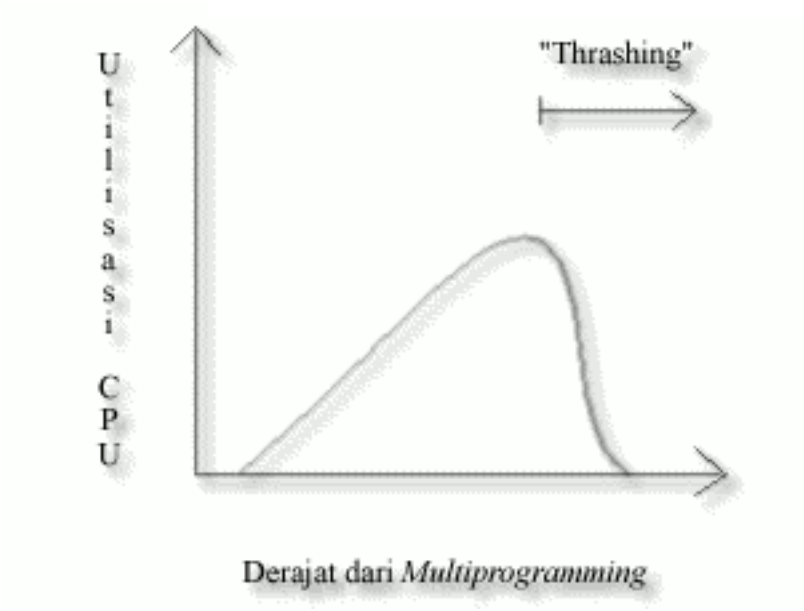
Penyebab *Thrashing*

Utilitas dari CPU selalu diharapkan tinggi hingga mendekati 100%. Jika proses yang dikerjakan CPU hanya sedikit, maka kita tidak bisa menjaga agar CPU sibuk. Utilitas dari CPU bisa ditingkatkan dengan meningkatkan jumlah proses. Jika Utilitas CPU rendah, maka sistem akan menambah derajat dari *multiprogramming* yang berarti menambah jumlah proses yang sedang berjalan. Pada titik tertentu, menambah jumlah proses justru akan menyebabkan utilitas CPU turun drastis dikarenakan proses-proses yang baru tidak mempunya memori yang cukup untuk berjalan secara efisien. Pada titik ini terjadi aktifitas penghalamanan yang tinggi yang akan menyebabkan *thrashing*.

Ketika sistem mendeteksi bahwa utilitas CPU menurun dengan bertambahnya proses, maka sistem meningkatkan lagi derajat dari *multiprogramming*. Proses-proses yang baru berusaha merebut frame-frame yang telah dialokasikan untuk proses yang sedang berjalan. Hal ini mengakibatkan kesalahan halaman meningkat tajam. Utilitas CPU akan menurun dengan sangat drastis diakibatkan oleh sistem yang terus menerus menambah derajat *multiprogramming*.

Pada gambar di bawah ini tergambar grafik dimana utilitas dari CPU akan terus meningkat seiiring dengan meningkatnya derajat dari *multiprogramming* hingga sampai pada suatu titik saat utilitas CPU menurun drastis. Pada titik ini, untuk menghentikan *thrashing*, derajat dari *multiprogramming* harus diturunkan.

Gambar 35.1. Derajat dari *Multiprogramming*



35.3. Membatasi Efek *Thrashing*

Efek dari *thrashing* dapat dibatasi dengan algoritma pergantian lokal atau prioritas. Dengan pergantian lokal, jika satu proses mulai *thrashing*, proses tersebut dapat mengambil *frame* dari proses yang lain dan menyebabkan proses itu tidak langsung *thrashing*. Jika proses mulai *thrashing*, proses itu akan berada pada antrian untuk melakukan penghalamanan yang mana hal ini memakan banyak waktu. Rata-rata waktu layanan untuk kesalahan halaman akan bertambah seiring dengan makin panjangnya rata-rata antrian untuk melakukan penghalamanan. Maka, waktu akses efektif

akan bertambah walaupun untuk suatu proses yang tidak *thrashing*.

Salah satu cara untuk menghindari *thrashing*, kita harus menyediakan sebanyak mungkin *frame* sesuai dengan kebutuhan suatu proses. Cara untuk mengetahui berapa *frame* yang dibutuhkan salah satunya adalah dengan strategi *Working Set*.

Selama satu proses di eksekusi, model lokalitas berpindah dari satu lokalitas satu ke lokalitas lainnnya. Lokalotas adalah kumpuulan halaman yang aktif digunakan bersama. Suatu program pada umumnya dibuat pada beberapa lokalitas sehingga ada kemungkinan terjadi *overlap*. *Thrashing* dapat muncul bila ukuran lokalitas lebih besar dari ukuran memori total.

Model *Working Set*

Strategi *Working set* dimulai dengan melihat berapa banyak *frame* yang sesungguhnya digunakan oleh suatu proses. *Working set model* mengatakan bahwa sistem hanya akan berjalan secara efisien jika masing-masing proses diberikan jumlah halaman *frame* yang cukup. Jika jumlah *frame* tidak cukup untuk menampung semua proses yang ada, maka akan lebih baik untuk menghentikan satu proses dan memberikan halamannya untuk proses yang lain.

Working set model merupakan model lokalitas dari suatu eksekusi proses. Model ini menggunakan parameter Δ (delta) untuk mendefinisikan *working set window*. Untuk menentukan halaman yang dituju, yang paling sering muncul. Kumpulan dari halaman dengan Δ halaman yang dituju yang paling sering muncul disebut *working set*. *Working set* adalah pendekatan dari program lokalitas.

Contoh:

Keakuratan *working set* tergantung pada pemilihan Δ.

1. Jika Δ terlalu kecil, tidak akan dapat mewakilkan keseluruhan dari lokalitas.
2. Jika Δ terlalu besar, akan menyebabkan *overlap* beberapa lokalitas.
3. Jika Δ tidak terbatas, *working set* adalah kumpulan *page* sepanjang eksekusi program.

Jika kita menghitung ukuran dari *Working Set*, WWSi, untuk setiap proses pada sistem, kita hitung dengan D = WSSi, dimana D merupakan total *demand* untuk *frame*.

Jika total perminataan lebih dari total banyaknya *frame* yang tersedia (D > m), *thrashing* dapat terjadi karena beberapa proses akan tidak memiliki *frame* yang cukup. Jika hal tersebut terjadi, dilakukan satu pengeblokan dari proses-proses yang sedang berjalan.

Strategi *Working Set* menangani *thrashing* dengan tetap mempertahankan derajat dari *multiprogramming* setinggi mungkin.

Contoh: Δ = 1000 referensi, Penghitung interupsi setiap 5000 referensi.

Ketika kita mendapat interupsi, kita menyalin dan menghapus nilai bit referensi dari setiap halaman. Jika kesalahan halaman muncul, kita dapat menentukan bit referensi sekarang dan 2 pada bit memori untuk memutuskan apakah halaman itu digunakan dengan 10000 ke 15000 referensi terakhir.

Jika digunakan, paling sedikit satu dari bit-bit ini akan aktif. Jika tidak digunakan, bit ini akan menjadi tidak aktif.

Halaman yang memiliki paling sedikit 1 bit aktif, akan berada di *working-set*.

Hal ini tidaklah sepenuhnya akurat karena kita tidak dapat memberitahukan dimana pada interval

5000 tersebut, referensi muncul. Kita dapat mengurangi ketidakpastian dengan menambahkan sejarah bit kita dan frekuensi dari interupsi.

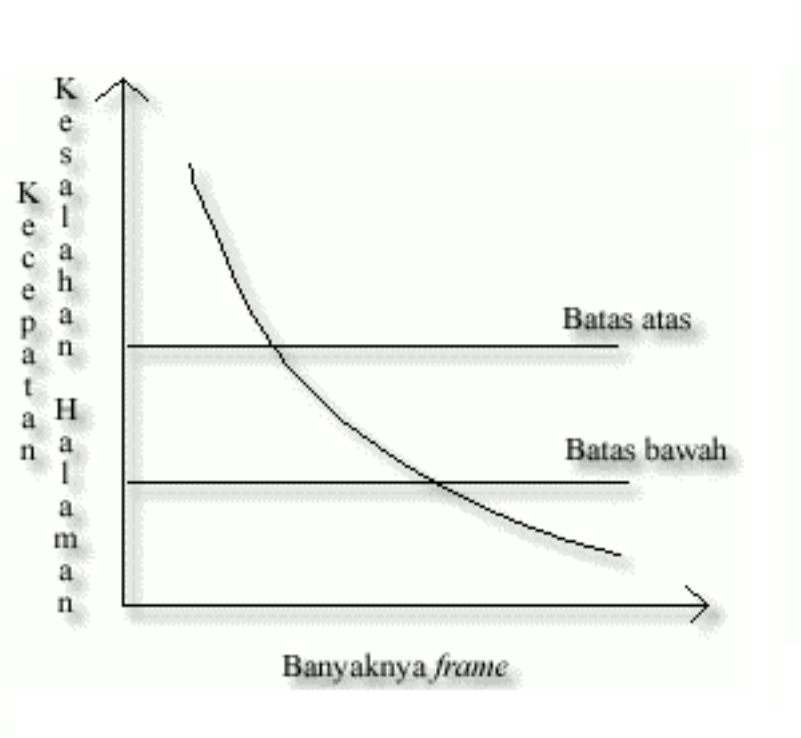
Contoh: 20 bit dan interupsi setiap 1500 referensi.

Frekuensi Kesalahan Halaman

Working-set dapat berguna untuk *prepaging*, tetapi kurang dapat mengontrol *thrashing*. Strategi menggunakan frekuensi kesalahan halaman mengambil pendekatan yang lebih langsung.

Thrashing memiliki kecepatan kasalahan halaman yang tinggi. Kita ingin mengontrolnya. Ketika terlalu tinggi, kita mengetahui bahwa proses membutuhkan *frame* lebih. Sama juga, jika terlalu rendah, maka proses mungkin memiliki terlalu banyak *frame*. Kita dapat menentukan batas atas dan bawah pada kecepatan kesalahan halaman seperti terlihat pada gambar berikut ini.

Gambar 35.2. Kecepatan *page-fault*



Jika kecepatan kasalahan halaman yang sesungguhnya melampaui batas atas, kita mengalokasikan *frame* lain ke proses tersebut, sedangkan jika kecepatan kasalahan halaman di bawah batas bawah, kita pindahkan *frame* dari proses tersebut. Maka kita dapat secara langsung mengukur dan mengontrol kecepatan kasalahan halaman untuk mencegah *thrashing*.

35.4. *Prepaging*

Pemilihan algoritma penggantian dan aturan alokasi adalah keputusan-keputusan utama yang kita buat untuk sistem pemberian halaman. Selain itu, masih banyak pertimbangan lain.

Sebuah ciri dari sistem permintaan pemberian halaman murni adalah banyaknya kesalahan halaman yang terjadi saat proses dimulai. Situasi ini merupakan hasil dari percobaan untuk mendapatkan tempat pada awalnya. Situasi yang sama mungkin muncul di lain waktu. Misalnya saat proses *swapped-out* dimulai kembali, seluruh halaman ada di cakram dan setiap halaman harus dibawa ke

dalam memori yang akan mengakibatkan banyaknya kesalahan halaman. *Prepaging* mencoba untuk mencegah pemberian halaman awal tingkat tinggi ini. Strateginya adalah untuk membawa seluruh halaman yang akan dibutuhkan pada satu waktu ke memori

Sebagai contoh, pada sistem yang menggunakan model *working set*, untuk setiap proses terdapat daftar dari semua halaman yang ada pada *working set* nya. Jika kita harus menunda sebuah proses (karena menunggu M/K atau kekurangan *frame* bebas), daftar *working set* untuk proses tersebut disimpan. Saat proses itu akan dilanjutkan kembali (permintaan M/K telah terpenuhi atau *frame* bebas yang cukup), secara otomatis seluruh *working set*-nya akan dibawa ke dalam memori sebelum memulai kembali proses tersebut.

Prepaging dapat berguna pada beberapa kasus. Yang harus dipertimbangkan adalah apakah biaya menggunakan *prepaging* lebih sedikit dari biaya menangani kesalahan halaman yang terjadi bila tidak memakai *prepaging*. Jika biaya *prepaging* lebih sedikit (karena hampir seluruh halaman yang di *prepage* digunakan) maka *prepaging* akan berguna. Sebaliknya, jika biaya *prepaging* lebih besar (karena hanya sedikit halaman dari yang di-*prepage* digunakan) maka *prepaging* akan merugikan.

35.5. Ukuran halaman

Para perancang sistem operasi untuk mesin yang sudah ada jarang memiliki pilihan terhadap ukuran halaman. Akan tetapi, saat merancang sebuah mesin baru, harus dipertimbangkan berapa ukuran halaman yang terbaik. Pada dasarnya tidak ada ukuran halaman yang paling baik, karena banyaknya faktor-faktor yang mempengaruhinya.

Salah satu faktor adalah ukuran tabel halaman. Untuk sebuah memori virtual dengan ukuran 4 *megabytes* (2^22), akan ada 4.096 halaman berukuran 1.024 *bytes*, tapi hanya 512 halaman jika ukuran halaman 8.192 *bytes*. Setiap proses yang aktif harus memiliki salinan dari tabel halaman-nya, jadi lebih masuk akal jika dipilih ukuran halaman yang besar.

Di sisi lain, pemanfaatan memori lebih baik dengan halaman yang lebih kecil. Jika sebuah proses dialokasikan di memori, mengambil semua halaman yang dibutuhkannya, mungkin proses tersebut tidak akan berakhir pada batas dari halaman terakhir. Jadi, ada bagian dari halaman terakhir yang tidak digunakan walaupun telah dialokasikan. Asumsikan rata-rata setengah dari halaman terakhir tidak digunakan, maka untuk halaman dengan ukuran 256 *bytes* hanya akan ada 128 *bytes* yang terbuang, bandingkan dengan halaman berukuran 8192 *bytes*, akan ada 4096 *bytes* yang terbuang. Untuk meminimalkan pemborosan ini, kita membutuhkan ukuran halaman yang kecil.

Masalah lain adalah waktu yang dibutuhkan untuk membaca atau menulis halaman. Waktu M/K terdiri dari waktu pencarian, *latency* dan transfer. Waktu transfer sebanding dengan jumlah yang dipindahkan (yaitu, ukuran halaman). Sedangkan waktu pencarian dan *latency* biasanya jauh lebih besar dari waktu transfer. Untuk laju pemindahan 2 MB/s, hanya dihabiskan 0.25 millidetik untuk memindahkan 512 *bytes*. Waktu *latency* mungkin sekitar 8 millidetik dan waktu pencarian 20 millidetik. Total waktu M/K 28.25 milidetik. Waktu transfer sebenarnya tidak sampai 1%. Sebagai perbandingan, untuk mentransfer 1024 *bytes*, dengan ukuran halaman 1024 *bytes* akan dihabiskan waktu 28.5 milidetik (waktu transfer 0.5 milidetik). Namun dengan halaman berukuran 512 *bytes* akan terjadi 2 kali transfer 512 *bytes* dengan masing-masing transfer menghabiskan waktu 28.25 milidetik sehingga total waktu yang dibutuhkan 56.5 milidetik. Kesimpulannya, untuk meminimalisasi waktu M/K dibutuhkan ukuran halaman yang lebih besar.

Masalah terakhir yang akan dibahas disini adalah mengenai kesalahan halaman. Misalkan ukuran halaman adalah 1 *byte*. Sebuah proses sebesar 100 KB, dimana hanya setengahnya yang menggunakan memori, akan menghasilkan kesalahan halaman sebanyak 51200. Sedangkan bila ukuran halaman sebesar 200 KB maka hanya akan terjadi 1 kali kesalahan halaman. Jadi untuk mengurangi jumlah kesalahan halaman dibutuhkan ukuran halaman yang besar.

Masih ada faktor lain yang harus dipertimbangkan (misalnya hubungan antara ukuran halaman dengan ukuran sektor pada peranti pemberian halaman). Tidak ada jawaban yang pasti berapa ukuran halaman yang paling baik. Sebagai acuan, pada 1990, ukuran halaman yang paling banyak dipakai adalah 4096 *bytes*. Sedangkan sistem modern saat ini menggunakan ukuran halaman yang jauh lebih besar dari itu.

35.6. Jangkauan *TLB*

Hit ratio dari *TLB* adalah persentasi alamat virtual yang diselesaikan dalam *TLB* daripada di tabel halaman. *Hit ratio* sendiri berhubungan dengan jumlah masukan dalam *TLB* dan cara untuk meningkatkan *hit ratio* adalah dengan menambah jumlah masukan dari *TLB*. Tetapi ini tidaklah murah karena memori yang dipakai untuk membuat *TLB* mahal dan haus akan tenaga.

Ada suatu ukuran lain yang mirip dengan *hit ratio* yaitu jangkauan *TLB*. Jangkauan *TLB* adalah jumlah memori yang dapat diakses dari *TLB*, jumlah tersebut merupakan perkalian dari jumlah masukan dengan ukuran halaman. Idealnya, *working set* dari sebuah proses disimpan dalam *TLB*. Jika tidak, maka proses akan menghabiskan waktu yang cukup banyak mengatasi referensi memori di dalam tabel halaman daripada di *TLB*. Jika jumlah masukan dari *TLB* dilipatgandakan, maka jangkauan *TLB* juga akan bertambah menjadi dua kali lipat. Tetapi untuk beberapa aplikasi hal ini masih belum cukup untuk menyimpan *working set*.

Cara lain untuk meningkatkan jangkauan *TLB* adalah dengan menambah ukuran halaman. Bila ukuran halaman dijadikan empat kali lipat dari ukuran awalnya (misalnya dari 32 KB menjadi 128 KB), maka jangkauan *TLB* juga akan menjadi empat kali lipatnya. Namun ini akan meningkatkan fragmentasi untuk aplikasi-aplikasi yang tidak membutuhkan ukuran halaman sebesar itu. Sebagai alternatif, OS dapat menyediakan ukuran halaman yang bervariasi. Sebagai contoh, UltraSparc II menyediakan halaman berukuran 8 KB, 64 KB, 512 KB, dan 4 MB. Sedangkan Solaris 2 hanya menggunakan halaman ukuran 8 KB dan 4 MB.

35.7. Tabel Halaman yang Dibalik

Kegunaan dari tabel halaman yang dibalik adalah untuk mengurangi jumlah memori fisik yang dibutuhkan untuk melacak penerjemahan alamat virtual-ke-fisik. Metode penghematan ini dilakukan dengan membuat tabel yang memiliki hanya satu masukan tiap halaman memori fisik, terdaftar oleh pasangan (proses-id, nomor-halaman).

Karena menyimpan informasi tentang halaman memori virtual yang mana yang disimpan di setiap *frame* fisik, tabel halaman yang dibalik mengurangi jumlah fisik memori yang dibutuhkan untuk menyimpan informasi ini. Bagaimana pun, tabel halaman yang dibalik tidak lagi mengandung informasi yang lengkap tentang ruang alamat logis dari sebuah proses, dan informasi itu dibutuhkan jika halaman yang direferensikan tidak sedang berada di memori. *Demand paging* membutuhkan informasi ini untuk memproses kesalahan halaman. Agar informasi ini tersedia, sebuah tabel halaman eksternal (satu tiap proses) harus tetap disimpan. Setiap tabel tampak seperti tabel halaman per proses tradisional, mengandung informasi dimana setiap halaman virtual berada.

Tetapi, apakah tabel halaman eksternal menegasikan kegunaan tabel halaman yang dibalik? Karena tabel-tabel ini direferensikan hanya saat kesalahan halaman terjadi, mereka tidak perlu tersedia secara cepat. Namun, mereka dimasukkan atau dikeluarkan dari memori sesuai kebutuhan. Sayangnya, sekarang kesalahan halaman mungkin terjadi pada manager memori virtual, menyebabkan kesalahan halaman lain karena pada saat mem-*page in* tabel halaman eksternal, ia harus mencari halaman virtual pada *backing store*. Kasus spesial ini membutuhkan penanganan di kernel dan penundaan pada proses *page-lookup*.

35.8. Struktur Program

Pemilihan struktur data dan struktur pemograman secara cermat dapat meningkatkan *locality* dan karenanya menurunkan tingkat kesalahan halaman dan jumlah halaman di *working set*. Sebuah *stack* memiliki *locality* yang baik, karena akses selalu dari atas. Sebuah *hash table*, di sisi lain, didesain untuk menyebar referensi-referensi, menghasilkan *locality* yang buruk. Tentunya, referensi akan *locality* hanyalah satu ukuran dari efisiensi penggunaan struktur data. Faktor-faktor lain yang berbobot berat termasuk kecepatan pencarian, jumlah total dari referensi dan jumlah total dari halaman yang disentuh.

35.9. M/K *Interlock*

Saat *demand paging* digunakan, kita terkadang harus mengizinkan beberapa halaman untuk dikunci di memori. Salah satu situasi muncul saat M/K dilakukan ke atau dari memori pengguna (virtual). M/K sering diimplementasikan oleh prosesor M/K yang terpisah. Sebagai contoh, sebuah pengendali pita magnetik pada umumnya diberikan jumlah *bytes* yang akan dipindahkan dan alamat memori untuk *buffer*. Saat pemindahan selesai, CPU diinterupsi.

Harus diperhatikan agar urutan dari kejadian-kejadian berikut tidak muncul: Sebuah proses mengeluarkan permintaan M/K, dan diletakkan di antrian untuk M/K tersebut. Sementara itu, CPU diberikan ke proses- proses lain. Proses-proses ini menimbulkan kesalahan halaman, dan, menggunakan algoritma penggantian global, salah satu dari mereka menggantikan halaman yang mengandung memori *buffer* untuk proses yang menunggu tadi. Halaman-halaman untuk proses tersebut dikeluarkan. Kadang-kadang kemudian, saat permintaan M/K bergerak maju menuju ujung dari antrian peranti, M/K terjadi ke alamat yang telah ditetapkan. Bagaimana pun, *frame* ini sekarang sedang digunakan untuk halaman berbeda milik proses lain.

Ada dua solusi untuk masalah ini. Salah satunya adalah jangan pernah menjalankan M/K kepada memori pengguna. Sedangkan solusi lainnya adalah dengan mengizinkan halaman untuk dikunci dalam memori.

35.10. Pemrosesan Waktu Nyata

Diskusi-diskusi di bab ini telah dikonsentrasikan dalam menyediakan penggunaan yang terbaik secara menyeluruh dari sistem komputer dengan meningkatkan penggunaan memori. Dengan menggunakan memori untuk data yang aktif, dan memindahkan data yang tidak aktif ke cakram, kita meningkatkan *throughput*. Bagaimana pun, proses individual dapat menderita sebagai hasilnya, sebab mereka sekarang mendapatkan kesalahan halaman tambahan selama eksekusi.

Pertimbangkan sebuah proses atau *thread* waktu-nyata. Proses tersebut berharap untuk memperoleh kendali CPU, dan untuk menjalankan penyelesaian dengan penundaan yang minimum. Memori virtual adalah kebalikan dari komputasi waktu nyata, sebab dapat menyebabkan penundaan jangka panjang yang tidak diharapkan pada eksekusi sebuah proses saat halaman dibawa ke memori. Untuk itulah, sistem-sistem waktu nyata hampir tidak memiliki memori virtual.

35.11. Windows NT

Pada bagian-bagian berikut ini akan dibahas bagaimana Windows NT, Solaris 2, dan Linux mengimplementasi memori virtual.

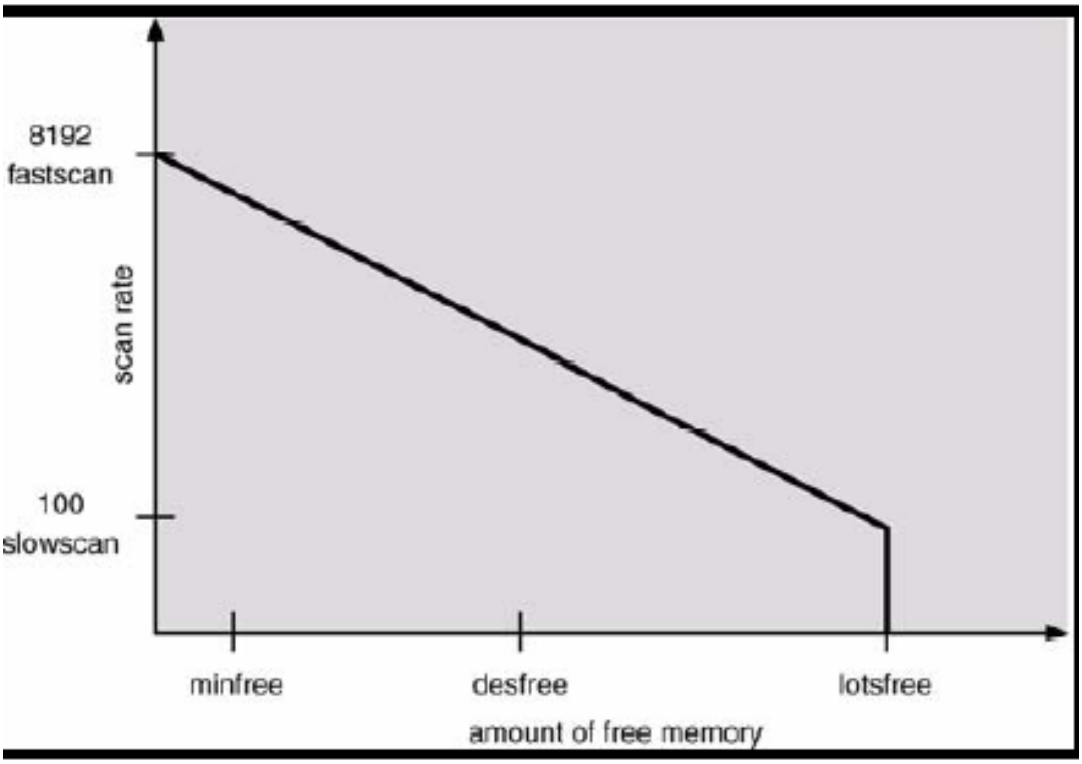
Windows NT mengimplementasikan memori virtual dengan menggunakan permintaan halaman melalui *clustering*. *Clustering* menanganani kesalahan halaman dengan menambahkan tidak hanya halaman yang terkena kesalahan, tetapi juga halaman-halaman yang berada disekitarnya. Saat proses pertama dibuat, dia diberikan *working set* minimum yaitu jumlah minimum halaman yang dijamin akan dimiliki oleh proses tersebut dalam memori. Jika memori yang cukup tersedia, proses dapat diberikan halaman sampai sebanyak *working set* maximum. Manager memori virtual akan menyimpan daftar dari halaman *frame* 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 halaman pengganti dengan aturan penggantian halaman 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 batasan. Hal ini bekerja dengan 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 dapat mendapatkan halaman tambahan.

35.12. Solaris 2

Dalam sistem operasi Solaris 2, jika sebuah proses menyebabkan terjadi kesalahan halaman, kernel akan memberikan halaman kepada proses tersebut dari daftar halaman bebas yang disimpan. Akibat dari hal ini adalah, kernel harus menyimpan sejumlah memori bebas. Terhadap daftar ini ada dua parameter yg disimpan yaitu *minfree* dan *lotsfree*, yaitu batasan minimum dan maksimum dari memori bebas yang tersedia. Empat kali dalam tiap detiknya, kernel memeriksa jumlah memori yang bebas. Jika jumlah tersebut jatuh di bawah *minfree*, maka sebuah proses *pageout* akan dilakukan, dengan pekerjaan sebagai berikut. Pertama *clock* akan memeriksa semua halaman dalam memori dan mengeset bit referensi menjadi 0. Saat berikutnya, *clock* kedua akan memeriksa bit referensi halaman dalam memori, dan mengembalikan bit yang masih di set ke 0 ke daftar memori bebas. Hal ini dilakukan sampai jumlah memori bebas melampaui parameter *lotsfree*. Lebih lanjut, proses ini dinamis, dapat mengatur kecepatan jika memori terlalu sedikit. Jika proses ini tidak bisa membebaskan memori , maka *kernel* memulai pergantian proses untuk membebaskan halaman yang dialokasikan ke proses-proses tersebut.

Gambar 35.3. ***Solar Page Scanner***



sumber: Silberschatz, "*Operating Systems: -- Fourth Edition*", Prentice Hall, 2001

35.13. Rangkuman

Masalah yang penting dari alokasi *frame* dengan penggunaan memori virtual adalah bagaimana membagi memori dengan bebas untuk beberapa proses yang sedang dikerjakan. Contoh algoritma yang lazim digunakan adalah *equal allocation* dan *proportional allocation*.

Algoritma *page replacement* dapat diklasifikasikan dalam 2 kategori, yaitu penggantian global dan penggantian lokal. Perbedaan antara keduanya terletak pada boleh tidaknya setiap proses memilih *frame* pengganti dari semua *frame* yang ada.

Utilitas dari CPU dapat menyebabkan *trashing*, dimana sistem sibuk melakukan *swapping* dikarenakan banyaknya *page-fault* yang terjadi. Efek dari trashing dapat dibatasi dengan algoritma penggantian lokal atau prioritas. Cara untuk mengetahui berapa banyak proses yang dibutuhkan suatu proses salah satunya adalah dengan strategi *working set*.

Pemilihan algoritma penggantian dan aturan alokasi adalah keputusan-keputusan utama yang kita buat untuk sistem *paging*. Selain itu, ada beberapa pertimbangan lain, antara lain *prepaging, TLB reach*, ukuran *page*, struktur program, *I/O interlock*, dan lain sebagainya. Beberapa contoh sistem operasi yang mengimplementasikan virtual memori adalah Windows NT, Solaris 2 dan Linux.

35.14. Latihan

1. Apakah penyebab terjadinya *thrasing*? Jelaskan!

35.15. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts*: 6th ed. John Wiley & Sons

Tanenbaum,Andrew S. Woodhull, Albert S. 1997. *Operating Systems Design and Implementation: Second Edition*. Prentice Hall.

http://css.uni.edu/

http://allan.ultra.nyu.edu/~gottlieb/courses/1999-00-spring/os/lecture-11.html

http://www.cs.wisc.edu/~solomon/cs537/paging.html

http://www.cs.princeton.edu/courses/archive/spring02/cs217/lectures/paging.pdf

http://cwx.prenhall.com/bookbind/pubbooks/tanenbaum/chapter0/deluxe.html

http://www.cs.jhu.edu/~yairamir/cs418/os5/

Bab 36. Memori Linux

36.1. Pendahuluan

Seperti pada Solaris 2, Linux juga menggunakan variasi dari algoritma *clock*. *Thread* dari kernel linux (kswapd) akan dijalankan secara periodik (atau dipanggil ketika penggunaan memori sudah berlebihan). Jika jumlah halaman yang bebas lebih sedikit dari batas atas halaman bebas, maka *thread* tersebut akan berusaha untuk membebaskan tiga halaman. Jika lebih sedikit dari batas bawah halaman bebas, *thread* tersebut akan berusaha untuk membebaskan enam halaman dan tidur untuk beberapa saat sebelum berjalan lagi. Saat dia berjalan, akan memeriksa mem_map, daftar dari semua halaman yang terdapat di memori. Setiap halaman mempunyai *byte* umur yang diinisialisasikan ke tiga. Setiap kali halaman ini diakses, maka umur ini akan ditambahkan (hingga maksimum 20), setiap kali kswapd memeriksa halaman ini, maka umur akan dikurangi. Jika umur dari sebuah halaman sudah mencapai 0 maka dia bisa ditukar. Ketika kswapd berusaha membebaskan halaman, dia pertama akan membebaskan halaman dari *cache*, jika gagal dia akan mengurangi *cache* sistem berkas, dan jika semua cara sudah gagal, maka dia akan menghentikan sebuah proses. Alokasi memori pada linux menggunakan dua buah alokasi yang utama, yaitu algoritma *buddy* dan *slab*. Untuk algoritma *buddy*, setiap rutin pelaksanaan alokasi ini dipanggil, dia memeriksa blok memori berikutnya, jika ditemukan dia dialokasikan, jika tidak maka daftar tingkat berikutnya akan diperiksa. Jika ada blok bebas, maka akan dibagi jadi dua, yang satu dialokasikan dan yang lain dipindahkan ke daftar yang di bawahnya.

36.2. Managemen Memori Fisik

Bagian ini menjelaskan bagaimana linux menangani memori dalam sistem. Memori managemen merupakan salah satu bagian terpenting dalam sistem operasi. Karena adanya keterbatasan memori, diperlukan suatu strategi dalam menangani masalah ini. Jalan keluarnya adalah dengan menggunakan memori virtual. Dengan memori virtual, memori tampak lebih besar daripada ukuran yang sebenarnya.

Dengan memori virtual kita dapat:

- Ruang alamat yang besar

  Sistem operasi membuat memori terlihat lebih besar daripada ukuran memori sebenarnya. Memori virtual bisa beberapa kali lebih besar daripada memori fisiknya.

- Pembagian memori fisik yang adil

  Managemen memori membuat pembagian yang adil dalam pengalokasian memori antara proses-proses.

- Perlindungan

  Memori managemen menjamin setiap proses dalam sistem terlindung dari proses-proses lainnya. Dengan demikian, program yang crash tidak akan mempengaruhi proses lain dalam sistem tersebut.

- Penggunaan memori virtual bersama

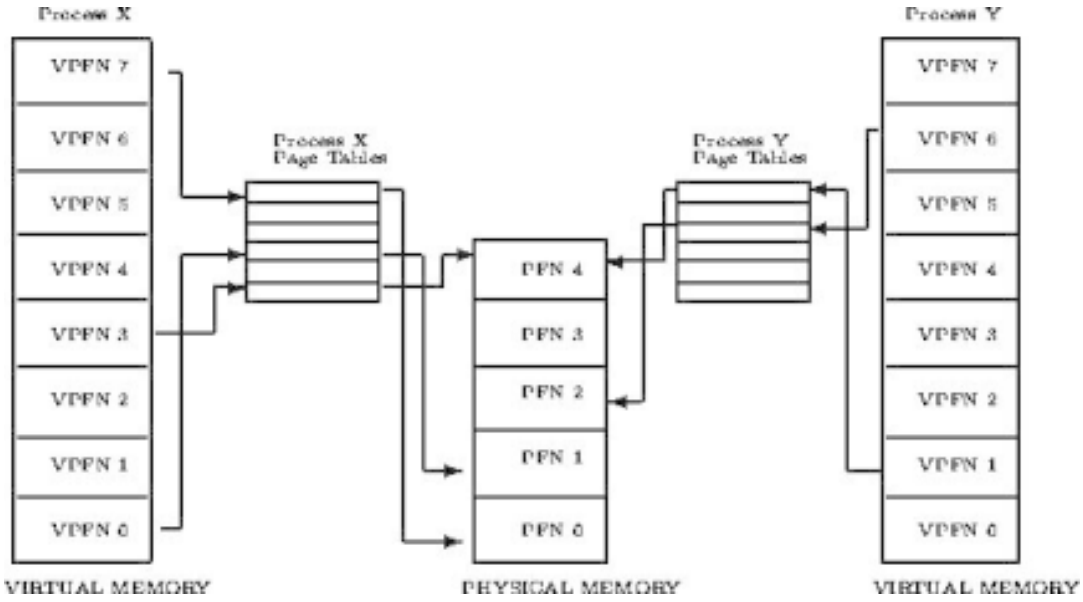
  Memori virtual mengizinkan dua buah proses berbagi memori diantara keduanya, contohnya dalam shared library. Kode library dapat berada di satu tempat, dan tidak dikopi pada dua program yang berbeda.

36.3. Memori Virtual

Memori fisik dan memori virtual dibagi menjadi bagian-bagian yang disebut page. Page ini

memiliki ukuran yang sama besar. Tiap page ini punya nomor yang unik, yaitu Page Frame Number (PFN). Untuk setiap instruksi dalam program, CPU melakukan mapping dari alamat virtual ke memori fisik yang sebenarnya.

Gambar 36.1. Pemetaan Memori Virtual ke Alamat Fisik. Sumber: . . .



Penerjemahan alamat di antara virtual dan memori fisik dilakukan oleh CPU menggunakan tabel page untuk proses x dan proses y. Ini menunjukkan virtial PFN 0 dari proses x dimap ke memori fisik PFN 1. Setiap anggota tabel page mengandung informasi berikut ini:

- Virtual PFN
- PFN fisik
- Informasi akses page dari page tersebut

Untuk menerjemahkan alamat virtual ke alamat fisik, pertama-tama CPU harus menangani alamat virtual PFN dan offsetnya di virtual page. CPU mencari tabel page proses dan mancari anggota yang sesuai degan virtual PFN. Ini memberikan PFN fisik yang dicari. CPU kemudian mengambil PFN fisik dan mengalikannya dengan besar page untuk mendapat alamat basis page tersebut di dalam memori fisik. Terakhir, CPU menambahkan offset ke instruksi atau data yang dibutuhkan. Dengan cara ini, memori virtual dapat dimap ke page fisik dengan urutan yang teracak.

36.4. *Demand Paging*

Cara untuk menghemat memori fisik adalah dengan hanya meload page virtual yang sedang digunakan oleh program yang sedang dieksekusi. Tehnik dimana hanya meload page virtual ke memori hanya ketika program dijalankan disebut demand paging.

Ketika proses mencoba mengakses alamat virtual yang tidak ada di dalam memori, CPU tidak dapat menemukan anggota tabel page. Contohnya, dalam gambar, tidak ada anggota tabel page untuk proses x untuk virtual PFN 2 dan jika proses x ingin membaca alamat dari virtual PFN 2, CPU tidak dapat menterjemahkan alamat ke alamat fisik. Saat ini CPU bergantung pada sistem operasi untuk menangani masalah ini. CPU menginformasikan kepada sistem operasi bahwa page fault telah terjadi, dan sistem operasi membuat proses menunggu selama sistem operasi menagani masalah ini.

CPU harus membawa page yang benar ke memori dari image di disk. Akses disk membutuhkan waktu yang sangat lama dan proses harus menunggu sampai page selesai diambil. Jika ada proses lain yang dapat dijalankan, maka sistem operai akan memilihnya untuk kemudian dijalankan. Page yang diambil kemudian dituliskan di dalam page fisik yang masih kosong dan anggota dari virtual PFN ditambahkan dalam tabel page proses. Proses kemudian dimulai lagi pada tempat dimana page fault terjadi. Saat ini terjadi pengaksesan memori virtual, CPU membuat penerjemahan dan kemudian proses dijalankan kembali.

Demand paging terjadi saat sistem sedang sibuk atau saat image pertama kali diload ke memori. Mekanisme ini berarti sebuah proses dapat mengeksekusi image dimana hanya sebagian dari image tersebut terdapat dalam memori fisik.

36.5. *Swaping*

Jika memori fisik tiba-tiba habis dan proses ingin memindahkan sebuah page ke memori, sistem operasi harus memutuskan apa yang harus dilakukan. Sistem operasi harus adil dalam mambagi page fisik dalam sistem diantara proses yang ada, bisa juga sistem operasi menghapus satu atau lebih page dari memori untuk membuat ruang untuk page baru yang dibawa ke memori. Cara page virtual dipilih dari memori fisik berpengaruh pada efisiensi sistem.

Linux menggunakan tehnik page aging agar adil dalam memilih page yang akan dihapus dari sistem. Ini berarti setiap page memiliki usia sesuai dengan berapa sering page itu diakses. Semakin sering sebuah page diakses, semakin muda page tersebut. Page yang tua adalah kandidat untuk diswap.

36.6. Pengaksesan Memori Virtual Bersama

Memori virtual mempermudah proses untuk berbagi memori saat semua akses ke memori menggunakan tabel page. Proses yang akan berbagi memori virtual yang sama, page fisik yang sama direference oleh banyak proses. Tabel page untuk setiap proses mengandung anggota page table yang mempunyai PFN fisik yang sama.

36.7. Efisiensi

Desainer dari CPU dan sistem operasi berusaha meningkatkan kinerja dari sistem. Disamping membuat prosesor, memori semakin cepat, jalan terbaik adalah manggunakan cache. Berikut ini adalah beberapa cache dalam managemen memori di linux:

- Page Cache

  Digunakan untuk meningkatkan akses ke image dan data dalam disk. Saat dibaca dari disk, page dicache di page cache. Jika page ini tidak dibutuhkan lagi pada suatu saat, tetapi dibutuhkan lagi pada saat yang lain, page ini dapat segera diambil dari page cache.

- Buffer Cache

  Page mungkin mengandung buffer data yang sedang digunakan oleh kernel, device driver dan lain-lain. Buffer cache tampak seperti daftar buffer. Contohnya, device driver membutuhkan buffer 256 bytes, adalah lebih cepat untuk mengambil buffer dari buffer cache daripada mengalokasikan page fisik lalu kemudian memecahnya menjadi 256 bytes buffer-buffer.

- Swap Cache

  Hanya page yang telah ditulis ditempatkan dalam swap file. Selama page ini tidak mengalami perubahan setelah ditulis ke dalam swap file, maka saat berikutnya page di swap out tidak perlu menuliskan kembali jika page telah ada di swap file. Di sistem yang sering mengalami swap, ini dapat menghemat akses disk yang tidak perlu.

  Salah satu implementasi yang umum dari hardware cache adalah di CPU, cache dari anggota

tabel page. Dalam hal ini, CPU tidak secara langsung membaca tabel page, tetap mencache terjemahan page yang dibutuhkan.

36.8. Load dan Eksekusi Program

1. Penempatan program dalam memori

   Linux membuat tabel-tabel fungsi untuk loading program, memberikan kesempatan kepada setiap fungsi untuk meload file yang diberikan saat sistem call exec dijalankan. Pertama-tama file binari dari *page* ditempatkan pada memori virtual. Hanya pada saat program mencoba mengakses *page* yang telah diberikan terjadi *page fault*, maka *page* akan diload ke memori fisik.

2. Linking statis dan linking dinamis

   a. Linking statis:

      librari-librari yang digunakan oleh program ditaruh secara langsung dalam file binari yang dapat dieksekusi. Kerugian dari linking statis adalah setiap program harus mengandung kopi library sistem yang umum.

   b. Linking dinamis:

      hanya sekali meload librari sistem menuju memori. Linking dinamis lebih efisien dalam hal memori fisik dan ruang disk.

36.9. Rangkuman

Algoritma *page replacement* dapat diklasifikasikan dalam 2 kategori, yaitu penggantian global dan penggantian lokal. Perbedaan antara keduanya terletak pada boleh tidaknya setiap proses memilih *frame* pengganti dari semua *frame* yang ada.

Utilitas dari CPU dapat menyebabkan *trashing*, dimana sistem sibuk melakukan *swapping* dikarenakan banyaknya *page-fault* yang terjadi. Efek dari trashing dapat dibatasi dengan algoritma penggantian lokal atau prioritas. Cara untuk mengetahui berapa banyak proses yang dibutuhkan suatu proses salah satunya adalah dengan strategi *working set*.

Pemilihan algoritma penggantian dan aturan alokasi adalah keputusan-keputusan utama yang kita buat untuk sistem *paging*. Selain itu, ada beberapa pertimbangan lain, antara lain *prepaging, TLB reach*, ukuran *page*, struktur program, *I/O interlock*, dan lain sebagainya. Beberapa contoh sistem operasi yang mengimplementasikan virtual memori adalah Windows NT, Solaris 2 dan Linux.

36.10. Latihan

1. Apakah penyebab terjadinya *thrasing*? Jelaskan!

36.11. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts*: 6th ed. John Wiley & Sons

Tanenbaum,Andrew S. Woodhull, Albert S. 1997. *Operating Systems Design and Implementation: Second Edition*. Prentice Hall.

http://css.uni.edu/

http://allan.ultra.nyu.edu/~gottlieb/courses/1999-00-spring/os/lecture-11.html

http://www.cs.wisc.edu/~solomon/cs537/paging.html

http://www.cs.princeton.edu/courses/archive/spring02/cs217/lectures/paging.pdf

http://cwx.prenhall.com/bookbind/pubbooks/tanenbaum/chapter0/deluxe.html

http://www.cs.jhu.edu/~yairamir/cs418/os5/

Bagian VI. Memori Sekunder

Daftar Isi

Bab 37. Sistem Berkas

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

Masalah berikutnya adalah apabila proses tersebut berhenti maka informasinya hilang. Padahal ada beberapa informasi yang penting dan harus bertahan beberapa waktu bahkan selamanya.

Ada pun masalah ketiga yaitu sangatlah perlu terkadang untuk lebih dari satu proses mengakses informasi secara berbarengan. Untuk memecahkan masalah ini, informasi tersebut harus dapat berdiri sendiri tanpa tergantung dengan sebuah proses.

Pada akhirnya kita memiliki masalah-masalah yang cukup signifikan dan penting untuk dicari solusinya. Pertama kita harus dapat menyimpan informasi dengan ukuran yang besar. Kedua, informasi harus tetap ketika proses berhenti. Ketiga, informasi harus dapat diakses oleh lebih dari satu proses secara bersamaan. Solusi dari ketiga masalah diatas adalah sesuatu yang disebut berkas.

Berkas adalah sebuah unit tempat menyimpan informasi. Berkas ini dapat diakses lebih dari satu proses, dapat dibaca, dan bahkan menulis yang baru. Informasi yang disimpan dalam berkas harus persisten, dalam artian tidak hilang sewaktu proses berhenti. Berkas-berkas ini diatur oleh sistem operasi, bagaimana strukturnya, namanya, aksesnya, penggunaannya, perlindungannya, dan implementasinya. Bagian dari sistem operasi yang mengatur masalah-masalah ini disebut sistem berkas.

Untuk kebanyakan pengguna, sistem berkas adalah aspek yang paling terlihat dari sebuah sistem operasi. Dia menyediakan mekanisme untuk penyimpanan *online* dan akses ke data dan program. Sistem berkas terbagi menjadi dua bagian yang jelas; koleksi berkas (masing-masing menyimpan data yang berkaitan) dan struktur direktori (mengatur dan menyediakan informasi mengenai semua berkas yang berada di sistem). Sekarang marilah kita memperdalam konsep dari berkas tersebut.

37.1. Konsep Berkas

Berkas adalah sebuah koleksi informasi berkaitan yang diberi nama dan disimpan di dalam *secondary storage*. Biasanya sebuah berkas merepresentasikan data atau program. Ada pun jenis-jenis dari berkas:

- *Text file*: yaitu urutan dari karakter-karakter yang diatur menjadi barisan dan mungkin halaman.
- *Source file*: yaitu urutan dari berbagai subroutine dan fungsi yang masing-masing kemudian diatur sebagai deklarasi-deklarasi diikuti oleh pernyataan-pernyataan yang dapat diexecute.
- *Object file*: yaitu urutan dari byte-byte yang diatur menjadi blok-blok yang dapat dipahami oleh penghubung system.
- *Executable file*: adalah kumpulan dari bagian-bagian kode yang dapat dibawa ke memori dan dijalankan oleh loader.

37.2. Atribut berkas

Selain nama dan data, sebuah berkas dikaitkan dengan informasi-informasi tertentu yang juga penting untuk dilihat pengguna, seperti kapan berkas itu dibuat, ukuran berkas, dan lain-lain. Kita akan sebut informasi-informasi ekstra ini atribut. Setiap sistem mempunyai sistem atribusi yang berbeda-beda, namun pada dasarnya memiliki atribut-atribut dasar seperti berikut ini:

- Nama: nama berkas simbolik ini adalah informasi satu-satunya yang disimpan dalam format yang dapat dibaca oleh pengguna.

- *Identifier*: tanda unik ini yang biasanya merupakan sebuah angka, mengenali berkas didalam sebuah sistem berkas; tidak dapat dibaca oleh pengguna.
- Jenis: informasi ini diperlukan untuk sistem-sistem yang mendukung jenis berkas yang berbeda.
- Lokasi: informasi ini adalah sebuah penunjuk pada sebuah *device* dan pada lokasi berkas pada *device* tersebut.
- Ukuran: ukuran dari sebuah berkas (dalam bytes, words, atau blocks) dan mungkin ukuran maksimum dimasukkan dalam atribut ini juga.
- Proteksi: informasi yang menentukan siapa yang dapat melakukan *read*, *write*, *execute*, dan lainnya.
- Waktu dan identifikasi pengguna: informasi ini dapat disimpan untuk pembuatan berkas, modifikasi terakhir, dan penggunaan terakhir. Data-data ini dapat berguna untuk proteksi, keamanan, dan *monitoring* penggunaan.

37.3. Jenis Berkas

Salah satu atribut dari sebuah berkas yang cukup penting adalah jenis berkas. Saat kita mendesain sebuah sistem berkas, kita perlu mempertimbangkan bagaimana operating sistem akan mengenali berkas-berkas dengan jenis yang berbeda. Apabila sistem operasi dapat mengenali, maka menjalankan berkas tersebut bukan suatu masalah. Seperti contohnya, apabila kita hendak mengeprint bentuk *binary-object* dari sebuah program, yang didapat biasanya adalah sampah, namun hal ini dapat dihindari apabila sistem operasi telah diberitahu akan adanya jenis berkas tersebut.

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

37.4. Operasi Berkas

Fungsi dari berkas adalah untuk menyimpan data dan mengizinkan kita membacanya. Dalam proses ini ada beberapa operasi yang dapat dilakukan berkas. Ada pun operasi-operasi dasar yang dilakukan berkas, yaitu:

- Membuat Berkas (*Create*):

  Kita perlu dua langkah untuk membuat suatu berkas. Pertama, kita harus temukan tempat didalam sistem berkas. Kedua, sebuah entri untuk berkas yang baru harus dibuat dalam direktori. Entri dalam direktori tersebut merekam nama dari berkas dan lokasinya dalam sistem berkas.

- Menulis sebuah berkas (*Write*):

  Untuk menulis sebuah berkas, kita membuat sebuah *system call* yang menyebutkan nama berkas dan informasi yang akan di-nulis kedalam berkas.

- Membaca Sebuah berkas (*Read*):

  Untuk membaca sebuah berkas menggunakan sebuah system call yang menyebut nama berkas yang dimana dalam blok memori berikutnya dari sebuah berkas harus diposisikan.

- Memposisikan Sebuah Berkas (*Reposition*):

  Direktori dicari untuk entri yang sesuai dan *current-file-position* diberi sebuah nilai. Operasi ini di dalam berkas tidak perlu melibatkan M/K, selain itu juga diketahui sebagai *file seek.*

- Menghapus Berkas (*Delete*):

  Untuk menghapus sebuah berkas kita mencari dalam direktori untuk nama berkas tersebut. Setelah ditemukan, kita melepaskan semua spasi berkas sehingga dapat digunakan kembali oleh berkas-berkas lainnya dan menghapus entry direktori.

- Menghapus Sebagian Isi Berkas (*Truncate*):

  User mungkin mau menghapus isi dari sebuah berkas, namun menyimpan atributnya. Daripada memaksa pengguna untuk menghapus berkas tersebut dan membuatnya kembali, fungsi ini tidak akan mengganti atribut, kecuali panjang berkas dan mendefinisikan ulang panjang berkas tersebut menjadi nol.

Keenam operasi diatas merupakan operasi-operasi dasar dari sebuah berkas yang nantinya dapat dikombinasikan untuk membentuk operasi-operasi baru lainnya. Contohnya apabila kita ingin menyalin sebuah berkas, maka kita memakai operasi *create* untuk membuat berkas baru, *read* untuk membaca berkas yang lama, dan *write* untuk menulisnya pada berkas yang baru.

37.5. Struktur Berkas

Berkas dapat di struktur dalam beberapa cara. Cara yang pertama adalah sebuah urutan *bytes* yang tidak terstruktur. Akibatnya sistem operasi tidak tahu atau peduli apa yang ada dalam berkas, yang dilihatnya hanya bytes. Ini menyediakan fleksibilitas yang maksimum. User dapat menaruh apa pun yang mereka mau dalam berkas, dan sistem operasi tidak membantu, namun tidak juga menghalangi.

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

Struktur berkas yang ketiga, adalah menggunakan sebuah *tree*. Dalam struktur ini sebuah berkas terdiri dari sebuah *tree* dari rekaman-rekaman tidak perlu dalam panjang yang sama, tetapi masing-masing memiliki sebuah *field key* dalam posisi yang telah ditetapkan dalam rekaman tersebut. Tree ini disort dalam *field key* dan mengizinkan pencarian yang cepat untuk sebuah *key tertentu*.

37.6. Metode Akses

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

- Akses Sekuensial

  Akses ini merupakan yang paling sederhana dan paling umum digunakan. Informasi di dalam berkas diproses secara berurutan. Sebagai contoh, editor dan kompilator biasanya mengakses berkas dengan cara ini.

- *Akses Langsung*

  Metode berikutnya adalah akses langsung atau dapat disebut *relative access*. Sebuah berkas dibuat dari rekaman-rekaman logical yang panjangnya sudah ditentukan, yang mengizinkan program untuk membaca dan menulis rekaman secara cepat tanpa urutan tertentu.

37.7. Rangkuman

Di dalam sebuah sistem operasi, salah satu hal yang paling penting adalah sistem berkas. Sistem berkas ini muncul karena ada tiga masalah utama yang cukup signifikan: kebutuhan untuk menyimpan data dalam jumlah yang besar, kebutuhan agar data tidak mudah hilang (*non-volatile*), dan informasi harus berdiri sendiri tidak bergantung pada proses. Pada sistem berkas ini, diatur segala rupa macam yang berkaitan dengan sebuah berkas mulai dari atribut, tipe, operasi, struktur, sampai metode akses berkas.

37.8. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:
   - *Dynamic Storage Allocation Strategy: "Best Fit" vs. "Worse Fit".*
   - *File Operations: "Deleting" vs. "Truncating".*
   - *Storage System: "Volatile" vs. "Non-volatile".*
   - *File Allocation Methods: "Contiguous" vs. "Linked".*
   - *Disk Management: "Boot Block" vs. "Bad Block".*
2. Berikan gambaran umum mengenai sistem berkas!
3. Operasi apa saja yang dijalankan untuk melakukan operasi *copy*?
4. Sebutkan salah satu cara mengimplementasikan tipe berkas!
5. Sebutkan dan jelaskan tiga contoh struktur berkas!
6. Apa bedanya *sequential access* dan *direct access*?
7. Apa kelebihan struktur direktori *Acyclic Graph* dengan struktur direktori *Tree*?
8. Jelaskan yang dimaksud dengan *Garbage Collection Scheme*!

37.9. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts, 6th ed*. John Wiley & Sons.

Tananbaum, Andrew S. 1992. *Modern Operating System 2nd ed*. Engrewood cliffs, New Jersey: Prentice Hall Inc.

Stallings, Williem. 2000. *Operating System 4th ed*. Prentice Hall.

http://infocom.cqu.edu.au/Courses/aut2001/85349/Resources/ Study_Guide/10.pdf

http://www.cs.utah.edu/classes/cs5460/lectures/lecture19- 2up.pdf

http://support.sitescape.com/ forum/ support/ dispatch.cgi/ _help/ showHelp/ page/ help/ en/ webfiles_tabs/share_files.html

>http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690/ IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm

http://www.atnf.csiro.au/people/rgooch/linux/docs/vfs.txt

Bab 38. Struktur Direktori

Beberapa sistem komputer menyimpan banyak sekali berkas-berkas dalam disk, sehingga diperlukan suatu struktur pengorganisasian data-data agar lebih mudah diatur.

38.1. Operasi Direktori

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

- Mencari Berkas

  Mencari lewat struktur direktori untuk dapat menemukan entri untuk suatu berkas tertentu. berkas-berkas dengan nama yang simbolik dan mirip, mengindikasikan adanya keterkaitan diantara berkas-berkas tersebut. Oleh karena itu, tentunya perlu suatu cara untuk menemukan semua berkas yang benar-benar memenuhi kriteria khusus yang diminta.

- Membuat berkas

  berkas-berkas baru perlu untuk dibuat dan ditambahkan ke dalam direktori.

- Menghapus berkas

  Saat suatu berkas tidak diperlukan lagi, berkas tsb perlu dihapus dari direktori.

- Menampillkan isi direktori

  Menampilkan daftar berkas-berkas yang ada di direktori, dan semua isi direktori dari berkas-berkas dalam daftar tsb.

- Mengubah nama berkas

  Nama berkas mencerminkan isi berkas terhadap pengguna. Oleh karena itu, nama berkas harus dapat diubah-ubah ketika isi dan kegunaannya sudah berubah atau tidak sesuai lagi. Mengubah nama berkas memungkinkan posisinya berpindah dalam struktur direktori.

- Akses Sistem berkas

  Mengakses tiap direktori dan tiap berkas dalam struktur direktori. Sangatlah dianjurkan untuk menyimpan isi dan stuktur dari keseluruhan sistem berkas setiap jangka waktu tertentu. Menyimpan juga dapat berarti menyalin seluruh berkas ke pita magnetik. Teknik ini membuat suatu cadangan salinan dari berkas tersebut jika terjadi kegagalan sistem atau jika berkas itu tidak diperlukan lagi.

Sedangkan Tanenbaum juga menambahkan hal-hal berikut sebagai operasi yang dapat dilakukan terhadap direktori tersebut:

- Membuka direktori
- Menutup direktori
- Menambah direktori
- Mengubah nama direktori
- Menghubungkan berkas-berkas di direktori berbeda
- Menghapus hubungan berkas-berkas di direktori berbeda

38.2. Direktori Satu Tingkat

Direktori Satu Tingkat (*Single Level Directory*) ini merupakan struktur direktori yang paling sederhana. Semua berkas disimpan dalam direktori yang sama.

Gambar 38.1. Single Level Directory

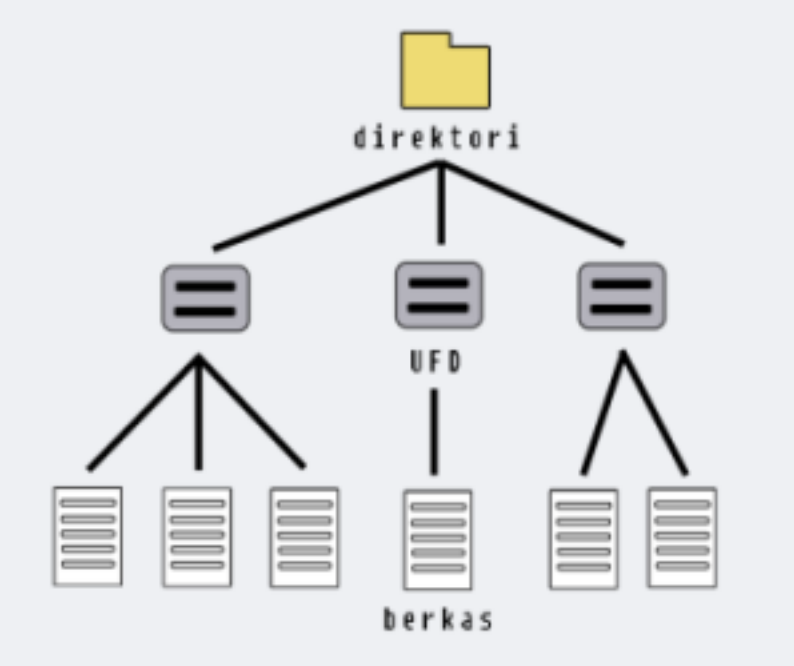
Direktori satu tingkat memiliki keterbatasan, yaitu bila berkas bertambah banyak atau bila sistem memiliki lebih dari satu pengguna. Hal ini disebabkan karena tiap berkas harus memiliki nama yang unik.

38.3. Direktori Dua Tingkat

Direktori Dua Tingkat (*Two Level Directory*) membuat direktori yang terpisah untuk tiap pengguna, yang disebut *User File Directory* (UFD). Ketika pengguna login, *master directory* berkas dipanggil. MFD memiliki indeks berdasarkan nama pengguna dan setiap entri menunjuk pada UFD pengguna tersebut. Maka, pengguna boleh memiliki nama berkas yang sama dengan berkas lain.

Meski pun begitu, struktur ini masih memiliki kerugian, terutama bila beberapa pengguna ingin mengerjakan tugas secara kerjasama dan ingin mengakses berkas dari salah satu pengguna lain. Beberapa sistem secara sederhana tidak mengizinkan berkas seorang pengguna diakses oleh pengguna lain.

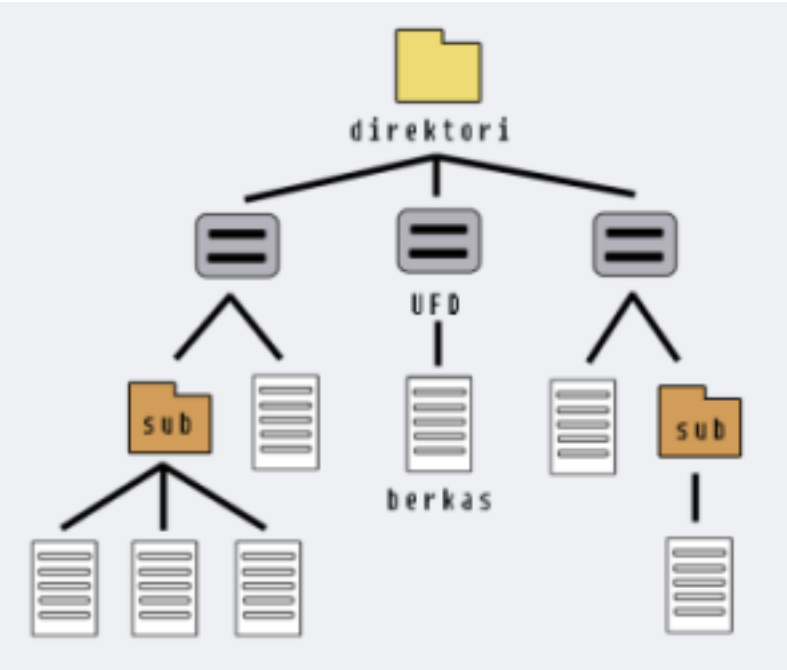
Gambar 38.2. Two Level Directory



38.4. Direktori dengan Struktur Tree

Pada direktori dengan Struktur Tree (*Tree-Structured Directory*), setiap pengguna dapat membuat subdirektori sendiri dan mengorganisasikan berkas-berkasnya. Dalam penggunaan normal, tiap pengguna memiliki apa yang disebut direktori saat ini. Direktori saat ini mengandung berkas-berkas yang baru-baru ini digunakan oleh pengguna.

Gambar 38.3. Tree-Structured Directory

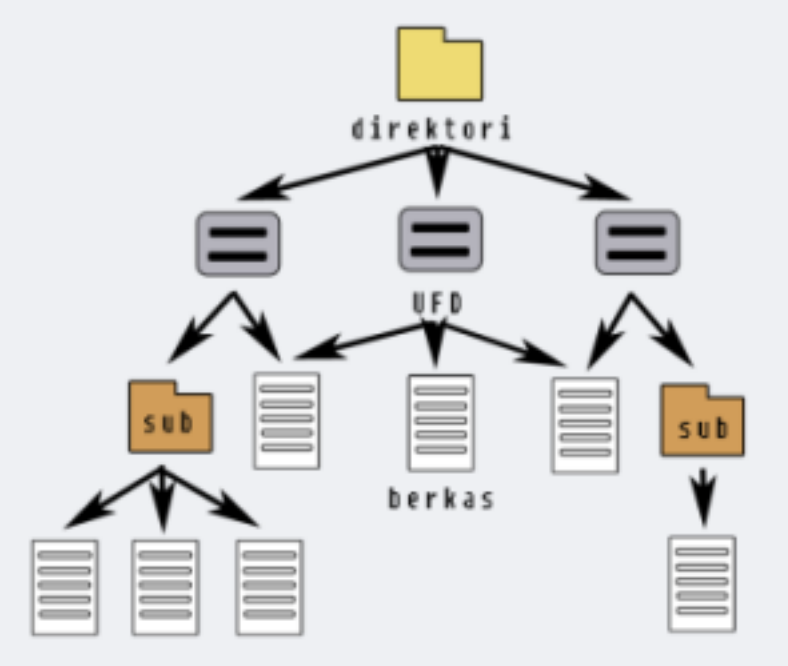


Terdapat dua istilah, *path* (lintasan) relatif dan lintasan mutlak. Lintasan relatif adalah lintasan yang dimulai dari direktori saat ini, sedangkan lintasan mutlak adalah path yang dimulai dari *root directory*.

38.5. Direktori dengan Struktur Graf Asiklik

Direktori dengan struktur *tree* melarang pembagian berkas/direktori. Oleh karena itu, struktur graf asiklik (*Acyclic-Structured Directory*) memperbolehkan direktori untuk berbagi berkas atau subdirektori. Jika ada berkas yang ingin diakses oleh dua pengguna atau lebih, maka struktur ini menyediakan fasilitas *sharing*.

Gambar 38.4. Acyclic-Structured Directory



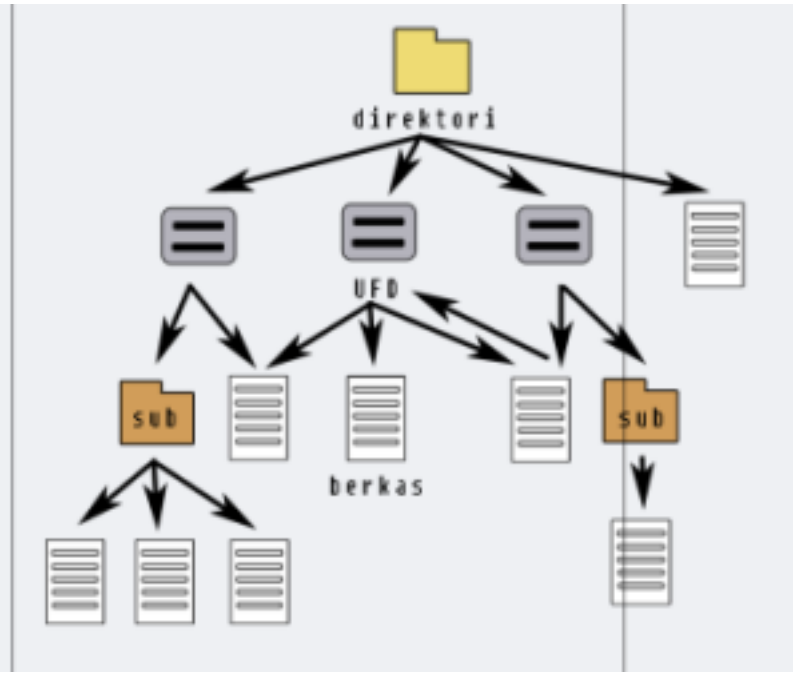
38.6. Direktori dengan Struktur Graf Umum

Masalah yang timbul dalam penggunaan struktur graf asiklik adalah meyakinkan apakah tidak ada siklus. Bila kita mulai dengan struktur direktori tingkat dua dan memperbolehkan pengguna untuk membuat subdirektori, maka kita akan mendapatkan struktur direktori *tree*. Sangatlah mudah untuk mempertahankan sifat pohon, akan tetapi, bila kita tambahkan sambungan pada direktori dengan struktur pohon, maka sifat pohon akan musnah dan menghasilkan struktur graf sederhana.

Bila siklus diperbolehkan dalam direktori, tentunya kita tidak ingin mencari sebuah berkas 2 kali. Algoritma yang tidak baik akan menghasilkan *infinite loop* dan tidak pernah berakhir. Oleh karena itu diperlukan skema pengumpulan sampah (*garbage-collection scheme*).

Skema ini menyangkut memeriksa seluruh sistem berkas dengan menandai tiap berkas yang dapat diakses. Kemudian mengumpulkan apa pun yang tidak ditandai pada tempat yang kosong. Hal ini tentunya dapat menghabiskan banyak waktu.

Gambar 38.5. General Graph Directory



38.7. Rangkuman

Beberapa sistem komputer menyimpan banyak sekali berkas-berkas dalam *disk*, sehingga diperlukan suatu struktur pengorganisasian data-data sehingga data lebih mudah diatur. Dalam struktur direktori satu tingkat, semua berkas diletakkan pada direktori yang sama, sehingga memiliki suatu keterbatasan karena nama berkas harus unik. Struktur direktori dua tingkat mencoba mengatasi masalah tersebut dengan membuat direktori yang terpisah untuk tiap pengguna yang disebut dengan *user file directory* (UFD). Sedangkan dalam struktur direktori *tree* setiap pengguna dapat membuat subdirektori sendiri dan mengorganisasikan berkas-berkasnya. Direktori dengan struktur *tree* melarang berbagi berkas atau direktori. Oleh karena itu, struktur dengan *acyclic-graph* memperbolehkan direktori untuk berbagi berkas atau sub-direktori. Struktur Direktori *general graph* mengatasi masalah yang timbul dalam struktur *acyclic* dengan metode *Garbage Collection*.

38.8. Latihan

1. Struktur Direktori apa saja yang menyediakan fasilitas *sharing*?
2. Apa yang terjadi jika kita mengubah nama suatu berkas?
3. Pada sistem UNIX, apa yg terjadi saat kita ingin menghapus suatu direktori yang masih mengandung suatu berkas?
4. Apakah yang dimaksud dengan pesan *error* berikut?

   pwd: Permission denied
5. Berikan sebuah contoh struktur direktori selain yang ada di buku dan jelaskan cara implementasi pada direktori!
6. Sistem Berkas I

   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.

7. Sebutkan pendekatan apa saja yang dipakai untuk mengatasi masalah proteksi berkas beserta keuntungan dan kerugiannya?

8. Apa perbedaan algoritma *Linear List* dan *Hash Table* untuk implentasi direktori? Menurut anda manakah yang lebih unggul? Jelaskan!

38.9. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts, 6th ed*. John Wiley & Sons.

Tananbaum, Andrew S. 1992. *Modern Operating System 2nd ed*. Engrewood cliffs, New Jersey: Prentice Hall Inc.

Stallings, Williem. 2000. *Operating System 4th ed*. Prentice Hall.

http://infocom.cqu.edu.au/Courses/aut2001/85349/Resources/ Study_Guide/10.pdf

http://www.cs.utah.edu/classes/cs5460/lectures/lecture19- 2up.pdf

http://support.sitescape.com/ forum/support/ dispatch.cgi/ _help/ showHelp/ page/ help/ en/ webfiles_tabs/ share_files.html

>http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690/ IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm

http://www.atnf.csiro.au/people/rgooch/linux/docs/vfs.txt

Bab 39. Sistem Berkas Jaringan

39.1. *Sharing*

Kita dapat berbagi berkas dengan pengguna lainnya yang teregistrasi. Hal pertama yang harus kita lakukan adalah menentukan dengan siapa berkas tersebut akan dibagi dan akses seperti apa yang akan diberikan kepada mereka. Berbagi bekas berguna bagi pengguna yang ingin bergabung dengan pengguna lain dan mengurangi usaha untuk mencapai sebuah hasil akhir.

Saat sebuah sistem operasi dibuat untuk *multiple user*, masalah berbagi berkas, penamaan berkas dan proteksi berkas menjadi sangat penting. Oleh karena itu, sistem operasi harus dapat mengakomodasikan/mengatur pembagian berkas dengan memberikan suatu struktur direktori yang membiarkan pengguna untuk saling berbagi.

Berkaitan dengan permasalahan akses berkas, kita dapat mengijinkan pengguna lain untuk melihat, mengedit atau menghapus suatu berkas. Proses mengedit berkas yang menggunakan *web-file system* berbeda dengan menggunakan aplikasi seperti Windows Explorer. Untuk mengedit sebuah file dengan *web-file system*, kita harus menduplikasi berkas tersebut dahulu dari *web-file system* ke komputer lokal, mengeditnya di komputer lokal, dan mengirim file tersebut kembali ke sistem dengan menggunakan nama berkas yang sama.

Sebagai contoh, kita dapat mengizinkan semua pengguna yang terdaftar untuk melihat berkas-berkas yang ada di direktori (tetapi mereka tidak dapat mengedit atau menghapus berkas tersebut). Contoh lainnya, kita dapat mengijinkan satu pengguna saja untuk melakukan apa pun terhadap sebuah direktori dan segala isinya (ijin untuk melihat semua berkas, mengeditnya, menambah berkas bahkan menghapus isi berkas). Kita juga dapat memberikan kesempatan bagi pengguna untuk mengubah izin dan kontrol akses dari sebuah isi direktori, namun hal tersebut biasanya di luar kebiasaan, sebab seharusnya satu-satunya pengguna yang berhak mengubah izin adalah kita sendiri.

Sistem berkas web memungkinkan kita untuk menspesifikasikan suatu akses dalam tingkatan berkas. Jadi, kita dapat mengijinkan seluruh orang untuk melihat isi dari sebuah direktori atau mengijinkan sebagian kecil pengguna saja untuk mengakses suatu direktori. Bahkan, dalam kenyataannya, kita dapat menspesifikasikan jenis akses yang berbeda dengan jumlah pengguna yang berbeda pula.

Kebanyakan pada sistem banyak pengguna menerapkan konsep direktor berkas *owner/user* dan group.

- *Owner*: pengguna yang dapat mengubah atribut, memberikan akses, dan memiliki sebagian besar kontrol di dalam sebuah berkas atau direktori.
- *Group*: sebagian pengguna yang sedang berbagi berkas.

39.2. *Remote File System*

Jaringan menyebabkan berbagi data terjadi di seluruh dunia. Dalam metode implementasi pertama, yang digunakan untuk berbagi data adalah program FTP (*File Transfer Protocol*). Yang kedua terbesar adalah DFS (*Disributed File System*) yang memungkinkan remote direktori terlihat dari mesin lokal. Metode yang ketiga adalah WWW (*World Wide Web*)

FTP digunakan untuk akses anonim (mentransfer file tanpa memiliki account di sistem remote) dan akses autentik (membutuhkan ijin). WWW biasanya menggunakan akses anonim, dan DFS menggunakan akses autentik.

39.3. *Client-Server Model*

- server: mesin yang berisi berkas
- klien: mesin yang mengakses berkas

Server dapat melayani banyak pengguna dan klien dapat menggunakan banyak server. Proses identifikasi klien biasanya sulit, dan cara yang biasa digunakan adalah melacak alamat IP, namun karena alamat IP dapat dipalsukan, cara ini menjadi kurang efektif. Ada juga yang menggunakan proses kunci terenkripsi, namun hal ini lebih rumit lagi, sebab klien-server harus menggunakan algoritma enkripsi yang sama dan pertukaran kunci yang aman.

39.4. Proteksi

Dalam pembahasan mengenai proteksi berkas, kita akan berbicara lebih mengenai sisi keamanan dan mekanisme bagaimana menjaga keutuhan suatu berkas dari gangguan akses luar yang tidak dikehendaki. Sebagai contoh bayangkan saja Anda berada di suatu kelompok kerja dimana masing-masing staf kerja disediakan komputer dan mereka saling terhubung membentuk suatu jaringan; sehingga setiap pekerjaan/dokumen/berkas dapat dibagi-bagikan ke semua pengguna dalam jaringan tersebut. Misalkan lagi Anda harus menyerahkan berkas RAHASIA.txt ke atasan Anda, dalam hal ini Anda harus menjamin bahwa isi berkas tersebut tidak boleh diketahui oleh staf kerja lain apalagi sampai dimodifikasi oleh orang yang tidak berwenang. Suatu mekanisme pengamanan berkas mutlak diperlukan dengan memberikan batasan akses ke setiap pengguna terhadap berkas tertentu.

39.5. Tipe Akses

Proteksi berkaitan dengan kemampuan akses langsung ke berkas tertentu. Panjangnya, apabila suatu sistem telah menentukan secara pasti akses berkas tersebut selalu ditutup atau selalu dibebaskan ke setiap pengguna lain maka sistem tersebut tidak memerlukan suatu mekanisme proteksi. Tetapi tampaknya pengimplementasian seperti ini terlalu ekstrim dan bukan pendekatan yang baik. Kita perlu membagi akses langsung ini menjadi beberapa jenis-jenis tertentu yang dapat kita atur dan ditentukan (akses yang terkontrol).

Dalam pendekatan ini, kita mendapatkan suatu mekanisme proteksi yang dilakukan dengan cara membatasi jenis akses ke suatu berkas. Beberapa jenis akses tersebut antara lain:

- Read/Baca: membaca berkas
- Write/Tulis: menulis berkas
- Execute/Eksekusi: memasukkan berkas ke memori dan dieksekusi
- Append/Sisip: menulis informasi baru pada baris akhir berkas
- Delete/Hapus: menghapus berkas
- List/Daftar: mendaftar nama dan atribut berkas

Operasi lain seperti *rename*, *copying*, atau *editing* yang mungkin terdapat di beberapa sistem merupakan gabungan dari beberapa jenis kontrol akses diatas. Sebagai contoh, menyalin sebuah berkas dikerjakan sebagai runtutan permintaan baca dari pengguna. Sehingga dalam hal ini, seorang pengguna yang memiliki kontrol akses *read* dapat pula meng-*copy*, mencetak dan sebagainya.

39.6. Kontrol Akses

Pendekatan yang paling umum dipakai dalam mengatasi masalah proteksi berkas adalah dengan membiarkan akses ke berkas ditentukan langsung oleh pengguna (dalam hal ini pemilik/pembuat berkas itu). Sang pemilik bebas menentukan jenis akses apa yang diperbolehkan untuk pengguna lain. Hal ini dapat dilakukan dengan menghubungkan setiap berkas atau direktori dengan suatu daftar kontrol-akses (*Access-Control Lists/ACL*) yang berisi nama pengguna dan jenis akses apa yang diberikan kepada pengguna tersebut.

Sebagai contoh dalam suatu sistem VMS, untuk melihat daftar direktori berikut daftar kontrol-akses, ketik perintah "DIR/SECURITY", atau "DIR/SEC". Salah satu keluaran perintah itu adalah daftar seperti berikut ini:

```
WWW-HOME.DIR;1       [HMC2000,WWART]              (RW,RWED,,E)
    (IDENTIFIER=WWW_SERVER_ACCESS,OPTIONS=DEFAULT,ACCESS=READ)
    (IDENTIFIER=WWW_SERVER_ACCESS,ACCESS=READ)
```

Baris pertama menunjukkan nama berkas tersebut WWW-HOME.DIR kemudian disebelahnya nama grup pemilik HMC2000 dan nama pengguna WWART diikuti dengan sekelompok jenis akses RW, RWED,,E (R=Baca, W=Tulis, E=Eksekusi, D=Hapus). Dua baris dibawahnya itulah yang disebut daftar konrol-akses. Satu-satu baris disebut sebagai masukan kontrol-akses (*Access Control Entry/ACE*) dan terdiri dari 3 bagian. Bagian pertama disebut sebagai IDENTIFIER/Identifikasi, menyatakan nama grup atau nama pengguna (seperti [HMC2000, WWART]) atau akses khusus (seperti WWW_SERVER_ACCESS). Bagian kedua merupakan daftar OPTIONS/Plihan-pilihan. Dan terakhir adalah daftar ijin ACCESS/akses, seperti *read* atau *execute*, yang diberikan kepada siapa saja yang mengacu pada bagian Identifikasi.

Cara kerjanya: apabila seorang pengguna meminta akses ke suatu berkas/direktori, sistem operasi akan memeriksa ke daftar kontrol-akses apakah nama pengguna itu tercantum dalam daftar tersebut. Apabila benar terdaftar, permintaan akses akan diberikan dan sebaliknya bila tidak, permintaan akses akan ditolak.

Pendekatan ini memiliki keuntungan karena penggunaan metodologi akses yang kompleks sehingga sulit ditembus sembarangan. Masalah utamanya adalah ukuran dari daftar akses tersebut. Bayangkan apabila kita mengijinkan semua orang boleh membaca berkas tersebut, kita harus mendaftar semua nama pengguna disertai ijin akses baca mereka. Lebih jauh lagi, tehnik ini memiliki dua konsekuensi yang tidak diinginkan:

- Pembuatan daftar semacam itu merupakan pekerjaan yang melelahkan dan tidak efektif.
- Entri direktori yang sebelumnya memiliki ukuran tetap, menjadi ukuran yang dapat berubah-ubah, mengakibatkan lebih rumitnya managemen ruang kosong.

Masalah ini dapat diselesaikan dengan penggunaan daftar akses yang telah disederhanakan.

Untuk menyederhanakan ukuran daftar kontrol akses, banyak sistem menggunakan tiga klasifikasi pengguna sebagai berikut:

- *Owner*: pengguna yang telah membuat berkas tersebut.
- *Group*: sekelompok pengguna yang saling berbagi berkas dan membutuhkan akses yang sama.
- *Universe*: keseluruhan pengguna.

Pendekatan yang dipakai belum lama ini adalah dengan mengkombinasikan daftar kontrol-akses

dengan konsep kontrol- akses pemilik, grup dan semesta yang telah dijabarkan diatas. Sebagai contoh, Solaris 2.6 dan versi berikutnya menggunakan tiga klasifikasi kontrol-akses sebagai pilihan umum, tetapi juga menambahkan secara khusus daftar kontrol-akses terhadap berkas/direktori tertentu sehingga semakin baik sistem proteksi berkasnya.

Contoh lain yaitu sistem UNIX dimana konrol-aksesnya dinyatakan dalam 3 bagian. Masing-masing bagian merupakan klasifikasi pengguna (yi.pemilik, grup dan semesta). Setiap bagian kemudian dibagi lagi menjadi 3 bit jenis akses -*rwx*, dimana *r* mengontrol akses baca, *w* mengontrol akses tulis dan *x* mengontrol eksekusi. Dalam pendekatan ini, 9 bit diperlukan untuk merekam seluruh informasi proteksi berkas.

Berikut adalah keluaran dari perintah "ls -al" di sistem UNIX:

```
-rwxr-x---  1  david karyawan 12210 Nov 14 20:12 laporan.txt
```

Baris di atas menyatakan bahwa berkas laporan.txt memiliki akses penuh terhadap pemilik berkas (yi.david), grupnya hanya dapat membaca dan mengeksekusi, sedang lainnya tidak memiliki akses sama sekali.

39.7. Pendekatan Pengamanan Lainnya

Salah satu pendekatan lain terhadap masalah proteksi adalah dengan memberikan sebuah kata kunci (*password*) ke setiap berkas. Jika kata-kata kunci tersebut dipilih secara acak dan sering diganti, pendekatan ini sangatlah efektif sebab membatasi akses ke suatu berkas hanya diperuntukkan bagi pengguina yang mengetahui kata kunci tersebut.

Meski pun demikian, pendekatan ini memiliki beberapa kekurangan, diantaranya:

- Kata kunci yang perlu diingat oleh pengguna akan semakin banyak, sehingga membuatnya menjadi tidak praktis.
- Jika hanya satu kata kunci yang digunakan di semua berkas, maka jika sekali kata kunci itu diketahui oleh orang lain, orang tersebut dapat dengan mudah mengakses semua berkas lainnya. Beberapa sistem (contoh: TOPS-20) memungkinkan seorang pengguna untuk memasukkaan sebuah kata kunci dengan suatu subdirektori untuk menghadapi masalah ini, bukan dengan satu berkas tertentu.
- Umumnya, hanya satu kata kunci yang diasosiasikan dengan semua berkas lain. Sehingga, pengamanan hanya menjadi semua-atau-tidak sama sekali. Untuk mendukung pengamanan pada tingkat yang lebih mendetail, kita harus menggunakan banyak kata kunci.

39.8. *Mounting*

Mounting adalah proses mengkaitkan sebuah sistem berkas yang baru ditemukan pada sebuah piranti ke struktur direktori utama yang sedang dipakai. Piranti-piranti yang akan di-*mount* dapat berupa *cd-rom*, disket atau sebuah *zip-drive*. Tiap-tiap sistem berkas yang akan di-*mount* akan diberikan sebuah *mount point*, atau sebuah direktori dalam pohon direktori sistem Anda, yang sedang diakses.

Sistem berkas yang dideskripsikan di /etc/fstab (fstab adalah singkatan dari *filesystem tables*) biasanya akan di-*mount* saat komputer baru mulai dinyalakan, tapi dapat juga me-*mount* sistem berkas tambahan dengan menggunakan perintah:

```
mount [nama piranti]
```

atau dapat juga dengan menambahkan secara manual *mount point* ke berkas /etc/fstab. Daftar sistem

berkas yang di-*mount* dapat dilihat kapan saja dengan menggunakan perintah *mount*. Karena izinnya hanya diatur *read-only* di berkas fstab, maka tidak perlu khawatir pengguna lain akan mencoba mengubah dan menulis *mount point* yang baru.

Seperti biasa saat ingin mengutak-atik berkas konfigurasi seperti mengubah isi berkas fstab, pastikan untuk membuat berkas cadangan untuk mencegah terjadinya kesalahan teknis yang dapat menyebabkan suatu kekacauan. Kita dapat melakukannya dengan cara menyediakan sebuah disket atau *recovery-disk* dan mem-*back-up* berkas fstab tersebut sebelum membukanya di editor teks untuk diutak-atik.

Red Hat Linux dan sistem operasi lainnya yang mirip dengan UNIX mengakses berkas dengan cara yang berbeda dari MS-DOS, Windows dan Macintosh. Di linux, segalanya disimpan di dalam sebuah lokasi yang dapat ditentukan dalam sebuah struktur data. Linux bahkan menyimpan perintah-perintah sebagai berkas. Seperti sistem operasi modern lainnya, Linux memiliki struktur *tree*, hirarki, dan organisasi direktori yang disebut sistem berkas.

Semua ruang kosong yang tersedia di *disk* diatur dalam sebuah pohon direktori tunggal. Dasar sistem ini adalah direktori *root* yang dinyatakan dengan sebuah garis miring ("/"). Pada linux, isi sebuah sistem berkas dibuat nyata tersedia dengan menggabungkan sistem berkas ke dalam sebuah sistem direktori melalui sebuah proses yang disebut *mounting*.

Sistem berkas dapat di-*mount* maupun di-*umount* yang berarti sistem berkas tersebut dapat tersambung atau tidak dengan struktur pohon direktori. Perbedaannya adalah sistem berkas tersebut akan selalu di-*mount* ke direktori *root* ketika sistem sedang berjalan dan tidak dapat di-*mount*. Sistem berkas yang lain di-*mount* seperlunya, contohnya yang berisi *hard drive* berbeda dengan *floppy disk* atau CD-ROM.

39.9. *Mounting Overview*

Mounting membuat sistem berkas, direktori, piranti dan berkas lainnya menjadi dapat digunakan di lokasi-lokasi tertentu, sehingga memungkinkan direktori itu menjadi dapat diakses. Perintah *mount* menginstruksikan sistem operasi untuk mengkaitkan sebuah sistem berkas ke sebuah direktori khusus.

Memahami *Mount Point*

Mount point adalah sebuah direktori dimana berkas baru menjadi dapat diakses. Untuk me-*mount* suatu sistem berkas atau direktori, titik *mount*-nya harus berupa direktori, dan untuk me-*mount* sebuah berkas, *mount point*-nya juga harus berupa sebuah berkas.

Biasanya, sebuah sistem berkas, direktori, atau sebuah berkas di-*mount* ke sebuah *mount point* yang kosong, tapi biasanya hal tersebut tidak diperlukan. Jika sebuah berkas atau direktori yang akan menjadi *mount point* berisi data, data tersebut tidak akan dapat diakses selama direktori/berkas tersebut sedang dijadikan *mount point* oleh berkas atau direktori lain. Sebagai akibatnya, berkas yang di-*mount* akan menimpa apa yang sebelumnya ada di direktori/berkas tersebut. Data asli dari direktori itu dapat diakses kembali bila proses *mounting* sudah selesai.

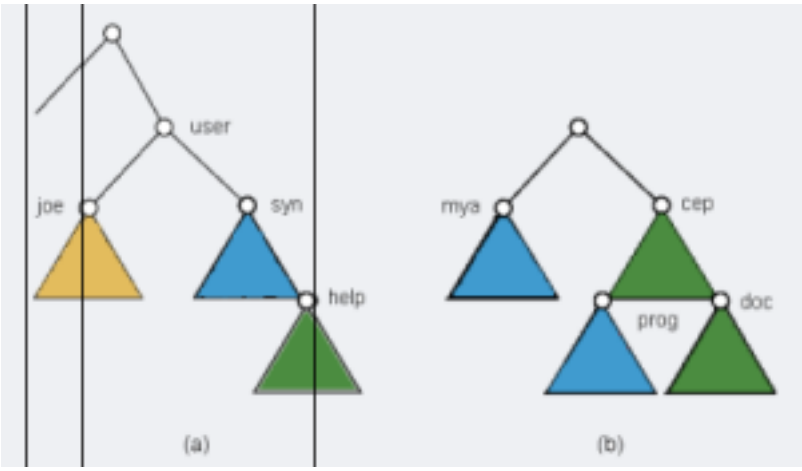
Saat sebuah sistem berkas di-*mount* ke sebuah direktori, izin direktori *root* dari berkas yang di-*mount* akan mengambil alih izin dari *mount point*. Pengecualiannya adalah pada direktori induk akan memiliki atribut .. (*double dot*). Agar sistem operasi dapat mengakses sistem berkas yang baru, direktori induk dari *mount point* harus tersedia.

Untuk segala perintah yang membutuhkan informasi direktori induk, pengguna harus mengubah izin dari direktori *mounted-over*. Kegagalan direktori *mounted-over* untuk mengabulkan izin dapat menyebabkan hasil yang tidak terduga, terutama karena izin dari direktori *mounted-over* tidak dapat terlihat. Kegagalan umum terjadi pada perintah pwd. Tanpa mengubah izin direktori *mounted-over*, akan timbul pesan error seperti ini:

```
pwd: permission denied
```

Masalah ini dapat diatasi dengan mengatur agar izin setidaknya di-set dengan 111.

Gambar 39.1. Mount Point



Mounting Sistem Berkas, Direktori, dan Berkas

Ada dua jenis *mounting*: *remote mounting* dan *mounting* lokal. *Remote mounting* dilakukan dengan sistem *remote* dimana data dikirimkan melalui jalur telekomunikasi. *Remote* sistem berkas seperti Network File Systems (NFS), mengharuskan agar file diekspor dulu sebelum di-*mount*. *mounting* lokal dilakukan di sistem lokal.

Tiap-tiap sistem berkas berhubungan dengan piranti yang berbeda. Sebelum kita menggunakan sebuah sistem berkas, sistem berkas tersebut harus dihubungkan dengan struktur direktori yang ada (dapat *root* atau berkas yang lain yang sudah tersambung).

Sebagai contoh, kita dapat me-*mount* dari /home/server/database ke *mount point* yang dispesifikasikan sebagai /home/user1, /home/user2, and /home/user3:

- /home/server/database /home/user1
- /home/server/database /home/user2
- /home/server/database /home/user3

39.10. Rangkuman

Mounting adalah proses mengaitkan sebuah sistem berkas yang baru ditemukan pada sebuah piranti ke struktur direktori utama yang sedang dipakai. *Mount point* adalah sebuah direktori dimana berkas baru menjadi dapat diakses.

39.11. Latihan

1. Apa perbedaan antara metode implementasi *sharing* antara FTP, DFS dan WWW?
2. Kemanakah sistem berkas akan selalu di-*mount* ketika sistem sedang berjalan?

39.12. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts, 6th ed*. John Wiley & Sons.

Tananbaum, Andrew S. 1992. *Modern Operating System 2nd ed*. Engrewood cliffs, New Jersey: Prentice Hall Inc.

Stallings, Williem. 2000. *Operating System 4th ed*. Prentice Hall.

http://infocom.cqu.edu.au/Courses/aut2001/85349/Resources/ Study_Guide/10.pdf

http://www.cs.utah.edu/classes/cs5460/lectures/lecture19- 2up.pdf

http://support.sitescape.com/ forum/ support/ dispatch.cgi/ _help/showHelp/ page/ help/ en/ webfiles_tabs/ share_files.html

>http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690/ IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm

http://www.atnf.csiro.au/people/rgooch/linux/docs/vfs.txt

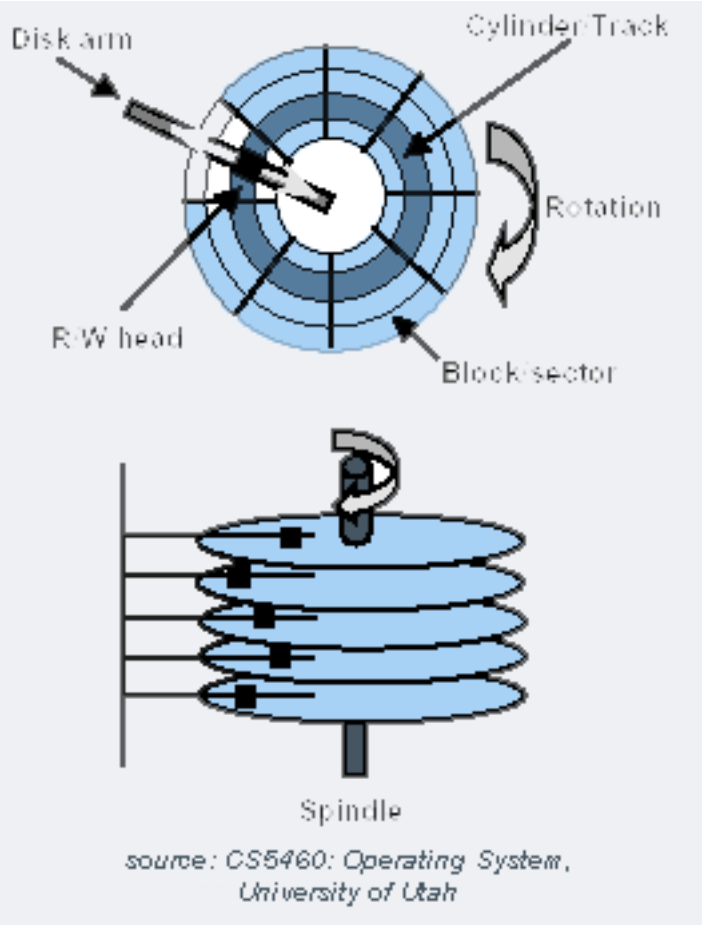
Bab 40. Implementasi Sistem Berkas

40.1. Struktur Sistem Berkas

Disk yang merupakan tempat terdapatnya sistem berkas menyediakan sebagian besar tempat penyimpanan dimana sistem berkas akan dikelola. *Disk* memiliki dua karakteristik penting yang menjadikan *disk* sebagai media yang tepat untuk menyimpan berbagai macam berkas, yaitu:

- Data dapat ditulis ulang di *disk* tersebut, hal ini memungkinkan untuk membaca, memodifikasi, dan menulis di *disk* tersebut.
- Dapat diakses langsung ke setiap blok di *disk*. Hal ini memudahkan untuk mengakses setiap berkas baik secara berurut maupun tidak berurut, dan berpindah dari satu berkas ke berkas lain dengan hanya mengangkat *head disk* dan menunggu *disk* berputar.

Gambar 40.1. *Disk Organization*



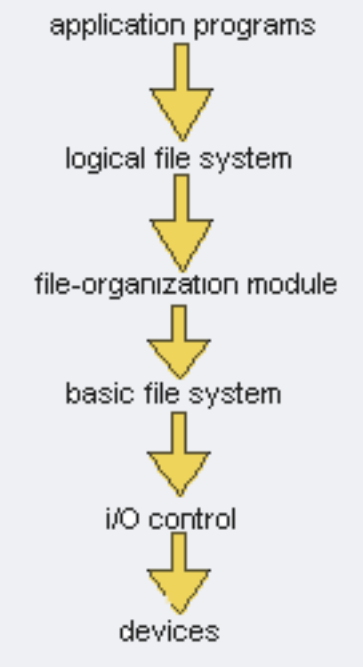
Untuk meningkatkan efisiensi *I/O*, pengiriman data antara memori dan *disk* dilakukan dalam setiap blok. Setiap *blok* merupakan satu atau lebih sektor. Setiap *disk* memiliki ukuran yang berbeda-beda, biasanya berukuran 512 *bytes*.

Sistem operasi menyediakan sistem berkas agar data mudah disimpan, diletakkan dan diambil kembali dengan mudah. Terdapat dua masalah desain dalam membangun suatu sistem berkas. Masalah pertama adalah definisi dari sistem berkas. Hal ini mencakup definisi berkas dan

atributnya, operasi ke berkas, dan struktur direktori dalam mengorganisasikan berkas-berkas. Masalah kedua adalah membuat algoritma dan struktur data yang memetakan struktur logikal sistem berkas ke tempat penyimpanan sekunder.

Sistem berkas dari sistem operasi yang sudah *modern* diimplementasikan dengan menggunakan struktur berlapis. Keuntungan struktur berlapis ini adalah fleksibilitas yang dimilikinya. Penggunaan dari struktur berlapis ini memungkinkan adanya implementasi yang lebih dari satu secara bersamaan, terutama pada *I/O Control* dan tingkatan organisasi berkas. Hal ini memungkinkan untuk mendukung lebih dari satu implementasi sistem berkas.

Gambar 40.2. *Layered File System*



Lapisan struktur sistem berkas menghubungkan antara perangkat keras dengan aplikasi program yang ada, yaitu (dari yang terendah):

- *I/O control*, terdiri atas *driver device* dan *interrupt handler*. *Driver device* adalah perantara komunikasi antara sistem operasi dengan perangkat keras. *Input* didalamnya berisikan perintah tingkat tinggi seperti "ambil blok 133", sedangkan *output*-nya adalah perintah tingkat rendah, instruksi spesifik perangkat keras yang digunakan oleh *controller* perangkat keras.
- *Basic file system*, diperlukan untuk mengeluarkan perintah *generic* ke *device driver* untuk *read* dan *write* pada suatu blok dalam *disk*.
- *File-organization module*, informasi tentang alamat logika dan alamat fisik dari berkas tersebut. Modul ini juga mengatur sisa *disk* dengan melacak alamat yang belum dialokasikan dan menyediakan alamat tersebut saat *pengguna* ingin menulis berkas ke dalam *disk*. Di dalam *File-organization module* juga terdapat *free- space manager*.
- *Logical file-system*, tingkat ini berisi informasi tentang simbol nama berkas, struktur dari direktori, dan proteksi dan sekuriti dari berkas tersebut. Sebuah *File Control Block* (FCB) menyimpan informasi tentang berkas, termasuk kepemilikan, izin dan lokasi isi berkas.

Di bawah ini merupakan contoh dari kerja struktur berlapis ini ketika suatu program mau membaca informasi dari *disk*. Urutan langkahnya:

1. *Application program* memanggil sistem berkas dengan *system call*.

   Contoh: *read (fd, input, 1024)* akan membaca *section* sebesar 1 Kb dari *disk* dan menempatkannya ke variabel *input*.

2. Diteruskan ke *system call interface*.

   System call merupakan *software interrupt*. Jadi, *interrupt handler* sistem operasi akan memeriksa apakah *system call* yang menginterupsi. *Interrupt handler* ini akan memutuskan bagian dari sistem operasi yang bertanggung-jawab untuk menangani *system call*. *Interrupt handler* akan meneruskan *system call*.

3. Diteruskan ke *logical file system*.

   Memasuki lapisan sistem berkas. Lapisan ini menyediakan *system call*, operasi yang akan dilakukan dan jenis berkas. Yang perlu ditentukan selanjutnya adalah *file organization module* yang akan meneruskan permintaan ini. *File organization module* yang akan digunakan tergantung dari jenis sistem berkas dari berkas yang diminta.

   Contoh: Misalkan kita menggunakan LINUX dan berkas yang diminta ada di Windows 95. Lapisan *logical file system* akan meneruskan permintaan ke *file organization module* dari Windows 95.

4. Diteruskan ke *file organization module*.

   File organization module yang mengetahui pengaturan (organisasi) direktori dan berkas pada *disk*. Sistem berkas yang berbeda memiliki organisasi yang berbeda. Windows 95 menggunakan VFAT-32. Windows NT menggunakan format NTFS. Linux menggunakan EXT2. Sistem operasi yang paling *modern* memiliki beberapa *file organization module* sehingga dapat membaca format yang berbeda.

   Pada contoh di atas, *logical file system* telah meneruskan permintaan ke *file organization module* VFAT32. Modul ini menterjemahkan nama berkas yang ingin dibaca ke lokasi fisik yang biasanya terdiri dari *disk* antarmuka, *disk drive*, *surface*, *cylinder*, *track*, *sector*.

5. Diteruskan ke *basic file system*.

   Dengan adanya lokasi fisik, kita dapat memberikan perintah ke piranti keras yang dibutuhkan. Hal ini merupakan tanggung-jawab *basic file system*. *Basic file system* ini juga memiliki kemampuan tambahan seperti *buffering* dan *caching*.

   Contoh: Sektor tertentu yang dipakai untuk memenuhi permintaan mungkin saja berada dalam *buffers* atau *caches* yang diatur oleh *basic file system*. Jika terjadi hal seperti ini, maka informasi akan didapatkan secara otomatis tanpa perlu membaca lagi dari *disk*.

6. *I/O Control*

   Tingkatan yang paling rendah ini yang memiliki cara untuk memerintah/memberitahu piranti keras yang diperlukan.

40.2. Implementasi Sistem Berkas

Untuk mengimplementasikan suatu sistem berkas biasanya digunakan beberapa struktur *on-disk* dan *in-memory*. Struktur ini bervariasi tergantung pada sistem operasi dan sistem berkas, tetapi beberapa prinsip dasar harus tetap diterapkan. Pada struktur *on-disk*, sistem berkas mengandung informasi tentang bagaimana mem-*boot* sistem operasi yang disimpan, jumlah blok, jumlah dan lokasi blok yang masih kosong, struktur direktori, dan berkas individu.

Struktur *on-disk*:

- *Boot Control Block*

Informasi yang digunakan untuk menjalankan mesin mulai dari partisi yang diinginkan untuk menjalankan mesin mulai dari partisi yang diinginkan. Dalam UPS disebut *boot block*. Dalam NTFS disebut *partition boot sector*.

- *Partition Block Control*

  Spesifikasi atau detil-detil dari partisi (jumlah blok dalam partisi, ukuran blok, ukuran blok, dsb). Dalam UPS disebut *superblock*. Dalam NTFS disebut tabel *master file*.

- Struktur direktori

  Mengatur berkas-berkas.

- File Control Block (FCB)

  Detil-detil berkas yang spesifik. Di UPS disebut *inode*. Di NTFS, informasi ini disimpan di dalam tabel *Master File*.

Struktur in-memory:

- Tabel Partisi *in-memory*

  Informasi tentang partisi yang di-*mount*.

- Struktur Direktori *in-memory*

  Menyimpan informasi direktori tentang direktori yang paling sering diakses.

- Tabel *system-wide open-file*

  - menyimpan *open count* (informasi jumlah proses yang membuka berkas tsb)
  - menyimpan atribut berkas (pemilik, proteksi, waktu akses, dsb), dan lokasi *file blocks*.
  - Tabel ini digunakan bersama-sama oleh seluruh proses.

- Tabel *per-process open-file*

  - menyimpan *pointer* ke entri yang benar dalam tabel *open- file*
  - menyimpan posisi pointer pada saat itu dalam berkas.
  - modus akses

Untuk membuat suatu berkas baru, program aplikasi memanggil *logical file system*. *Logical file system* mengetahui format dari struktur direktori. Untuk membuat berkas baru, *logical file system* akan mengalokasikan FCB, membaca direktori yang benar ke memori, memperbaharui dengan nama berkas dan FCB yang baru dan menulisnya kembali ke dalam *disk*.

Beberapa sistem operasi, termasuk UNIX, memperlakukan berkas sebagai direktori. Sistem operasi Windows NT mengimplementasi beberapa *system calls* untuk berkas dan direktori. Windows NT memperlakukan direktori sebagai sebuah kesatuan yang berbeda dengan berkas. *Logical file system* dapat memanggil *file-organization module* untuk memetakan direktori *I/O* ke *disk-block numbers*, yang dikirimkan ke sistem berkas dasar dan *I/O control system*. *File- organization module* juga mengalokasikan blok untuk penyimpanan data-data berkas.

Setelah berkas selesai dibuat, mula-mula harus dibuka terlebih dahulu. Perintah *open* dikirim nama berkas ke sistem berkas. Ketika sebuah berkas dibuka, struktur direktori mencari nama berkas yang diinginkan. Ketika berkas ditemukan, FCD disalin ke ke tabel *system-wide open-file* pada memori. Tabel ini juga mempunyai entri untuk jumlah proses yang membuka berkas tersebut.

Selanjutnya, entri dibuat di tabel *per-process open-file* dengan penunjuk ke entri di dalam tabel *system-wide open-file*. Seluruh operasi pada berkas akan diarahkan melalui penunjuk ini.

40.3. Partisi dan *Mounting*

Setiap partisi dapat merupakan *raw* atau *cooked*. *Raw* adalah partisi yang tidak memiliki sistem berkas dan cooked sebaliknya. *Raw disk* digunakan jika tidak ada sistem berkas yang tepat. *Raw disk* juga dapat menyimpan informasi yang dibutuhkan oleh sistem *disk* RAID dan *database* kecil yang menyimpan informasi konfigurasi RAID.

Informasi *boot* dapat disimpan di partisi yang berbeda. Semuanya mempunyai formatnya masing-masing karena pada saat *boot*, sistem tidak punya sistem berkas dari perangkat keras dan tidak dapat memahami sistem berkas.

Root partition yang mengandung *kernel* sistem operasi dan sistem berkas yang lain, di-*mount* saat *boot*. Partisi yang lain di-*mount* secara otomatis atau manual (tergantung sistem operasi). Sistem operasi menyimpan dalam struktur tabel *mount* dimana sistem berkas di-mount dan jenis dari sistem berkas.

Pada UNIX, sistem berkas dapat di-*mount* di direktori mana pun. Ini diimplementasikan dengan mengatur *flag* di salinan *in-memory* dari jenis direktori itu. *Flag* itu mengindikasikan bahwa direktori adalah puncak *mount*.

40.4. Sistem Berkas Virtual

Suatu direktori biasanya menyimpan beberapa berkas dengan jenis-jenis yang berbeda. Sistem operasi harus dapat menyatukan berkas-berkas berbeda itu di dalam suatu struktur direktori. Untuk menyatukan berkas-berkas tersebut digunakan metode implementasi beberapa jenis sistem berkas dengan menulis di direktori dan *file routine* untuk setiap jenis.

Sistem operasi pada umumnya, termasuk UNIX, menggunakan teknik berorientasi objek untuk menyederhakan, mengorganisir dan mengelompokkannya sesuai dengan implementasinya. Penggunaan metode ini memungkinkan berkas-berkas yang berbeda jenisnya diimplementasikan dalam struktur yang sama.

Implementasi spesifiknya menggunakan struktur data dan prosedur untuk mengisolasi fungsi dasar dari *system call*.

Implementasi sistem berkas terdiri dari 3 lapisan utama:

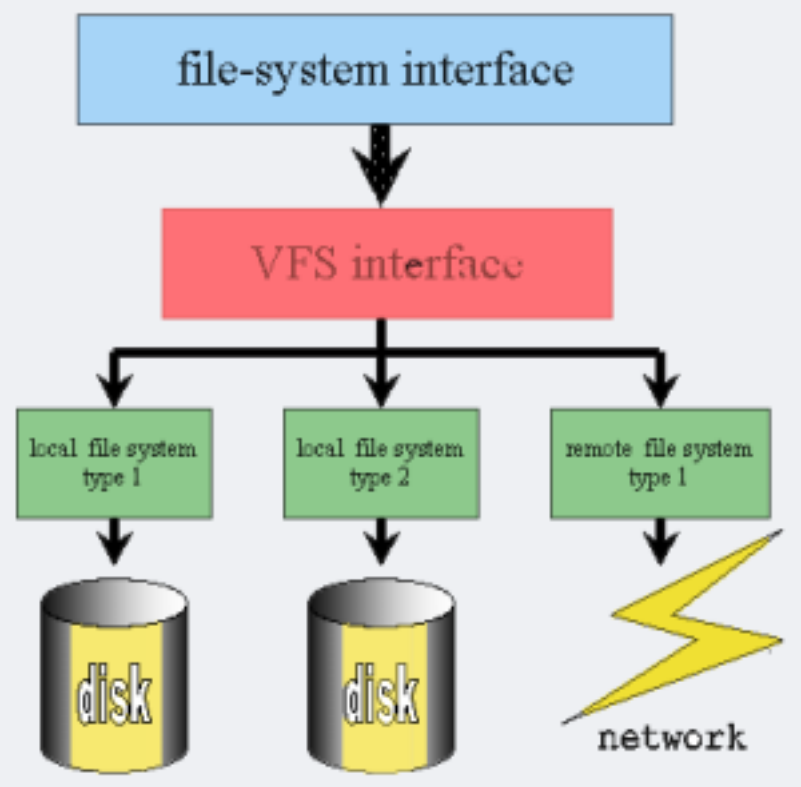
- *Interface* sistem berkas: perintah *open, read, write, close* dan *file descriptor*.
- *Virtual File System*(VFS)

  Virtual file system adalah suatu lapisan perangkat lunak dalam kernel yang menyediakan antar muka sistem berkas untuk program *userspace*. VFS juga menyediakan suatu abstraksi dalam kernel yang mengijinkan implementasi sistem berkas yang berbeda untuk muncul.

  VFS ini memiliki 2 fungsi yang penting yaitu:

  - Memisahkan operasi berkas *generic* dari implementasinya dengan mendefinisikan VFS antar muka yang masih baru.
  - VFS didasarkan pada struktur *file-representation* yang dinamakan *vnode*, yang terdiri dari *designator* numerik untuk berkas unik *network-wide*.
- Sistem berkas lokal dan sistem berkas *remote* untuk jaringan.

Gambar 40.3. ***Schematic View of Virtual File System***



40.5. Implementasi Direktori

Sebelum sebuah berkas dapat dibaca, berkas tersebut harus dibuka terlebih dahulu. Saat berkas tersebut dibuka, sistem operasi menggunakan *path name* yang dimasukkan oleh pengguna untuk mengalokasikan direktori entri yang menyediakan informasi yang dibutuhkan untuk menemukan *block disk* tempat berkas itu berada. Tergantung dari sistem tersebut, informasi ini dapat berupa alamat *disk* dari berkas yang bersangkutan (*contiguous allocation*), nomor dari blok yang pertama (kedua skema *linked list*), atau nomor dari *inode*. Dalam semua kasus, fungsi utama dari direktori entri adalah untuk memetakan nama ASCII dari berkas yang bersangkutan kepada informasi yang dibutuhkan untuk mengalokasikan data.

Masalah berikutnya yang kemudian muncul adalah dimana atribut yang dimaksud akan disimpan. Kemungkinan paling nyata adalah menyimpan secara langsung di dalam direktori entri, dimana kebanyakan sistem menggunakannya. Untuk sistem yang menggunakan *inodes*, kemungkinan lain adalah menyimpan atribut ke dalam *inode*, selain dari direktori entri. Cara yang terakhir ini mempunyai keuntungan lebih dibandingkan menyimpan dalam direktori entri.

Cara pengalokasian direktori dan pengaturan direktori dapat meningkatkan efisiensi, performa dan kehandalan. Ada beberapa macam algoritma yang dapat digunakan.

40.6. Algoritma *Linear List*

Metode paling sederhana. Menggunakan nama berkas dengan penunjuk ke data blok.

Proses:

- Mencari (tidak ada nama berkas yang sama).

- Menambah berkas baru pada akhir direktori.
- Menghapus (mencari berkas dalam direktori dan melepaskan tempat yang dialokasikan).

Penggunaan suatu berkas:

Memberi tanda atau menambahkan pada daftar direktori bebas.

Kelemahan:

Pencarian secara *linier (linier search)* untuk mencari sebuah berkas, sehingga implementasi sangat lambat saat mengakses dan mengeksekusi berkas.

Solusi:

Linked list dan *Software Cache*

40.7. Algoritma *Hash Table*

Linear List menyimpan direktori entri, tetapi sruktur data *hash* juga digunakan.

Proses:

Hash table mengambil nilai yang dihitung dari nama berkas dan mengembalikan sebuah penunjuk ke nama berkas yang ada di *linier list*.

Kelemahan:

- Ukuran tetap:
- Adanya ketergantungan fungsi *hash* dengan ukuran *hash table*

Alternatif:

Chained-overflow hash table yaitu setiap *hash table* mempunyai *linked list* dari nilai individual dan *crash* dapat diatasi dengan menambah tempat pada *linked list* tersebut. Namun penambahan ini dapat memperlambat.

40.8. Direktori pada CP/M

Direktori pada CP/M merupakan direktori entri yang mencakup nomor *block disk* untuk setiap berkas. Contoh direktori ini (Golden dan Pechura, 1986), berupa satu direktori saja. Jadi, Semua sistem berkas harus melihat nama berkas dan mencari dalam direktori satu-satunya ini.

Direktori ini terdiri dari 3 bagian yaitu:

- *User Code*

  Merupakan bagian yang menetapkan *track* dari *user* mana yang mempunyai berkas yang bersangkutan, saat melakukan pencarian, hanya entri tersebut yang menuju kepada *logged-in user* yang bersangkutan. Dua bagian berikutnya terdiri dari nama berkas dan ekstensi dari berkas.

- *Extent*

  Bagian ini diperlukan oleh berkas karena berkas yang berukuran lebih dari 16 blok menempati direktori entri yang banyak. Bagian ini digunakan untuk memberitahukan entri mana yang datang pertama, kedua, dan seterusnya.

- *Block Count*

  Bagian ini memberitahukan seberapa banyak dari ke-enambelas *block disk* potensial, sedang digunakan. Enambelas bagian akhir berisi nomor *block disk* yang bersangkutan. Bagian blok yang terakhir dapat saja penuh, jadi sistem tidak dapat menentukan kapasitas pasti dari berkas sampai ke *byte* yang terakhir.

  Saat CP/M menemukan entri, CP/M juga memakai nomor *block disk*, saat berkas disimpan dalam direktori entri, dan juga semua atributnya. Jika berkas menggunakan *block disk* lebih dari satu entri, berkas dialokasikan dalam direktori yang ditambahkan.

40.9. Direktori pada MS-DOS

Merupakan sistem dengan *tree hierarchy directory*. Mempunyai panjang 32 *bytes*, yang mencakup nama berkas, atribut, dan nomor dari *block disk* yang pertama. Nomor dari *block disk* yang pertama digunakan sebagai indeks dari tabel MS-DOS direktori entri. Dengan sistem rantai, semua blok dapat ditemukan.

Dalam MS-DOS, direktori dapat berisi direktori lain, tergantung dari hirarki sistem berkas. Dalam MS-DOS, program aplikasi yang berbeda dapat dimulai oleh setiap program dengan membuat direktori dalam direktori *root*, dan menempatkan semua berkas yang bersangkutan di dalam sana. Jadi antar aplikasi yang berbeda tidak dapat terjadi konflik.

40.10. Direktori pada UNIX

Struktur direktori yang digunakan dalam UNIX adalah struktur direktori tradisional. Seperti yang terdapat dalam gambar direktori entri dalam UNIX, setiap entri berisi nama berkas dan nomor *inode* yang bersangkutan. Semua informasi dari jenis, kapasitas, waktu dan kepemilikan, serta *block disk* yang berisi *inode*. Sistem UNIX terkadang mempunyai penampakan yang berbeda,tetapi pada beberapa kasus, direktori entri biasanya hanya *string* ASCII dan nomor *inode*.

Gambar 40.4. *A UNIX directory entry*

Saat berkas dibuka, sistem berkas harus mengambil nama berkas dan mengalokasikan *block disk* yang bersangkutan, sebagai contoh, nama *path* /usr/ast/mbox dicari, dan kita menggunakan UNIX sebagai contoh, tetapi algoritma yang digunakan secara dasar sama dengan semua hirarki sistem direktori sistem.

Pertama, sistem berkas mengalokasikan direktori *root*. Dalam UNIX *inode* yang bersangkutan ditempatkan dalam tempat yang sudah tertentu dalam *disk*. Kemudian, UNIX melihat komponen pertama dari *path*, usr dalam direktori root menemukan nomor *inode* dari direktori /usr. Mengalokasikan sebuah nomor *inode* adalah secara *straight-forward*, sejak setiap *inode* mempunyai lokasi yang tetap dalam *disk*. Dari *inode* ini, sistem mengalokasikan direktori untuk /usr dan melihat komponen berikutnya, dst. Saat dia menemukan entri untuk ast, dia sudah mempunyai *inode* untuk direktori /ust/ast. Dari *inode* ini, dia dapat menemukan direktorinya dan melihat mbox. *Inode* untuk berkas ini kemudian dibaca ke dalam memori dan disimpan disana sampai berkas tersebut ditutup.

Nama *path* dilihat dengan cara yang relatif sama dengan yang absolut. Dimulai dari direktori yang bekerja sebagai pengganti *root directory*. Setiap direktori mempunyai entri untuk. dan ".." yang dimasukkan ke dalam saat direktori dibuat. Entri "." mempunyai nomor *inode* yang menunjuk ke direktori di atasnya/orangtua (*parent*), "." kemudian melihat ../dick/prog.c hanya melihat tanda ".." dalam direktori yang bekerja, dengan menemukan nomor *inode* dalam direktori di atasnya/parent

dan mencari direktori *disk*. Tidak ada mekanisme spesial yang dibutukan untuk mengatasi masalah nama ini. Sejauh masih di dalam sistem direktori, mereka hanya merupakan ASCII string yang biasa.

40.11. Rangkuman

Sebagai implementasi direktori yang merupakan implementasi dari Implementasi Sistem Berkas, implementasi direktori memiliki algoritma seperti *Linear List dan Hashtable*. Direktori pada MS/Dos merupakan sistem dengan direktori hirarki tree. Direktori pada UNIX merupakan struktur direktori tradisional.

Sebagai implementasi direktori yang merupakan implementasi dari Implementasi Sistem Berkas, implementasi direktori memiliki algoritma seperti *Linear List dan Hashtable*. Direktori pada MS/Dos merupakan sistem dengan direktori hirarki tree. Direktori pada UNIX merupakan struktur direktori tradisional.

40.12. Latihan

1. Sebutkan dua sistem operasi yang implementasi sistem berkasnya menggunakan *Layered File System*!
2. Jelaskan cara membuat suatu berkas baru!
3. Apakah hubungan partisi dan *mounting* dengan implementasi sistem berkas?

40.13. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts, 6th ed*. John Wiley & Sons.

Tananbaum, Andrew S. 1992. *Modern Operating System 2nd ed*. Engrewood cliffs, New Jersey: Prentice Hall Inc.

Stallings, Williem. 2000. *Operating System 4th ed*. Prentice Hall.

http://infocom.cqu.edu.au/Courses/aut2001/85349/Resources/ Study_Guide/10.pdf

http://www.cs.utah.edu/classes/cs5460/lectures/lecture19- 2up.pdf

http://support.sitescape.com/ forum/ support/ dispatch.cgi/ _help/ showHelp/ page/ help/ en/ webfiles_tabs/ share_files.html

>http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690/ IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm

http://www.atnf.csiro.au/people/rgooch/linux/docs/vfs.txt

Bab 41. *Filesystem Hierarchy Standard*

41.1. Pendahuluan

Filesystem Hierarchy Standard (FHS) adalah standar yang digunakan oleh perangkat lunak dan pengguna untuk mengetahui lokasi dari berkas atau direktori yang berada pada komputer. Hal ini dilakukan dengan cara menetapkan prinsip-prinsip dasar pada setiap daerah pada sistem berkas, menetapkan berkas dan direktori minimum yang dibutuhkan, mengatur banyaknya pengecualian dan mengatur kasus yang sebelumnya pernah mengalami konflik secara spesifik.

Dokumen FHS ini digunakan oleh pembuat perangkat lunak untuk menciptakan suatu aplikasi yang *compliant* dengan FHS. Selain itu, dokumen ini juga digunakan oleh para pembuat sistem operasi untuk menyediakan sistem yang *compliant* dengan FHS.

Komponen dari nama berkas yang dapat berubah-ubah, akan diapit oleh tanda < dan >, sedangkan komponen yang bersifat pilihan, akan diapit oleh tanda "[" dan "]" dan dapat dikombinasi dengan '<' dan '>'. Sebagai contoh, jika nama berkas diperbolehkan untuk menggunakan atau tidak menggunakan ekstensi, akan ditulis sebagai <nama berkas>[.<ekstensi>]. Sedangkan, variabel *substring* dari nama direktori atau nama berkas akan ditulis sebagai "*".

41.2. Sistem Berkas

Terdapat dua perbedaan yang saling independen dalam berkas, yaitu *shareable* vs. *unshareable* dan *variable* vs. *static*. Secara umum, berkas-berkas yang memiliki perbedaan seperti di atas sebaiknya diletakkan dalam direktori yang berbeda. Hal ini mempermudah penyimpanan berkas dengan karakteristik yang berbeda dalam sistem berkas yang berbeda.

Berkas *shareable* adalah berkas yang disimpan di satu komputer, namun masih dapat digunakan oleh komputer lainnya. Sedangkan berkas *unshareable* tidak dapat digunakan bersama-sama antar komputer yang satu dan lainnya.

Berkas *static* meliputi berkas biner, pustaka, dokumentasi dan berkas-berkas lain yang tidak dapat diubah tanpa intervensi administrator sistem. Sedangkan, berkas *variable* adalah semua berkas yang bukan merupakan berkas *static*.

41.3. Sistem Berkas *Root*

Tujuan dan Prasyarat

Isi dari sistem berkas *root* harus memadai untuk melakukan operasi *boot*, *restore*, *recover*, dan atau perbaikan pada sistem.

Untuk melakukan operasi *boot* pada sistem, perlu dilakukan hal-hal untuk *mounting* sistem berkas lain. Hal ini meliputi konfigurasi data, informasi *boot loader* dan keperluan-keperluan lain yang mengatur *start-up* data.

Untuk melakukan *recovery* dan atau perbaikan dari sistem, hal-hal yang dibutuhkan untuk mendiagnosa dan memulihkan sistem yang rusak harus diletakkan dalam sistem berkas *root*.

Untuk *restore* suatu sistem, hal-hal yang dibutuhkan untuk *back-up* sistem, seperti *floppy disk*, *tape*, dsb, harus berada dalam sistem berkas *root*.

Aplikasi pada komputer tidak diperbolehkan untuk membuat berkas atau subdirektori di dalam direktori *root*, karena untuk meningkatkan *performance* dan keamanan, partisi *root* sebaiknya dibuat seminimum mungkin. Selain itu, lokasi-lokasi lain dalam FHS menyediakan fleksibilitas yang lebih

dari cukup untuk *package* mana pun.

Terdapat beberapa direktori yang merupakan persyaratan dari sistem berkas *root*. Setiap direktori akan dibahas dalam sub-bagian di bawah. /usr dan /var akan dibahas lebih mendetail karena direktori tersebut sangat kompleks.

Tabel 41.1. Direktori/link yang dibutuhkan dalam "/."

Direktori	Keterangan
bin	Instruksi dasar biner
boot	Berkas statik untuk me-*load boot*
dev	Berkas peranti
etc	Konfigurasi sistem *host-specific*
lib	Pustaka dasar bersama dan modul *kernel*
media	*Mount point* untuk media-media *removable*
mnt	*Mount point* untuk *mounting* sistem berkas secara temporer
opt	Penambahan aplikasi *package* perangkat lunak
sbin	Sistem biner dasar
srv	Data untuk servis yang disediakan oleh sistem
tmp	Berkas temporer
usr	Hirarki sekunder
var	Data variabel
home	Direktori *home* pengguna
lib<qual>	Format alternatif dari pustaka dasar bersama
root	Direktori *home* untuk *root* pengguna

"/bin": Perintah biner dasar (untuk digunakan oleh semua pengguna)

"/bin" berisi perintah-perintah yang dapat digunakan oleh administrator sistem dan pengguna, namun dibutuhkan apabila tidak ada sistem berkas lain yang di-*mount*. "/bin" juga berisi perintah-perintah yang digunakan secara tidak langsung oleh *script*.

"/boot": Berkas statik untuk me-*load boot*

Dalam direktori ini, terdapat segala sesuatu yang dibutuhkan untuk melakukan *boot* proses. "/boot" menyimpan data yang digunakan sebelum *kernel* mulai menjalankan program mode pengguna. Hal ini dapat meliputi sektor *master boot* dan sektor berkas map.

"/dev": Berkas peranti

Direktori "/dev" adalah lokasi dari berkas-berkas peranti. Direktori ini harus memiliki perintah bernama "MAKEDEV" yang dapat digunakan untuk menciptakan peranti secara manual. Jika dibutuhkan, "MAKEDEV" harus memiliki segala ketentuan untuk menciptakan peranti-peranti yang ditemukan dalam sistem, bukan hanya implementasi partikular yang di-*install*.

"/etc": Konfigurasi sistem *host-specific*

Direktori "/etc" mernyimpan berkas-berkas konfigurasi. Yang dimaksud berkas konfigurasi adalah berkas lokal yang digunakan untuk mengatur operasi dari sebuah program. Berkas ini harus statik dan bukan merupakan biner *executable*.

"/home": Direktori home pengguna

"/home" adalah konsep standar sistem berkas yang *site-specific*, artinya *setup* dalam *host* yang satu dan yang lainnya akan berbeda-beda. Maka, program sebaiknya tidak diletakkan dalam direktori ini.

"/lib": Pustaka dasar bersama dan modul *kernel*

Direktori "/lib" meliputi gambar-gambar pustaka bersama yang dibutuhkan untuk *boot* sistem tersebut dan menjalankan perintah dalam sistem berkas *root*, contohnya berkas biner di "/bin" dan "/sbin".

"/lib<qual>": Format alternatif dari pustaka dasar bersama

Pada sistem yang mendukung lebih dari satu format biner, mungkin terdapat satu atau lebih perbedaan dari direktori "/lib". Jika direktori ini terdapat lebih dari satu, maka persyaratan dari isi tiap direktori adalah sama dengan direktori "/lib" normalnya, namun "/lib<qual>/cpp" tidak dibutuhkan.

"/media": Mount point media *removable*

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

"/mnt": *Mount point* untuk sistem berkas yang di-*mount* secara temporer

Direktori ini disediakan agar administrator sistem dapat *mount* suatu sistem berkas yang dibutuhkan secara temporer. Isi dari direktori ini adalah *issue* lokal, dan tidak mempengaruhi sifat-sifat dari program yang sedang dijalankan.

/opt: Aplikasi tambahan untuk paket peringkat lunak

"/opt" disediakan untuk aplikasi tambahan paket peringkat lunak. Paket yang di-*install* di "/opt" harus menemukan berkas statiknya di direktori "/opt/<package>" atau "/opt/<provider>", dengan <package> adalah nama yang mendeskripsikan paket perangkat lunak tersebut, dan <provider> adalah nama dari *provider* yang bersangkutan.

"/root": Direktori *home* untuk *root* pengguna

Direktori *home root* dapat ditentukan oleh pengembang atau pilihan-pilihan lokal, namun direktori ini adalah lokasi *default* yang direkomendasikan.

"/sbin": Sistem Biner

Kebutuhan yang digunakan oleh administrator sistem disimpan di "/sbin", "/usr/sbin", dan "/usr/local/sbin". "/sbin" berisi biner dasar untuk *boot* sistem, mengembalikan sistem, memperbaiki sistem sebagai tambahan untuk biner-biner di "/bin". Program yang dijalankan setelah /usr diketahui harus di-*mount*, diletakkan dalam "/usr/bin". Sedangkan, program-program milik administrator sistem yang di-*install* secara lokal sebaiknya diletakkan dalam "/usr/local/sbin".

"/srv": Data untuk servis yang disediakan oleh sistem

"/srv" berisi data-data *site-specific* yang disediakan oleh sistem.

"/tmp": Berkas-berkas temporer

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

41.4. Hirarki "/usr"

Tujuan

"/usr" adalah bagian utama yang kedua dari sistem berkas. "/usr" bersifat *shareable* dan *read-only*. Hal ini berarti "/usr" bersifat *shareable* diantara bermacam-macam *host* FHS-*compliant*, dan tidak boleh di-*write*. *Package* perangkat lunak yang besar tidak boleh membuat subdirektori langsung di bawah hirarki "/usr" ini.

Persyaratan

Tabel 41.2. Direktori/link yang dibutuhkan dalam ''/usr''.

Direktori	Keterangan
bin	Sebagian besar perintah pengguna
include	Berkas *header* yang termasuk dalam program-program C
lib	Pustaka
local	Hirarki lokal (kosong sesudah instalasi main)
sbin	Sistem biner non-vital
share	Data arsitektur yang independen

Pilihan spesifik

Tabel 41.3. Direktori/link yang merupakan pilihan dalam ''/usr''.

Direktori	Keterangan
X11R6	Sistem X Window, Versi 11 *Release* 6
games	*Games*dan *educational* biner
lib<qual>	Format pustaka alternatif
src	Kode *source*

Link-link simbolik seperti di bawah ini dapat terjadi, apabila terdapat kebutuhan untuk menjaga keharmonisan dengan sistem yang lama, sampai semua implementasi dapat diasumsikan untuk menggunakan hirarki "/var":

- /usr/spool --> /var/spool
- /usr/temp --> /var/tmp
- /usr/spool/locks --> /var/lock

Saat sistem tidak lagi membutuhkan *link-link* di atas, *link* tersebut dapat dihapus.

"/usr/X11R6": Sistem X Window, Versi 11 Release 6

Hirarki ini disediakan untuk Sistem X Window, Versi 11 *Release* 6 dan berkas-berkas yang berhubungan. Untuk menyederhanakan persoalan dan membuat XFree86 lebih kompatibel dengan Sistem X Window, link simbolik di bawah ini harus ada jika terdapat direktori "/usr/X11R6":

- /usr/bin/X11 --> /usr/X11R6/bin
- /usr/lib/X11 --> /usr/X11R6/lib/X11
- /usr/include/X11 --> /usr/X11R6/include/X11

Link-link di atas dikhususkan untuk kebutuhan dari pengguna saja, dan perangkat lunak tidak boleh di-*install* atau diatur melalui *link-link* tersebut.

"/usr/bin": Sebagian perintah pengguna

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

"/usr/include": Direktori untuk *include-files* standar

Direktori ini berisi penggunaan umum berkas *include* oleh sistem, yang digunakan untuk bahasa pemrograman C.

"/usr/lib": Pustaka untuk pemrograman dan *package*

"/usr/lib" meliputi berkas obyek, pustaka dan biner internal yang tidak dibuat untuk dieksekusi secara langsung melalui pengguna atau *shell script*. Aplikasi-aplikasi dapat menggunakan subdirektori tunggal di bawah "/usr/lib". Jika aplikasi tersebut menggunakan subdirektori, semua data yang arsitektur-*dependent* yang digunakan oleh aplikasi tersebut, harus diletakkan dalam subdirektori tersebut juga.

Untuk alasan historis, "/usr/lib/sendmail" harus merupakan *link* simbolik ke "/usr/sbin/sendmail". Demikian juga, jika "/lib/X11" ada, maka "/usr/lib/X11" harus merupakan *link* simbolik ke "/lib/X11", atau ke mana pun yang dituju oleh *link* simbolik "/lib/X11".

"/usr/lib<qual>": Format pustaka alternatif

"/usr/lib<qual>" melakukan peranan yang sama seperti "/usr/lib" untuk format biner alternatif, namun tidak lagi membutuhkan *link* simbolik seperti "/usr/lib<qual>/sendmail" dan "/usr/lib<qual>/X11".

"/usr/local/share"

Direktori ini sama dengan "/usr/share". Satu-satunya pembatas tambahan adalah bahwa direktori '/usr/local/share/man' dan '/usr/local/man' harus *synonomous* (biasanya ini berarti salah satunya harus merupakan *link* simbolik).

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

Direktori ini berisi biner non-vital mana pun yang digunakan secara eksklusif oleh administrator sistem. Program administrator sistem yang diperlukan untuk perbaikan sistem, *mounting* "/usr" atau kegunaan penting lainnya harus diletakkan di "/sbin".

"/usr/share": Data arsitektur independen

Hirarki "/usr/share" hanya untuk data-data arsitektur independen yang *read-only*. Hirarki ini

ditujukan untuk dapat di-*share* diantara semua arsitektur *platform* dari sistem operasi; sebagai contoh: sebuah *site* dengan *platform* i386, Alpha dan PPC dapat me-*maintain* sebuah direktori /usr/share yang di-*mount* secara sentral.

Program atau paket mana pun yang berisi dan memerlukan data yang tidak perlu dimodifikasi harus menyimpan data tersebut di "/usr/share" (atau "/usr/local/share", apabila di- *install* secara lokal). Sangat direkomendasikan bahwa sebuah subdirektori digunakan dalam /usr/share untuk tujuan ini.

"/usr/src": Kode *source*

Dalam direktori ini, dapat diletakkan kode-kode *source*, yang digunakan untuk tujuan referensi.

41.5. Hirarki "/var"

Tujuan

"/var" berisi berkas data variabel, meliputi berkas dan direktori *spool*, data administratif dan *logging*, serta berkas *transient* dan temporer. Beberapa bagian dari "/var" tidak *shareable* diantara sistem yang berbeda, antara lain: "/var/log", "/var/lock" dan "/var/run". Sedangkan, "/var/mail", "/var/cache/man", "/var/cache/fonts" dan "/var/spool/news" dapat di-*share* antar sistem yang berbeda.

"/var" ditetapkan di ini untuk memungkinkan operasi *mount* "/usr" *read-only*. Segala sesuatu yang melewati "/usr", yang telah ditulis selama operasi sistem, harus berada di "/var". Jika "/var" tidak dapat dibuatkan partisi yang terpisah, biasanya "/var" dipindahkan ke luar dari partisi *root* dan dimasukkan ke dalam partisi "/usr".

Bagaimana pun, "/var" tidak boleh di-*link* ke "/usr", karena hal ini membuat pemisahan antara "/usr" dan "/var" semakin sulit dan biasa menciptakan konflik dalam penamaan. Sebaliknya, buat *link* "/var" ke "/usr/var".

Persyaratan

Tabel 41.4. Direktori/link yang dibutuhkan dalam ''/var''

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 yang disimpan di dalam *reboot* sistem

Pilihan Spesifik

Direktori atau link simbol yang menuju ke direktori di bawah ini, dibutuhkan dalam "/var", jika subsistem yang berhubungan dengan direktori tersebut di-*install*:

Tabel 41.5. Direktori/link yang dibutuhkan di dalam ''/var''

Direktori	Keterangan
account	*Log accounting* proses
crash	*System crash dumps*
games	Data variabel *game*
mail	Berkas *mailbox* pengguna
yp	Network Information Service (NIS) berkas *database*

"/var/account": *Log accounting*proses

Direktori ini memegang *log accounting* dari proses yang sedang aktif dan gabungan dari penggunaan data.

"/var/cache": Aplikasi data *cache*

"/var/cache" ditujukan untuk data *cache* dari aplikasi. Data tersebut diciptakan secara lokal sebagai *time-consuming* M/K atau kalkulasi. Aplikasi ini harus dapat menciptakan atau mengembalikan data. Tidak seperti "/var/spool", berkas *cache* dapat dihapus tanpa kehilangan data.

Berkas yang ditempatkan di bawah "/var/cache" dapat *expired* oleh karena suatu sifat spesifik dalam aplikasi, oleh administrator sistem, atau keduanya, maka aplikasi ini harus dapat *recover* dari penghapusan berkas secara manual.

"/var/crash": *System crash dumps*

Direktori ini mengatur *system crash dumps*. Saat ini, *system crash dumps* belum dapat di-*support* oleh Linux, namun dapat di-*support* oleh sistem lain yang dapat memenuhi FHS.

"/var/games": Data variabel *game*

Data variabel mana pun yang berhubungan dengan *games* di "/usr" harus diletakkan di direktori ini. "/var/games" harus meliputi data variabel yang ditemukan di /usr; data statik, seperti *help text*, deskripsi level, dll, harus ditempatkan di lain direktori, seperti "/usr/share/games".

"/var/lib": Informasi status variabel

Hirarki ini berisi informasi status suatu aplikasi dari sistem. Yang dimaksud dengan informasi status adalah data yang dimodifikasi program saat program sedang berjalan. Pengguna tidak diperbolehkan untuk memodifikasi berkas di "/var/lib" untuk mengkonfigurasi operasi *package*. Informasi status ini digunakan untuk memantau kondisi dari aplikasi, dan harus tetap valid setelah *reboot*, tidak berupa *output logging* atau pun data *spool*.

Sebuah aplikasi harus menggunakan subdirektory "/var/lib" untuk data-datanya. Terdapat satu subdirektori yang dibutuhkan lagi, yaitu "/var/lib/misc", yang digunakan untuk berkas-berkas status yang tidak membutuhkan subdirektori.

"/var/lock": *Lock* berkas

Lock berkas harus disimpan dalam struktur direktori /var/lock. *Lock* berkas untuk peranti dan sumber lain yang di-*share* oleh banyak aplikasi, seperti *lock* berkas pada serial peranti yang ditemukan dalam "/usr/spool/locks" atau "/usr/spool/uucp", sekarang disimpan di dalam "/var/lock".

Format yang digunakan untuk isi dari *lock* berkas ini harus berupa format *lock* berkas HDB UUCP. Format HDB ini adalah untuk menyimpan pengidentifikasi proses (Process Identifier - PID) sebagai

10 byte angka desimal ASCII, ditutup dengan baris baru. Sebagai contoh, apabila proses 1230 memegang *lock* berkas, maka HDO formatnya akan berisi 11 karakter: spasi, spasi, spasi, spasi, spasi, spasi, satu, dua, tiga, nol dan baris baru.

"/var/log": Berkas dan direktori *log*

Direktori ini berisi bermacam-macam berkas *log*. Sebagian besar *log* harus ditulis ke dalam direktori ini atau subdirektori yang tepat.

"/var/mail": Berkas *mailbox*pengguna

Mail spool harus dapat diakses melalui "/var/mail" dan berkas mail spool harus menggunakan form <nama_pengguna>. Berkas *mailbox* pengguna dalam lokasi ini harus disimpan dengan format standar *mailbox* UNIX.

"/var/opt": Data variabel untuk "/opt"

Data variabel untuk paket di dalam "/opt" harus di-*install* dalam "/var/opt/<subdir>", di mana <subdir> adalah nama dari *subtree* dalam "/opt" tempat penyimpanan data statik dari *package* tambahan perangkat lunak.

"/var/run": Data variabel *run-time*

Direktori ini berisi data informasi sistem yang mendeskripsikan sistem sejak di *boot*. Berkas di dalam direktori ini harus dihapus dulu saat pertama memulai proses *boot*. Berkas pengidentifikasi proses (PID), yang sebelumnya diletakkan di "/etc", sekarang diletakkan di "/var/run".

Program yang membaca berkas-berkas PID harus fleksibel terhadap berkas yang diterima, sebagai contoh: program tersebut harus dapat mengabaikan ekstra spasi, baris-baris tambahan, angka nol yang diletakkan di depan, dll.

"/var/spool": Aplikasi data *spool*

"/var/spool" berisi data yang sedang menunggu suatu proses. Data di dalam "/var/spool" merepresentasikan pekerjaan yang harus diselesaikan dalam waktu depan (oleh program, pengguna atau administrator); biasanya data dihapus sesudah selesai diproses.

"/var/tmp": Berkas temporer ang diletakkan di dalam *reboot* sistem

Direktori "/var/tmp" tersedia untuk program yang membutuhkan berkas temporer atau direktori yang diletakkan dalam *reboot* sistem. Karena itu, data yang disimpan di "/var/tmp" lebih bertahan daripada data di dalam "/tmp". Berkas dan direktori yang berada dalam "/var/tmp" tidak boleh dihapus saat sistem di-*boot*. Walaupun data-data ini secara khusus dihapus dalam *site-specific manner*, tetap direkomendasikan bahwa penghapusan dilakukan tidak sesering penghapusan di "/tmp".

"/var/yp": Berkas database Network Information Service (NIS)

Data variabel dalam Network Information Service (NIS) atau yang biasanya dikenal dengan Sun Yellow Pages (YP) harus diletakkan dalam direktori ini.

41.6. Rangkuman

Standar Hirarki Sistem Berkas (*File Hierarchy Standard*) adalah rekomendasi penulisan direktori

dan berkas-berkas yang diletakkan di dalamnya. FHS ini digunakan oleh perangkat lunak dan *user* untuk menentukan lokasi dari berkas dan direktori.

41.7. Latihan

1. Sebutkan dua sistem operasi yang implementasi sistem berkasnya menggunakan *Layered File System*!
2. Jelaskan cara membuat suatu berkas baru!
3. Apakah hubungan partisi dan *mounting* dengan implementasi sistem berkas?
4. Apa yang Anda ketahui mengenai *Filesystem Hierarchy Standard*?
5. Jelaskan 3 tujuan dari sistem berkas *root*!
6. Jelaskan tujuan dan persyaratan dari hirarki /usr!
7. Jelaskan tujuan dan persyaratan dari hirarki /var!
8. Apakah kegunaan dari direktori di bawah ini:
   - /boot
   - /media
   - /mnt
   - /root
   - /usr/lib
   - /var/cache

41.8. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts, 6th ed*. John Wiley & Sons.

Tananbaum, Andrew S. 1992. *Modern Operating System 2nd ed*. Engrewood cliffs, New Jersey: Prentice Hall Inc.

Stallings, Williem. 2000. *Operating System 4th ed*. Prentice Hall.

http://infocom.cqu.edu.au/Courses/aut2001/85349/Resources/ Study_Guide/10.pdf

http://www.cs.utah.edu/classes/cs5460/lectures/lecture19- 2up.pdf

http://support.sitescape.com/ forum/ support/ dispatch.cgi/ _help/ showHelp/ page/ help/ en/ webfiles_tabs/ share_files.html

>http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690/ IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm

http://www.atnf.csiro.au/people/rgooch/linux/docs/vfs.txt

Bab 42. Konsep Alokasi Blok Sistem Berkas

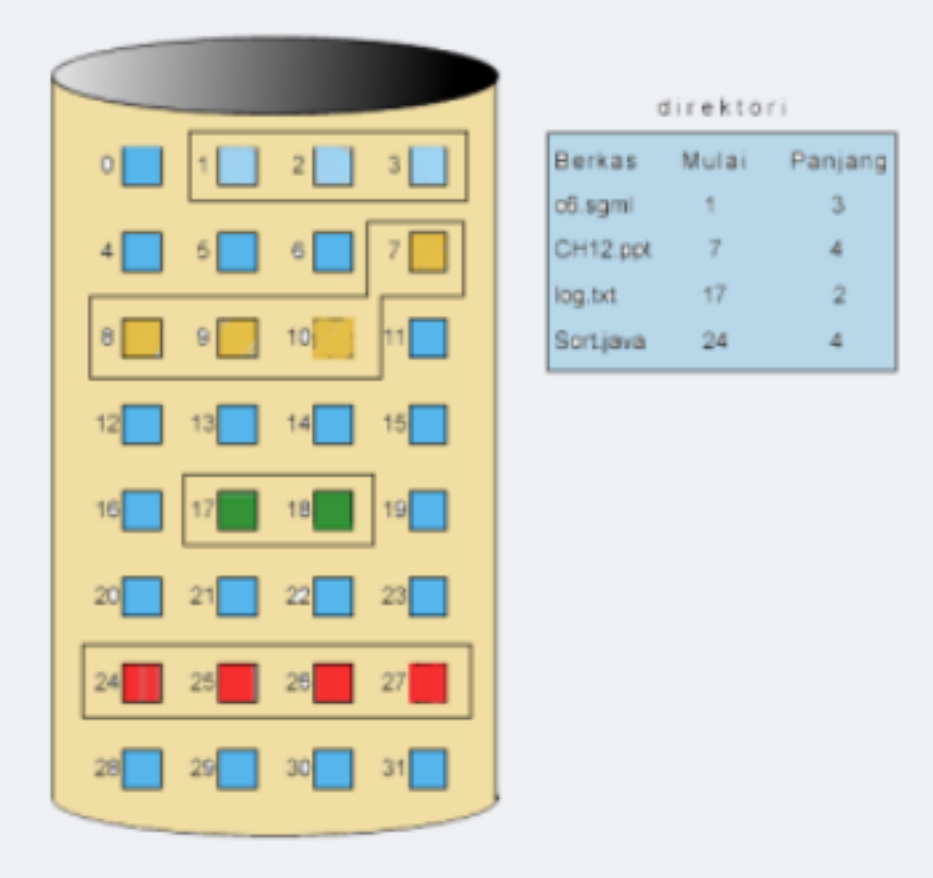
42.1. Metode Alokasi

Kegunaan penyimpanan sekunder yang utama adalah menyimpan berkas-berkas yang kita buat, karena sifat disk akan mempertahankan berkas walaupun tidak ada arus listrik. Oleh karena itu, agar kita dapat mengakses berkas-berkas dengan cepat dan memaksimalisasikan ruang yang ada di disk tersebut, maka lahirlah metode-metode untuk mengalokasikan berkas ke disk. Metode-metode yang akan dibahas lebih lanjut dalam buku ini adalah *contiguous allocation*, *linked allocation*, dan *indexed allocation*. Metode-metode tersebut memiliki beberapa kelebihan dan juga kekurangan. Biasanya sistem operasi memilih satu dari metode diatas untuk mengatur keseluruhan berkas.

Contiguous Allocation

Metode ini akan mengalokasikan satu berkas kedalam blok-blok disk yang berkesinambungan atau berurutan secara linier dari disk, jadi sebuah berkas didenifinikan oleh alamat disk blok pertama dan panjangnya dengan satuan blok atau berapa blok yang diperlukannya. Bila suatu berkas memerlukan n buah blok dan blok awalnya adalah a, berarti berkas tersebut disimpan dalam blok dialamat a, a + 1, a + 2, a + 3, ..., a + n - 1. Direktori mengindentifikasi setiap berkas hanya dengan alamat blok pertama berkas tersebut disimpan yang dalam contoh di atas adalah a, dan banyaknya blok yang diperlukan untuk mengalokasikan berkas tersebut yang dalam contoh di atas adalah n.

Gambar 42.1. ***Contiguous allocation***



Berkas yang dialokasikan dengan metode ini akan mudah diakses, karena pengaksesan alamat a + 1 setelah alamat a tidak diperlukan perpindahan head, jika diperlukan pemindahan head, maka head tersebut akan hanya akan berpindah satu track. Hal tersebut menjadikan metode ini mendukung pengaksesan secara berurutan, tapi metode ini juga mendukung pengaksesan secara langsung, karena bila ingin mengakses blok ke i berarti kita akan mengakses blok a + i.

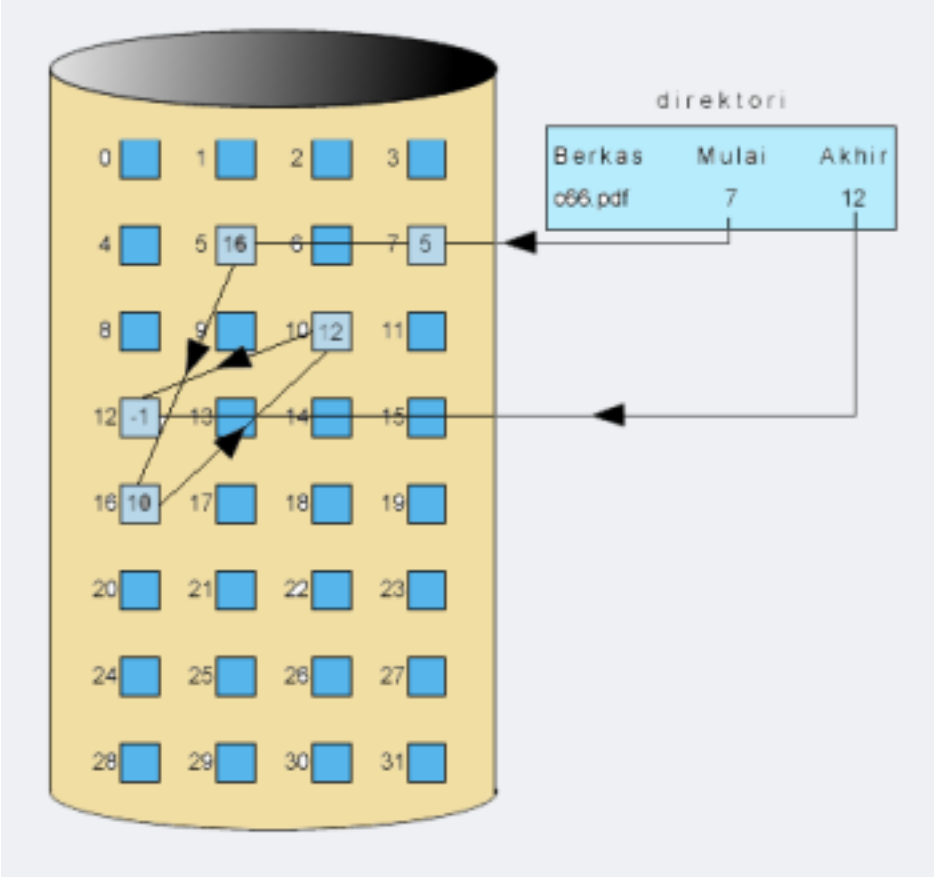
Metode *contiguous allocation* juga mempunyai beberapa masalah. Diantaranya adalah mencari ruang untuk berkas baru, menentukan seberapa besar ruang yang diperlukan untuk sebuah berkas. Untuk masalah mencari ruang untuk berkas baru, akan di implementasikan oleh managemen ruang kosong.

Untuk penentuan ruang kita tidak boleh terlalu kecil atau terlalu besar, bila kita menentukannya terlalu kecil maka ada saatnya berkas tersebut tidak dapat dikembangkan, tapi bila terlalu besar maka akan ada ruang yang sia-sia bila berkas tersebut hanya memerlukan ruang yang kecil.

Metode ini dapat menimbulkan fragmentasi eksternal disaat ruang kosong yang ada diantara berkas-berkas yang sudah terisi tidak cukup untuk mengisi berkas baru. Hal ini terjadi karena blok pertama dari suatu berkas itu ditentukan oleh sistem operasi, bila berkas pertama blok pertamanya itu di 1 dan memerlukan 9 blok untuk pengalokasiannya dan berkas kedua blok pertamanya di 11 dan memerlukan 5 blok untuk pengalokasiannya, berarti ruang-kosong diantara berkas tersebut ada 1 blok, yaitu dialamat 10. Blok tersebut dapat untuk menyimpan berkas, tetapi hanya berkas yang berukuran 1 blok yang dapat disimpan di blok tersebut.

Linked Allocation

Gambar 42.2. ***Linked allocation***



Metode ini dapat mengatasi masalah yang terjadi pada metode *contiguous allocation*. Dalam metode ini setiap berkas diidentifikasikan dengan *linked list* dari blok-blok, jadi blok-blok tersebut tidak

harus berkesinambungan dengan blok yang lain. Direktori hanya menyimpan alamat blok pertama dan alamat blok terakhir. Jika kita ingin mengaksess blok kedua, maka harus melihat alamatnya di blok pertama dan begitu seterusnya. Oleh karena itu, metode ini hanya mendukung pengaksesan secara berurutan.

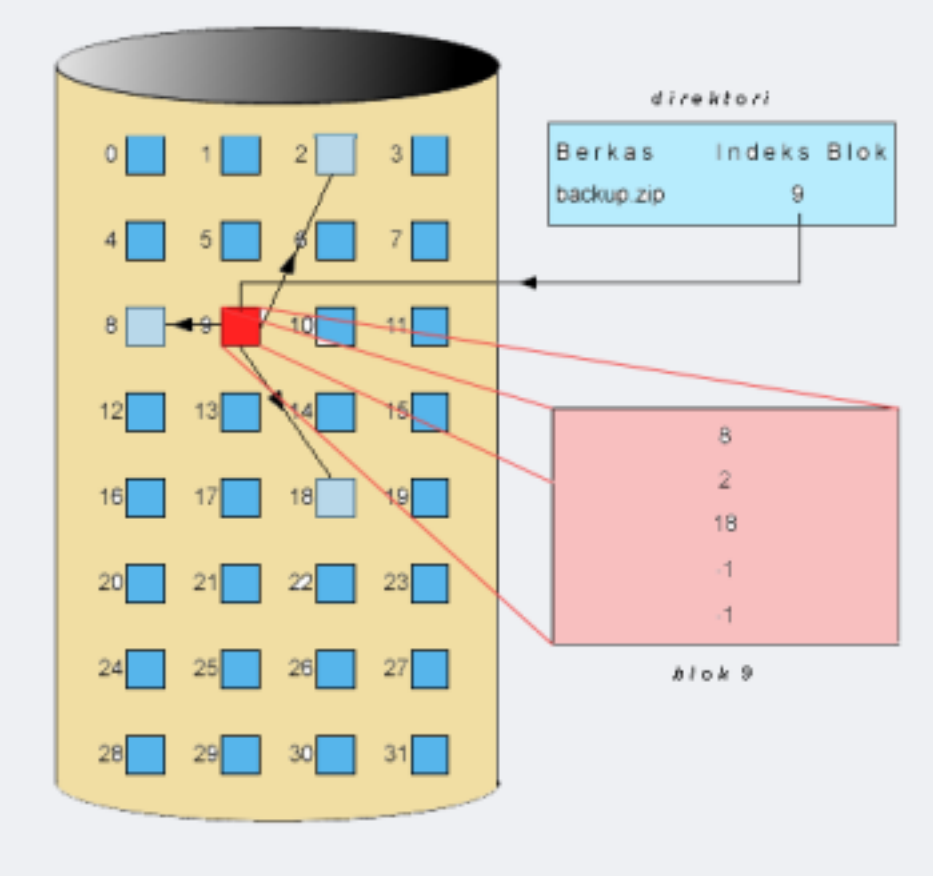
Metode *linked allocation* memiliki beberapa kerugian, karena petunjuk ke blok berikutnya memerlukan ruang. Bila ukuran petunjuknya 4 byte dari blok yang ukurannya 512 byte, berarti 0,78% dari ruang disk hanya digunakan untuk petunjuk saja. Hal ini dapat diminimalisasikan dengan menggunakan *cluster* yang menggabungkan 4 blok dalam satu *cluster*, jadi jumlah petunjuknya akan berkurang dari yang tidak memakai *cluster*.

Paling penting dalam metode ini adalah menggunakan *file-allocation table* (FAT). Tabel tersebut menyimpan setiap blok yang ada di disk dan diberi nomor sesuai dengan nomor blok. Jadi, direktori hanya menyimpan alamat dari blok pertama saja, dan untuk selanjutnya dilihat dari tabel tersebut yang menunjukkan ke blok berikutnya. Jika kita memakai metode ini, akan menyebabkan mudahnya untuk membuat berkas baru atau mengembangkan berkas sebelumnya. Mencari tempat kosong untuk berkas baru lebih mudah, karena kita hanya mencari angka 0 yang pertama dari isi tabel tersebut. Dan bila kita ingin mengembangkan berkas sebelumnya carilah alamat terakhirnya yang memiliki ciri tertentu dan ubahlah isi dari tabel tersebut dengan alamat blok penambahan. Alamat terakhir berisi hal yang unik, sebagai contoh ada yang menuliskan -1, tapi ada juga yang menuliskannya EOF (*End Of File*).

Metode *linked allocation* yang menggunakan FAT akan mempersingkat waktu yang diperlukan untuk mencari sebuah berkas. Karena bila tidak menggunakan FAT, berarti kita harus ke satu blok tertentu dahulu dan baru diketahui alamat blok selanjutnya. Dengan menggunakan FAT kita dapat melihat alamat blok selanjutnya disaat kita masih menuju blok yang dimaksud. Tetapi bagaimana pun ini belum dapat mendukung pengaksesan secara langsung.

Indexed Allocation

Gambar 42.3. ***Indexed allocation***



Metode yang satu ini memecahkan masalah fragmentasi eksternal dari metode *contiguous allocation* dan ruang yang cuma-cuma untuk petunjuk pada metode *linked allocation*, dengan cara menyatukan semua petunjuk kedalam blok indeks yang dimiliki oleh setiap berkas. Jadi, direktori hanya menyimpan alamat dari blok indeks tersebut, dan blok indeks tersebut yang menyimpan alamat dimana blok-blok berkas berada. Untuk berkas yang baru dibuat, maka blok indeksnya di set dengan *null*.

Metode ini mendukung pengaksesan secara langsung, bila kita ingin mengakses blok ke-i, maka kita hanya mencari isi dari blok indeks tersebut yang ke-i untuk dapatkan alamat blok tersebut.

Metode *indexed allocation* tidak menyia-nyiakan ruang disk untuk petunjuk, karena dibandingkan dengan metode *linked allocation*, maka metode ini lebih efektif, kecuali bila satu berkas tersebut hanya memerlukan satu atau dua blok saja.

Metode ini juga memiliki masalah. Masalah itu timbul disaat berkas berkembang menjadi besar dan blok indeks tidak dapat menampung petunjuk-petunjuknya itu dalam satu blok. Salah satu mekanisme dibawah ini dapat dipakai untuk memecahkan masalah yang tersebut. Mekanisme-mekanisme itu adalah:

- *Linked scheme*: Untuk mengatasi petunjuk untuk berkas yang berukuran besar mekanisme ini menggunakan tempat terakhir dari blok indeks untuk alamat ke blok indeks selanjutnya. Jadi, bila berkas kita masih berukuran kecil, maka isi dari tempat yang terakhir dari blok indeks berkas tersebut adalah *null*. Namun, bila berkas tersebut berkas besar, maka tempat terakhir itu berisikan alamat untuk ke blok indeks selanjutnya, dan begitu seterusnya.

- Indeks bertingkat: Pada mekanisme ini blok indeks itu bertingkat-tingkat, blok indeks pada tingkat pertama akan menunjukkan blok-blok indeks pada tingkat kedua, dan blok indeks pada tingkat kedua menunjukkan alamat-alamat dari blok berkas, tapi bila dibutuhkan dapat dilanjutkan kelevel ketiga dan keempat tergantung dengan ukuran berkas tersebut. Untuk blok indeks 2 level dengan ukuran blok 4.096 byte dan petunjuk yang berukuran 4 byte, dapat mengalokasikan berkas hingga 4 GB, yaitu 1.048.576 blok berkas.

- *Combined scheme*: Mekanisme ini menggabungkan *direct block* dan *indirect block*. *Direct block* akan langsung menunjukkan alamat dari blok berkas, tetapi pada *indirect block* akan menunjukkan blok indeks terlebih dahulu seperti dalam mekanisme indeks bertingkat. *Single indirect block* akan menunjukkan ke blok indeks yang akan menunjukkan alamat dari blok berkas, *double indirect block* akan menunjukkan suatu blok yang bersifat sama dengan blok indeks 2 level, dan *triple indirect block* akan menunjukkan blok indeks 3 level. Dimisalkan ada 15 petunjuk dari mekanisme ini, 12 pertama dari petunjuk tersebut adalah *direct block*, jadi bila ukuran blok 4 byte berarti berkas yang dapat diakses secara langsung didukung sampai ukurannya 48 KB. 3 petunjuk berikutnya adalah *indirect block* yang berurutan dari *single indirect block* sampai *triple indirect block*. Yang hanya mendukung 32 bit petunjuk berkas berarti akan hanya mencapai 4 GB, namun yang mendukung 64 bit petunjuk berkas dapat mengalokasikan berkas berukuran sampai satuan terabyte.

Kinerja Sistem Berkas

Keefisiensian penyimpanan dan waktu akses blok data adalah kriteria yang penting dalam memilih metode yang cocok untuk sistem operasi untuk mengimplementasikan sesuatu. Sebelum memilih sebuah metode alokasi, kita butuh untuk menentukan bagaimana sistem ini akan digunakan.

Untuk beberapa tipe akses, *contiguous allocation* membutuhkan hanya satu akses untuk mendapatkan sebuah blok disk. Sejak kita dapat dengan mudah menyimpan alamat inisial dari sebuah berkas di memori, kita dapat menghitung alamat disk dari blok ke-i (atau blok selanjutnya) dengan cepat dan membacanya dengan langsung.

Untuk *linked allocation*, kita juga dapat menyimpan alamat dari blok selanjutnya di memori dan membacanya dengan langsung. Metode ini bagus untuk akses secara berurutan; untuk akses langsung, bagaimana pun, sebuah akses menuju blok ke-i harus membutuhkan pembacaan disk ke-i.

Masalah ini menunjukkan mengapa alokasi yang berurutan tidak digunakan untuk aplikasi yang membutuhkan akses langsung.

Sebagai hasilnya, beberapa sistem mendukung berkas-barkas yang diakses langsung dengan menggunakan *contiguous allocation* dan yang diakses berurutan dengan *linked allocation*. Di dalam kasus ini, sistem operasi harus mempunyai struktur data yang tepat dan algoritma untuk mendukung kedua metode alokasi.

Indexed allocation lebih komplek. Jika blok indeks sudah ada dimemori, akses dapat dibuat secara langsung. Bagaimana pun, menyimpan blok indeks tersebut di memori membutuhkan tempat yang dapat ditolerir. Dengan begitu, kinerja dari *indexed allocation* tergantung dari struktur indeks, ukuran file, dan posisi dari blok yang diinginkan.

Beberapa sistem menggabungkan *contiguous allocation* dengan *indexed allocation* dengan menggunakan *contiguous allocation* untuk berkas-berkas yang kecil (diatas tiga atau empat berkas), dan secara otomatis mengganti ke *indexed allocation* jika berkas bertambah besar.

42.2. Managemen Ruang Kosong

Sejak ruang disk terbatas, kita butuh menggunakan lagi ruang tersebut dari berkas yang sudah dihapus menjadi berkas yang baru, jika memungkinkan. Untuk menyimpan *track* dari ruang disk yang kosong, sistem membuat daftar ruang-kosong. Daftar ruang-kosong tersebut merekam semua blok-blok disk yang kosong itu semua tidak dialokasikan di beberapa berkas atau direktori.

Bit Vector

Seringkali, daftar ruang yang kosong diimplementasikan sebagai sebuah *bit map* atau *bit vector*. Setiap blok direpresentasikan dengan 1 bit. Jika bloknya kosong, bitnya adalah 1; jika bloknya ditempati, bitnya adalah 0.

Sebagai contoh, mepertimbangkan sebuah disk dimana blok-blok 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, dan 27 kosong, dan sisa dari blok-blok tersebut ditempati. *Bit map* dari ruang-kosong yaitu

00111100111111000110000011100000...

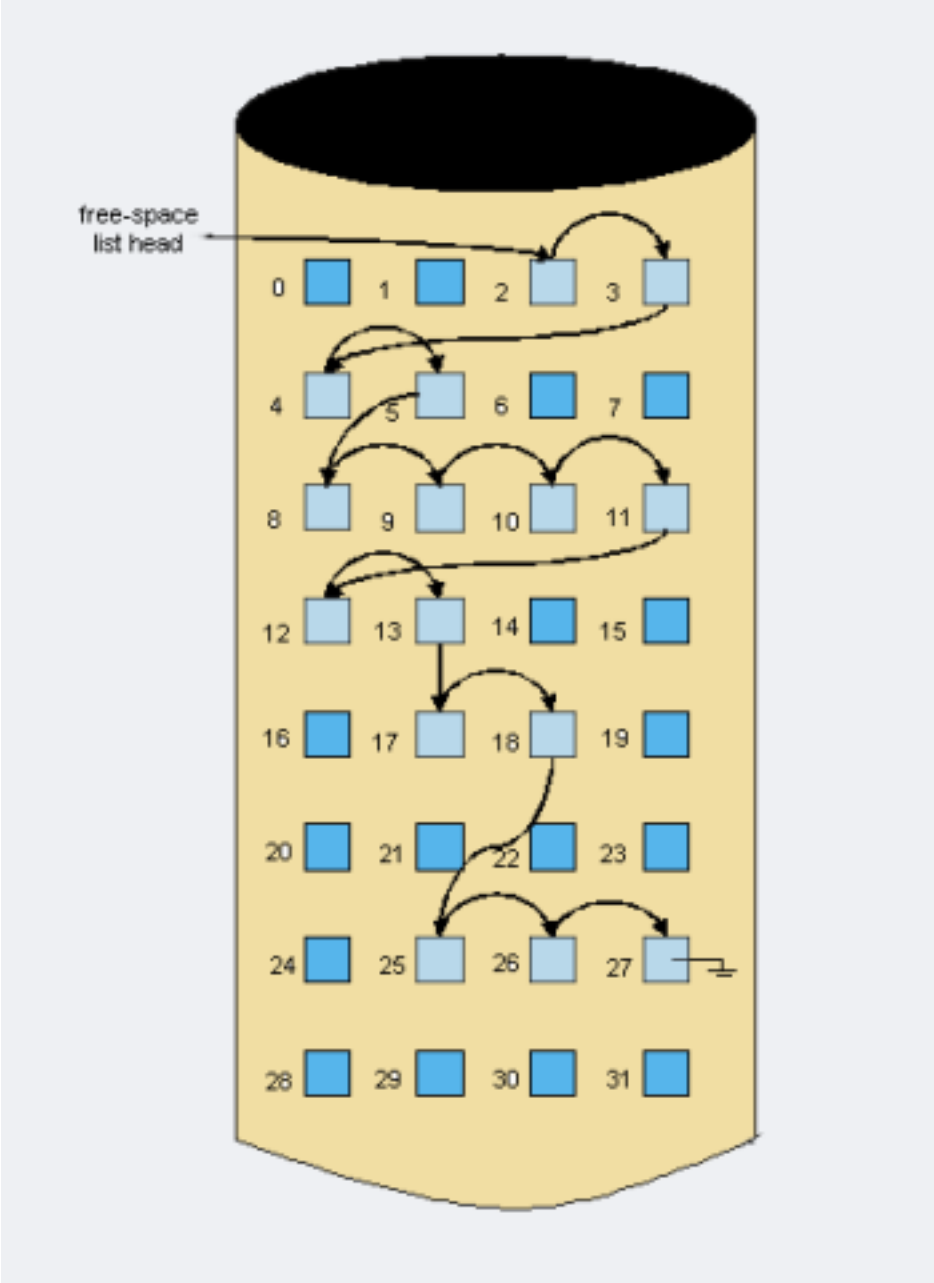
Keuntungan utama dari pendekatan ini adalah relatif sederhana dan keefisiensian dalam menemukan blok kosong yang pertama, atau blok-blok kosong n yang berurutan di dalam disk. Sayangnya, *bit vectors* tidak efisien kecuali seluruh vektor disimpan di memori utama (dan ditulis ke disk secara rutin untuk kebutuhan *recovery*. Menyimpan vektor tersebut di memori utama memungkinkan untuk disk-disk yang kecil, seperti pada *microcomputers*, tetapi tidak untuk disk-disk yang besar.

Linked List

Pendekatan yang lainnya untuk managemen ruang-kosong adalah menghubungkan semua blok-blok disk kosong, menyimpan sebuah penunjuk ke blok kosong yang pertama di lokasi yang khusus di disk dan menyimpannya di memori. Blok pertama ini mengandung sebuah penunjuk ke blok disk kosong selanjutnya, dan seterusnya. Sebagai contoh, kita akan menyimpan sebuah penunjuk ke blok 2, sebagai blok kosong pertama. Blok 2 mengandung sebuah penunjuk ke blok 3, yang akan menunjuk ke blok4, yang akan menunjuk ke blok 5, yang akan menunjuk ke blok 8, dan seterusnya.

Bagaimana pun, skema ini tidak efisien untuk mengakses daftar tersebut, kita harus membaca setiap blok, yang membutuhkan tambahan waktu M/K. Untungnya, mengakses daftar kosong tersebut itu tidak eksekusi yang teratur. Biasanya, sistem operasi tersebut membutuhkan sebuah blok kosong supaya sistem operasi dapat mengalokasikan blok tersebut ke berkas, lalu blok yang pertama di daftar kosong digunakan.

Gambar 42.4. Ruang kosong *linked list*



Grouping

Sebuah modifikasi dari pendekatan daftar-kosong adalah menyimpan alamat-alamat dari n blok-blok kosong di blok kosong yang pertama. n-1 pertama dari blok-blok ini sebenarnya kosong. Blok terakhir mengandung alamat-alamat dari n blok kosong lainnya, dan seterusnya. Pentingnya implementasi ini adalah alamat-alamat dari blok-blok kosong yang banyak dapat ditemukan secara cepat, tidak seperti di pendekatan *linked-list* yang standard.

Counting

Daripada menyimpan daftar dari n alamat-alamat disk kosong, kita dapat menyimpan alamat dari blok kosong yang pertama tersebut dan angka n dari blok *contiguous* kosong yang diikuti blok yang

pertama. Setiap masukan di daftar ruang-kosong lalu mengandung sebuah alamat disk dan sebuah jumlah. Meski pun setiap masukan membutuhkan ruang lebih daripada alamat-alamat disk yang sederhana, daftar kesemuanya akan lebih pendek, selama jumlahnya rata-rata lebih besar daripada 1.

42.3. Efisiensi dan Kinerja

Kita sekarang dapat mempertimbangkan mengenai efek dari alokasi blok dan manajeman direktori dalam kinerja dan penggunanan disk yang efisien. Di bagian ini, kita mendiskusikan tentang bermacam-macam teknik yang digunakan untuk mengembangkan efisiensi dan kinerja dari penyimpanan kedua.

Efisiensi

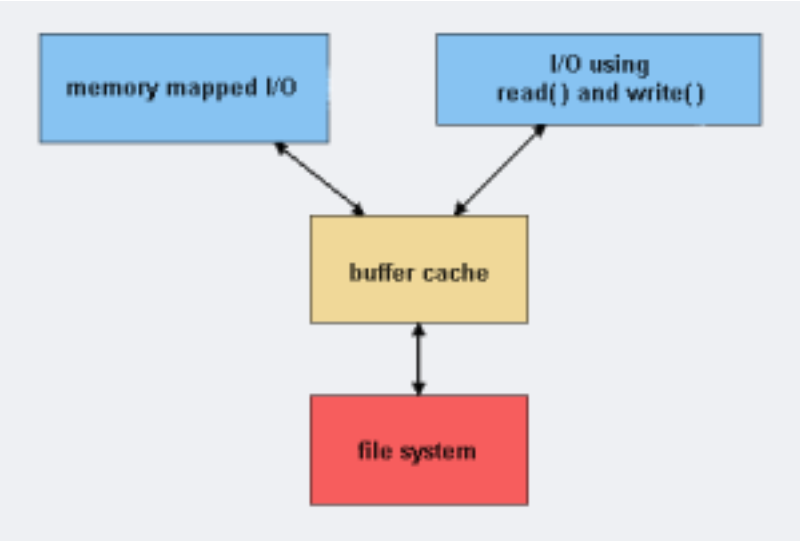
Penggunaan yang efisien dari ruang disk sangat tergantung pada alokasi disk dan algoritma direktori yang digunakan. Sebagai contoh, UNIX mengembangakan kinerjanya dengan mencoba untuk menyimpan sebuah blok data berkas dekat dengan blok inode berkas untuk mengurangi waktu pencarian.

Tipe dari data normalnya disimpan di masukan direktori berkas (atau inode) juga membutuhkan pertimbangan. Biasanya, tanggal terakhir penulisan direkam untuk memberikan informasi kepada pengguna dan untuk menentukan jika berkas ingin di *back up*. Beberapa sistem juga menyimpan sebiuah "last access date", supaya seorang pengguna dapat menentukan kapan berkas terakhir dibaca. Hasil dari menyimpan informasi ini adalah ketika berkas sedang dibaca, sebuah field di struktur direktori harus ditulisi. Prasyarat ini dapat tidak efisien untuk pengaksesan berkas yang berkala. Umumnya setiap persatuan data yang berhubungan dengan berkas membutuhkan untuk dipertimbangkan efeknya pada efisiensi dan kinerja.

Sebagai contoh, mempertimbangkan bagaimana efisiensi dipengaruhi oleh ukuran penunjuk-penunjuk yang digunakan untuk mengakses data. Bagaimana pun, penunjuk-penunjuk membutuhkan ruang lebih untuk disimpan, dan membuat metode alokasi dan managemen ruang-kosong menggunakan ruang disk yang lebih. Satu dari kesulitan memilih ukuran penunjuk, atau juga ukuran alokasi yang tetap diantara sistem operasi, adalah rencana untuk efek dari teknologi yang berubah.

Kinerja

Gambar 42.5. Menggunakan *unified buffer cache*



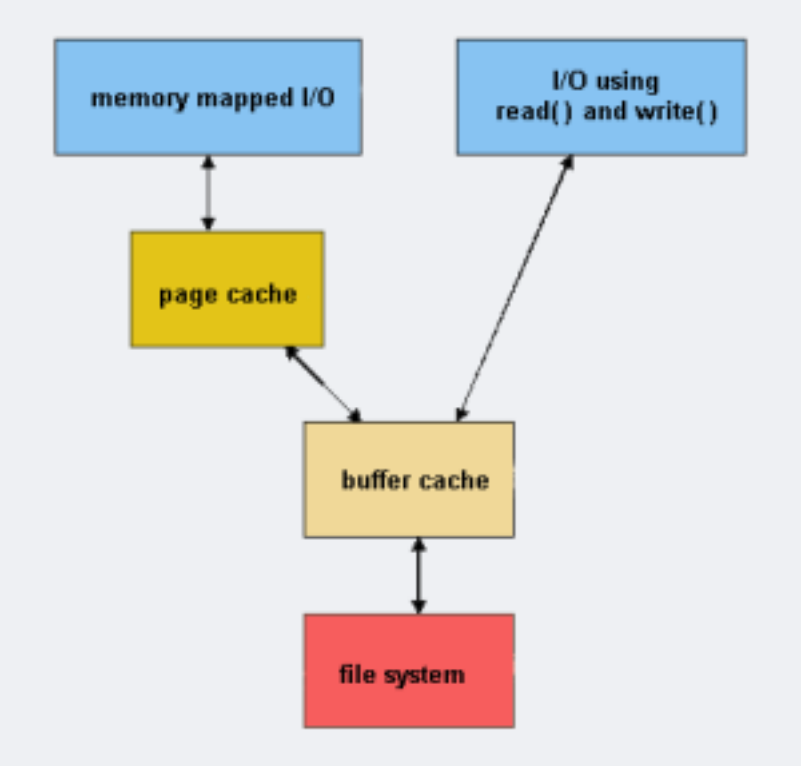
Sekali algoritma sistem berkas dipilih, kita tetap dapat mengembangkan kinerja dengan beberapa cara. Kebanyakan dari *disk controller* mempunyai memori lokal untuk membuat *on-board cache*

yang cukup besar untuk menyimpan seluruh *tracks* dengan sekejap.

Beberapa sistem membuat seksi yang terpisah dari memori utama untuk digunakan sebagai *disk cache*, dimana blok-blok disimpan dengan asumsi mereka akan digunakan lagi dengan secepatnya. Sistem lainnya menyimpan data berkas menggunakan sebuah *page cache*. *Page cache* tersebut menggunakan teknik memori virtual untuk menyimpan data berkas sebagai halaman-halaman daripada sebagai blok-blok *file-system-oriented*. Menyimpan data berkas menggunakan alamat-alamat virtual jauh lebih efisien daripada menyimpannya melalui blok disk fisik. Ini dikenal sebagai *unified virtual memory*.

Sebagian sistem operasi menyediakan sebuah *unified buffer cache*. Tanpa sebuah *unified buffer cache*, kita mempunyai situasi panggilan *mapping* memori butuh menggunakan dua cache, *page cache* dan *buffer cache*. Karena sistem memori virtual tidak dapat menggunakan dengan *buffer cache*, isi dari berkas di dalam *buffer cache* harus diduplikat ke *page cache*. Situasi ini dikenal dengan *double caching* dan membutuhkan menyimpan data sistem-berkas dua kali. Tidak hanya membuang-buang memori, tetapi ini membuang CPU dan perputaran M/K dikerenakan perubahan data ekstra diantara memori sistem. Juga dapat menyebabkan korupsi berkas. Sebuah *unified buffer cache* mempunyai keuntungan menghindari *double caching* dan menunjuk sistem memori virtual untuk mengatur data sistem berkas.

Gambar 42.6. Tanpa *unified buffer cache*



42.4. *Recovery*

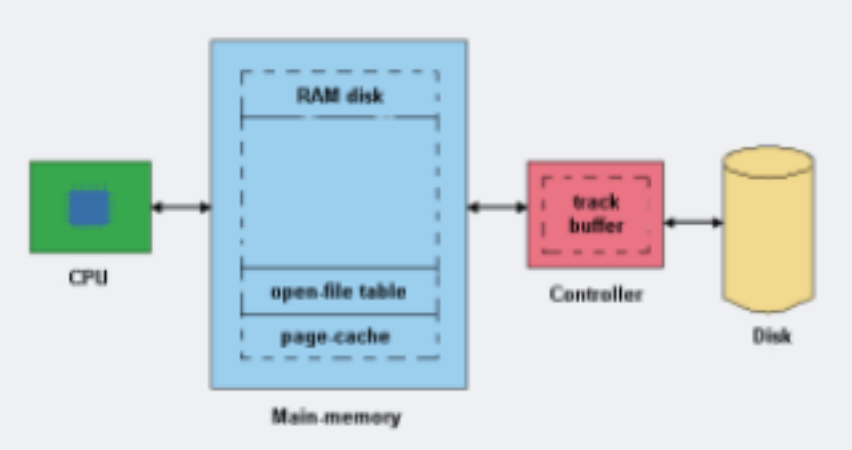
Sejak berkas-berkas dan direktori-direktori dua-duanya disimpan di memori utama dan pada disk, perawatan harus dilakukan untuk memastikan kegagalan sistem tidak terjadi di kehilangan data atau di tidakkonsistennya data.

Pengecekan Rutin

Informasi di direktori di memori utama biasanya lebih baru daripada informasi yang ada di disk, karena penulisan dari informasi direktori yang disimpan ke disk tidak terlalu dibutuhkan secepat

terjadinya pembaharuan. Mempertimbangkan efek yang memungkinkan terjadinya *crash* pada komputer. Secara berkala, program khusus akan dijalankan pada saat waktu *reboot* untuk mengecek dan mengoreksi disk yang tidak konsisten. Pemerikasaan rutin membandingkan data yang ada di struktur direktori dengan blok data pada disk, dan mencoba untuk memperbaiki ketidakkonsistenan yang ditemukan.

Gambar 42.7. Macam-macam lokasi *disk-caching*



Backup dan Restore

Dikarenakan disk magnetik kadang-kadang gagal, perawatan harus dijalankan untuk memastikan data tidak hilang selamanya. Oleh karena itu, program sistem dapat digunakan untuk *back up* data dari disk menuju ke media penyimpanan yang lainnya, seperti sebuah *floppy disk*, tape magnetik, atau disk optikal. *Recovery* dari kehilangan sebuah berkas individu, atau seluruh disk, mungkin menjadi masalah dari *restoring* data dari *backup*.

Untuk meminimalis kebutuhan untuk menduplikat, kita dapat menggunakan inforamsi dari, masing-masing masukan direktori. Sebagai contoh, jika program *backup* mengetahui kapan *backup* terakhir dari berkas telah selesai, dan tanggal terakhir berkas di direktori menunjukkan bahwa berkas tersebut tidak dirubah sejak tanggal tersebut, lalu berkas tersebut tidak perlu diduplikat lagi.

Sebuah tipe jadual *backup* yaitu sebagai berikut:

- Day 1:

  Menduplikat ke sebuah medium *back up* semua berkas ke disk. Ini disebut sebuah *full backup*.

- Day 2:

  Menduplikat ke medium lainnya semua berkas yang dirubah sejak hari pertama. Ini adalah *incremental backup*.

- Day 3:

  Menduplikat ke medium lainnya semua berkas yang dirubah sejak hari ke-2.

- Day N:

  Menduplikat ke medium lainnya semua berkas yang dirubah sejak hari ke N-1.

Perputaran baru dapat mempunyai *backupnya* ditulis ke semua set sebelumnya, atau ke set yang baru dari media *backup*. N yang terbesar, tentu saja memerlukan tape atau disk yang lebih untuk

dibaca untuk penyimpanan yang lengkap. Keuntungan tambahan dari perputaran *backup* ini adalah kita dapat menyimpan berkas apa saja yang tidak sengaja terhapus selama perputaran dengan mengakses berkas yang terhapus dari *backup* hari sebelumnya.

42.5. *Log-Structured File System*

Algoritma *logging* sudah dilakukan dengan sukses untuk manangani masalah dari pemeriksaan rutin. Hasil dari implementasinya dikenal dengan *log-based transaction-oriented* (atau *journaling* sistem berkas.

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

Ada beberapa masalah dengan adanya pendekatan dari menunjuk struktur untuk memechkan dan memperbaikinya pada *recovery*. Salah satunya adalah ketidakkonsistenan tidak dapat diperbaiki. Pemeriksaan rutin mungkin tidak dapat untuk *recover* struktur tersebut, yang hasilnya kehilangan berkas dan mungkin seluruh direktori.

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

42.6. Sistem Berkas Linux Virtual

Obyek dasar dalam layer-layer virtual file system

1. File

   File adalah sesuatu yang dapat dibaca dan ditulis. File ditempatkan pada memori. Penempatan pada memori tersebut sesuai dengan konsep file deskriptor yang dimiliki unix.

2. Inode

   Inode merepresentasikan obyek dasar dalam file sistem. Inode bisa saja file biasa, direktori, simbolik link dan lain sebagainya. Virtual file sistem tidak memiliki perbedaan yang jelas di antara obyek, tetapi mengacu kepada implementasi file sistem yang menyediakan perilaku yang sesuai. Kernel tingkat tinggi menangani obyek yang berbeda secara tidak sama. File dan inode hampir mirip diantara keduanya. Tetapi terdapat perbedaan yang penting diantara keduanya. Ada sesuatu yang memiliki inode tetapi tidak memiliki file, contohnya adalah simbolik link. Ada juga file yang tidak memiliki inode seperti pipes dan socket.

3. File sistem

   File system adalah kumpulan dari inode-inode dengan satu inode pembeda yaitu root. Inode lainnya diakses mulai dari root inode dan pencarian nama file untuk menuju ke inode lainnya. File sistem mempunyai beberapa karakteristik yang mencakup seluruh inode dalam file sistem. Salah satu yang terpenting adalah blocksize.

4. Nama inode

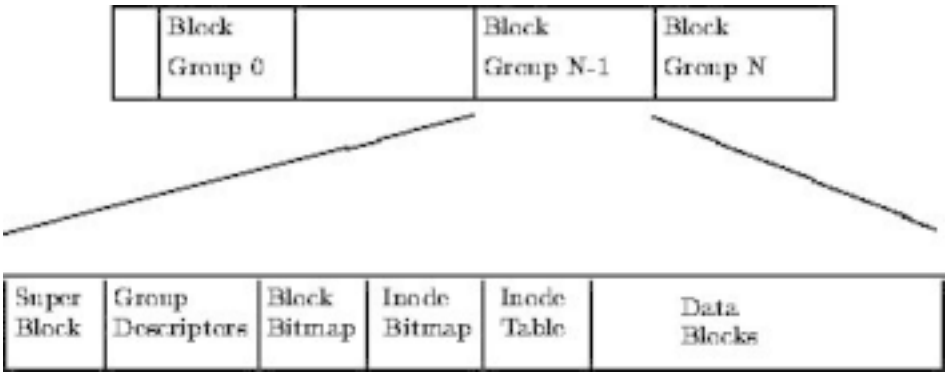
   Semua inode dalam file sistem diakses melalui namanya. Walaupun pencarian nama inode bisa menjadi terlalu berat untuk beberapa sistem, virtual file sistem pada linux tetap memantau cache dan nama inode yang baru saja terpakai agar kinerja meningkat. Cache terdapat di memori sebagai tree, ini berarti jika sembarang inode dari file terdapat di dalam cache, maka

parent dari inode tersebut juga terdapat di dalam cache. Virtual file system layer menangani semua pengaturan nama path dari file dan mengubahnya menjadi masukan di dalam cache sebelum mengizinkan file sistem untuk mengaksesnya. Ada pengecualian pada target dari simbolik link, akan diakses file sistem secara langsung. File sistem diharapkan untuk menginterpretasikannya.

42.7. Operasi-operasi Dalam Inode

Linux menyimpan cache dari inode aktif maupun dari inode yang telah terakses sebelumnya. Ada dua path dimana inode ini dapat diakses. Yang pertama telah disebutkan sebelumnya, setiap entri dalam cache menunjuk pada suatu inode dan menjaga inode tetap dalam cache. Yang kedua melalui inode hash table. Setiap inode mempunyai alamat 8 bit sesuai dengan alamat dari file sistem superblok dan nomor inode. Inode dengan nilai hash yang sama kemudian dirangkai di doubly linked list. Perubahan pada cache melibatkan penambahan dan penghapusan entri-entri dari cache itu sendiri. Entri-entri yang tidak dibutuhkan lagi akan di unhash sehingga tidak akan tampak dalam pencarian berikutnya. Operasi diperkirakan akan mengubah struktur cache harus dikunci selama melakukan perubahan. Unhash tidak memerlukan semaphore karena ini bisa dilakukan secara atomik dalam kernel lock. Banyak operasi file memerlukan dua langkah proses. Yang pertama adalah melakukan pencarian nama di dalam direktori. Langkah kedua adalah melakukan operasi pada file yang telah ditemukan. Untuk menjamin tidak terdapatnya proses yang tidak kompatibel diantara kedua proses itu, setelah proses kedua, virtual file sistem protokol harus memeriksa bahwa parent entry tetap menjadi parent dari entri child-nya. Yang menarik dari cache locking adalah proses rename, karena mengubah dua entri dalam sekali operasi.

Gambar 42.8. Struktur Sistem Berkas EXT2. Sumber: . . .



42.8. Sistem Berkas Linux

Sistem Berkas EXT2

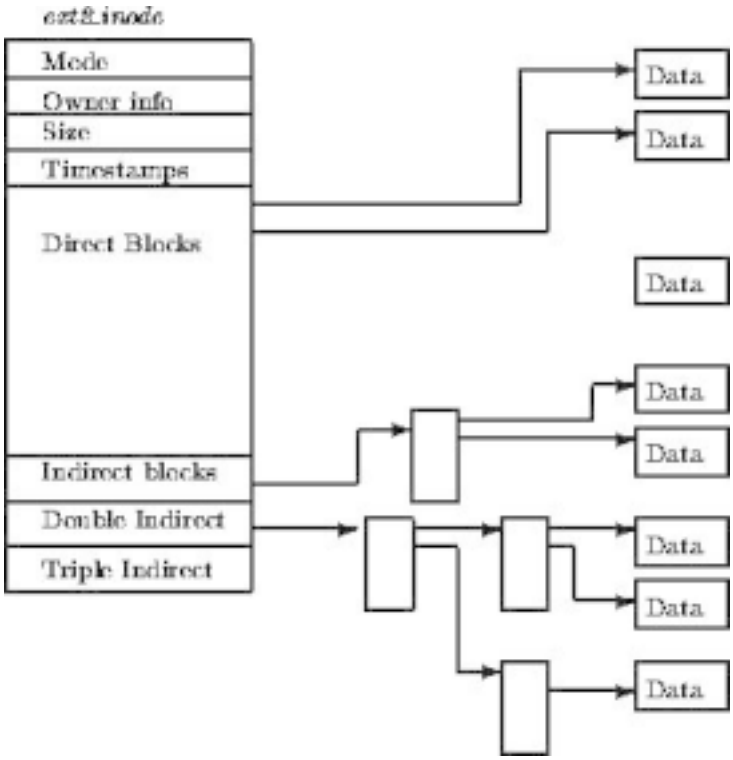
EXT2 adalah file sistem yang ampuh di linux. EXT2 juga merupakan salah satu file sistem yang paling ampuh dan menjadi dasar dari segala distribusi linux. Pada EXT2 file sistem, file data disimpan sebagai data blok. Data blok ini mempunyai panjang yang sama dan meski pun panjangnya bervariasi diantara EXT2 file sistem, besar blok tersebut ditentukan pada saat file sistem dibuat dengan perintah mk2fs. Jika besar blok adalah 1024 bytes, maka file dengan besar 1025 bytes akan memakai 2 blok. Ini berarti kita membuang setengah blok per file. EXT2 mendefinisikan topologi file sistem dengan memberikan arti bahwa setiap file pada sistem diasosiasiakan dengan struktur data inode. Sebuah inode menunjukkan blok mana dalam suatu file tentang hak akses setiap file, waktu modifikasi file, dan tipe file. Setiap file dalam EXT2 file sistem terdiri dari inode tunggal dan setiap inode mempunyai nomor identifikasi yang unik. Inode-inode file sistem disimpan dalam tabel inode. Direktori dalam EXT2 file sistem adalah file khusus yang mengandung pointer ke inode masing-masing isi direktori tersebut.

Inode adalah kerangka dasar yang membangun EXT2. Inode dari setiap kumpulan blok disimpan dalam tabel inode bersama dengan peta bit yang menyebabkan sistem dapat mengetahui inode mana yang telah teralokasi dana inode mana yang belum. MODE: mengandung dia informasi, inode apa dan izin akses yang dimiliki user. OWNER INFO: user atau grop yang memiliki file atau direktori SIZE: besar file dalam bytes TIMESTAMPS: kapan waktu pembuatan inode dan waktu terakhir dimodifikasi. DATABLOKS: pointer ke blok yang mengandung data. EXT2 inode juga dapat menunjuk pada device khusus, yang mana device khusus ini bukan merupakan file, tatapi dapat menangani program sehingga program dapat mengakses ke device. Semua file device di dalam drektori /dev dapat membantu program mengakses device.

Sistem Berkas EXT3

EXT3 adalah peningkatan dari EXT2 file sistem. Peningkatan ini memiliki beberapa keuntungan, diantaranya:

Gambar 42.9. Inode Sistem Berkas EXT2. Sumber: . . .



Setelah kegagalan sumber daya, "unclean shutdown", atau kerusakan sistem, EXT2 file sistem harus melalui proses pengecekan dengan program e2fsck. Proses ini dapat membuang waktu sehingga proses booting menjadi sangat lama, khususnya untuk disk besar yang mengandung banyak sekali data. Dalam proses ini, semua data tidak dapat diakses. Jurnal yang disediakan oleh EXT3 menyebabkan tidak perlu lagi dilakukan pengecekan data setelah kegagalan sistem. EXT3 hanya dicek bila ada kerusakan hardware seperti kerusakan hard disk, tetapi kejadian ini sangat jarang. Waktu yang diperlukan EXT3 file sistem setelah terjadi "unclean shutdown" tidak tergantung dari ukuran file sistem atau banyaknya file, tetapi tergantung dari besarnya jurnal yang digunakan untuk menjaga konsistensi. Besar jurnal default memerlukan waktu kira-kira sedetik untuk pulih, tergantung kecepatan hardware.

Integritas data. EXT3 menjamin adanya integritas data setelah terjadi kerusakan atau "unclean shutdown". EXT3 memungkinkan kita memilih jenis dan tipe proteksi dari data.

Kecepatan. Daripada menulis data lebih dari sekali, EXT3 mempunyai throughput yang lebih besar daripada EXT2 karena EXT3 memaksimalkan pergerakan head hard disk. Kita bisa memilih tiga

jurnal mode untuk memaksimalkan kecepatan, tetapi integritas data tidak terjamin.

Mudah dilakukan migrasi. Kita dapat berpindah dari EXT2 ke sistem EXT3 tanpa melakukan format ulang.

Sistem Berkas Reiser

Reiser file sistem memiliki jurnal yang cepat. Ciri-cirinya mirip EXT3 file sistem. Reiser file sistem dibuat berdasarkan balance tree yang cepat. Balance tree unggul dalam hal kinerja, dengan algoritma yang lebih rumit tentunya. Reiser file sistem lebih efisien dalam pemenfaatan ruang disk. Jika kita menulis file 100 bytes, hanya ditempatkan dalam satu blok. File sistem lain menempatkannya dalam 100 blok. Reiser file sistem tidak memiliki pengalokasian yang tetap untuk inode. Resier file sistem dapat menghemat disk sampai dengan 6 persen.

Sistem Berkas X

X file sistem juga merupakan jurnaling file sistem. X file sistem dibuat oleh SGI dan digunakan di sistem operasi SGI IRIX. X file sistem juga tersedia untuk linux dibawah lisensi GPL. X file sistem mengunakan B-tree untuk menangani file yang sangat banyak. X file sistem digunakan pada server-server besar.

Sistem Berkas Proc

Sistem Berkas Proc (Proc File Sistem) menunjukkan bagaimana hebatnya virtual file sistem yang ada pada linux. Proc file sistem sebenarnya tidak ada secara fisik, baik subdirektorinya, maupun file-file yang ada di dalamnya. Proc file sistem diregister oleh linux virtual file sistem, jika virtual file sistem memanggilnya dan meminta inode-inode dan file-file, proc file sistem membuat file tersebut dengan informasi yang ada di dalam kernel. Contohnya, /proc/devices milik kernel dibuat dari data struktur kernel yang menjelaskan device tersebut.

Sistem Berkas Web

Sistem Berkas Web (WFS) adalah sistem berkas Linux baru yang menyediakan sebuah sistem berkas seperti antarmuka untuk World Wide Web. Sistem ini dibangun sebagai modul kernel untuk Linux Kernel 2.2.1, dan meng-utilisasi proses level user (web daemon) untuk melayani permintaan pengambilan dokumen HTTP. Sistem berkas ini menyediakan fasilitas caching untuk dokumen remote, dan dapat memproses permintaan-permintaan terkenal multiple secara konkuren. Ketika suatu dokumen remote diambil, isi yang berada dalam hyperlink dalam dokumen tersebut diekstrak dan dipetakan kedalam sistem berkas local. Informasi ini direktori remote ini dibuat untuk setiap direktori yang diatur oleh WFS dalam sebuah file khusus. Utility lsw digunakan untuk daftar dan mengatur ini direktori remote. Partisi yang diatur WFS bersifat read-only bagi client WFS. Namun client dapat menyegarkan entri dari partisi WFS dengan menggunakan utility khusus rwm. Suatu studi unjuk kerja memperlihatkan bahwa WFS lebih kurang 30% lebih lambat daripada AFS untuk penelusuran akses berkas yang berisi 100% cache miss. Unjuk kerja yang lebih rendah ini kemungkinan karena antara lain jumlah proses yang lebih besar dilakukan dalam proses user dalam WFS, dan karena penggunaan general HTTP library untuk pengambilan dokumen.

Sistem Berkas Transparent Cryptographic (TCFS)

TCFS adalah sebuah sistem berkas terdistribusi. Sistem ini mengizinkan akses berkas sensitif yang disimpan dalam sebuah server remote dengan cara yang aman. Sistem ini mengatasi eavesdropping dan data tampering baik pada server maupun pada jaringan dengan menggunakan enkripsi dan message digest. Aplikasi mengakses data pada sistem berkas TCFS ini menggunakan system call regular untuk mendapatkan transparency yang lengkap bagi pengguna.

Sistem Berkas Steganographic

Sistem Berkas Cryptographic menyediakan sedikit perlindungan terhadap instrumen-instrumen legal atau pun ilegal yang memaksa pemilik data untuk melepaskan kunci deskripsinya demi data yang

disimpan saat hadirnya data terenkripsi dalam sebuah komputer yang terinfeksi. Sistem Berkas Cryptographic dapat diperluas untuk menyediakan perlindungan tambahan untuk scenario di atas dan telah diperluas sistem berkas Linux (ext2fs) dengan sebuah fungsi enkripsi yang plausible-deniability. Walaupun nyata bahwa komputer kita mempunyai software enkripsi hardisk yang sudah terinstal dan mungkin berisi beberapa data terenkripsi, sebiah inspector tetap akan dapat untuk menentukan apakah kita memberikan kunci akses kepada semua level keamanan atau terbatas. Implementasi ini disebut sistem berkas Steganographic.

42.9. Pembagian Sistem Berkas Secara Ortogonal

Shareable dan Unshareable

1. Shareable. Isinya dapat dishare (digunakan bersama) dengan sistem lain, gunanya untuk menghemat tempat.

2. Unshareable. Isinya tidak dapat dishare(digunakan bersama) dengan sistem lain, biasanya untuk alasan keamanan.

Variabel dan Statik

1. Variabel. Isinya sering berubah-ubah.

2. Statik. Sekali dibuat, kecil kemungkinan isinya akan berubah. Bisa berubah jika ada campur tangan sistem admin.

42.10. Rangkuman

Informasi yang disimpan di dalam suatu berkas harus disimpan ke dalam disk. Artinya, sistem operasi harus memutuskan tempat informasi itu akan disimpan. Ada 3 method untuk menentukan bagaimana sistem operasi menyimpan informasi ke disk yakni managemen ruang kosong (mengetahui seberapa luang kapasitas disk yang tersedia), efisiensi dan kinerja, dan *recovery*.

Salah satu tujuan OS adalah menyembunyikan kerumitan device hardware dari sistem penggunanya. Contohnya, Sistem Berkas Virtual menyamakan tampilan sistem berkas yang dimount tanpa memperdulikan devices fisik yang berada di bawahnya. Bab ini akan menjelaskan bagaimana kernel Linux mengatur device fisik di sistem.

Salah satu fitur yang mendasar adalah kernel mengabstraksi penanganan device. Semua device hardware terlihat seperti berkas pada umumnya: mereka dapat dibuka, ditutup, dibaca, dan ditulis menggunakan calls sistem yang sama dan standar untuk memanipulasi berkas. Setiap device di sistem direpresentasikan oleh sebuah file khusus device, contohnya disk IDE yang pertama di sistem direpresentasikan dengan /dev/hda. Devices blok (disk) dan karakter dibuat dengan perintah mknod dan untuk menjelaskan device tersebut digunakan nomor devices besar dan kecil. Devices jaringan juga direpresentasikan dengan berkas khusus device, tapi berkas ini dibuat oleh Linux setelah Linux menemukan dan menginisialisasi pengontrol-pengontrol jaringan di sistem. Semua device yang dikontrol oleh driver device yang sama memiliki nomor device besar yang umum. Nomor devices kecil digunakan untuk membedakan antara device-device yang berbeda dan pengontrol-pengontrol mereka, contohnya setiap partisi di disk IDE utama punya sebuah nomor device kecil yang berbeda. Jadi, /dev/hda2, yang merupakan partisi kedua dari disk IDE utama, punya nomor besar 3 dan nomor kecil yaitu 2. Linux memetakan berkas khusus device yang diteruskan ke system call (katakanlah melakukan mount ke sistem berkas device blok) pada driver si device dengan menggunakan nomor device besar dan sejumlah tabel sistem, contohnya tabel device karakter, chrdevs.

42.11. Latihan

1. Sebutkan 3 metode yang sering digunakan untuk mengalokasikan berkas?
2. Bandingkan masalah-masalah yang dapat terjadi di metode alokasi *contiguous allocation*
3. Bandingkan masalah-masalah yang dapat terjadi di metode alokasi *contiguous allocation* dan *linked allocation*?
4. Sebutkan 3 contoh mekanisme dari *Indexed Allocation*?
5. Jelaskan dengan singkat mengenai *Combined Scheme*!
6. Sebutkan akses berkas yang didukung oleh sistem yang menggunakan metode alokasi *contiguous allocation* dan *linked allocation*?
7. Jika ruang kosong di *bit map* sebagai berikut: 00111011101100010001110011 maka blok mana saja yang kosong?
8. Sebutkan keuntungan dari M/K yang menggunakan *unified buffer cache*?
9. Jelaskan cara membuat *backup* data!
10. Solusi apa yang diberikan untuk memastikan adanya *recovery* setelah adanya *system crash*?
11. 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.

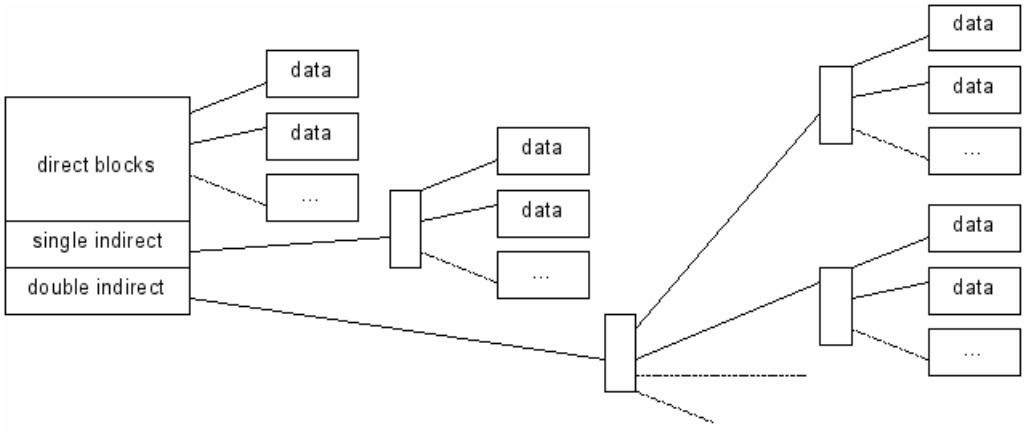
    e. Terangkan/berikan ilustrasi sebuah direktori yang shareable dan variable.

    f. Terangkan/berikan ilustrasi sebuah direktori yang unshareable dan static.

    g. Terangkan/berikan ilustrasi sebuah direktori yang unshareable dan variable.

12. Sistem Berkas II

    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 masingmasing 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?

13. Sistem Berkas III

a) Terangkan persamaan dan perbedaan antara operasi dari sebuah sistem direktori dengan operasi dari sebuah sistem 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).

14. 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!

42.12. Rujukan

Silberschatz, Galvin, Gagne. 2002. *Operating System Concepts, 6th ed*. John Wiley & Sons.

Tananbaum, Andrew S. 1992. *Modern Operating System 2nd ed*. Engrewood cliffs, New Jersey: Prentice Hall Inc.

Stallings, Williem. 2000. *Operating System 4th ed*. Prentice Hall.

http://infocom.cqu.edu.au/Courses/aut2001/85349/Resources/ Study_Guide/10.pdf

http://www.cs.utah.edu/classes/cs5460/lectures/lecture19- 2up.pdf

http://support.sitescape.com/ forum/ support/ dispatch.cgi/ _help/ showHelp/ page/ help/ en/ webfiles_tabs/ share_files.html

>http://www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690/ IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm

http://www.atnf.csiro.au/people/rgooch/linux/docs/vfs.txt

Bagian VII. Masukan/Keluaran (M/K)

Daftar Isi

Bab 43. Perangkat Keras Keluaran/Masukan

Menurut Silberschatz et. al. [Silberschatz2002], salah satu tujuan utama dari suatu sistem operasi ialah mengatur semua perangkat M/K (Masukan/Keluaran) atau I/O (*Input*/*Output*) komputer. Sistem operasi harus dapat memberikan perintah ke perangkat-perangkat itu, menangkap dan menangani interupsi, dan menangani *error* yang terjadi. Sistem operasi juga harus menyediakan antarmuka antara sistem operasi itu sendiri dengan keseluruhan sistem itu yang sederhana dan mudah digunakan. Antarmuka itu harus sama untuk semua perangkat (*device independent*), agar dapat dikembangkan.

Secara umum, terdapat beberapa jenis perangkat M/K, seperti perangkat penyimpanan (*disk*, *tape*), perangkat transmisi (*network card*, modem), dan perangkat antarmuka dengan pengguna (*screen*, *keyboard*, *mouse*). Perangkat tersebut dikendalikan oleh instruksi M/K. Alamat-alamat yang dimiliki oleh perangkat akan digunakan oleh *direct I/O instruction* dan *memory-mapped I/O*.

Beberapa konsep yang umum digunakan ialah *port*, bus (*daisy chain*/*shared direct access*), dan pengendali (*host adapter*). *Port* ialah koneksi yang digunakan oleh perangkat untuk berkomunikasi dengan mesin. Bus ialah koneksi yang menghubungkan beberapa perangkat menggunakan kabel-kabel. Pengendali ialah alat-alat elektronik yang berfungsi untuk mengoperasikan *port*, bus, dan perangkat.

Langkah yang ditentukan untuk perangkat ialah *command-ready*, *busy*, dan *error*. *Host* mengeset *command-ready* ketika perintah telah siap untuk dieksekusi oleh pengendali. Pengendali mengeset *busy* ketika sedang mengerjakan sesuatu, dan men-*clear busy* ketika telah siap untuk menerima perintah selanjutnya. *Error* diset ketika terjadi kesalahan.

43.1. Perangkat M/K

Pendapat orang-orang mengenai M/K berbeda-beda. Seorang insinyur mungkin akan memandang perangkat keras M/K sebagai kumpulan chip-chip, kabel-kabel, catu daya, dan komponen fisik lainnya yang membangun perangkat keras ini. Seorang programer akan memandangnya sebagai antarmuka yang disediakan oleh perangkat lunak -- perintah yang diterima perangkat keras, fungsi yang dikerjakannya, dan *error* yang ditimbulkan.

Perangkat M/K dapat dibagi secara umum menjadi dua kategori, yaitu: perangkat blok (*block devices*), dan perangkat karakter (*character devices*). Perangkat blok menyimpan informasi dalam sebuah blok yang ukurannya tertentu, dan memiliki alamat masing-masing. Umumnya blok berukuran antara 512 bytes sampai 32.768 bytes. Keuntungan dari perangkat blok ini ialah mampu membaca atau menulis setiap blok secara independen. Disk merupakan contoh perangkat blok yang paling banyak digunakan.

Tipe lain perangkat M/K ialah perangkat karakter. Perangkat karakter mengirim atau menerima sebarisan karakter, tanpa menghiraukan struktur blok. Tipe ini tidak memiliki alamat, dan tidak memiliki kemampuan mencari (*seek*). *Printer* dan antarmuka jaringan merupakan contoh perangkat jenis ini.

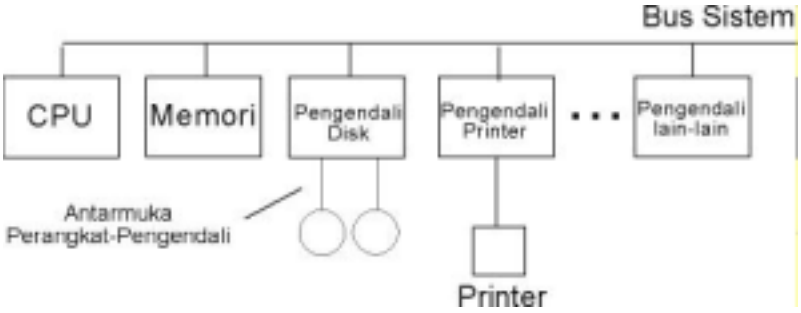
Pembagian ini tidaklah sempurna. Beberapa perangkat tidak memenuhi kriteria tersebut. Contohnya: *clock* yang tidak memiliki alamat dan juga tidak mengirim dan menerima barisan karakter. Yang ia lakukan hanya menimbulkan interupsi dalam jangka waktu tertentu.

43.2. Pengendali Perangkat

Unit M/K mengandung komponen mekanis dan elektronis. Komponen elektronis ini disebut pengendali perangkat (*device controllers*) atau adapter. Pada komputer personal (PC), komponen ini biasanya berupa kartu sirkuit yang dapat dimasukkan ke dalam slot pada *motherboard* komputer. Perangkat mekanis berupa perangkat itu sendiri.

Kartu pengendali biasanya memiliki sebuah penghubung. Beberapa pengendali dapat menangani dua, empat, atau bahkan delapan perangkat yang sejenis. Sistem operasi hampir selalu berhubungan dengan pengendali, bukan dengan perangkat secara langsung. Sebagian besar komputer yang berukuran kecil menggunakan model bus tunggal seperti pada Gambar 43.1, “Model Bus Tunggal” untuk berkomunikasi antara CPU dan pengendali. Sedangkan *mainframe* yang berukuran besar umumnya menggunakan model yang berbeda, dengan bus yang banyak dan *I/O channels*.

Gambar 43.1. Model Bus Tunggal

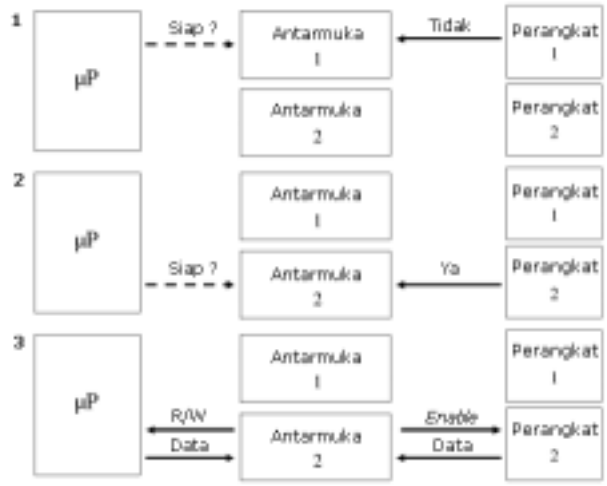


Sebuah model untuk menghubungkan CPU, memori, pengendali, dan perangkat M/K.

43.3. Polling

Busy-waiting/polling ialah ketika *host* mengalami *looping* yaitu membaca status *register* secara terus-menerus sampai status *busy* di-*clear*. Pada dasarnya *polling* dapat dikatakan efisien. Akan tetapi *polling* menjadi tidak efisien ketika setelah berulang-ulang melakukan *looping*, hanya menemukan sedikit perangkat yang siap untuk menservis, karena *CPU processing* yang tersisa belum selesai.

Gambar 43.2. Proses *Polling*



Contoh proses *polling* yang tipikal.

43.4. Interupsi

Mekanisme Dasar Interupsi

Ketika CPU mendeteksi bahwa sebuah pengendali telah mengirimkan sebuah sinyal ke *interrupt request line* (membangkitkan sebuah interupsi), CPU kemudian menjawab interupsi tersebut (juga disebut menangkap interupsi) dengan menyimpan beberapa informasi mengenai keadaan terkini CPU--contohnya nilai instruksi pointer, dan memanggil *interrupt handler* agar *handler* tersebut dapat melayani pengendali atau alat yang mengirim interupsi tersebut.

Fitur Tambahan pada Komputer Modern

Pada arsitektur komputer modern, tiga fitur disediakan oleh CPU dan pengendali interupsi (pada perangkat keras) untuk dapat menangani interrupsi dengan lebih bagus. Fitur-fitur ini antara lain ialah kemampuan menghambat sebuah proses penanganan interupsi selama proses berada dalam *critical state*, efisiensi penanganan interupsi sehingga tidak perlu dilakukan *polling* untuk mencari perangkat yang mengirimkan interupsi, dan fitur yang ketiga ialah adanya sebuah konsep interupsi multilevel sedemikian rupa sehingga terdapat prioritas dalam penanganan interupsi (diimplementasikan dengan *interrupt priority level system*).

Interrupt Request Line

Pada peranti keras CPU terdapat kabel yang disebut *interrupt request line*, kebanyakan CPU memiliki dua macam *interrupt request line*, yaitu *nonmaskable interrupt* dan *maskable interrupt*. *Maskable interrupt* dapat dimatikan/dihentikan oleh CPU sebelum pengeksekusian deretan *critical instruction* (*critical instruction sequence*) yang tidak boleh diinterupsi. Biasanya, interrupt jenis ini digunakan oleh pengendali perangkat untuk meminta pelayanan CPU.

Interrupt Vector dan *Interrupt Chaining*

Sebuah mekanisme interupsi akan menerima alamat *interrupt handling routine* yang spesifik dari sebuah set, pada kebanyakan arsitektur komputer yang ada sekarang ini, alamat ini biasanya berupa sekumpulan bilangan yang menyatakan offset pada sebuah tabel (biasa disebut vektor interupsi). Tabel ini menyimpan alamat-alamat *interrupt handler* spesifik di dalam memori. Keuntungan dari pemakaian vektor ialah untuk mengurangi kebutuhan akan sebuah interrupt handler yang harus mencari semua kemungkinan sumber interupsi untuk menemukan pengirim interupsi. Akan tetapi, vektor interupsi memiliki hambatan karena pada kenyataannya, komputer yang ada memiliki perangkat (dan *interrupt handler*) yang lebih banyak dibandingkan dengan jumlah alamat pada vektor interupsi. Karena itulah, digunakan teknik *interrupt chaining* setiap elemen dari vektor interupsi menunjuk pada elemen pertama dari sebuah daftar *interrupt handler*. Dengan teknik ini, *overhead* yang dihasilkan oleh besarnya ukuran tabel dan inefisiensi dari penggunaan sebuah *interrupt handler* (fitur pada CPU yang telah disebutkan sebelumnya) dapat dikurangi, sehingga keduanya menjadi kurang lebih seimbang.

Penyebab Interupsi

Interupsi dapat disebabkan berbagai hal, antara lain *exception*, *page fault*, interupsi yang dikirimkan oleh pengendali perangkat, dan *system call*. *Exception* ialah suatu kondisi dimana terjadi sesuatu, atau dari sebuah operasi didapat hasil tertentu yang dianggap khusus sehingga harus mendapat perhatian lebih, contohnya pembagian dengan 0 (nol), pengaksesan alamat memori yang *restricted* atau bahkan tidak valid, dan lain-lain. *System call* ialah sebuah fungsi pada aplikasi (perangkat lunak) yang dapat mengeksekusikan instruksi khusus berupa interupsi perangkat lunak atau *trap*.

43.5. *Direct Memory Access* (DMA)

DMA ialah sebuah prosesor khusus (*special purpose processor*) yang berguna untuk menghindari pembebanan CPU utama oleh program M/K (PIO).

Untuk memulai sebuah transfer DMA, *host* akan menuliskan sebuah DMA *command block* yang berisi *pointer* yang menunjuk ke sumber transfer, *pointer* yang menunjuk ke tujuan transfer, dan jumlah byte yang ditransfer, ke memori. CPU kemudian menuliskan alamat *command block* ini ke pengendali DMA, sehingga pengendali DMA dapat kemudian mengoperasikan bus memori secara langsung dengan menempatkan alamat-alamat pada bus tersebut untuk melakukan transfer tanpa bantuan CPU.

Tiga langkah dalam transfer DMA:

1. Prosesor menyiapkan DMA transfer dengan menyedia kan data-data dari perangkat, operasi yang akan ditampilkan, alamat memori yang menjadi sumber dan tujuan data, dan banyaknya byte yang ditransfer.

2. Pengendali DMA memulai operasi (menyiapkan bus, menyediakan alamat, menulis dan membaca data), sampai seluruh blok sudah di transfer.

3. Pengendali DMA meng-interupsi prosesor, dimana selanjutnya akan ditentukan tindakan berikutnya.

Pada dasarnya, DMA mempunyai dua metode yang berbeda dalam mentransfer data. Metode yang pertama ialah metode yang sangat baku dan sederhana disebut *HALT*, atau *Burst Mode DMA*, karena pengendali DMA memegang kontrol dari sistem bus dan mentransfer semua blok data ke atau dari memori pada *single burst*. Selagi transfer masih dalam prosres, sistem mikroprosessor di-set *idle*, tidak melakukan instruksi operasi untuk menjaga internal *register*. Tipe operasi DMA seperti ini ada pada kebanyakan komputer.

Metode yang kedua, mengikut-sertakan pengendali DMA untuk memegang kontrol dari sistem bus untuk jangka waktu yang lebih pendek pada periode dimana mikroprosessor sibuk dengan operasi internal dan tidak membutuhkan akses ke sistem bus. Metode DMA ini disebut *cycle stealing mode*. *Cycle stealing DMA* lebih kompleks untuk diimplementasikan dibandingkan HALT DMA, karena pengendali DMA harus mempunyai kepintaran untuk merasakan waktu pada saat sistem bus terbuka.

Handshaking

Proses *handshaking* antara pengendali DMA dan pengendali perangkat dilakukan melalui sepasang kabel yang disebut *DMA-request* dan *DMA-acknowledge*. Pengendali perangkat mengirimkan sinyal melalui *DMA-request* ketika akan mentransfer data sebanyak satu word. Hal ini kemudian akan mengakibatkan pengendali DMA memasukkan alamat-alamat yang dinginkan ke kabel alamat memori, dan mengirimkan sinyal melalui kabel *DMA-acknowledge*. Setelah sinyal melalui kabel *DMA-acknowledge* diterima, pengendali perangkat mengirimkan data yang dimaksud dan mematikan sinyal pada *DMA-request*.

Hal ini berlangsung berulang-ulang sehingga disebut *handshaking*. Pada saat pengendali DMA mengambil alih memori, CPU sementara tidak dapat mengakses memori (dihalangi), walaupun masih dapat mengaksees data pada cache primer dan sekunder. Hal ini disebut *cycle stealing*, yang walaupun memperlambat komputasi CPU, tidak menurunkan kinerja karena memindahkan pekerjaan data transfer ke pengendali DMA meningkatkan performa sistem secara keseluruhan.

Cara-cara Implementasi DMA

Dalam pelaksanaannya, beberapa komputer menggunakan memori fisik untuk proses DMA , sedangkan jenis komputer lain menggunakan alamat virtual dengan melalui tahap "penerjemahan" dari alamat memori virtual menjadi alamat memori fisik, hal ini disebut *Direct Virtual-Memory Address* atau DVMA. Keuntungan dari DVMA ialah dapat mendukung transfer antara dua memori *mapped device* tanpa intervensi CPU.

43.6. Rangkuman

FIXME

43.7. Latihan

1. Gambarkan diagram dari *Interrupt Driven I/O Cycle*!
2. Sebutkan langkah-langkah dari transfer DMA!
3. Apakah perbedaan dari *pooling* dan interupsi?
4. Apakah yang dimaksud dengan proses *pooling*? Jelaskan!

43.8. Rujukan

FIXME

Bibliografi

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

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

[Tanenbaum1992] Andrew Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. First Edition. Edisi Pertama. Prentice-Hall.

Bab 44. Subsistem M/K Kernel

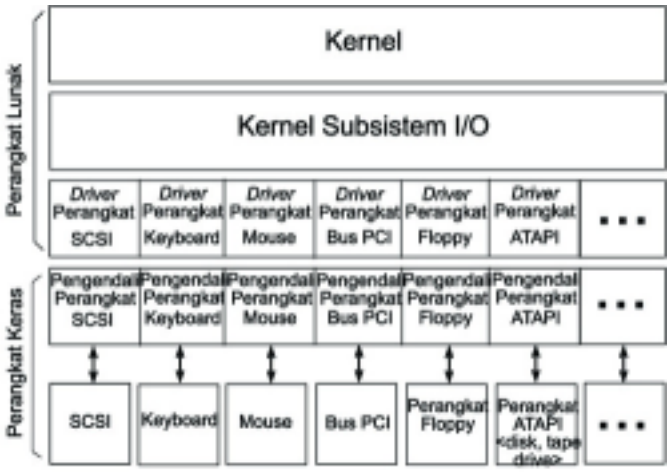
44.1. Aplikasi Antarmuka M/K

Bagian ini akan membahas bagaimana teknik dan struktur antarmuka yang memungkinkan M/K diperlakukan secara seragam. Salah satu contohnya adalah ketika suatu aplikasi ingin membuka data yang ada dalam suatu disk tanpa mengetahui jenis disk apa yang akan diaksesnya. Untuk mempermudah pengaksesan, sistem operasi melakukan standarisasi pengaksesan pada perangkat M/K. Pendekatan inilah yang dinamakan **aplikasi antarmuka M/K**.

Seperti layaknya permasalahan dari *software-engineering* yang rumit lainnya, aplikasi antarmuka M/K melibatkan abstraksi, enkapsulasi, dan *software layering*. Abstraksi dilakukan dengan membagi-bagi detail perangkat-perangkat M/K ke dalam kelas-kelas yang lebih umum. Dengan adanya kelas-kelas yang umum ini, maka akan lebih mudah bagi fungsi-fungsi standar (antarmuka) untuk mengaksesnya. Selanjutnya, keberadaan *device driver* pada masing-masing peralatan M/K akan berfungsi meng-enkapsulasi perbedaan-perbedaan yang ada dari setiap anggota kelas-kelas yang umum tadi.

Tujuan dari adanya lapisan *device driver* ini adalah untuk menyembunyikan perbedaan-perbedaan yang ada pada pengendali perangkat dari subsistem M/K yang terdapat dalam kernel. Dengan demikian, subsistem M/K dapat bersifat mandiri dari perangkat keras. Hal ini sangat menguntungkan dari segi pengembangan perangkat keras, karena tidak perlu menunggu vendor sistem operasi untuk mengeluarkan *support code* untuk perangkat-perangkat keras baru yang akan dikeluarkan oleh para vendor perangkat keras tersebut.

Gambar 44.1. Struktur Kernel



Gambar ini diadaptasi dari [Silberschatz2002, halaman 467].

Sayangnya untuk manufaktur perangkat keras, masing-masing sistem operasi memiliki standarnya sendiri untuk *device driver* antarmukanya. Karakteristik dari perangkat-perangkat tersebut sangat bervariasi, beberapa yang dapat membedakannya adalah dari segi:

1. ***Character-stream* atau *block***: Sebuah *stream* karakter memindahkan per satu *bytes*, sedangkan blok memindahkan sekumpulan *bytes* dalam 1 unit.

2. ***Sequential* atau *Random-access*:** Sebuah perangkat yang sekuensial memindahkan data dalam susunan yang sudah pasti seperti yang ditentukan oleh perangkat, sedangkan pengguna akses *random* dapat meminta perangkat untuk mencari ke seluruh lokasi penyimpanan data yang tersedia.

3. ***Synchronous* atau *asynchronous*:** perangkat yang *synchronous* menampilkan data-data transfer dengan waktu reaksi yang dapat diduga, sedangkan perangkat yang *asynchronous* menampilkan waktu reaksi yang tidak dapat diduga.

4. ***Sharable* atau *dedicated***: perangkat yang dapat dibagi dapat digunakan secara bersamaan oleh beberapa prosesor atau *thread*, sedangkan perangkat yang *dedicated*tidak dapat.

5. ***Speed of operation*:** Rentangan kecepatan perangkat dari beberapa bytes per detik sampai beberapa gigabytes per detik.

6. ***Read-write, read only*, atau *write only*:** Beberapa perangkat memungkinkan baik input-output dua arah, tapi beberapa lainnya hanya menunjang data satu arah.

Pada umumnya sistem operasi juga memiliki sebuah "*escape*" atau "pintu belakang" yang secara terbuka mengirim perintah yang *arbitrary* dari sebuah aplikasi ke *device driver*. Dalam UNIX, ada **ioctl()** yang memungkinkan aplikasi mengakses seluruh fungsi yang tersedia di *device driver* tanpa perlu membuat sebuah sistem *call* yang baru.

ioctl() ini mempunyai tiga argumen, yang pertama adalah sebuah pendeskripsi berkas yang menghubungkan aplikasi ke *driver* dengan menunjuk perangkat keras yang diatur oleh *driver* tersebut. Kedua, adalah sebuah integer yang memilih satu perintah yang terimplementasi di dalam *driver*. Ketiga, sebuah pointer ke struktur data *arbitrary* di memori, yang memungkinkan aplikasi dan *driver* berkomunikasi dengan data dan mengendalikan informasi data.

Peralatan Blok dan Karakter

Peralatan blok diharapkan dapat memenuhi kebutuhan akses pada berbagai macam *disk drive* dan juga peralatan blok lainnya, memenuhi/mengerti perintah baca, tulis dan juga perintah pencarian data pada peralatan yang memiliki sifat *random-access*.

Keyboard adalah salah satu contoh alat yang dapat mengakses *stream*-karakter. *System call* dasar dari antarmuka ini dapat membuat sebuah aplikasi mengerti tentang bagaimana cara untuk mengambil dan menuliskan sebuah karakter. Kemudian pada pengembangan lanjutannya, kita dapat membuat *library* yang dapat mengakses data/pesan baris demi baris.

Peralatan Jaringan

Karena adanya perbedaan dalam kinerja dan pengalamatan dari jaringan M/K, maka biasanya sistem operasi memiliki antarmuka M/K yang berbeda dari baca, tulis dan pencarian pada disk. Salah satu yang banyak digunakan pada sistem operasi adalah *socket interface*.

Socket berfungsi untuk menghubungkan komputer ke jaringan. *System call* pada *socket interface* dapat memudahkan suatu aplikasi untuk membuat *local socket*, dan menghubungkannya ke *remote socket*. Dengan menghubungkan komputer ke *socket*, maka komunikasi antar komputer dapat dilakukan.

Jam dan *Timer*

Adanya jam dan *timer* pada perangkat keras komputer, setidaknya memiliki tiga fungsi, memberi informasi waktu saat ini, memberi informasi lamanya waktu sebuah proses, sebagai *trigger* untuk suatu operasi pada suatu waktu. Fungsi-fungsi ini sering digunakan oleh sistem operasi. Sayangnya, *system call* untuk pemanggilan fungsi ini tidak distandarisasi antar sistem operasi.

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

Sistem operasi dapat mendukung lebih dari banyak *timer request* daripada banyaknya jumlah *timer hardware*. Dengan kondisi seperti ini, maka kernel atau *device driver* mengatur daftar dari interupsi

dengan urutan yang pertama kali datang akan dilayani terlebih dahulu.

Blocking dan *Nonblocking I/O*

Ketika suatu aplikasi menggunakan sebuah *blocking system call*, eksekusi aplikasi itu akan dihentikan sementara lalu dipindahkan ke *wait queue*. Setelah *system call* tersebut selesai, aplikasi tersebut dikembalikan ke *run queue*, sehingga pengeksekusiannya akan dilanjutkan. *Physical action* dari peralatan M/K biasanya bersifat *asynchronous*. Akan tetapi, banyak sistem operasi yang bersifat *blocking*, hal ini terjadi karena *blocking application* lebih mudah dimengerti dari pada *nonblocking application*.

44.2. Penjadualan M/K

Kernel menyediakan banyak layanan yang berhubungan dengan M/K. Pada bagian ini, kita akan mendeskripsikan beberapa layanan yang disediakan oleh subsistem kernel M/K, dan kita akan membahas bagaimana caranya membuat infrastruktur perangkat keras dan *device driver*. Layanan-layanan yang akan kita bahas adalah penjadualan M/K, *buffering*, *caching*, *spooling*, reservasi perangkat, *error handling*.

Menjadual sekumpulan permintaan M/K sama dengan menentukan urutan yang sesuai untuk mengeksekusi permintaan tersebut. Penjadualan dapat meningkatkan performa sistem secara keseluruhan, dapat membagi perangkat secara adil di antara proses-proses, dan dapat mengurangi waktu tunggu rata-rata untuk menyelesaikan operasi M/K.

Berikut adalah contoh sederhana untuk menggambarkan definisi di atas. Jika sebuah *arm disk* terletak di dekat permulaan disk, dan ada tiga aplikasi yang memblokir panggilan untuk membaca disk tersebut. Aplikasi pertama meminta sebuah blok dekat akhir disk, aplikasi kedua meminta blok yang dekat dengan awal, dan aplikasi tiga meminta bagian tengah dari disk. Sistem operasi dapat mengurangi jarak yang harus ditempuh oleh *arm disk* dengan melayani aplikasi tersebut dengan urutan 2, 3, 1. Pengaturan urutan pekerjaan kembali seperti ini merupakan inti dari penjadualan M/K.

Pengembang sistem operasi mengimplementasikan penjadualan dengan mengatur antrian permintaan untuk tiap perangkat. Ketika sebuah aplikasi meminta sebuah *blocking* sistem M/K, permintaan tersebut dimasukkan ke dalam antrian untuk perangkat tersebut. *Scheduler M/K* mengurutkan kembali antrian untuk meningkatkan efisiensi dari sistem dan waktu respon rata-rata yang harus dialami oleh aplikasi. Sistem operasi juga mencoba untuk bertindak secara adil agar tidak ada aplikasi yang menerima layanan yang lebih sedikit, atau dapat memberikan prioritas layanan untuk permintaan penting yang ditunda. Contohnya, pemintaan dari sub sistem mungkin akan mendapatkan prioritas lebih tinggi daripada permintaan dari aplikasi. Beberapa algoritma penjadualan untuk *I/O disk* akan dijelaskan pada bagian Penjadualan Disk.

Salah satu cara untuk meningkatkan efisiensi M/K sub sistem dari sebuah komputer adalah dengan mengatur operasi M/K tersebut. Cara lain adalah dengan menggunakan tempat penyimpanan pada memori utama atau pada disk, melalui teknik yang disebut *buffering*, *caching*, dan *spooling*.

44.3. *Buffering*

Buffer adalah area memori yang menyimpan data ketika mereka sedang dipindahkan antara dua perangkat atau antara perangkat dan aplikasi.

Tiga alasan melakukan buffering:

1. Mengatasi perbedaan kecepatan antara produsen dengan konsumen dari sebuah *stream* data.

   Contoh, sebuah berkas sedang diterima melalui *modem* dan akan disimpan di *hard disk*. Kecepatan *modem* tersebut ribuan kali lebih lambat daripada *hard disk*, sehingga *buffer* dibuat di dalam memori utama untuk mengumpulkan jumlah *byte* yang diterima dari *modem*. Ketika keseluruhan data di *buffer* sudah sampai, *buffer* tersebut dapat ditulis ke disk dengan operasi tunggal.

Karena penulisan disk tidak terjadi dengan seketika dan *modem* masih memerlukan tempat untuk menyimpan data yang berdatangan, maka dua buah *buffer* digunakan untuk melakukan operasi ini. Setelah *modem* memenuhi *buffer* pertama, akan terjadi permintaan untuk menulis di disk. *Modem* kemudian mulai memenuhi *buffer* kedua sementara *buffer* pertama dipakai untuk penulisan ke disk. Seiring *modem* sudah memenuhi *buffer* kedua, penulisan ke disk dari *buffer* pertama seharusnya sudah selesai, jadi *modem* akan berganti kembali memenuhi *buffer* pertama sedangkan *buffer* kedua dipakai untuk menulis. Metode *double buffering* ini membuat pasangan ganda antara produsen dan konsumen sekaligus mengurangi kebutuhan waktu diantara mereka.

2. Untuk menyesuaikan perangkat-perangkat yang mempunyai perbedaan dalam ukuran transfer data.

   Hal ini sangat umum terjadi pada jaringan komputer, dimana *buffer* dipakai secara luas untuk fragmentasi dan pengaturan kembali pesan-pesan yang diterima. Pada bagian pengirim, sebuah pesan yang besar akan dipecah ke dalam paket-paket kecil. Paket-paket tersebut dikirim melalui jaringan, dan penerima akan meletakkan mereka di dalam *buffer* untuk disusun kembali.

3. Untuk mendukung *copy semantics* untuk aplikasi M/K. Sebuah contoh akan menjelaskan apa arti dari *copy semantics*. Jika ada sebuah aplikasi yang mempunyai *buffer* data yang ingin dituliskan ke disk, aplikasi tersebut akan memanggil sistem penulisan, menyediakan *pointer* ke *buffer*, dan sebuah *integer* untuk menunjukkan ukuran *bytes* yang ingin ditulis. Setelah pemanggilan tersebut, apakah yang akan terjadi jika aplikasi tersebut merubah isi dari buffer?

   Dengan *copy semantics*, versi data yang ingin ditulis sama dengan versi data waktu aplikasi ini memanggil sistem untuk menulis, tidak tergantung dengan perubahan yang terjadi pada *buffer*. Sebuah cara sederhana untuk sistem operasi untuk menjamin *copy semantics* adalah membiarkan sistem penulisan untuk menyalin data aplikasi ke dalam *buffer kernel* sebelum mengembalikan kontrol kepada aplikasi. Jadi penulisan ke disk dilakukan pada *buffer kernel*, sehingga perubahan yang terjadi pada *buffer* aplikasi tidak akan membawa dampak apa-apa. Menyalin data antara *buffer kernel* data aplikasi merupakan sesuatu yang umum pada sistem operasi, kecuali *overhead* yang terjadi karena operasi *clean semantics*. Kita dapat memperoleh efek yang sama yang lebih efisien dengan memanfaatkan pemetaan virtual-memori dan proteksi *copy-on-wire* dengan lebih pintar.

44.4. *Caching*

Sebuah *cache* adalah daerah memori yang cepat yang berisikan data kopian. Akses ke sebuah kopian yang di-*cached* lebih efisien daripada akses ke data asli. Sebagai contoh, instruksi-instruksi dari proses yang sedang dijalankan disimpan ke dalam disk, dan ter-*cached* di dalam memori fisik, dan kemudian dikopi lagi ke dalam *cache secondary and primary* dari CPU. Perbedaan antara sebuah *buffer* dan *cache* adalah *buffer* dapat menyimpan satu-satunya informasi data sedangkan sebuah *cache* secara definisi hanya menyimpan sebuah data dari sebuah tempat untuk dapat diakses lebih cepat.

Caching dan *buffering* adalah dua fungsi yang berbeda, tetapi terkadang sebuah daerah memori dapat digunakan untuk keduanya. sebagai contoh, untuk menghemat *copy semantics* dan membuat penjadualan M/K menjadi efisien, sistem operasi menggunakan *buffer* pada memori utama untuk menyimpan data.

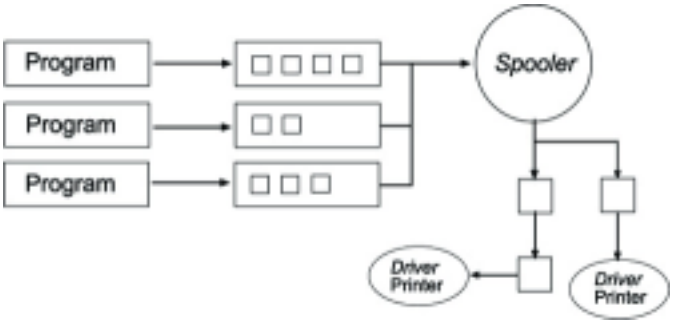
Buffer ini juga digunakan sebagai *cache*, untuk meningkatkan efisiensi IO untuk berkas yang digunakan secara bersama-sama oleh beberapa aplikasi, atau yang sedang dibaca dan ditulis secara berulang-ulang.

Ketika kernel menerima sebuah permintaan berkas M/K, kernel tersebut mengakses *buffer cache* untuk melihat apakah daerah memori tersebut sudah tersedia dalam memori utama. Jika sudah tersedia, sebuah *physical disk I/O* dapat dihindari atau bahkan tidak dipakai. Penulisan disk juga terakumulasi ke dalam *buffer cache* selama beberapa detik, jadi transfer yang besar akan dikumpulkan untuk mengefisiensikan jadual penulisan. Cara ini akan menunda penulisan untuk

meningkatkan efisiensi M/K akan dibahas pada bagian *Remote File Access*.

44.5. *Spooling* dan Reservasi Perangkat

Gambar 44.2. Spooling



Spooling adalah proses yang sangat berguna saat berurusan dengan perangkat M/K dalam sistem multiprogram. Sebuah *spool* adalah sebuah buffer yang menyimpan keluaran untuk sebuah perangkat yang tidak dapat menerima *interleaved data* streams. Salah satu perangkat *spool* yang paling umum adalah printer.

Printer hanya dapat melayani satu pekerjaan pada waktu tertentu, namun beberapa aplikasi dapat meminta printer untuk mencetak. *Spooling* memungkinkan keluaran mereka tercetak satu per satu, tidak tercampur. Untuk mencetak sebuah berkas, pertama-tama sebuah proses mengeneralisasi berkas secara keseluruhan untuk di cetak dan ditempatkan pada *spooling directory*. Sistem operasi akan menyelesaikan masalah ini dengan meng-*intercept* semua keluaran kepada printer. Tiap keluaran aplikasi sudah di-*spooled* ke disk berkas yang berbeda. Ketika sebuah aplikasi selesai mencetak, sistem *spooling* akan melanjutkan ke antrian berikutnya.

Di dalam beberapa sistem operasi, *spooling* ditangani oleh sebuah sistem proses *daemon*. Pada sistem operasi yang lain, sistem ini ditangani oleh *in-kernel thread*. Pada kedua penanganan tersebut, sistem operasi menyediakan antarmuka kontrol yang membuat *users* and sistem administrator dapat menampilkan antrian tersebut, untuk mengenyahkan antrian-antrian yang tidak diinginkan sebelum mulai dicetak.

Contoh lain adalah penggunaan *spooling* pada transfer berkas melalui jaringan yang biasanya menggunakan *daemon* jaringan. Untuk mengirim berkas ke suatu tempat, *user* menempatkan berkas tersebut dalam *spooling directory* jaringan. Selanjutnya, *daemon* jaringan akan mengambilnya dan mentransmisikannya. Salah satu bentuk nyata penggunaan *spooling* jaringan adalah sistim *email* via Internet. Keseluruhan sistem untuk mail ini berlangsung di luar sistem operasi.

Beberapa perangkat, seperti drive tape dan printer, tidak dapat me-*multiplex* permintaan M/K dari beberapa aplikasi. Selain dengan *spooling*, dapat juga diatasi dengan cara lain, yaitu dengan membagi koordinasi untuk *multiple concurrent* ini. Beberapa sistem operasi menyediakan dukungan untuk akses perangkat secara eksklusif, dengan mengalokasikan proses ke *device idle* dan membuang perangkat yang sudah tidak diperlukan lagi. Sistem operasi lainnya memaksakan limit suatu berkas untuk menangani perangkat ini. Banyak sistem operasi menyediakan fungsi yang membuat proses untuk menangani koordinat *exclusive akses* diantara mereka sendiri.

44.6. *Error Handling*

Sebuah sistem operasi yang menggunakan *protected memory* dapat menjaga banyak kemungkinan *error* akibat perangkat keras maupun aplikasi. Perangkat dan transfer M/K dapat gagal dalam banyak cara, dapat karena alasan transient, seperti *overloaded* pada jaringan, maupun alasan permanen yang seperti kerusakan yang terjadi pada *disk controller*. Sistem operasi seringkali dapat mengkompensasikan untuk kesalahan transient. Seperti, sebuah kesalahan baca pada disk akan mengakibatkan pembacaan ulang kembali dan sebuah kesalahan pengiriman pada jaringan akan mengakibatkan pengiriman ulang apabila protokolnya diketahui. Akan tetapi untuk kesalahan

permanen, sistem operasi pada umumnya tidak akan dapat mengembalikan situasi seperti semula.

Sebuah ketentuan umum, yaitu sebuah sistem M/K akan mengembalikan satu bit informasi tentang status panggilan tersebut, yang akan menandakan apakah proses tersebut berhasil atau gagal. Sistem operasi pada UNIX menggunakan *integer* tambahan yang dinamakan ***ERRNO*** untuk mengembalikan kode kesalahan sekitar 1 dari 100 nilai yang mengindikasikan sebab dari kesalahan tersebut. Sebaliknya, beberapa perangkat keras dapat menyediakan informasi kesalahan yang detail, walaupun banyak sistem operasi yang tidak mendukung fasilitas ini.

Sebagai contoh, kesalahan pada perangkat SCSI dilaporkan oleh protokol SCSI dalam bentuk ***sense key*** yang mengindentifikasi kesalahan yang umum seperti *error* pada perangkat keras atau permintaan yang ilegal; sebuah ***additional sense code*** yang mengkategorikan kesalahan yang muncul, seperti kesalahan parameter atau kesalahan *self-test*; dan sebuah ***additional sense code qualifier*** yang memberitahukan kesalahan secara lebih mendalam dan mendetil, seperti parameter yang error.

44.7. Struktur Data Kernel

Kernel membutuhkan informasi keadaan tentang penggunakan komponen M/K. *Kernel* menggunakan banyak struktur yang mirip untuk melacak koneksi jaringan, komunikasi perangkat karakter, dan aktivitas M/K lainnya.

UNIX menyediakan akses sistem berkas untuk beberapa entiti, seperti berkas pengguna, *raw devices*, dan alamat tempat proses. Walaupun tiap entiti ini didukung sebuah operasi baca, semantiknya berbeda untuk tiap entiti. Seperti untuk membaca berkas pengguna, kernel perlu memeriksa *buffer cache* sebelum memutuskan apakah akan melaksanakan *I/O disk*. Untuk membaca sebuah *raw disk*, kernel perlu untuk memastikan bahwa ukuran permintaan adalah kelipatan dari ukuran sektor disk, dan masih terdapat di dalam batas sektor. Untuk memproses citra, cukup perlu untuk mengkopi data ke dalam memori. UNIX mengkapsulasikan perbedaan-perbedaan ini di dalam struktur yang seragam dengan menggunakan teknik *object oriented*.

Beberapa sistem operasi bahkan menggunakan metode *object oriented* secara lebih ekstensif. Sebagai contoh, Windows NT menggunakan implementasi *message-passing* untuk M/K. Sebuah permintaan I/O akan dikonversikan ke sebuah pesan yang dikirim melalui kernel kepada M/K manager dan kemudian ke *device driver*, yang masing-masing dapat mengubah isi pesan. Untuk output, isi message adalah data yang akan ditulis. Untuk input, message berisikan *buffer* untuk menerima data. Pendekatan *message-passing* ini dapat menambah *overhead*, dengan perbandingan dengan teknik prosedural yang membagi struktur data, tetapi akan mennyederhanakan struktur dan design dari sistem M/K tersebut dan menambah fleksibilitas.

Kesimpulannya, subsistem M/K mengkoordinasi kumpulan-kumpulan service yang banyak sekali, yang tersedia dari aplikasi maupun bagian lain dari kernel. Subsistem M/K mengawasi:

1. Managemen nama untuk berkas dan perangkat.
2. Kontrol akses untuk berkas dan perangkat.
3. Kontrol operasi, contoh: model yang tidak dapat dikenali.
4. Alokasi tempat sistem berkas.
5. Alokasi perangkat.
6. *Buffering*, *caching*, *spooling*.
7. Penjadualan M/K
8. Mengawasi status perangkat, *error handling*, dan kesalahan dalam *recovery*.
9. Konfigurasi dan utilisasi *driver device*.

44.8. Penanganan Permintaan M/K

Di bagian sebelumnya, kita mendeskripsikan *handshaking* antara *device driver* dan pengendali perangkat, tapi kita tidak menjelaskan bagaimana Sistem Operasi menyambungkan permintaan aplikasi untuk menyiapkan jaringan menuju sektor disk yang spesifik.

Sistem Operasi yang modern mendapatkan fleksibilitas yang signifikan dari tahapan-tahapan tabel lookup di jalur diantara permintaan dan pengendali perangkat *physical*. Kita dapat mengenalkan perangkat dan *driver* baru ke komputer tanpa harus meng-compile ulang kernelnya. Sebagai fakta, ada beberapa sistem operasi yang mampu untuk me-*load device drivers* yang diinginkan. Pada waktu *boot*, sistem mula-mula meminta *bus* perangkat keras untuk menentukan perangkat apa yang ada, kemudian sistem me-*load* ke dalam *driver* yang sesuai; baik sesegera mungkin, maupun ketika diperlukan oleh sebuah permintaan M/K.

Sistem V UNIX mempunyai mekanisme yang menarik, yang disebut *streams*, yang membolehkan aplikasi untuk men-*assemble pipeline* dari kode *driver* secara dinamis. Sebuah *stream* adalah sebuah koneksi *full duplex* antara sebuah *device driver* dan sebuah proses user-level. Stream terdiri atas sebuah *stream head* yang merupakan antarmuka dengan user process, sebuah *driver end* yang mengontrol perangkat, dan nol atau lebih *stream modules* di antara mereka. *Modules* dapat didorong ke *stream* untuk menambah fungsionalitas di sebuah *layered fashion*. Sebagai gambaran sederhana, sebuah proses dapat membuka sebuah alat *port serial* melalui sebuah *stream*, dan dapat mendorong ke sebuah modul untuk memegang *edit* input. *Stream* dapat digunakan untuk interproses dan komunikasi jaringan. Faktanya, di Sistem V, mekanisme soket diimplementasikan dengan *stream*.

Berikut dideskripsikan sebuah *lifecycle* yang tipikal dari sebuah permintaan pembacaan blok:

1. Sebuah proses mengeluarkan sebuah *blocking read system call* ke sebuah berkas deskriptor dari berkas yang telah dibuka sebelumnya.

2. Kode *system-call* di kernel mengecek parameter untuk kebenaran. Dalam kasus input, jika data telah siap di *buffer cache*, data akan dikembalikan ke proses dan permintaan M/K diselesaikan.

3. Jika data tidak berada dalam *buffer cache*, sebuah physical M/K akan bekerja, sehingga proses akan dikeluarkan dari antrian jalan (*run queue*) dan diletakkan di antrian tunggu (*wait queue*) untuk alat, dan permintaan M/K pun dijadualkan. Pada akhirnya, subsistem M/K mengirimkan permintaan ke *device driver*. Bergantung pada sistem operasi, permintaan dikirimkan melalui *call* subrutin atau melalui pesan *in-kernel*.

4. *Device driver* mengalokasikan ruang *buffer* pada kernel untuk menerima data, dan menjadualkan M/K. Pada akhirnya, driver mengirim perintah ke pengendali perangkat dengan menulis ke *register device control*.

5. Pengendali perangkat mengoperasikan perangkat keras perangkat untuk melakukan transfer data.

6. *Driver* dapat menerima status dan data, atau dapat menyiapkan transfer DMA ke memori kernel. Kita mengasumsikan bahwa transfer diatur oleh sebuah *DMA controller*, yang menggunakan interupsi ketika transfer selesai.

7. *Interrupt handler* yang sesuai menerima interupsi melalui tabel vektor-interupsi, menyimpan sejumlah data yang dibutuhkan, menandai *device driver*, dan kembali dari interupsi.

8. *Device driver* menerima tanda, menganalisa permintaan M/K mana yang telah diselesaikan, menganalisa status permintaan, dan menandai subsistem M/K kernel yang permintaannya telah terselesaikan.

9. Kernel mentransfer data atau mengembalikan kode ke ruang alamat dari proses permintaan, dan memindahkan proses dari antrian tunggu kembali ke antrian siap.

10. Proses tidak diblok ketika dipindahkan ke antrian siap. Ketika penjadual (*scheduler*) mengembalikan proses ke CPU, proses meneruskan eksekusi pada penyelesaian dari *system call*.

44.9. *I/O Streams*

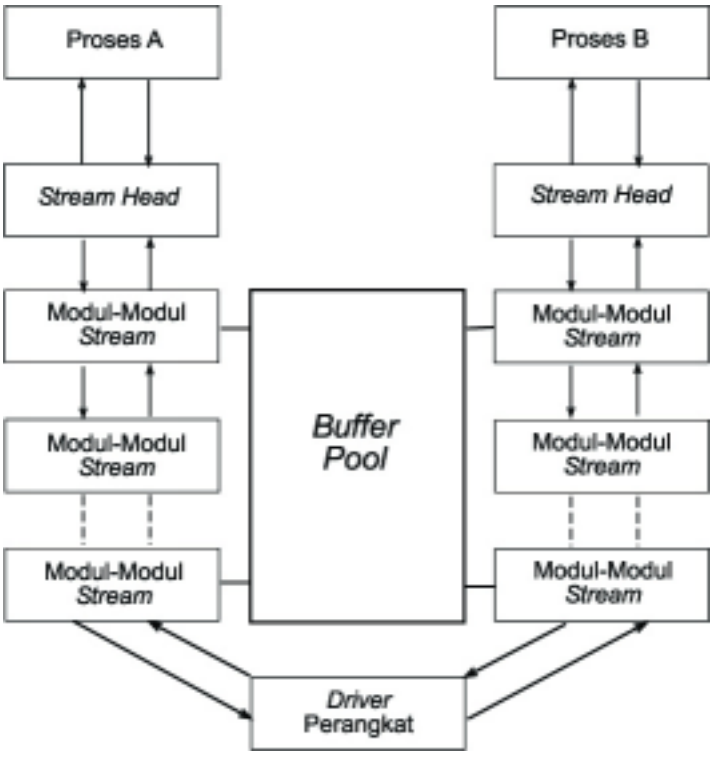
I/O stream adalah suatu mekanisme pengiriman data secara bertahap dan terus menerus melalui suatu aliran data dari proses ke peranti (begitu pula sebaliknya).

I/O Stream terdiri dari:

1. *stream head* yang berhubungan langsung dengan proses.
2. *driver ends* yang mengatur peranti-peranti
3. *stream modules* yang berada di antara *stream head* dan *driver end*, yang bertugas menyampaikan data ke *driver end* melalui *write queue*, maupun menyampaikan data ke proses melalui *read queue* dengan cara *message passing*.

Untuk memasukkan ke dalam stream digunakan *ioctl() system call*, sedangkan untuk menuliskan data ke peranti digunakan *write()*/ *putmsg() system calls*, dan untuk membaca data dari peranti digunakan *read()*/ *getmsg() system calls*.

Gambar 44.3. Struktur *Stream*



44.10. Kinerja M/K

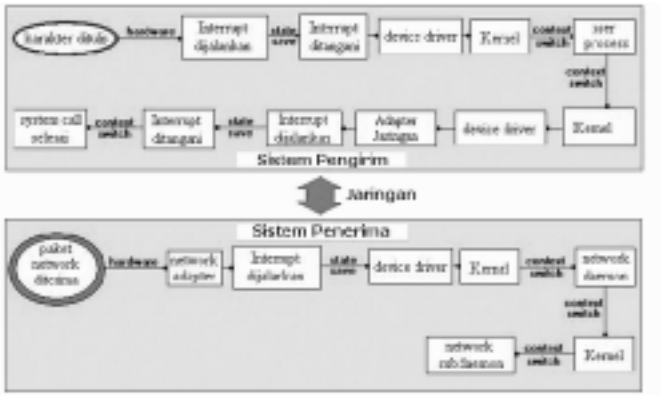
M/K adalah faktor penting dalam kinerja sistem. M/K sering meminta CPU untuk mengeksekusi *device-driver code* dan menjadual proses secara efisien sewaktu *memblock* dan *unblock*. Hasil *context switch* men-*stress* ke CPU dan *hardware cache*-nya. M/K juga memberitahukan ketidakefisienan mekanisme penanganan interupsi dalam kernel, dan M/K me-*load down memory bus* saat *data copy* antara pengendali dan memori fisik, dan juga saat *copy* antara *kernel buffers* dan *application data space*. Mengkopi dengan semua permintaan ini adalah salah satu kekhawatiran dalam arsitektur komputer.

Walaupun komputer modern dapat menangani beribu-ribu interupsi per detik, namun penanganan interupsi adalah pekerjaan yang sulit. Setiap interupsi mengakibatkan sistem melakukan perubahan status, mengeksekusi *interrupt handler* lalu mengembalikan statusnya kembali. M/K yang terprogram dapat lebih efisien dibanding *interrupt-driven I/O*, jika waktu *cycle* yang dibutuhkan untuk *busy-waiting* tidak berlebihan. M/K yang sudah selesai biasanya meng-*unblock* sebuah proses lalu membawanya ke *full overhead of context switch*.

Network traffic juga dapat menyebabkan *high context-switch rate*. Coba diperhatikan, misalnya sebuah *remote login* dari sebuah mesin ke mesin lainnya. Setiap karakter yang diketikkan pada *local machine* harus dikirim ke *remote machine*. Pada *local machine* karakter akan diketikkan, lalu *keyboard interrupt* dibuat, dan karakter melewati *interrupt handler* menuju *device-driver* lalu ke kernel, setelah itu ke proses. Proses memanggil *network I/O system call* untuk mengirim karakter ke *remote machine*. Karakter lalu melewati local kernel, menuju ke lapisan-lapisan network yang membuat paket network, lalu ke *network device driver*. *Network device driver* mengirim paket itu ke *network controller*, yang mengirim karakter dan membuat interupsi. Interupsi kemudian dikembalikan ke kernel supaya *I/O system call* dapat selesai.

Sekarang *remote system's network hardware* sudah menerima paket, dan interupsi dibuat. Karakter di-*unpack* dari *network protocol* dan dikirim ke *network daemon* yang sesuai. *Network daemon* mengidentifikasi *remote login session* mana yang terlibat, dan mengirim paket ke *subdaemon* yang sesuai untuk *session* itu. Melalui alur ini, ada *context switch* dan *state switch* (lihat Gambar 44.4, “Gambar Komunikasi Interkomputer”). Biasanya, penerima mengirim kembali karakter ke pengirim.

Gambar 44.4. Gambar Komunikasi Interkomputer



Gambar ini diadaptasi dari [Silberschatz2002, halaman 484].

Developer Solaris mengimplementasikan kembali telnet daemon menggunakan kernel-thread untuk menghilangkan *context switch* yang terlibat dalam pemindahan karakter dari daemon ke kernel. Sun memperkirakan bahwa perkembangan ini akan menambah jumlah maksimum *network logins* dari beberapa ratus hingga beberapa ribu (pada server besar).

Sistem lain menggunakan *front-end processor* yang terpisah untuk terminal M/K, supaya mengurangi beban interupsi pada *main CPU*. Misalnya, sebuah *terminal concentrator* dapat mengirim sinyal secara bersamaan dari beratus-ratus terminal ke satu port di *large computer*. Sebuah *I/O channel* adalah sebuah CPU yang memiliki tujuan khusus yang ditemukan pada mainframe dan pada *sistem high-end* lainnya. Kegunaan dari *I/O channel* adalah untuk meng-*offload I/O work* dari *main CPU*. Prinsipnya adalah *channel* tersebut menjaga supaya lalu lintas data lancar, sehingga main CPU dapat bebas memproses data. Seperti *device controller* dan *DMA controller* yang ada pada *smaller computer*, sebuah *channel* dapat memproses program-program yang umum dan kompleks, jadi *channel* dapat digunakan untuk *workload* tertentu.

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

1. Mengurangi *context switch*.

2. Mengurangi jumlah pengkopian data dalam memori sewaktu pengiriman antara peranti dan aplikasi.

3. Mengurangi jumlah interupsi dengan menggunakan transfer besar-besaran, *smart controller*, dan *polling* (jika *busy-waiting* dapat diminimalisir).

4. Menambah konkurensi dengan menggunakan pengendali atau *channel* DMA yang sudah diketahui untuk meng-*offload* kopi data sederhana dari CPU.

5. Memindahkan *processing primitives* ke perangkat keras, supaya operasi pada *device controller* konkuren dengan CPU dan operasi *bus*.

6. Keseimbangan antara CPU, *memory subsystem*, *bus* dan kinerja M/K, karena sebuah *overload* pada salah satu area akan menyebabkan keterlambatan pada yang lain.

Kompleksitas peranti berbeda-beda, misalnya *mouse*. *Mouse* adalah peranti yang sederhana. Pergerakan *mouse* dan *button click* diubah menjadi nilai numerik yang dikirim dari perangkat keras (melalui *mouse device driver*) menuju aplikasinya. Kebalikan dari *mouse*, fungsionalitas yang disediakan *NT disk device driver* sangatlah kompleks. *NT disk device driver* tidak hanya mengatur *individual disk*, tapi juga mengimplementasikan *RAID array*. Untuk dapat melakukannya, *NT disk device driver* mengubah *read* atau pun *write request* dari aplikasi menjadi *coordinated set of disk I/O operations*. Terlebih lagi, *NT disk device driver* mengimplementasikan penanganan error dan algoritma *data-recovery*, lalu mengambil langkah-langkah untuk mengoptimalkan kinerja disk, karena kinerja penyimpanan sekunder adalah hal penting untuk keseluruhan kinerja sistem.

Kapan fungsionalitas M/K dapat diimplementasikan? Pada *device hardware*, *device driver*, atau pada aplikasi perangkat lunak?

Mula-mula kita implementasikan eksperimen algoritma M/K pada *application level*, karena *application code* lebih fleksibel, dan *application bug* tidak membuat sistem *crash*. Terlebih lagi dengan mengembangkan kode pada *application level*, kita dapat menghindari *reboot* atau pun *reload device driver* setiap mengganti kode. Bagaimana pun juga sebuah implementasi pada *application level* dapat tidak efisien, karena *overhead of context switch*, dan karena aplikasi tidak dapat menerima kemudahan dari *internal kernel data structure* dan *fungsionalitas kernel* (seperti *internal kernel messaging*, *threading*, dan *locking* yang efisien).

Ketika algoritma *application level* memperlihatkan kegunaannya, kita dapat mengimplementasikan kembali kernel, sehingga dapat menambah kinerja. Akan tetapi, usaha pengembangan sulit dilakukan karena sistem operasi kernel adalah sistem perangkat lunak yang besar dan kompleks.

Terlebih lagi, dalam pengimplementasian internal kernel harus di-*debug* secara hati-hati untuk menghindari *data corrupt* dan sistem *crash*.

Kinerja tertinggi dapat didapatkan dengan cara implementasi spesial dalam perangkat keras, baik dalam peranti atau pun pengendali. Kerugian dari implementasi perangkat keras termasuk kesulitan dan pengorbanan dari membuat kemajuan atau dari pembetulan *bug*, dan bertambahnya *development time* (dalam satuan bulan, bukan hari), dan berkurangnya fleksibilitas.

Misalnya, sebuah *hardware RAID controller* mungkin saja tidak memberikan izin kepada kernel untuk mempengaruhi urutan atau pun lokasi dari *individual block reads and writes*, walaupun kernel memiliki informasi tertentu tentang *workload* yang mampu membuat kernel meningkatkan kinerja M/K.

44.11. Rangkuman

Subsistem kernel M/K menyediakan layanan yang berhubungan langsung dengan perangkat keras. Layanan Penjadualan M/K mengurutkan antrian permintaan pada tiap perangkat dengan tujuan untuk meningkatkan efisiensi dari sistem dan waktu respon rata-rata yang harus dialami oleh aplikasi.

Ada tiga alasan melakukan layanan Buffering, yaitu menyangkut perbedaan kecepatan produsen-konsumen, perbedaan ukuran transfer data dan dukungan copy semantics untuk aplikasi M/K. Fungsi buffering dan caching memiliki perbedaan dalam hal tujuan. Caching menyimpan salinan data asli pada area memori dengan tujuan agar bisa diakses lebih cepat, sedangkan buffering menyalin data asli agar dapat menyimpan satu-satunya informasi data.

Subsistem M/K mengkoordinasi kumpulan-kumpulan sevice yang banyak sekali, yang tersedia dari aplikasi atau bagian lain dari kernel. Penanganan permintaan M/K dilakukan dengan suatu mekanisme yang dideskripsikan sebagai sebuah life cycle.

Layanan I/O Streams menggunakan suatu mekanisme pengiriman data secara bertahap dan terus menerus melalui suatu aliran data dari piranti ke proses.

44.12. Latihan

1. Terangkan dengan singkat, pasangan konsep berikut ini. Terangkan pula perbedaan atau/dan persamaan pasangan konsep tersebut:
   - *I/O Data-Transfer Mode: "Character" vs. "Block".*
   - *I/O Access Mode: "Sequential" vs. "Random".*
   - *I/O Transfer Schedulle: "Synchronous" vs. "Asynchronous".*
   - *I/O Sharing: "Dedicated" vs. "Sharable".*
   - *I/O direction: "Read only" vs. "Write only".*
   - *"I/O Structure" vs. "Storage Structure".*
2. Apa hubungan arsitektur kernel yang di-thread dengan implementasi interupsi?
3. Mengapa antarmuka dibutuhkan pada aplikasi M/K?
4. Apa tujuan adanya *device driver*? Berikan contoh keuntungan pengimplementasiannya!
5. Jelaskan dengan singkat mengenai penjadualan M/K?
6. Apakah kegunaan *Streams* pada Sistem V UNIX?
7. Antarmuka M/K

   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.
8. M/K 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 M/K yang saling bertentangan (konflik).

e) Sebutkan dua aspek dari sub-sistem M/K kernel yang menjadi perhatian utama para perancang Sistem Operasi!

f) Bagaimana USB dapat mengatasi trend dan aspek tersebut di atas?

44.13. Rujukan

FIXME

Bibliografi

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

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

[Tanenbaum1992] Andrew Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. First Edition. Edisi Pertama. Prentice-Hall.

Bab 45. Managemen Disk I

45.1. Struktur *Disk*

Struktur *disk* merupakan suatu hal yang penting bagi penyimpanan informasi. Sistem komputer modern menggunakan *Disk* sebagai media penyimpanan sekunder. Dulu pita magnetik digunakan sebelum penggunaan *disk* sebagai media penyimpanan, sekunder yang memiliki waktu akses yang lebih lambat dari *disk*. Sejak digunakan *disk*, tape digunakan untuk backup, untuk menyimpan informasi yang tidak sering digunakan, sebagai media untuk memindahkan informasi dari satu sistem ke sistem lain, dan untuk menyimpan data yang cukup besar bagi sistem *disk*.

Bentuk penulisan *Disk* drive modern adalah *array* blok logika satu dimensi yang besar. Blok logika merupakan satuan unit terkecil dari transfer. Ukuran blok logika umumnya sebesar 512 bytes walaupun *disk* dapat diformat di level rendah (*low level formatted*) sehingga ukuran blok logika dapat ditentukan, misalnya 1024 bytes.

Array adalah blok logika satu dimensi yang dipetakan ke sektor dari *disk* secara sekuensial. Sektor 0 merupakan sektor pertama dari *track* pertama yang terletak di silinder paling luar (*outermost cylinder*). Proses pemetaan dilakukan secara berurut dari Sektor 0, lalu ke seluruh *track* dari silinder tersebut, lalu ke seluruh silinder mulai dari silinder terluar sampai silinder terdalam.

Kita dapat mengubah sebuah nomor blok logika dengan pemetaan menjadi sebuah alamat *disk* yang terdiri atas nomor silinder, nomor *track* di silinder tersebut, dan nomor sektor dari *track* tersebut. Dalam prakteknya, sulit untuk menerapkan pengubahan tersebut karena ada dua alasan. Pertama, kebanyakan *disk* memiliki sejumlah sektor yang rusak, tetapi pemetaan menyembunyikan hal ini dengan mensubstitusikan dengan sektor lain yang diambil dari suatu tempat di *disk*. Kedua, jumlah dari sektor tidak *track* tidak konstan. Pada media yang menggunakan ketentuan *Constant Linear Velocity* (CLV) kepadatan bit tiap *track* sama, jadi semakin jauh sebuah *track* dari tengah *disk*, semakin besar panjangnya, dan juga semakin banyak sektor yang dimilikinya. Trek di zona terluar memiliki 40% sektor lebih banyak dibandingkan dengan *track* di zona terdalam. Untuk menjamin aliran data yang sama, sebuah drive menaikan kecepatan putarannya ketika *disk head* bergerak dari zona luar ke zona dalam. Metode ini digunakan dalam CD-ROM dan DVD-ROM. Metode lain yang digunakan agar rotasi tetap konstan dan aliran data juga konstan dikenal dengan metode CAV (*Constant Angular Velocity*). CAV memungkinkan aliran data yang konstan karena kepadatan bit dari zona terdalam ke zona terluar semakin berkurang, sehingga dengan kecepatan rotasi yang konstan diperoleh aliran data yang konstan.

45.2. Penjadualan *Disk*

Penjadualan disk merupakan salah satu hal yang sangat penting dalam mencapai efisiensi perangkat keras. Bagi *disk* drives, efisiensi dipengaruhi oleh kecepatan waktu akses dan besarnya *disk bandwith*. Waktu akses memiliki dua komponen utama yaitu waktu pencarian dan waktu rotasi *disk* (*rotational latency*). Waktu pencarian adalah waktu yang dibutuhkan *disk arm* untuk menggerakkan *head* ke bagian silinder *disk* yang mengandung sektor yang diinginkan. Waktu rotasi *disk* adalah waktu tambahan yang dibutuhkan untuk menunggu perputaran *disk* agar *head* dapat berada di atas sektor yang diinginkan. *Disk bandwith* adalah total jumlah *bytes* yang ditransfer dibagi dengan total waktu dari awal permintaan transfer sampai transfer selesai. Kita dapat meningkatkan waktu akses dan *bandwidth* dengan menjadualkan permintaan dari M/K dalam urutan tertentu.

Apabila suatu proses membutuhkan pelayanan M/K dari atau menuju *disk*, maka proses tersebut akan melakukan *system call* ke sistem operasi. Permintaan tersebut membawa beberapa informasi, antara lain:

1. Apakah operasi *input* atau *output*.
2. Alamat *disk* untuk proses tersebut.
3. Alamat memori untuk proses tersebut

4. Jumlah *bytes* yang akan ditransfer

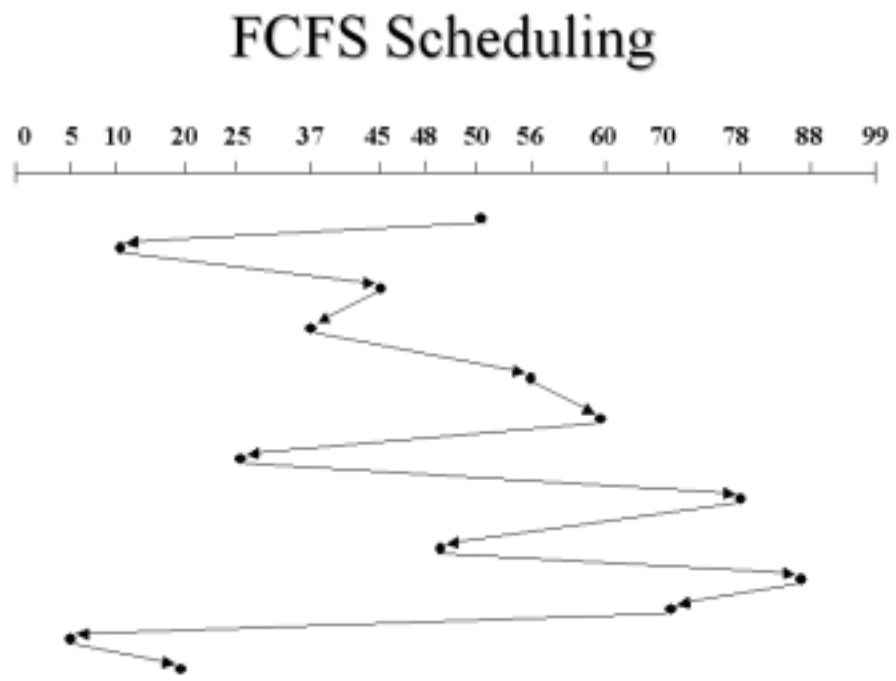
Pelayanan akan dilayani pada suatu proses apabila *disk drive* beserta pengendali tersedia untuk proses tersebut. Apabila *disk drive* dan pengendali sedang sibuk melayani proses lain, maka semua permintaan yang memerlukan pelayanan *disk* tersebut akan diletakkan pada suatu antrian permintaan untuk *disk* tersebut. Dengan demikian, jika suatu permintaan telah dilayani, maka sistem operasi melayani permintaan dari antrian berikutnya.

45.3. Penjadualan FCFS

Penjadualan *disk* FCFS melayani permintaan sesuai dengan antrian dari banyak proses yang meminta layanan. Secara umum algoritma FCFS ini sangat adil walaupun ada kelemahan dalam algoritma ini dalam hal kecepatannya yang lambat. Sebagai contoh, antrian permintaan pelayanan *disk* untuk proses M/K pada blok dalam silinder adalah sebagai berikut: 10, 45, 37, 56, 60, 25, 78, 48, 88, 70, 5, 20. Jika *head* pada awalnya berada pada 50, maka *head* akan bergerak dulu dari 50 ke 10, kemudian 45, 37, 56, 60, 25, 78, 48, 88, 70, 5 dan terakhir 20, dengan total pergerakan *head* sebesar 362 silinder.

Dari contoh diatas, kita dapat melihat permasalahan dengan menggunakan penjadualan jenis ini yaitu pergerakan dari 78 ke 48 dan kembali lagi ke 88. Jika permintaan terhadap silinder 88 dapat dilayani setelah permintaan 78, setelah selesai baru melayani permintaan 48, maka pergerakan total *head* dapat dikurangi, sehingga dengan demikian pendayagunaan akan meningkat.

Gambar 45.1. Penjadualan FCFS



Gambar ini diadaptasi dari [Silberschatz2002, halaman 494].

45.4. Penjadualan SSTF

Shortest-Seek-Time-First (SSTF) merupakan algoritma yang melayani permintaan berdasarkan waktu pencarian atau waktu pencarian paling kecil dari posisi *head* terakhir. Karena waktu pencarian meningkat seiring dengan jumlah silinder yang dilewati oleh *head*, maka SSTF memilih permintaan yang paling dekat posisinya di *disk* terhadap posisi *head* terakhir. Pergerakan dari

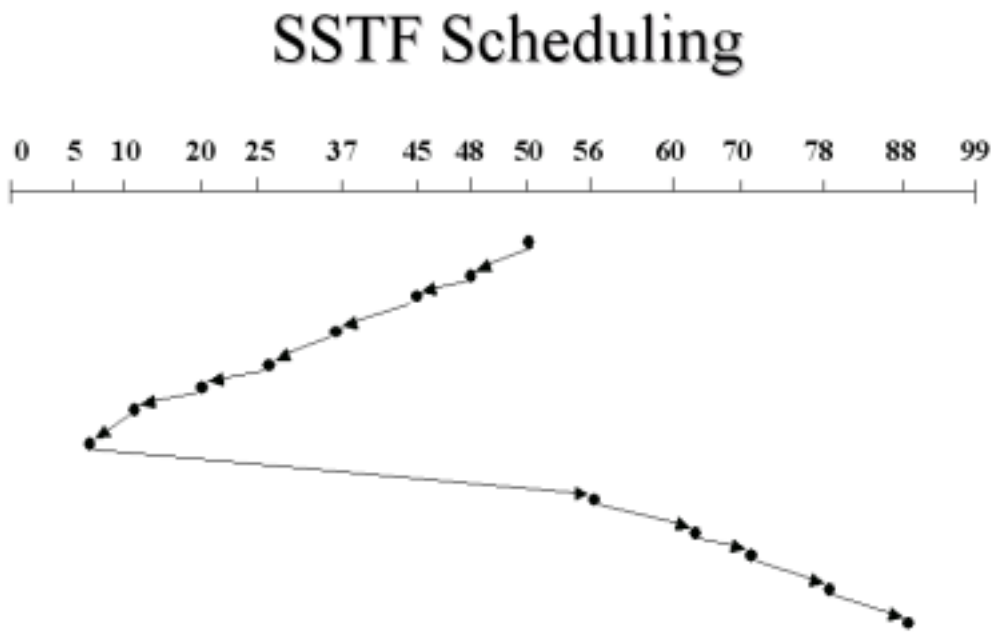
contoh diatas yaitu 50 ke 48, lalu ke 45, 37, 25, 20, 10, 5, 56, 60, 70, 78, 88.

Perhatikan contoh antrian permintaan yang kita sajikan pada penjadualan FCFS, permintaan paling dekat dengan posisi *head* saat itu (50) adalah silinder 48. Jika kita penuhi permintaan 48, maka yang terdekat berikutnya adalah silinder 45. Dari 45, silinder 37 letaknya lebih dekat ke 45 dibandingkan silinder 56,jadi 37 dilayani duluan. Selanjutnya, dilanjutkan ke silinder 25, 20, 10, 5, 56, 60, 70, 78 dan terakhir adalah 88.

Metode penjadualan ini hanya menghasilkan total pergerakan *head* sebesar 128 silinder -- kira-kira sepertiga dari yang dihasilkan penjadualan FCFS. Algoritma SSTF ini memberikan peningkatan yang cukup signifikan dalam hal pendayagunaan atau kinerja sistem.

Penjadualan SSTF merupakan salah satu bentuk dari penjadualan *shortest-job-first* (SJF), dan karena itu maka penjadualan SSTF juga dapat mengakibatkan *starvation* pada suatu saat tertentu. Hal ini dapat terjadi bila ada permintaan untuk mengakses bagian yang berada di silinder terdalam. Jika kemudian berdatangan lagi permintaan-permintaan yang letaknya lebih dekat dengan permintaan terakhir yang dilayani maka permintaan dari silinder terluar akan menunggu lama dan sebaliknya. Walaupun algoritma SSTF jauh lebih cepat dibandingkan dengan FCFS, namun untuk keadilan layanan SSTF lebih buruk dari penjadualan FCFS.

Gambar 45.2. Penjadualan SSTF



Gambar ini diadaptasi dari [Silberschatz2002, halaman 494]

45.5. Penjadualan SCAN

Pada algoritma ini *disk arm* bergerak menuju ke silinder paling ujung dari *disk*, kemudian setelah sampai di silinder paling ujung, *disk arm* akan berbalik arah geraknya menuju ke silinder paling ujung lainnya. Algoritma SCAN disebut juga Algoritma lift/ *elevator* karena algoritma ini cara kerjanya sama seperti algoritma yang umum dipakai oleh lift untuk melayani penggunanya, yaitu lift akan melayani orang-orang yang akan naik ke atas dulu, setelah sampai di lantai tertinggi, baru lift akan berbalik arah geraknya untuk melayani orang-orang yang akan turun. Dalam pergerakannya yang seperti lift itu, *disk arm* hanya dapat melayani permintaan-permintaan yang berada di depan arah geraknya terlebih dahulu. Bila ada permintaan yang berada di belakang arah geraknya, permintaan tersebut harus menunggu sampai *disk arm* mencapai salah satu silinder paling ujung dari

disk, kemudian berbalik arah geraknya untuk melayani permintaan tersebut.

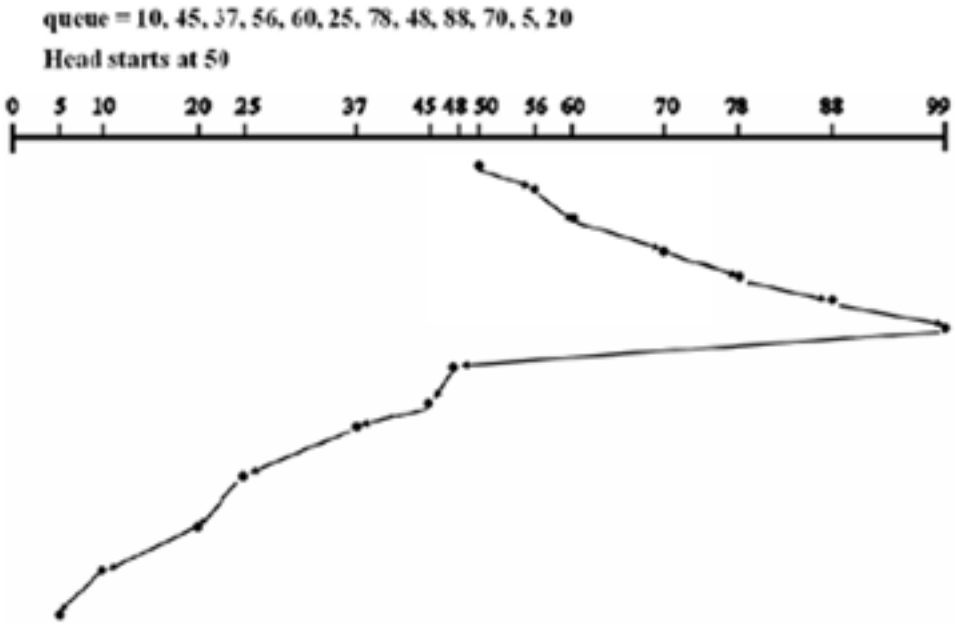
Contoh: (lihat Gambar 45.3, “Penjadualan SCAN”) Jika *disk head* sedang berada di silinder 50, dan sedang bergerak menuju silinder 99, maka permintaan yang dapat dilayani berikutnya adalah yang terdekat dengan silinder 50, tetapi masih berada di depan arah geraknya, yaitu: silinder 56. Begitu seterusnya *disk arm* melayani permintaan yang berada di depannya sampai *disk arm* mencapai silinder 99 dan berbalik arah gerak menuju ke silinder 0. Maka setelah *disk arm* berbalik arah gerak, permintaan di silinder 45 baru dapat dilayani.

Keunggulan dari algoritma SCAN adalah total pergerakan *disk arm* memiliki batas atas, yaitu 2 kali dari jumlah total silinder pada *disk*. Tetapi di samping itu masih ada beberapa kelemahan yang dimiliki oleh algoritma ini.

Dari contoh Gambar 45.3, “Penjadualan SCAN” terlihat salah satu kelemahan algoritma SCAN: permintaan di silinder 88 sebenarnya sudah merupakan permintaan yang paling ujung, tetapi *disk arm* harus bergerak sampai silinder 99 dulu, baru kemudian dapat berbalik arah geraknya. Bukankah hal seperti itu sangat tidak efisien? Mengapa *disk arm* tidak langsung berbalik arah geraknya sesudah sampai di silinder 88? Kelemahan ini akan dijawab oleh algoritma LOOK yang akan dibahas pada sub-bab berikutnya.

Kelemahan lain dari algoritma SCAN yaitu dapat menyebabkan terjadinya *starvation*. Begitu *disk arm* berbalik arah geraknya dari silinder 99, maka silinder yang berada dekat di depan arah gerak *disk arm* baru saja dilayani, sedangkan silinder-silinder yang dekat dengan silinder 0 sudah lama menunggu untuk dilayani. Bila kemudian bermunculan permintaan-permintaan baru yang dekat dengan silinder 99 lagi, maka permintaan-permintaan baru itulah yang akan dilayani, sehingga permintaan-permintaan yang dekat dengan silinder 0 akan semakin "lapar". Karena kelemahan yang kedua inilah muncul modifikasi dari algoritma SCAN, yaitu C-SCAN yang akan kita bahas berikutnya.

Gambar 45.3. Penjadualan SCAN



Gambar ini diadaptasi dari [Silberschatz2002, halaman 495]

45.6. Penjadualan C-SCAN

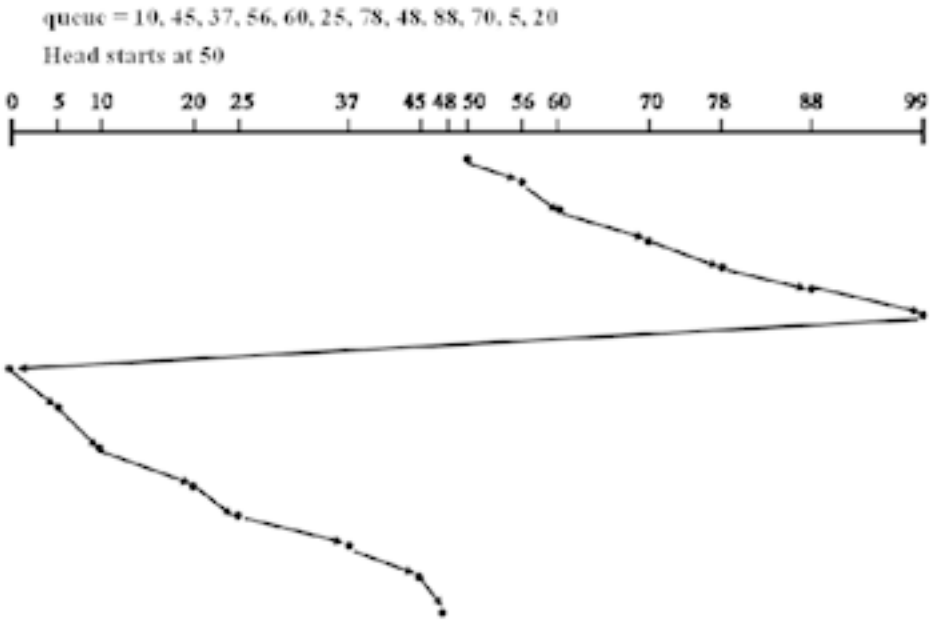
Algoritma Circular SCAN (C-SCAN) merupakan hasil modifikasi algoritma SCAN untuk mengurangi kemungkinan *starvation* yang dapat terjadi pada SCAN. Perbedaan C-SCAN dengan SCAN hanya pada bagaimana pergerakan *disk arm* setelah sampai ke salah satu silinder paling

ujung. Pada algoritma SCAN, *disk arm* akan berbalik arah menuju ke silinder paling ujung yang lain sambil tetap melayani permintaan yang berada di depan arah pergerakan *disk arm*, sedangkan pada algoritma C-SCAN sesudah mencapai silinder paling ujung, maka *disk arm* akan bergerak cepat ke silinder paling ujung lainnya tanpa melayani permintaan.

Contoh: (lihat Gambar 45.4, “Penjadualan C-SCAN”) Setelah sampai di silinder 99, *disk arm* akan bergerak dengan cepat ke silinder 0 tanpa melayani permintaan selama dalam perjalanannya. Kemudian setelah sampai di silinder 0, baru *disk arm* akan bergerak ke arah silinder 99 lagi sambil melayani permintaan.

Dengan pergerakan yang seperti demikian, seolah-olah *disk arm* hanya bergerak 1 arah dalam melayani permintaan. Tetapi dalam algoritma C-SCAN masih terkandung kelemahan yang juga dimiliki oleh algoritma SCAN, yaitu *disk arm* harus sampai di silinder 99 atau silinder 0 terlebih dahulu sebelum dapat berbalik arah. Untuk itulah dibuat algoritma LOOK yang akan kita bahas berikutnya.

Gambar 45.4. Penjadualan C-SCAN



Gambar ini diadaptasi dari [Silberschatz2002, halaman 496].

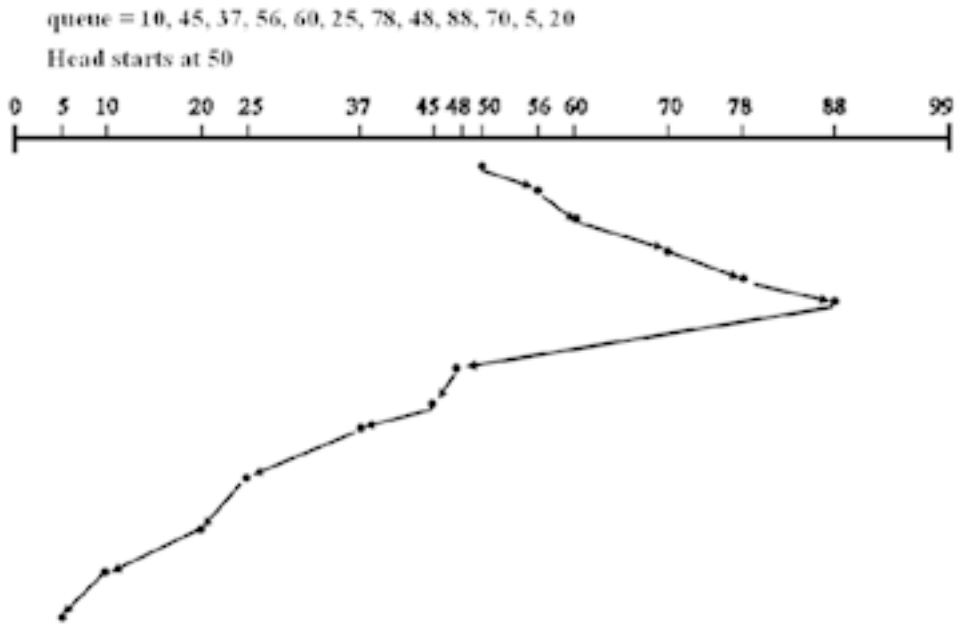
45.7. Penjadualan LOOK

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. Bila tidak ada lagi permintaan di depannya, *disk arm* dapat langsung berbalik arah geraknya. Penjadualan LOOK seperti SCAN yang lebih "pintar".

Contoh: (lihat Gambar 45.5, “Penjadualan LOOK”). Ketika *disk head* sudah selesai melayani permintaan di silinder 88, algoritma ini akan "melihat" bahwa ternyata di depan arah pegerakannya sudah tidak ada lagi permintaan yang harus dilayani. Oleh karena itu *disk arm* dapat langsung berbalik arah geraknya sehingga permintaan yang menunggu untuk dilayani dapat mendapatkan pelayanan lebih cepat.

Kelemahan algoritma ini sama seperti kelemahan algoritma SCAN bahwa dapat terjadi *starvation* untuk situasi yang sama pula dengan yang menyebabkan terjadinya *starvation*pada algoritma SCAN. Oleh karena itulah dibuat lagi suatu algoritma yang lebih baik untuk memperbaiki algoritma ini, yaitu: C-LOOK.

Gambar 45.5. Penjadualan LOOK

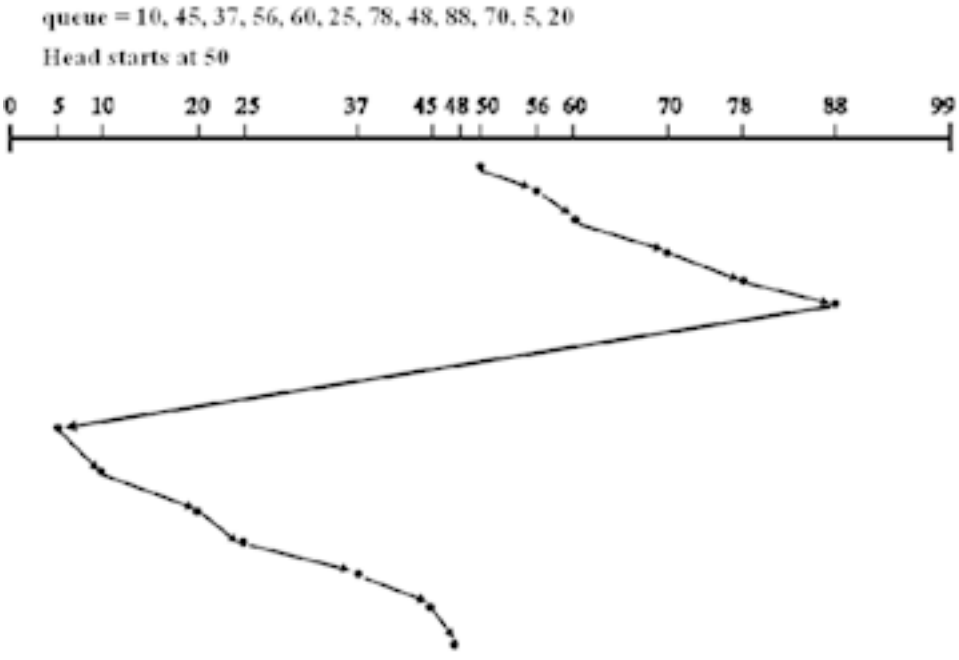


Gambar ini diadaptasi dari [Silberschatz2002, halaman 497].

45.8. Penjadualan C-LOOK

Algoritma ini berhasil memperbaiki kelemahan-kelemahan algoritma SCAN, C-SCAN, dan LOOK. Algoritma C-LOOK memperbaiki kelemahan LOOK sama seperti algoritma C-SCAN memperbaiki kelemahan SCAN, yaitu pada cara pergerakan *disk arm* setelah mencapai silinder yang paling ujung.

Gambar 45.6. Penjadualan C-LOOK



Gambar ini diadaptasi dari [Silberschatz2002, halaman 497].

Contoh: (lihat Gambar 45.6, “Penjadualan C-LOOK”) dengan memiliki kemampuan "melihat" algoritma LOOK, setelah melayani permintaan di silinder 88, *disk arm* akan bergerak dengan cepat ke silinder 5, yaitu permintaan di silinder yang terletak paling dekat dengan silinder 0.

Dengan cara pergerakan *disk arm* yang mengadaptasi keunggulan dari C-SCAN dan LOOK, algoritma ini dapat mengurangi terjadinya *starvation*, dengan tetap menjaga efektifitas pergerakan *disk arm*.

45.9. Pemilihan Algoritma Penjadualan *Disk*

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

Dengan algoritma penjadualan yang mana pun, kinerja sistem sangat tergantung pada jumlah dan tipe permintaan. Sebagai contoh, misalnya kita hanya memiliki satu permintaan, maka semua algoritma penjadualan akan dipaksa bertindak sama. Sedangkan permintaan sangat dipengaruhi oleh metode penempatan berkas. Karena kerumitan inilah, maka algoritma penjadualan *disk* harus ditulis dalam modul terpisah dari sistem operasi, jadi dapat saling mengganti dengan algoritma lain jika diperlukan.

Namun perlu diingat bahwa algoritma-algoritma di atas hanya mempertimbangkan jarak pencarian, sedangkan untuk *disk* modern, *rotational latency* dari *disk* sangat menentukan. Tetapi sangatlah sulit jika sistem operasi harus memperhitungkan algoritma untuk mengurangi *rotational latency* karena *disk* modern tidak memperlihatkan lokasi fisik dari blok-blok logikanya. Oleh karena itu para produsen *disk* telah mengurangi masalah ini dengan mengimplementasikan algoritma penjadualan *disk* di dalam pengendali perangkat keras, sehingga kalau hanya kinerja M/K yang diperhatikan, maka sistem operasi dapat menyerahkan algoritma penjadualan *disk* pada perangkat keras itu sendiri.

45.10. Rangkuman

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

1. Constant Linear Velocity (CLV)

   Kepadatan bit setiap track sama, semakin jauh sebuah track dari tengah disk, maka semakin besar panjangnya, dan juga semakin banyak sektor yang dimilikinya. Digunakan pada CD-ROM dan DVD-ROM.

2. Constant Angular Velocity (CAV)

   Kepadatan bit dari zona terdalam ke zona terluar semakin berkurang, kecepatan rotasi konstan sehingga aliran data pun konstan.

   Penjadualan disk sangat penting dalam hal meningkatkan efisiensi penggunaan perangkat keras. Efisiensi penggunaan disk terkait dengan kecepatan waktu akses dan besarnya disk bandwith. Untuk meningkatkan efisiensi tersebut dibutuhkan algoritma penjadualan yang tepat dalam penggunaan disk.

Terdapat berbagai macam algoritma penjadualan disk, yaitu:

1. FCFS (First Come First Serve)

2. SSTF (Shortest-Seek-Time-First)
3. SCAN
4. C-SCAN (Circular SCAN)
5. LOOK
6. C-LOOK (Circular LOOK)

45.11. Latihan

1. Andaikan suatu disk memiliki 100 silinder (silinder 0 - silinder 99), posisi *head* sekarang di silinder 25, sebelumnya *head* melayani silinder 13. Berikut ini adalah antrian silinder yang meminta layanan secara *FIFO*: 86, 37, 12, 90, 46, 77, 24, 48, 86, 65.

   Hitung total pergerakan head untuk memenuhi permintaan tersebut dimulai dari posisi *head* sekarang, dengan algoritma:

   a. *FCFS*

   b. *SSTF*

   c. *SCAN*

   d. *LOOK*

   e. *C-SCAN*

   f. *C-LOOK*

2. Jelaskan perbedaan, persamaan serta kelebihan dan kekurangan dari 2 perbandingan algoritma berikut:

   a. *FCFS* vs *SSTF*

   b. *SCAN* vs *C-SCAN*

   c. *LOOK* vs *C-LOOK*

   d. *SSTF* vs *SCAN*

45.12. Rujukan

FIXME

Bibliografi

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

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

[Tanenbaum1992] Andrew Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. First Edition. Edisi Pertama. Prentice-Hall.

Bab 46. Managemen Disk II

46.1. Komponen Disk

Beberapa aspek yang termasuk aspek penting dalam Managemen Disk.

Format Disk

Disk adalah salah satu tempat penyimpanan data. Sebelum sebuah disk dapat digunakan, disk harus dibagi-bagi dalam beberapa sektor. Sektor-sektor ini yang kemudian akan dibaca oleh pengendali. Pembentukan sektor-sektor ini disebut *low level formatting* atau *physical formatting*. *Low level formatting* juga akan mengisi disk dgn beberapa struktur data penting seperti *header* dan *trailer*. *Header* dan *trailer* mempunyai informasi seperti nomor sektor, dan *Error Correcting Code* (ECC). ECC ini berfungsi sebagai *correcting code* karena mempunyai kemampuan untuk mendeteksi bit yang salah, menghitung nilai yang benar dan kemudian mengubahnya. Ketika proses penulisan, ECC di-update dengan menghitung bit di area data. Pada proses pembacaan, ECC dihitung ulang dan dicocokan dengan nilai ECC yang tersimpan saat penulisan. Jika nilainya berbeda maka dipastikan ada sektor yang terkorup.

Agar dapat menyimpan data, OS harus menyimpan struktur datanya dalam disk tersebut. Proses itu dilakukan dalam dua tahap, yaitu partisi dan *logical formatting*. Partisi akan membagi disk menjadi beberapa silinder yang dapat diperlakukan secara independen. *Logical formatting* akan membentuk sistem berkas disertai pemetaan disk. Terkadang sistem berkas ini dirasakan menggangu proses alokasi suatu data, sehingga diadakan sistem partisi lain yang tidak mengikutkan pembentukan sistem berkas, disebut *raw disk*.

Boot Block

Saat sebuah komputer dijalankan, sistem akan mencari sebuah *initial program* yang akan memulai segala sesuatunya. *Initial program*-nya (*initial bootstrap*) bersifat sederhana dan akan menginisialisasi seluruh aspek yang diperlukan bagi komputer untuk beroperasi dengan baik seperti CPU registers, controller, dan yang terakhir adalah Sistem Operasinya. Pada kebanyakan komputer, *bootstrap* disimpan di ROM (*read only memory*) karena letaknya yang tetap dan dapat langsung dieksekusi ketika pertama kali listrik dijalankan. Letak *bootstrap* di ROM juga menguntungkan karena sifatnya yang read only memungkinkan dia untuk tidak terinfeksi virus. Untuk melakukan tugasnya, *bootstrap* mencari *kernel* di disk dan me-*load kernel* ke memori dan kemudian loncat ke *initial address* untuk memulai eksekusi OS.

Untuk alasan praktis, *bootstrap* sering dibuat berbentuk kecil (*tiny loader*) dan diletakkan di ROM, yang kemudian akan me-*load full bootstrap* dari disk bagian disk yang disebut *boot block*. Perubahan menjadi bentuk simple ini bertujuan jika diadakan perubahan pada *bootstrap*, maka struktur ROM tidak perlu dirubah semuanya.

Bad Block

Bad block adalah satu atau lebih sektor yang cacat atau rusak. Kerusakan ini dapat diakibatkan karena kerentanan disk jika sering dipindah-pindah atau kemasukan benda asing. Dalam disk sederhana seperti IDE *controller*, *bad block* akan ditangani secara manual seperti dengan perintah format pada MS-DOS yang akan mencari *bad block* dan menulis nilai spesial ke FAT *entry* agar tidak mengalokasikan *branch routine* ke blok tersebut.

SCSI mengatasi *bad block* dengan cara yang lebih baik. Daftar *bad block*-nya dipertahankan oleh *controller* pada saat *low level formatting*, dan terus diperbarui selama disk itu digunakan. *Low level formatting* akan memindahkan *bad sector* itu ke tempat lain yang kosong dengan algoritma *sector sparing* atau *forwarding*. *Sector sparing* dijalankan dengan ECC mendeteksi *bad sector* dan melaporkannya ke OS, sehingga saat sistem dijalankan sekali lagi, *controller* akan menggantikan *bad sector* tersebut dengan sektor kosong. algoritma lain yang sering digunakan adalah *sector slipping*. Ketika sebuah *bad sector* terdeteksi, sistem akan mengopi semua isi sektor ke sektor

selanjutnya secara bertahap satu-satu sampai ditemukan sektor kosong. Misal *bad sector* di sektor 7, maka isinya akan dipindahkan ke sektor 8, isi sektor 8 dipindahakan ke 9 dan seterusnya.

46.2. Managemen Ruang *Swap*

Managemen ruang *swap* adalah salah satu *low level task* dari OS. Memori virtual menggunakan ruang disk sebagai perluasan dari memori utama. Tujuan utamanya adalah untuk menghasilkan output yang baik. Namun di lain pihak, penggunaan disk akan memperlambat akses karena akses dari memori jauh lebih cepat.

Penggunaan Ruang *Swap*

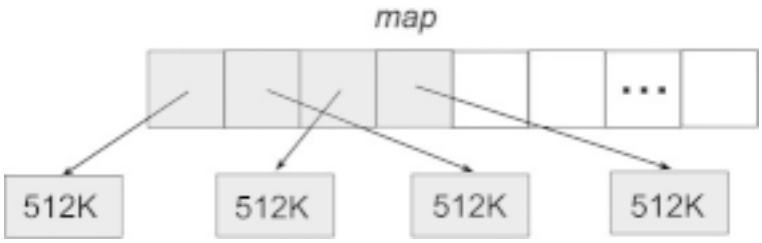
Ruang *swap* digunakan dalam beberapa cara tergantung penerapan algoritma. Sebagai contoh, sistem yang menggunakan *swapping* dapat menggunakan ruang *swap* untuk memegang seluruh proses pemetaan termasuk data dan segmen. Jumlah dari ruang *swap* yang dibutuhkan tergantung dari jumlah memori fisik, jumlah dari memori virtual yang dijalankan, cara penggunaan memori virtual tersebut. Beberapa OS seperti UNIX menggunakan banyak ruang *swap*, yang biasa diletakan di disk terpisah.

Ketika kita akan menentukan besarnya ruang *swap*, sebaiknya kita tidak terlalu banyak atau terlalu sedikit. Jika sistem dijalankan dan ruang *swap* terlalu sedikit, maka proses akan dihentikan dan mungkin akan merusak sistem. Sebaliknya jika terlalu banyak juga akan mengakibatkan lambatnya akses dan pemborosan ruang disk.

Lokasi Ruang *Swap*

Ruang *swap* dapat diletakan di dua tempat yaitu: ruang *swap* dapat berada di sistem berkas normal atau dapat juga berada di partisi yang terpisah. Jika ruang *swap* berukuran besar dan diletakan di sistem berkas normal, *routine*-nya dapat menciptakan, menamainya dan menentukan besar space. Walaupun lebih mudah dijalankan, cara ini cenderung tidak efisien. Pengaksesannya akan sangat memakan waktu dan akan meningkatkan fragmentasi karena pencarian data yang berulang terus selama proses baca atau tulis.

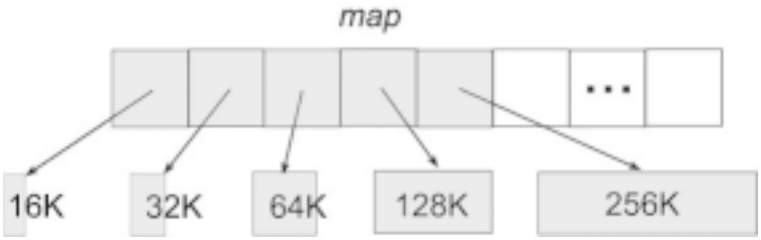
Gambar 46.1. Contoh Managemen ruang swap: pemetaan swap segmen teks 4.3 BSD



Gambar ini diadaptasi dari [Silberschatz2002, halaman 504].

Ruang *swap* yang diletakan di partisi disk terpisah, menggunakan manajer ruang *swap* terpisah untuk melakukan pengalokasian space. Manajer ruang *swap* tersebut menggunakan algoritma yang mengutamakan peningkatan kecepatan dari pada efisiensi. Walaupun fragmentasi masih juga terjadi, tapi masih dalam batas-batas toleransi mengingat ruang *swap* sangat sering diakses. Dengan partisi terpisah, alokasi ruang *swap* harus sudah pasti. Proses penambahan besar ruang *swap* dapat dilakukan hanya dengan partisi ulang atau penambahan dengan lokasi yang terpisah.

Gambar 46.2. Contoh Managemen ruang swap: pemetaan swap segmen data 4.3 BSD



Gambar ini diadaptasi dari [Silberschatz2002, halaman 504].

46.3. Struktur RAID

Disk memiliki resiko untuk mengalami kerusakan. Kerusakan ini dapat berakibat turunnya kinerja atau pun hilangnya data. Meski pun terdapat *backup data*, tetap saja ada kemungkinan data yang hilang karena adanya perubahan setelah terakhir kali data di-backup. Karenanya reliabilitas dari suatu disk harus dapat terus ditingkatkan.

Berbagai macam cara dilakukan untuk meningkatkan kinerja dan juga reliabilitas dari disk. Biasanya untuk meningkatkan kinerja, dilibatkan banyak disk sebagai satu unit penyimpanan. Tiap-tiap blok data dipecah ke dalam beberapa subblok, dan dibagi-bagi ke dalam disk-disk tersebut. Ketika mengirim data disk-disk tersebut bekerja secara paralel, sehingga dapat meningkatkan kecepatan transfer dalam membaca atau menulis data. Ditambah dengan sinkronisasi pada rotasi masing- masing disk, maka kinerja dari disk dapat ditingkatkan. Cara ini dikenal sebagai RAID -- *Redundant Array of Independent* (atau *Inexpensive*) *Disks*. Selain masalah kinerja RAID juga dapat meningkatkan realibilitas dari disk dengan jalan melakukan redundansi data.

Tiga karakteristik umum dari RAID ini, yaitu:

1. Menurut Stallings [Stallings2001], RAID adalah sebuah sebuah set dari beberapa physical drive yang dipandang oleh sistem operasi sebagai sebuah logical drive.
2. Data didistribusikan ke dalam array dari beberapa *physical drive*.
3. Kapasitas disk yang berlebih digunakan untuk menyimpan informasi paritas, yang menjamin data dapat diperbaiki jika terjadi kegagalan pada salah satu disk.

Peningkatan Kehandalan dan Kinerja

Peningkatan Kehandalan dan Kinerja dari disk dapat dicapai melalui dua cara:

1. Redundansi

   Peningkatan Kehandalan disk dapat dilakukan dengan redundansi, yaitu menyimpan informasi tambahan yang dapat dipakai untuk membentuk kembali informasi yang hilang jika suatu disk mengalami kegagalan. Salah satu teknik untuk redundansi ini adalah dengan cara *mirroring* atau *shadowing*, yaitu dengan membuat duplikasi dari tiap-tiap disk. Jadi, sebuah disk *logical* terdiri dari 2 disk *physical*, dan setiap penulisan dilakukan pada kedua disk, sehingga jika salah satu disk gagal, data masih dapat diambil dari disk yang lainnya, kecuali jika disk kedua gagal sebelum kegagalan pada disk pertama diperbaiki. Pada cara ini, berarti diperlukan media penyimpanan yang dua kali lebih besar daripada ukuran data sebenarnya. Akan tetapi, dengan cara ini pengaksesan disk yang dilakukan untuk membaca dapat ditingkatkan dua kali lipat. Hal ini dikarenakan setengah dari permintaan membaca dapat dikirim ke masing-masing disk. Cara

lain yang digunakan adalah paritas blok *interleaved*, yaitu menyimpan blok-blok data pada beberapa disk dan blok paritas pada sebuah (atau sebagian kecil) disk.

2. Paralelisme

   Peningkatan kinerja dapat dilakukan dengan mengakses banyak disk secara paralel. Pada *disk mirroring*, di mana pengaksesan disk untuk membaca data menjadi dua kali lipat karena permintaan dapat dilakukan pada kedua disk, tetapi kecepatan transfer data pada setiap disk tetap sama. Kita dapat meningkatkan kecepatan transfer ini dengan cara melakukan data *striping* ke dalam beberapa disk. Data *striping*, yaitu menggunakan sekelompok disk sebagai satu kesatuan unit penyimpanan, menyimpan bit data dari setiap *byte* secara terpisah pada beberapa disk (paralel).

Level RAID

RAID terdiri dapat dibagi menjadi enam level yang berbeda:

1. **RAID level 0**

   RAID level 0 menggunakan kumpulan disk dengan *striping* pada level blok, tanpa redundansi. Jadi hanya menyimpan melakukan *striping* blok data ke dalam beberapa disk. Level ini sebenarnya tidak termasuk ke dalam kelompok RAID karena tidak menggunakan redundansi untuk peningkatan kinerjanya.

2. RAID level 1

   RAID level 1 ini merupakan *disk mirroring*, menduplikat setiap disk. Cara ini dapat meningkatkan kinerja disk, tetapi jumlah disk yang dibutuhkan menjadi dua kali lipat, sehingga biayanya menjadi sangat mahal.

3. RAID level 2

   RAID level 2 ini merupakan pengorganisasian dengan *error-correcting-code* (ECC). Seperti pada memori di mana pendeteksian terjadinya error menggunakan paritas bit. Setiap *byte* data mempunyai sebuah paritas bit yang bersesuaian yang merepresentasikan jumlah bit di dalam *byte* data tersebut di mana paritas bit=0 jika jumlah bit genap atau paritas=1 jika ganjil. Jadi, jika salah satu bit pada data berubah, paritas berubah dan tidak sesuai dengan paritas bit yang tersimpan. Dengan demikian, apabila terjadi kegagalan pada salah satu disk, data dapat dibentuk kembali dengan membaca *error-correction bit* pada disk lain.

4. RAID level 3

   RAID level 3 merupakan pengorganisasian dengan paritas bit *interleaved*. Pengorganisasian ini hampir sama dengan RAID level 2, perbedaannya adalah RAID level 3 ini hanya memerlukan sebuah disk redundan, berapa pun jumlah kumpulan disk-nya. Jadi tidak menggunakan ECC, melainkan hanya menggunakan sebuah bit paritas untuk sekumpulan bit yang mempunyai posisi yang sama pada setiap disk yang berisi data. Selain itu juga menggunakan data *striping* dan mengakses disk-disk secara paralel.

5. RAID level 4

   RAID level 4 merupakan pengorganisasian dengan paritas blok *interleaved*, yaitu menggunakan *striping* data pada level blok, menyimpan sebuah paritas blok pada sebuah disk yang terpisah untuk setiap blok data pada disk-disk lain yang bersesuaian. Jika sebuah disk gagal, blok paritas tersebut dapat digunakan untuk membentuk kembali blok-blok data pada disk yang gagal tadi. Kecepatan transfer untuk membaca data tinggi, karena setiap disk-disk data dapat diakses secara paralel. Demikian juga dengan penulisan, karena disk data dan paritas dapat ditulis secara paralel.

6. RAID level 5

RAID level 5 merupakan pengorganisasian dengan paritas blok *interleaved* tersebar. Data dan paritas disebar pada semua disk termasuk sebuah disk tambahan. Pada setiap blok, salah satu dari disk menyimpan paritas dan disk yang lainnya menyimpan data. Sebagai contoh, jika terdapat kumpulan dari 5 disk, paritas blok ke n akan disimpan pada disk (n mod 5) + 1; blok ke n dari empat disk yang lain menyimpan data yang sebenarnya dari blok tersebut. Sebuah paritas blok tidak menyimpan paritas untuk blok data pada disk yang sama, karena kegagalan sebuah disk akan menyebabkan data hilang bersama dengan paritasnya dan data tersebut tidak dapat diperbaiki. Penyebaran paritas pada setiap disk ini menghindari penggunaan berlebihan dari sebuah paritas disk seperti pada RAID level 4.

7. RAID level 6

   RAID level 6 disebut juga redundansi P+Q, seperti RAID level 5, tetapi menyimpan informasi redundan tambahan untuk mengantisipasi kegagalan dari beberapa disk sekaligus. RAID level 6 melakukan dua perhitungan paritas yang berbeda, kemudian disimpan di dalam blok-blok yang terpisah pada disk-disk yang berbeda. Jadi, jika disk data yang digunakan sebanyak n buah disk, maka jumlah disk yang dibutuhkan untuk RAID level 6 ini adalah n+2 disk. Keuntungan dari RAID level 6 ini adalah kehandalan data yang sangat tinggi, karena untuk menyebabkan data hilang, kegagalan harus terjadi pada tiga buah disk dalam interval rata-rata untuk perbaikan data *Mean Time To Repair* (MTTR). Kerugiannya yaitu penalti waktu pada saat penulisan data, karena setiap penulisan yang dilakukan akan mempengaruhi dua buah paritas blok.

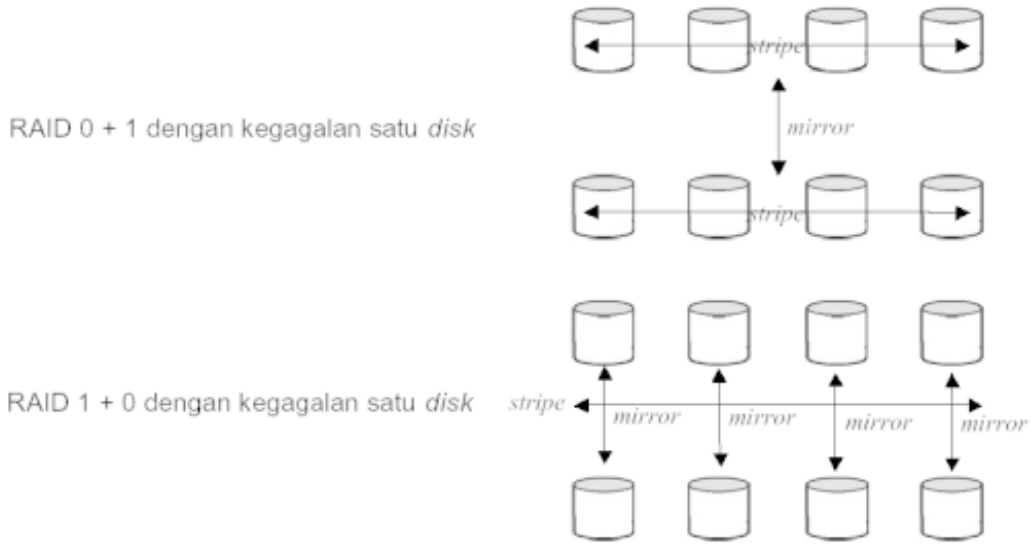
8. RAID level 0+1 dan 1+0

   RAID level 0+1 dan 1+0 ini merupakan kombinasi dari RAID level 0 dan 1. RAID level 0 memiliki kinerja yang baik, sedangkan RAID level 1 memiliki kehandalan. Namun, dalam kenyataannya kedua hal ini sama pentingnya. Dalam RAID 0+1, sekumpulan disk di-strip, kemudian strip tersebut di-mirror ke disk-disk yang lain, menghasilkan strip- strip data yang sama. Kombinasi lainnya yaitu RAID 1+0, di mana disk-disk di-mirror secara berpasangan, dan kemudian hasil pasangan mirrornya di-strip. RAID 1+0 ini mempunyai keuntungan lebih dibandingkan dengan RAID 0+1. Sebagai contoh, jika sebuah disk gagal pada RAID 0+1, seluruh strip-nya tidak dapat diakses, hanya sebagian strip saja yang dapat diakses, sedangkan pada RAID 1+0, disk yang gagal tersebut tidak dapat diakses, tetapi pasangan mirror-nya masih dapat diakses, yaitu disk-disk selain dari disk yang gagal.

Gambar 46.3. Level RAID

Gambar ini diadaptasi dari [Silberschatz2002, halaman 507].

Gambar 46.4. RAID 0 + 1 dan 1 + 0



Gambar ini diadaptasi dari [Silberschatz2002, halaman 511].

46.4. *Host-Attached Storage*

Host-Attached Storage merupakan sistem penyimpanan yang terhubung secara langsung dengan komputer tersebut. *Host-Attached Storage* terhubung secara langsung dengan komputer menggunakan *interface* bus dan IDE.

Dalam implementasinya dalam jaringan, *Host-Attached Storage* dapat juga disebut dengan *Server-Attached Storage* karena sistem penyimpanannya terdapat didalam server itu.

46.5. *Storage-Area Network* dan *Network-Attached Storage*

Network-Attached Storage device

Network-attached storage (NAS) adalah suatu konsep penyimpanan bersama pada suatu jaringan. NAS berkomunikasi menggunakan *Network File Sistem* (NFS) untuk UNIX, *Common Internet File System* (CIFS) untuk Microsoft Windows, FTP, http, dan protokol networking lainnya. NAS membawa kebebasan platform dan meningkatkan kinerja bagi suatu jaringan, seolah-olah adalah suatu dipasang peralatan. NAS device biasanya merupakan *dedicated single-purpose machine*. NAS dimaksudkan untuk berdiri sendiri dan melayani kebutuhan penyimpanan yang spesifik dengan sistem operasi mereka dan perangkat keras/perangkat lunak yang terkait. NAS mirip dengan alat plug-and-play, akan tetapi manfaatnya adalah untuk melayani kebutuhan penyimpanan. NAS cocok digunakan untuk melayani network yang memiliki banyak *client*, *server*, dan operasi yang mungkin menangani task seperti *web cache* dan *proxy*, *firewall*, *audio-video streeming*, *tape backup*, dan penyimpanan data dengan *file serving*.

Network-Attached Storage Versus Storage Area Networks

NAS dan *Storage-Area Network* (SAN) memiliki sejumlah atribut umum. Kedua-Duanya menyediakan konsolidasi optimal, penyimpanan data yang dipusatkan, dan akses berkas yang efisien. Kedua-Duanya mengijinkan untuk berbagi storage antar host, mendukung berbagai sistem operasi yang berbeda pada waktu yang sama, dan memisahkan *storage* dari server aplikasi. Sebagai tambahan, kedua- duanya dapat menyediakan ketersediaan data yang tinggi dan dapat memastikan integritas dengan banyak komponen dan *Redundant Arrays of Independent Disk* (RAID). Banyak yang berpendapat bahwa NAS adalah saingan dari SAN, akan tetapi keduanya dalam kenyataannya dapat bekerja dengan cukup baik ketika digunakan bersama.

NAS dan SAN menghadirkan dua teknologi penyimpanan yang berbeda dan menghubungkan jaringan pada tempat yang sangat berbeda. NAS berada diantar server aplikasi dan sistem berkas. SAN berada diantar sistem berkas dan mendasari *physical storage*. SAN merupaka jaringan itu sendiri, menghubungkan semua storage dan semua server. Karena pertimbangan ini, masing-masing mendukung kebutuhan penyimpanan dari area bisnis yang berbeda.

NAS: Memikirkan Pengguna Jaringan

NAS adalah *network-centric*. Biasanya digunakan Untuk konsolidasi penyimpanan client pada suatu LAN, NAS lebih disukai dalam solusi kapasitas penyimpanan untuk memungkinkan *client* untuk mengakses berkas dengan cepat dan secara langsung. Hal ini menghapuskan *bottleneck* user ketika mengakses berkas dari suatu *general-purpose server*.

NAS menyediakan keamanan dan melaksanakan semua berkas dan storage service melalui protokol standard network, menggunakan TCP/IP untuk transfer data, Ethernet Dan Gigabit Ethernet untuk media akses, dan CIFS, http, dan NFS untuk *remote file service*. Sebagai tambahan, NAS dapat melayani UNIX dan Microsoft Windows user untuk berbagi data yang sama antar arsitektur yang berbeda. Untuk user client, NAS adalah teknologi pilihan untuk menyediakan penyimpanan dengan akses *unen-cumbered* ke berkas.

Walaupun NAS menukar kinerja untuk manajebilitas dan kesederhanaan, bukan merupakan lazy technology. Gigabit Ethernet mengijinkan NAS untuk memilih kinerja yang tinggi dan latensi yang rendah, sehingga mungkin untuk mendukung banyak sekali *client* melalui suatu antarmuka tunggal. Banyak NAS devices yang mendukung berbagai antarmuka dan dapat mendukung berbagai jaringan pada waktu yang sama.

SAN: Memikirkan *Back-End*/Kebutuhan Ruang Penyimpanan Komputer

SAN adalah data-centric, jaringan khusus penyimpanan data. Tidak sama dengan NAS, SAN terpisah dari traditional LAN atau messaging network. Oleh karena itu, SAN dapat menghindari lalu lintar jaringan standar, yang sering menghambat kinerja. SAN dengan *fibre channel* lebih meningkatkan kinerja dan pengurangan latency dengan menggabungkan keuntungan M/K channel dengan suatu jaringan dedicated yang berbeda.

SAN menggunakan *gateway*, *switch*, dan *router* untuk memudahkan pergerakan data antar sarana penyimpanan dan server yang heterogen. Ini mengijinkan untuk menghubungkan kedua jaringan dan potensi untuk *semi-remote storage* (memungkinkan hingga jarak 10km) ke *storage management effort*. Arsitektur SAN optimal untuk memindahkan *storage block*. Di dalam ruang komputer, SAN adalah pilihan yang lebih disukai untuk menujukan isu *bandwidth* dan data aksesibilitas seperti halnya untuk menangani konsolidasi.

Dalam kaitan dengan teknologi dan tujuan mereka yang berbeda,salah satu maupun kedua-duanya dapat digunakan untuk kebutuhan penyimpanan. Dalam kenyataannya, batas antara keduanya samar sedikit menurut Kelompok Penilai, Analis Inc.. Sebagai contoh, dalam aplikasinya anda boleh memilih untuk mem-backup NAS *device* anda dengan SAN, atau menyertakan NAS *device* secara langsung ke SAN untuk mengijinkan *non-bottlenecked access* segera ke storage. (Sumber: An

Overview of Network-Attached Storage, ¨ 2000, Evaluator Group, Inc.)

46.6. Implementasi Penyimpanan Stabil

Pada bagian sebelumnya, kita sudah membicarakan mengenai write-ahead log, yang membutuhkan ketersediaan sebuah storage yang stabil. Berdasarkan definisi, informasi yang berada di dalam stable storage tidak akan pernah hilang. Untuk mengimplementasikan storage seperti itu, kita perlu mereplikasi informasi yang dibutuhkan ke banyak peralatan storage (biasanya disk-disk) dengan failure modes yang independen. Kita perlu mengkoordinasikan penulisan update-update dalam sebuah cara yang menjamin bila terjadi kegagalan selagi meng-update tidak akan membuat semua kopi yang ada menjadi rusak, dan bila sedang recover dari sebuah kegagalan, kita dapat memaksa semua kopi yang ada ke dalam keadaan yang bernilai benar dan konsisten, bahkan bila ada kegagalan lain yang terjadi ketika sedang recovery. Untuk selanjutnya, kita akan membahas bagaimana kita dapat mencapai kebutuhan kita.

Sebuah disk write menyebabkan satu dari tiga kemungkinan:

1. *successful completion*
2. *partial failure*
3. *total failure*

Kita memerlukan, kapan pun sebuah kegagalan terjadi ketika sedang menulis ke sebuah blok, sistem akan mendeteksinya dan memanggil sebuah prosedur recovery untuk me-restore blok tersebut ke sebuah keadaan yang konsisten. Untuk melakukan itu, sistem harus menangani dua blok physical untuk setiap blok logical. Sebuah operasi output dieksekusi seperti berikut:

1. Tulis informasinya ke blok physical yang pertama.
2. Ketika penulisan pertama berhasil, tulis informasi yang sama ke blok *physical* yang kedua.
3. Operasi dikatakan berhasil hanya jika penulisan kedua berhasil.

Pada saat perbaikan dari sebuah kegagalan, setiap pasang blok physical diperiksa. Jika keduanya sama dan tidak terdeteksi adanya kesalahan, tetapi berbeda dalam isi, maka kita mengganti isi dari blok yang pertama dengan isi dari blok yang kedua. Prosedur recovery seperti ini memastikan bahwa sebuah penulisan ke stable storage akan sukses atau tidak ada perubahan sama sekali.

Kita dapat menambah fungsi prosedur ini dengan mudah untuk memboleh kan penggunaan dari kopi yang banyak dari setiap blok pada stable storage. Meski pun sejumlah besar kopi semakin mengurangi kemungkin an untuk terjadinya sebuah kegagalan, maka biasanya wajar untuk men simulasi stable storage hanya dengan dua kopi. Data di dalam stable storage dijamin aman kecuali sebuah kegagalan menghancurkan semua kopi yang ada.

46.7. Rangkuman

Aspek-aspek penting mengenai managemen ruang swap, yaitu:

1. Pengunaan ruang swap

Penggunaan ruang swap tergantung pada penerapan algoritma.

2. Lokasi ruang swap

Ruang swap dapat diletakkan di: - sistem berkas normal

- partisi yang terpisah

RAID (Redundant Array of Independent Disks) merupakan salah satu cara untuk meningkatkan kinerja dan reliabilitas dari disk. Peningkatan Kehandalan dan Kinerja dari disk dapat dicapai melalui:

1. Redudansi

Dengan cara menyimpan informasi tambahan yang dapat dipakai untuk mengembalikan informasi yang hilang jika suatu disk mengalami kegagalan.

2. Paralelisme

Dengan cara mengakses banyak disk secara paralel.

Host-Attached Storage merupakan sistem penyimpanan yang terhubung secara langsung dengan komputer. Dalam implementasinya, Host-Attached Storage dapat disebut juga dengan Server-Attached Storage karena sistem penyimpanannya terdapat di dalam server.

Sedangkan Network-Attached Storage adalah suatu konsep penyimpanan bersama pada suatu jaringan. Network-Attached Storage mengutamakan pengguna jaringan. Berbeda dengan Storage-Area networks yang lebih mengutamakan mengenai kebutuhan ruang penyimpanan komputer.

46.8. Latihan

1. Jelaskan siklus hidup dari permintaan pembacaan blok!
2. Bagaimana cara meningkatkan efisiensi performa M/K?
3. Apa keuntungan penggunaan pemetaan pada disk?
4. Bagaimana cara disk SCSI memulihkan kondisi blok yang rusak?
5. Bagaimana penanganan ruang *swap* pada disk?
6. Bagaimanakah suatu operasi output dieksekusi?
7. Sebutkan kelebihan *tertiary storage structure*?
8. 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

9. Mass Storage System I

   Bandingkan jarak tempuh (dalam satuan silinder) antara penjadualan FCFS (First Come First Served), SSTF (Shortest-Seek-Time-First), dan LOOK. Isi antrian permintaan akses berturut-turut untuk silinder:

   100, 200, 300, 101, 201, 301.

   Posisi awal disk head pada silinder 0.

10. 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 penjadualan "First Come First Served " (FCFS).

    b) Hitunglah jarak tempuh (dalam satuan silinder) untuk algoritma penjadualan "Shortest Seek Time First " (STTF).

11. 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?

12. 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!

13. HardDisk I

    Diketahui sebuah perangkat DISK dengan spesifikasi:

    Kapasitas 100 Gbytes (asumsi 1Gbytes = 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?

14. HardDisk II

    Diketahui sebuah disk dengan spesifikasi berikut ini:

    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?

15. Mengapa file sistem ext3 membutuhkan waktu recovery yang lebih sedikit daripada file sistem ext2 setelah terjadi "unclean shutdown"?

16. Jelaskan pengertian proc file sistem!

46.9. Rujukan

FIXME

Bibliografi

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

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

[Tanenbaum1992] Andrew Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. First Edition. Edisi Pertama. Prentice-Hall.

Bab 47. Perangkat Penyimpanan Tersier

Karakteristik dari perangkat penyimpanan tersier pada dasarnya adalah menggunakan *removable media* yang tentu saja berdampak pada biaya produksi yang lebih murah. Sebagai contoh: sebuah VCR dengan banyak kaset akan lebih murah daripada sebuah VCR yang hanya dapat memainkan satu kaset saja.

47.1. Macam-macam Struktur Penyimpanan Tersier

Floppy Disk

Menurut Silberschatz et. al. [Silberschatz2002], *floppy disk* adalah sebuah media penyimpanan yang terbuat dari cakram fleksibel tipis yang dilapisi oleh bahan magnetik dan ditutupi oleh plastik.

Ciri-ciri *floppy disk*:

1. Memiliki kapasitas kecil (1-2 Mb).
2. Kemampuan aksesnya hampir secepat *hard disk*.
3. Lebih rentan terhadap gesekan di permukaan magnetiknya.

Prinsip ini juga digunakan oleh disk magnetik yang memiliki kapasitas sebesar 1 GB yang memiliki kecepatan akses yang hampir sama dengan *hard disk*.

Magneto-optic disk

Dalam *magneto-optic disk*, data ditulis di atas sebuah piringan keras yang dilapisi oleh suatu bahan magnetik lalu dilapisi pelindung untuk melindungi *head* dari *disk* tsb. Dalam suhu ruangan, medan magnet yang ada tidak dapat digunakan untuk menyimpan bit data sehingga harus ditembakkan laser dari *disk head*. Tempat yang terkena sinar laser ini dapat digunakan untuk menyimpan bit.

Head membaca data yang telah disimpan dengan bantuan *Kerr Effect*. Efek ini timbul karena *head* dari *magneto-optic disk* terlalu jauh dari permukaan *disk* sehingga tidak dapat dibaca dengan cara yang sama yang diterapkan ke *hard disk*. Oleh karena itu digunakan *Kerr Effect*.

Menurut Silberschatz et. al. [Silberschatz2002], prinsip dari *Kerr Effect* adalah ketika suatu sinar laser dipantulkan dari sebuah titik magnetik, polarisasinya akan diputar searah atau berlawanan arah dengan arah jarum jam, tergantung dari orientasi medan magnetiknya. Rotasi inilah yang dibaca oleh *head* sebagai sebuah bit.

Optical disk

Disk tipe ini tidak menggunakan magnetik melainkan suatu bahan yang dapat dibelokkan oleh sinar laser. Setelah dimodifikasi dengan dengan sinar laser pada *disk* akan terdapat *spot* yang gelap atau terang. *Spot* ini menyimpan satu bit.

Teknologi *optical-disk* dapat dibagi menjadi:

1. *Phase-change disk*, dilapisi oleh material yang dapat membeku menjadi *crystalline* atau *amorphous state*. Kedua *state* ini memantulkan sinar laser dengan kekuatan yang berbeda.

Drive menggunakan sinar laser pada kekuatan yang berbeda. Kekuatan rendah digunakan untuk membaca data yang telah ditulis, kekuatan medium untuk menghapus data dengan cara melelehkan permukaannya dan kemudian dibekukan lagi ke dalam keadaan *crystalline*. Kekuatan tinggi digunakan untuk melelehkan *disk*-nya ke dalam *amorphous state* sehingga dapat digunakan untuk menulis data.

2. *Dye-polimer disk*, merekam data dengan membuat *bump*. *Disk* dilapisi plastik yang mengandung *dye* yang dapat menyerap sinar laser. Sinar laser membakar *spot* yang kecil sehingga *spot* membengkak dan membentuk *bump*. Sinar laser juga dapat menghangatkan *bump* sehingga *spot* menjadi lunak dan *bump* menjadi datar.

Write Once Read Many-times (WORM)

WORM terbentuk dari sebuah aluminium film yang dilapisi oleh plastik di bagian atas dan bagian bawahnya. Untuk menulis data, pada media ini digunakan sinar laser untuk membuat lubang pada aluminiumnya sehingga *disk* ini hanya dapat ditulis sekali.

Ciri-ciri *WORM Disk*:

1. Hanya dapat ditulis sekali.
2. Data lebih tahan lama dan dapat dipercaya.

WORM ini dianggap tahan banting dan paling terpercaya karena lapisan metalnya dilindungi dengan aman oleh lapisan plastiknya dan juga datanya tidak dapat dirusak dengan pengaruh medan magnet.

Kebanyakan *removable-disk* lebih lambat dari *non-removable-disk* karena kinerja mereka juga dipengaruhi oleh waktu yang dibutuhkan untuk menulis data. Waktu ini dipengaruhi oleh waktu rotasi, dan juga kadang-kadang *seek time*.

Tapes

Sebuah *tape* dapat menyimpan data lebih banyak dari *optical* maupun *magnetic disk cartridge*, harga *cartridge* dari *tape drive* lebih murah namun memiliki *random access* yang lebih lambat karena membutuhkan operasi *fast-forward* dan *rewind* yang kadang-kadang dapat membutuhkan waktu beberapa detik bahkan menit.

Tape ini biasa digunakan oleh *supercomputer center* untuk menyimpan data yang besar dan tidak membutuhkan *random access* yang cepat.

Dalam skala yang besar biasanya digunakan *Robotic Tape Changers* yaitu sebuah alat yang dipakai untuk mengganti *tape* dalam sebuah *library*.

Stacker menyimpan beberapa *tape*, sedangkan *silo* untuk menyimpan ribuan tape.

47.2. *Future Technology*

Penyimpanan *Holographic*

Teknologi ini digunakan untuk menyimpan foto hologram di media khusus. Misalkan pada foto hitam putih digunakan array 2 dimensi yang merepresentasikan warna hitam dan putih (bit 0 dan 1) maka di teknologi *holographic* ini satu pixel gambar dapat menyimpan jutaan bit sehingga gambarnya menjadi tajam dan pixelnya ditransfer menggunakan sinar laser sehingga *transfer rate*-nya tinggi.

Microelectronic Mechanical Systems (MEMS)

Teknologi yang bertujuan mengembangkan sebuah media penyimpanan yang bersifat *non-volatile* dengan kecepatan yang lebih cepat dan lebih murah dari *semiconductor* DRAM.

47.3. Aplikasi Antarmuka

Sistem operasi tidak menangani *tapes* sebagaimana sistem operasi menangani *removable disk* maupun *fixed disk*. Sistem operasi biasanya menampilkan *tape* sebagai media penyimpanan secara keseluruhan.

Suatu aplikasi tidak membuka suatu berkas pada *tape*, melainkan membuka *tape drive* secara keseluruhan sebagai *raw device*.

Biasanya *tape drive* disediakan untuk penggunaan aplikasi tersebut secara eksklusif, sampai aplikasi tersebut berakhir atau aplikasi tersebut menutup *tape device*. Eksklusivitas ini masuk akal, karena *random access* pada *tape* dapat memakan waktu yang lama, sehingga membiarkan beberapa aplikasi melakukan *random access* pada *tape* dapat menyebabkan *thrashing*.

Sistem operasi tidak menyediakan sistem berkas sehingga aplikasi harus memutuskan bagaimana cara menggunakan blok-blok array.

Tiap aplikasi membuat peraturannya masing-masing tentang bagaimana mengatur *tape* supaya suatu *tape* yang penuh terisi dengan data hanya dapat digunakan oleh program yang membuatnya.

Tape drive mempunyai set operasi-operasi dasar yang berbeda dengan *disk* drive. Sebagai pengganti operasi *seek* (sebagaimana yang digunakan pada *disk drive*), *tape drive* menggunakan operasi *locate*. Operasi *locate* ini lebih akurat dibandingkan dengan operasi *seek* karena operasi ini memposisikan *tape* ke *logical block* yang spesifik.

Sebagian besar tape drive mempunyai operasi *read* position yang berfungsi memberitahu posisi *tape head* dengan menunjukkan nomor *logical blok*. Selain itu banyak juga *tape drive* yang menyediakan operasi *space* yang berfungsi memindahkan posisi *tape head*. Misalnya operasi *space* akan memindahkan posisi *tape head* sejauh dua blok ke belakang.

Untuk sebagian jenis *tape drive*, menulis pada blok mempunyai efek samping menghapus apa pun yang berada pada posisi sesudah posisi penulisan. Hal ini menunjukkan bahwa *tape drive* adalah *append-only devices*, maksudnya adalah apabila kita meng-*update* blok yang ada di tengah berarti kita akan menghapus semua data yang terletak sesudah blok tersebut. Untuk mencegah hal ini terjadi maka digunakan tanda EOT (*end-of-tape*) yang diletakkan pada posisi sesudah posisi blok yang ditulis. Drive menolak untuk mencari lokasi sesudah tanda EOT, tetapi adalah suatu hal yang penting untuk mencari lokasi EOT kemudian mulai menulis menulis data. Cara ini menyebabkan tanda EOT yang lama tertimpa, lalu tanda yang baru diletakkan pada posisi akhir dari blok yang baru saja ditulis.

Penamaan Berkas

Penamaan berkas pada *removable disk* cukup sulit terutama pada saat kita mau menulis data pada *removable cartridge* di suatu komputer, kemudian menggunakan *cartridge* tersebut di komputer yang lain. Jika kedua komputer memiliki tipe mesin yang sama dan memiliki jenis *removable drive* yang sama, maka satu- satunya kesulitan yang ada adalah bagaimana cara mengetahui isi dan *data layout* pada *cartridge*. Namun jika tipe kedua mesin maupun drive berbeda, banyak masalah dapat muncul. Sekali pun kedua drive-nya kompatibel, komputer yang berbeda dapat menyimpan bytes dengan urutan yang berbeda, dan dapat menggunakan *encoding* yang berbeda untuk *binary number* maupun huruf.

Pada umumnya sistem operasi sekarang membiarkan masalah *name-space* tidak terselesaikan untuk *removable media*, dan bergantung kepada aplikasi dan user untuk memecahkan bagaimana cara mengakses dan menerjemahkan data. Untungnya, beberapa jenis *removable media* sudah distandarkan dengan sangat baik sehingga semua komputer dapat menggunakannya dengan cara yang sama, contoh: CD.

Managemen Penyimpanan Hierarkis

Robotic jukebox memungkinkan komputer untuk mengganti *removable cartridge* di *tape* atau *disk drive* tanpa bantuan manusia. Dua penggunaan utama dari teknologi ini adalah untuk kepentingan *backup* dan sistem penyimpanan hirarkis. Sistem penyimpanan hirarkis ini sendiri melingkupi hirarkis penyimpanan yang merupakan cakupan lebih luas daripada memori primer dan penyimpanan sekunder untuk membentuk penyimpanan tersier. Penyimpanan tersier biasanya diimplementasikan sebagai *jukebox* dari *tapes* atau *removable media*.

Walaupun penyimpanan tersier dapat mempergunakan sistem memori virtual, cara ini tidak baik. Karena pengambilan data dari *jukebox* membutuhkan waktu yang agak lama. Selain itu diperlukan waktu yang agak lama untuk *demand paging* dan untuk bentuk lain dari penggunaan *virtual-memory*.

Berkas yang kapasitasnya kecil dan sering digunakan dibiarkan berada di disk magnetik, sementara berkas yang kapasitasnya besar, sudah lama, dan tidak aktif digunakan akan diarsipkan di *jukebox*.

Pada beberapa sistem *file-archiving*, *directory entry* untuk berkas selalu ada, tetapi isi berkas tidak lagi berada di penyimpanan sekunder. Jika suatu aplikasi mencoba membuka berkas, pemanggilan *open system* akan ditunda sampai isi berkas dikirim dari penyimpanan tersier. Ketika isi berkas sudah dikirimkan dari disk magnetik, operasi *open* mengembalikan kontrol kepada aplikasi.

Managemen penyimpanan hierarkis biasanya ditemukan pada pusat *supercomputing* dan instalasi besar lainnya yang mempunyai data yang besar.

47.4. Masalah Kinerja

Tiga aspek utama dari kinerja penyimpanan tersier berdasarkan Silberschatz et. al. [Silberschatz2002]:

1. Kecepatan

   Kecepatan dari penyimpanan tersier memiliki dua aspek: *bandwidth* dan *latency*. Menurut Silberschatz et. al. [Silberschatz2002], *Sustained bandwidth* adalah rata-rata tingkat data pada proses transfer, yaitu jumlah byte dibagi dengan waktu transfer. *Effective bandwidth* menghitung rata-rata pada seluruh waktu I/O, termasuk waktu untuk *seek* atau *locate*. Istilah *bandwidth* dari suatu *drive* sebenarnya adalah *sustained bandwidth*.

2. Kehandalan

   Removable magnetic disk tidak begitu dapat diandalkan dibandingkan dengan *fixed hard-disk* karena *cartridge* lebih rentan terhadap lingkungan yang berbahaya seperti debu, perubahan besar pada temperatur dan kelembaban, dan gangguan mekanis seperti tekukan. *Optical disks* dianggap sangat dapat diandalkan karena lapisan yang menyimpan bit dilindungi oleh plastik transparan atau lapisan kaca.

3. Harga

47.5. Rangkuman

Karakteristik utama dari perangkat penyimpanan tersier adalah bahwa mereka bersifat removable, sebagai contoh floppy disk, CD_ROM dan tape. Berbagai teknologi digunakan untuk perangkat lunak tersier seperti magnetic tape, magneto-optic disk dan optical disk. Meskipun keuntungan utamanya itu adalah removable namun mereka memiliki kecepatan akses yang jauh lebih lama dibandingkan perangkat penyimpanan primer dan sekunder. Berbagai upaya telah dilakukan untuk meningkatkan kecepatan akses perangkat penyimpanan sekunder.

Tiga aspek kinerja yang utama adalah kecepatan, kehandalan dan harga. Perbedaan yang cukup besar dari bandwith dapat ditemukan antara disk dan tape dimana akses terhadap tape lebih lama.

Kehandalan dari removable magnetic disk masih kurang karena masih rentan terhadap lingkungan yang berbahaya seperti debu, temperatur dan kelembaban. Sedangkan optical disk lebih handal dibandingkan magnetic media karena memiliki lapisan pelindung dari plastik transparan atau kaca.

47.6. Latihan

1. Sebutkan macam-macam perangkat penyimpanan tersier, dan apa karakteristik dari perangkat penyimpanan tersier?
2. Jelaskan secara singkat tentang future teknologi pada perangkat penyimpanan tersier saat ini?
3. Apa beda magneto optic disk dan optical disk?
4. Jelaskan dengan singkat tentang aplikasi antar muka?

47.7. Rujukan

FIXME

Bibliografi

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

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

[Tanenbaum1992] Andrew Tanenbaum. Hak Cipta © 1992. *Modern Operating Systems*. First Edition. Edisi Pertama. Prentice-Hall.

Bab 48. Keluaran/Masukan Linux

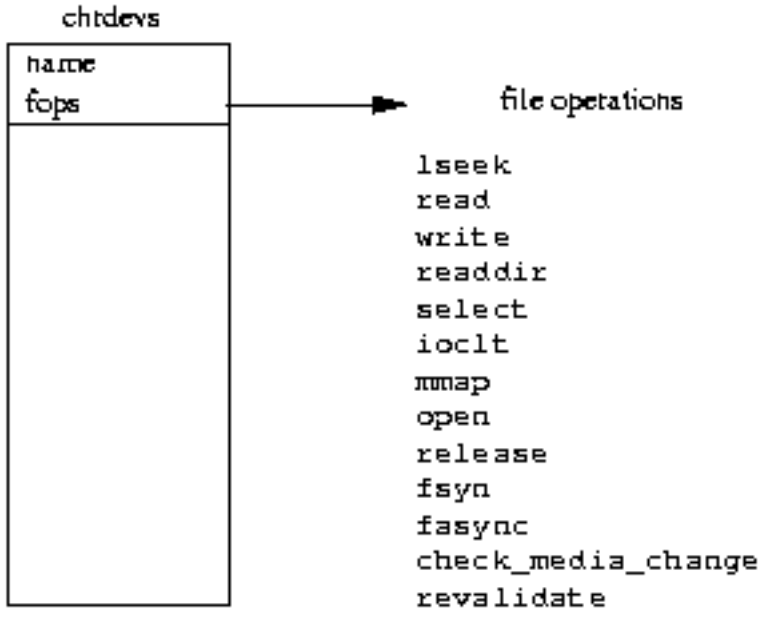
Salah satu tujuan sistem operasi adalah menyembunyikan kerumitan device hardware dari sistem penggunanya. Contohnya, Sistem Berkas Virtual menyamakan tampilan sistem berkas yang dimount tanpa memperdulikan devices fisik yang berada di bawahnya. Bab ini akan menjelaskan bagaimana kernel Linux mengatur device fisik di sistem.

Salah satu fitur yang mendasar adalah kernel mengabstraksi penanganan device. Semua device hardware terlihat seperti berkas pada umumnya: mereka dapat dibuka, ditutup, dibaca, dan ditulis menggunakan calls sistem yang sama dan standar untuk memanipulasi berkas. Setiap device di sistem direpresentasikan oleh sebuah file khusus device, contohnya disk IDE yang pertama di sistem direpresentasikan dengan /dev/hda. Devices blok (disk) dan karakter dibuat dengan perintah mknod dan untuk menjelaskan device tersebut digunakan nomor devices besar dan kecil. Devices jaringan juga direpresentasikan dengan berkas khusus device, tapi berkas ini dibuat oleh Linux setelah Linux menemukan dan menginisialisasi pengontrol-pengontrol jaringan di sistem. Semua device yang dikontrol oleh driver device yang sama memiliki nomor device besar yang umum. Nomor devices kecil digunakan untuk membedakan antara device-device yang berbeda dan pengontrol-pengontrol mereka, contohnya setiap partisi di disk IDE utama punya sebuah nomor device kecil yang berbeda. Jadi, /dev/hda2, yang merupakan partisi kedua dari disk IDE utama, punya nomor besar 3 dan nomor kecil yaitu 2. Linux memetakan berkas khusus device yang diteruskan ke system call (katakanlah melakukan mount ke sistem berkas device blok) pada driver si device dengan menggunakan nomor device besar dan sejumlah tabel sistem, contohnya tabel device karakter, chrdevs.

Linux membagi devices ke tiga kelas: devices karakter, devices blok dan devices jaringan.

48.1. *Device Karakter*

Gambar 48.1. CharDev. Sumber: . . .



Device karakter, device paling sederhana dari Linux, diakses sebagai berkas. Aplikasi menggunakan *system calls* standar untuk membukanya, membacanya dan menulisnya dan menutupnya persis seolah devices adalah berkas. Memang benar, meski pun devices ini merupakan modem yang sedang digunakan oleh PPP daemon untuk menghubungkan sistem Linux ke jaringan. Saat sebuah

device karakter diinisialisasi, driver devicenya mendaftarkan sang device pada kernel Linux dengan menambahkan sebuah entry ke vektor chrdevs dari struk data device_struct. Pengenal utama devicenya digunakan sebagai indeks ke vektor ini. Pengenal utama untuk suatu device tidak pernah berubah.

Setiap entry di vektor chrdevs, sebuah struk data device_struct, mengandung dua elemen: sebuah penunjuk nama dari driver devices yang terdaftar dan sebuah penunjuk ke operasi-operasi berkas seperti buka, baca, tulis, dan tutup. Isi dari /proc/devices untuk devices karakter diambil dari vektor chrdevs.

Saat sebuah berkas khusus karakter yang merepresentasikan sebuah devices karakter (contohnya /dev/cua0) dibuka, kernelnya harus mengatur beberapa hal sehingga routine operasi berkas yang benar dari driver devices karakter akan terpanggil.

Seperti sebuah berkas atau direktori pada umumnya, setiap berkas khusus device direpresentasikan dengan sebuah inode VFS. Inode VFS untuk sebuah berkas khusus karakter tersebut, sebenarnya untuk semua berkas yang berada dibawahnya, contohnya EXT2. Hal ini terlihat dari informasi di berkas yang sebenarnya ketika nama berkas khusus device dilihat.

Setiap inode VFS memiliki keterkaitan dengan seperangkat operasi berkas dan operasi-operasi ini berbeda tergantung pada obyek sistem berkas yang direpresentasikan oleh inode tersebut. Kapan pun sebuah VFS yang merepsentasikan berkas khusus karakter dibuat, operasi-operasi berkasnya diset ke operasi device karakter default.

VFS inode memiliki hanya satu operasi berkas, yaitu operasi membuka berkas. Saat berkas khusus karakter dibuka oleh sebuah aplikasi, operasi buka berkas yang umum atau generik menggunakan pengenal utama dari device tersebut. Pengenal ini digunakan sebagai index ke vektor chrdevs untuk memperoleh blok operasi berkas untuk device tertentu ini. Ia juga membangun struk data berkas yang menjelaskan berkas khusus karakter ini, yang membuat penunjuk operasi berkas menunjuk ke driver device itu. Setelah itu semua aplikasi dari operasi-operasi berkas aplikasi akan dipetakan untuk memanggil perangkat devices karakter dari operasi berkas itu.

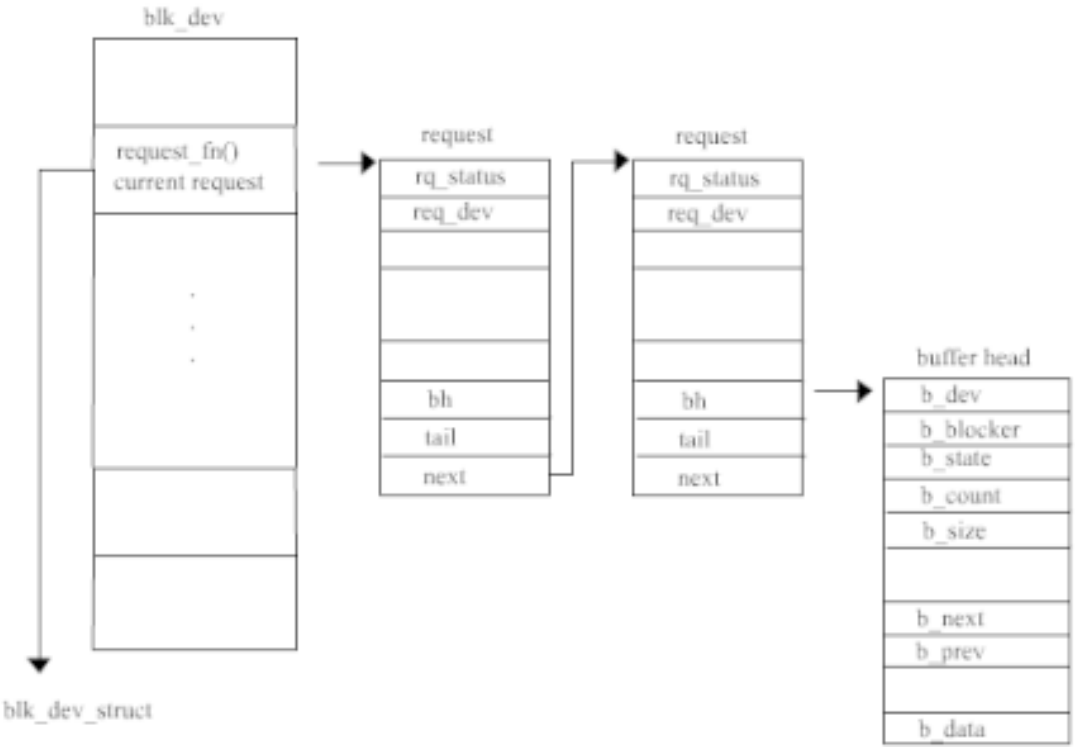
48.2. *Device* Blok

Device ini pun diakses seperti berkas. Mekanisme untuk menyediakan perangkat operasi berkas yang benar bagi berkas khusus blok yang terbuka sama seperti devices karakter. Linux memelihara operasi dari perangkat device blok yang terdaftar sebagai vektor blkdevs. Vektor ini, seperti halnya vektor chrdevs, diindeks dengan menggunakan nomor device besar dari sang device. Entrynya juga merupakan struk data device_struct. Tidak seperti devices karakter, ada sejumlah kelas yang dimiliki device blok. Device-device SCSI adalah salah satu kelasnya dan device IDE adalah kelas lainnya. Kelaslah yang mendaftarkan dirinya sendiri pada kernel Linux dan menyediakan operasi berkas kepada kernel. Driver-driver device untuk sebuah kelas device blok menyediakan interface khusus kelas kepada kelas tersebut. Jadi, contohnya, sebuah driver device SCSI harus menyediakan interface untuk subsistem SCSI agar dapat menyediakan operasi berkas bagi devices ini ke kernel.

Setiap driver device blok harus menyediakan sebuah interface ke cache buffernya, demikian pula interface operasi umum berkas. Setiap driver device blok mengisi entrynya di vektor blk_dev dari struk data blk_dev_struct. Indeksnya ke vektor ini, lagi-lagi, nomor utama devicenya. Struk data blk_dev_struct mengandung alamat routine permintaan dan sebuah penunjuk ke sekumpulan struk data request,yang masing-masingnya merepresentasikan sebuah request dari cache buffernya untuk driver untuk membaca atau menulis atau menulis satu blok data.

Setiap kali cache buffer ingin membaca dari, atau pun menuliskan satu blok data ke device terdaftar, ia menambahkan struk data request kedalam blk_dev_struct nya. Gambar di atas ini menunjukkan bahwa setiap request memiliki *pointer* (penunjuk) ke satu atau lebih struk data buffer_head. Masing-masingnya merupakan suatu request untuk membaca atau menulis sebuah blok data. Struk buffer_head tersebut dikunci (oleh cache buffer) dan mungkin ada suatu proses yang menunggu buffer ini selesai di operasi blok tersebut. Setiap struk request dialokasikan dari suatu daftar yang statik, yaitu daftar all_request. Jika proses tersebut sedang dimasukkan sebuah ke list request yang kosong, fungsi request dari drivernya akan dipanggil agar memulai proses antrian request. Jika tidak driver tersebut hanya akan memproses setiap request di daftar request.

Gambar 48.2. Buffer. Sumber: . . .



Sekali driver device telah menyelesaikan sebuah request, ia harus membuang setiap stuk buffer_request dari struk requestnya, kemudian mencapnya *up to date* dan membuka kuncinya. Pembukaan kunci buffer_head akan membangunkan proses apa pun yang tidur akibat menunggu operasi blok selesai. Contoh dari kasus ini misalnya dimana sebuah nama berkas sedang ditangani dan sistem berkas EXT2 harus membaca blok data yang mengandung entry direktori EXT2 berikutnya dari device blok yang menyimpan sistem berkas tersebut. Proses ini tidur di buffer_head yang akan mengandung entri direktorinya sampai driver devicenya membangunkannya. Struk data request tersebut ditandai bebas sehingga ia dapat digunakan di request blok lainnya.

48.3. *Device* Jaringan

Device jaringan merupakan sebuah entity yang mengirimkan dan menerima paket-paket data. Biasanya ia merupakan device fisik seperti kartu ethernet. Beberapa devices jaringan bagaimana pun hanyalah software, seperti device loopback yang digunakan untuk mengirimkan data ke Anda. Setiap device direpresentasikan dengan struk data device. Driver device jaringan mendaftarkan device-device yang ia kontrol pada Linux selama inisialisasi jaringan yaitu saat kernel melakukan booting. Struk data device tersebut berisi informasi mengenai device dan alamat fungsi-fungsi yang memungkinkan bermacam-macam protokol jaringan menggunakan layanan dari device tersebut. Fungsi-fungsi ini kebanyakan terkait dengan mentransmisikan data dengan menggunakan device jaringan. Device tersebut menggunakan mekanisme pendukung jaringan standar untuk melewatkan data yang diterima sampai ke lapisan protokol yang semestinya. Semua data jaringan atau paket yang ditransmisikan dan diterima, direpresentasikan dengan struk-struk data sk_buff. Struk-struk data yang bersifat fleksibel ini memungkinkan header-header protokol jaringan menjadi mudah ditambahkan dan dibuang. Bagian ini hanya memfokuskan pada struk data device serta bagaimana jaringan ditemukan dan diinsialisasi.

Struk data device ini mengandung informasi tentang device jaringan berikut.

Nama

Berbeda dengan device karakter dan blok yang menggunakan berkas khusus device yang dibuat dengan perintah mknod, berkas khusus device terlihat sekilas seperti device jaringan sistem yang ditemukan dan diinsialisasi. Nama mereka standar, yaitu setiap nama merepsentasikan jenis device masing-masing. Device multiple dari jenis yang sama dinomori lebih besar dari 0. Oleh sebab itu device-device ethernet dikenal sebagai /dev/eth0, /dev/eth1, /dev/eth2 dan seterusnya.

Beberapa device jaringan yang umum adalah

- /dev/ethN Device ethernet
- /dev/slN Device SLIP
- /dev/pppN Device PPP
- /dev/lo Device Loopback

Informasi Bus

Berikut ini adalah informasi yang driver device butuhkan untuk mengontrol devicenya. Nomor irq merupakan interrupt yang digunakan oleh device ini. Alamat basisnya adalah alamat dari segala register status dan control dari device yang ada di memori M/K. Channel DMA adalah nomor DMA yang device jaringan ini gunakan. Semua informasi ini diset pada waktu booting, yaitu saat device ini diinisialisasi.

Flags Interface

Hal-hal berikut ini akan menjelaskan karakteristik dan kemampuan dari device jaringan:

- IFF_UP Interface bangkit dan berjalan,
- IFF_BROADCAST Alamat broadcast di device adalah sah
- IFF_DEBUG Penghilangan error dinyalakan
- IFF_LOOPBACK Merupakan device loopback
- IFF_POINTTOPOINT Merupakan link point to point (SLIP dan PPP)
- IFF_NOTRAILERS Tidak ada pengangkut jaringan
- IFF_RUNNING Sumberdaya yang dialokasikan
- IFF_NOARP Tidak mendukung protokol ARP
- IFF_PROMISC Device di mode penerimaan acak, ia akan menerima semua paket tanpa memperdulikan kemana paket-paket ini dialamatkan
- IFF_ALLMULTI Menerima seluruh frame multicast IP
- IFF_MULTICAST Dapat menerima frame multicast IP

Informasi Protokol

Setiap device menjelaskan bagaimana ia digunakan oleh lapisan protokol jaringan.

MTU

Ukuran paket terbesar yang jaringan dapat kirim, tidak termasuk header lapisan link yang ia perlu tambahkan.

Keluarga

Keluarga ini menandakan bahwa keluarga protokol yang dapat didukung oleh device tersebut. Keluarga untuk seluruh device jaringan Linux adalah AF_INET, keluarga alamat internet.

Jenis

Jenis menjelaskan media di mana device jaringan terpasang. Ada banyak jenis media yang didukung oleh device jaringan Linux. Termasuk diantaranya adalah Ethernet, X.25, Token Ring, Slip, PPP dan Apple Localtalk.

Alamat

Struk data device tersebut memiliki sejumlah alamat yang relevan bagi device jaringan ini, termasuk alamat-alamat IP-nya.

Antrian Paket

Merupakan antrian paket-paket sk_buff yang antri menunggu untuk dikirmkan lewat device jaringan ini.

Fungsi Pendukung

Setiap device menyediakan seperangkat routine standar yang lapisan-lapisan protokol sebut sebagai bagian dari interface mereka ke lapisan link device ini. Hal ini termasuk pembuatannya dan routine-routine pengirim frame dan routine-routine penambah header standar dan pengumpul statistik. Statistik ini bisa dilihat dengan memakai perintah ifconfig.

48.4. Rangkuman

Dasar dari elemen perangkat keras yang terkandung pada M/K adalah *bus*, *device controller*, dan M/K itu sendiri. Kinerja kerja pada data yang bergerak antara device dan memori utama di jalankan oleh CPU, di program oleh M/K atau mungkin DMA *controller*. Modul kernel yang mengatur *device* adalah *device driver*. *System-call interface* yang disediakan aplikasi dirancang untuk menghandle beberapa dasar kategori dari perangkat keras, termasuk *block devices*, *character devices*, *memory mapped files*, *network sockets*, dan *programmed interval timers*.

Subsistem M/K kernel menyediakan beberapa servis. Diantaranya adalah *I/O schedulling*, *buffering*, *spooling*, *error handling*, dan *device reservation*. Salah satu servis dinamakan *translation*, untuk membuat koneksi antara perangkat keras dan nama file yang digunakan oleh aplikasi.

I/O system calls banyak dipakai oleh CPU, dikarenakan oleh banyaknya lapisan dari perangkat lunak antara *physical device* dan aplikasi. Lapisan ini mengimplikasikan *overhead* dari *context switching* untuk melewati *kernel's protection boundary*, dari sinyal dan *interrupt handling* untuk melayani *I/O devices*.

Disk drives adalah *major secondary-storage I/O device* pada kebanyakan komputer. Permintaan untuk disk M/K digenerate oleh sistem file dan sistem virtual memori. Setiap permintaan menspesifikasikan alamat pada disk untuk dapat direferensikan pada *form* di *logical block number*.

Algoritma *disk schedulling* dapat meningkatkan efektifitas *bandwidth*, *average response time*, dan *variance response time*. Algoritma seperti SSTF, SCAN, C-SCAN, LOOK dan C-LOOK didesain untuk membuat perkembangan dengan menyusun ulang antrian disk untuk meningkatkan total waktu pencarian.

Performa dapat rusak karena *external fragmentation*. Satu cara untuk menyusun ulang disk untuk mengurangi fragmentasi adalah untuk *back up* dan *restore* seluruh disk atau partisi. Blok-blok dibaca dari lokasi yang tersebar, me-*restore* tulisan mereka secara berbeda. Beberapa sistem mempunyai kemampuan untuk men-*scan* sistem file untuk mengidentifikasi file terfragmentasi, lalu menggerakan blok-blok mengelilingi untuk meningkatkan fragmentasi. Mendefragmentasi file yang sudah di fragmentasi (tetapi hasilnya kurang optimal) dapat secara signifikan meningkatkan performa, tetapi sistem ini secara umum kurang berguna selama proses defragmentasi sedang berjalan. Sistem operasi me-*manage* blok-blok pada disk. Pertama, disk baru di format secara *low level* untuk menciptakan sektor pada perangkat keras yang masih belum digunakan. Lalu, disk dapat di partisi dan sistem file diciptakan, dan blok-blok boot dapat dialokasikan. Terakhir jika ada blok yang terkorupsi, sistem harus mempunyai cara untuk me-*lock out* blok tersebut, atau menggantikannya dengan cadangan.

Tertiary storage di bangun dari disk dan *tape drives* yang menggunakan media yang dapat dipindahkan. Contoh dari *tertiary storage* adalah *magnetic tape*, *removable magnetic*, dan *magneto-optic disk*.

Untuk *removable disk*, sistem operasi secara general menyediakan servis penuh dari sistem file *interface*, termasuk *space management* dan *request-queue schedulling* . Untuk tape, sistem operasi secara general hanya menyediakan *interface* yang baru. Banyak sistem operasi yang tidak memiliki *built-in support* untuk *jukeboxes*. *Jukebox support* dapat disediakan oleh *device driver*.

Setiap aplikasi yang dijalankan di linux mempunyai pengenal yang disebut sebagai process identification number (PID). PID disimpan dalam 32 bit dengan angka berkisar dari 0-32767 untuk menjamin kompatibilitas dengan unix. Dari nomor PID inilah linux dapat mengawasi dan mengatur proses-proses yang terjadi didalam system. Proses yang dijalankan atau pun yang baru dibuat mempunyai struktur data yang disimpan di task_struct. Linux mengatur semua proses di dalam sistem melalui pemeriksaan dan perubahan terhadap setiap struktur data task_struct yang dimiliki setiap proses. Sebuah daftar pointer ke semua struktur data task_struct disimpan dalam task vector. Jumlah maksimum proses dalam sistem dibatasi oleh ukuran dari task vector. Linux umumnya memiliki task vector dengan ukuran 512 entries. Saat proses dibuat, task_struct baru dialokasikan dari memori sistem dan ditambahkan ke task vector. Linux juga mendukung proses secara real time. Proses semacam ini harus bereaksi sangat cepat terhadap event eksternal dan diperlakukan berbeda dari proses biasa lainnya oleh penjadual.

Obyek-obyek yang terdapat di sistem berkas linux antara lain file, inode, file sistem dan nama inode. Sedangkan macam-macam sistem berkas linux antar lain : ext2fs, ext3fs, reiser, x, proc dan tiga tambahan : sistem berkas web, sistem berkas transparent cryptographic dan sistem berkas steganographic

48.5. Latihan

1. FIXME

48.6. Rujukan

FIXME

Bagian VIII. Topik Lanjutan

Daftar Isi

Bab 49. Sistem Waktu Nyata dan Multimedia

49.1. Pendahuluan

Sistem yang memiliki persyaratan tertentu, tentunya memiliki tujuan yang berbeda dengan yang selama ini kita pelajari. Seperti halnya pula dengan sistem waktu nyata, dimana sistem ini mempersyaratkan bahwa komputasi yang dihasilkan benar tapi juga harus sesuai dengan waktu yang dikehendaki. Oleh karena itulah algoritma penjadualan yang tradisional haruslah dimodifikasi sehingga dapat memenuhi persyaratan deadline yang diminta. Hal ini pula yang dipersyaratkan oleh sistem multimedia yang tidak hanya memiliki data konvensional (seperti berkas teks, program, dll), tetapi juga memiliki data multimedia. Hal ini disebabkan karena data multimedia terdiri dari continous-media data (audio dan video), seperti contohnya adalah frame video, yang mempersyaratkan juga pengirimannya dalam batas waktu tertentu, misalnya 30 frame/detik. Untuk dapat memenuhi permintaan ini, dibutuhkan perubahan yang cukup signifikan pada struktur sistem operasi, yang kebanyakan pada memori, data dan juga managemen jaringan.

49.2. Kernel Waktu Nyata

Sebelum memasuki lebih jauh tentang kernel waktu nyata, kita perlu tahu apa yang dimaksud dengan waktu nyata. Waktu nyata merujuk pada bentuk aplikasi yang mengontrol proses dimana masalah waktu merupakan hal yang sangat penting. Sistem waktu nyata digunakan ketika ada persyaratan waktu yang ketat pada operasi di prosesor atau flow dari data; yang sering digunakan sebagai alat kontrol yang pada aplikasi yang terpisah. Atau dengan kata lain, sebuah sistem waktu nyata tidak hanya perlu untuk menjalankan software melalui proses dengan benar, tapi juga perlu untuk menjalankannnya dalam waktu yang tepat, kalau tidak sistem akan gagal.

Pada sistem operasi yang mendukung sistem waktu nyata, tidak dibutuhkan fitur yang penting untuk standar desktop dan server system. Hal ini dikarenakan:

1. Kebanyakan sistem waktu nyata melayani untuk sebuah tujuan saja, sehingga tidak perlu membutuhkan banyak fitur pada desktop PC. Sistem waktu nyata tertentu juga tidak memasukkan notion pada user karena sistem hanya mendukung sejumlah kecil task saja, yang sering menunggu input dari peralatan H/W.

2. Fitur yang didukung oleh standar desktop dan server system tidak memungkinkan untuk menyediakan prosesor yang cepat dan memori yang banyak. Kekurangan space, menyebabkan sistem waktu nyata tidak dapat untuk mendukung drive disk yang peripheral atau mendisplay grafik.

3. Mendukung fitur yang biasa ada pada standar desktop dan server system akan sangat meningkatkan cost dari sistem waktu nyata.

Fitur-fitur minimal yang dibutuhkan oleh sistem operasi yang mendukung sistem yang real time adalah:

1. Penjadualan berdasarkan prioritas dan preemptif

2. Kernel preemptif

3. Latency yang minimal

Fitur yang dihilangkan pada daftar di atas adalah dukungan jaringan. Hal ini dikarenakan hanya dipersyaratkan pada sistem komputer yang memang berinteraksi dengan sistem komputer lain

melalui jaringan.

49.2.1. Penjadualan Berdasarkan Prioritas

Algoritma penjadualan berdasarkan prioritas dan preemptif mampu menjalankan proses berdasarkan tingkat kepentingannya, dimana proses yang sedang berjalan akan dipreemptifkan apabila sebuah proses berprioritas tinggi menjadi tersedia untuk dijalankan. Oleh karena itulah algoritma ini dibutuhkan, mengingat fitur utama yang paling dibutuhkan oleh sistem yang waktu nyata adalah mampu merespon proses yang real time secepat proses membutuhkan CPU.

Sistem waktu nyata dibagi menjadi dua, yaitu sistem waktu nyata keras dan sistem waktu nyata lembut. Algoritma ini adalah hanya menjamin fungsionalitas waktu nyata lembut. Hal ini dikarenakan sistem waktu nyata lembut memberikan aturan yang kurang, sehingga memungkinkan proses yang kritis untuk mendapatkan prioritas belakangan. Walaupun menambahkan fungsionalitas waktu nyata lembut pada sistem baru dapat menyebabkan alokasi sumber daya yang tidak adil dan dapat pula menyebabkan penundaan yang lebih lama, atau bahkan kelaparan, namun setidaknya hal tersebut untuk beberapa proses mungkin untuk dicapai. Hasilnya adalah sebuah sistem bertujuan umum yang dapat mendukung multimedia berupa grafik interaktif berkecepatan tinggi dan berbagai jenis pekerjaan yang secara fungsi tidak akan diterima pada lingkungan yang tidak mendukung perhitungan waktu nyata lembut. Algoritma ini tidak mampu mendukung sistem waktu nyata keras, karena kedepannya sistem waktu nyata keras harus memiliki jaminan bahwa task waktu nyata akan dilayani sesuai dengan persyaratan waktu tenggangnya. Pernyataan tentang jumlah waktu yang dibutuhkan untuk penyelesaian proses, disubmit bersamaan dengan proses itu sendiri. Kemudian penjadual akan memberikan ijin bagi proses tersebut, memberikan jaminan bahwa proses tersebut dapat diselesaikan tepat waktu, dan apabila tidak memungkinkan untuk diselesaikan tepat waktu, maka akan ditolak. Jaminan semacam ini tidak akan mungkin dilakukan pada sistem dengan secondary storage atau memori virtual, karena subsistem ini dapat menyebabkan hal-hal yang tidak dapat dicegah dan tidak dapat diperkirakan pada sejumlah waktu untuk mengeksekusi proses tertentu. Oleh karenanyalah, sistem waktu nyata keras disusun dari software yang bertujuan khusus yang berjalan pada hardware yang khusus dipersembahkan bagi proses yang kritis, dan membutuhkan fungsionalitas yang penuh dari komputer modern dan sistem operasi.

49.2.2. Kernel Preemptif

Kernel yang non-preemptif tidak mengizinkan preemption pada proses yang berjalan pada mode kernel, dimana proses yang mode kernel akan berjalan sampai dengan keluar dari mode kernel, blok atau voluntarily dari yields control dari CPU. Sebaliknya, kernel yang preemptif mengizinkan preemption dari task untuk berjalan pada mode kernel.

Untuk dapat memenuhi persyaratan waktu dari sistem waktu nyata keras, adanya kernel preemptif menjadi penting. Kalau tidak maka task waktu nyata akan dapat menunggu dalam waktu yang panjang padahal task yang lain aktif dalam kernel.

Ada beberapa cara untuk membuat kernel yang dapat preemptif. Salah satunya adalah dengan memasukkan preemption point pada system call berdurasi panjang, dimana preemption point akan mengecek apakah proses dengan berprioritas tinggi perlu untuk dijalankan. Jika demikian maka context switch yang berperan. Maka, ketika proses berprioritas tinggi terminasi, proses yang diinterupsi akan melanjutkan system call. Preemption point akan ditempatkan hanya pada lokasi aman pada kernel, yaitu hanya pada saat dimana struktur data kernel yang belum dimodifikasi. Strategi kedua adalah dengan membuat sebuah kernel yang dapat preemptif melalui penggunaan mekanisme sinkronisasi. Dengan metodologi ini, kernel dapat selalu dipreemptifkan karena date kernel tertentu yang diupdate yang akan diproteksi dari proses berprioritas tinggi. Cara seperti ini digunakan di Solaris 2.

Fase konflik dari keterlambatan kehadiran memiliki dua komponen, yaitu:

1. Preemption dari berbagai proses terjadi di kernel
2. Pelepasan oleh proses berprioritas rendah yang sumber dayanya

dibutuh oleh prioritas tinggi. Sebagai contoh pada Solaris 2 dengan meniadakan kemampuan untuk

preemption, keterlambatan berada di atas 100 ms; dengan memungkinkan untuk preemption, maka akan berkurang menjadi 2 ms.

49.2.3. Mengurangi Latency

Event latency merupakan sejumlah waktu yang berlalu, mulai dari ketika sebuah event terjadi sampai dengan ketika event tersebut dilayani. Biasanya event yang berbeda memiliki requirement latency yang berbeda.

Performa dari sistem waktu nyata dipengaruhi oleh dua jenis keterlambatan berikut ini:

1. Interrupt latency
2. Dispatch latency

Interrupt latency merujuk kepada periode waktu dari kedatangan interupsi sampai dengan pada CPU sampai dengan dimulainya rutin dimana service diinterupsi. Ketika interupsi terjadi, sistem operasi pertama kali harus melengkapi instruksi yang dieksekusinya dan menentukan jenis dari interupsi yang terjadi. Kemudian harus disimpan state dari proses saat ini sebelum melayani interupsi menggunakan interrupt service routine (ISR) tertentu. Sebenarnya merupakan hal yang krusial bagi sistem operasi real time untuk mengurangi interrupt latency untuk menjamin bahwa real time task menerima perhatian yang cepat.

Satu faktor yang penting dalam berkontribusi untuk interrupt latency adalah jumlah waktu interupsi dapat di disable ketika kernel struktur data sedang diperbaiki. Sistem operasi real time membutuhkan interupsi untuk didisable untuk periode waktu yang sangat pendek. Walaupun demikian untuk sistem waktu nyata keras, interrupt latency tidak boleh hanya diminimalisir. Teknik yang paling efektif untuk menjaga dispatch latency, yaitu yang terdiri dari dua komponen yang merupakan tahapan konflik pada dispatch latency:

1. Preemptif pada setiap proses berjalan dikernel
2. Dikeluarkan oleh sumber daya dari proses berpriporitas rendah

Satu isu yang dapat menyebabkan dispatch latency bertambah ketika proses berprioritas rendah butuh untuk membaca atau memodifikasi data kernel yang saat itu sedang diakses oleh oleh proses prioritas rendah- atau sebuah rantai proses berprioritas rendah. Sebagai kernel data yang dilindungi dengan sebuah lock, sebuah proses berprioritas tinggi harus menunggu yang prioritasnya rendah dalam menyelesaikan sumber dayanya. Situasi menjadi lebih rumit ketika proses prioritas rendah preemptif dengan proses lain yang prioritasnya lebih tinggi. Misalnya ada tiga proses L, M, H yang prioritasnya sebagai berikut L < M < H. Diasumsikan proses H membutuhkan sumber daya R, yang sedang diakses oleh proses L. Sekarang misalkan proses M dapat berjalan, menyelak proses L. Secara tidak langsung, proses dengan prioritas rendah, yaitu proses M, yang telah menyebabkan proses H semakin lama menunggu L untuk menyerahkan sumberdaya R.

Masalah ini dikenal dengan inversi prioritas, yang dapat dipecahkan dengan protokol priority-inheritance. Dengan adanya protokol ini, ketika ada sebuah proses yang sedang menggunakan sumber daya, kemudian ada proses lain dengan prioritas yang lebih tinggi yang juga membutuhkan sumber daya tersebut, maka proses tersebut akan mewariskan prioritasnya. Hal ini terjadi sampai dengan sumber daya selesai dieksekusi. Untuk contoh kasus diatas, maka L akan mewarisi prioritas H. Hal ini menyebabkan H tidak dapat menyelak untuk mengakses sumber daya R. Nilai dari L akan dikembalikan seperti semula sampai dengan R selesai digunakan. Hal ini menyebabkan H dapat langsung mengakses R, setelahnya.

49.3. Penjadual Proses

Penjadualan untuk sistem pada sistem waktu nyata lembut tidak memberikan jaminan kapan proses yang kritis akan dijadualkan, akan tetapi memberikan jaminan bahwa proses tersebut akan didahulukan daripada proses yang tidak kritis. Untuk sistem waktu nyata keras, persyaratan

penjadualan lebih ketat, yaitu berdasarkan deadline. Sebuah task harus dilayani berdasarkan deadline-nya. Sehingga ketika deadline sudah kadaluarsa (melebihi deadline), maka task tidak akan mendapat pelayanan.

Karakteristik dari proses yang ada pada sistem waktu nyata adalah proses tersebut dianggap periodik, karena membutuhkan CPU pada interval yang konstan (periode). Setiap proses berperiode memiliki waktu pemrosesan yang fix yaitu w, setiap kali mendapatkan CPU, juga memiliki sebuah deadline d ketika harus dilayani oleh CPU, dan sebuah periode p, yang dapat diekspresikan sebagai $0 < t < d < p$. Rate dari task berpriodik adalah 1/p.

Hal yang tidak biasa dari bentuk penjadualan ini adalah, proses akan mengumumkan deadlinenya pada penjadual. Kemudian dengan algoritma admission-control, penjadual akan menyatakan bahwa proses akan diselesaikan tepat waktu atau ditolak permohonannya karena tidak dapat menjamin bahwa task akan dapat dilayani sesuai deadline.

Berikut merupakan algoritma penjadualan untuk sistem waktu nyata keras.

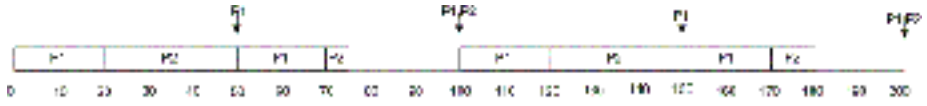
49.3.1. Penjadualan Rate-Monotonic

Algoritma ini menjadualkan task berpriodik berdasarkan ketentuan prioritas statik dengan preemption. Jika proses berprioritas rendah sedang jalan dan prioritas tinggi siap untuk jalan, maka akan didahulukan proses dengan prioritas rendah.Untuk memasuki sistem, setiap task berpriodik mendapatkan prioritas dengan inversi dari periodenya. Semakin rendah periodenya makan akan semakin tinggi prioritasnya, dan demikian pula sebaliknya. Ketentuan ini sebenarnya memprioritaskan proses yang lebih sering menggunakan CPU.

Sebagai contoh adalah proses P1 dan P2, dimana periode P1 yaitu 50 (p1=50) dan periode dari P2 adalah 100 (p2=100). Sedangkan waktu pemrosesannya adalah t1=20 dan t2=35. Deadline dari proses mempersyaratkan untuk menyelesaikan CPU burst-nya pada awal dari periode berikutnya. Utilisasi CPU dari proses P1 yang merupakan rasio dari t1/p1, adalah 20/50=0,40 dan utilisasi CPU dari P2 adalah 35/100=0,35 sehingga total utilisasi CPU-nya adalah 0,75%.Dengan ini, tampaknya dapat memenuhi deadline dan menyisakan burst time.

Dengan menggunakan penjadualan rate-monotonic maka P1 akan mendapat prioritas lebih tinggi dari P2, karena P1 lebih pendek daripada P2. P1 mulai terlebih dahulu dan menyelesaikan CPU burst pada waktu 20 (memenuhi deadline pertama). Kemudian dilanjutkan dengan P2 sampai dengan waktu 50, dimana tersisa 5 ms pada CPU burst-nya. P1 melanjutkan sampai waktu 70 (memenuhi deadline kedua), kemudian P2 menyelesaikan CPU burst-nya pada waktu 75 (memenuhi deadline pertama dari P2).

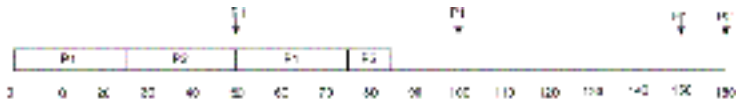
Gambar 49.1. Gambar ??



Sumber: Silberschatz

Algoritma rate-monotonic dianggap optimal apabila sebuah set proses tidak dapat dijadualkan oleh algoritma ini, serta tidak dapat pula dijadualkan oleh algoritma lain yang menggunakan prioritas statik. Sebagai contoh, diasumsikan proses P1 memiliki periode p1=50 dan CPU burst t1=25. Selain itu, ada proses P2 yang memiliki nilai p2=80 dan t2=35. Maka, proses P1 akan mendapat prioritas lebih besar. Total utilisasi CPU dari 2 proses diatas adalah (25/50) + (35/80)=0,94. Nampaknya, secara logikal keduanya dapat dijadualkan dan masih meninggalkan CPU dengan 6% waktu tersedia. Awalnya, P1 berjalan sampai dengan menyelesaikan CPU burst-nya pada waktu 25. Kemudian dilanjutkan dengan proses P2 yang berjalan sampai dengan waktu 50. P2 masih menyelesaikan burstnya 10. Kemudian dilanjutkan lagi oleh P1 sampai dengan waktu 75. Namun, P1 kehilangan deadline-nya untuk menyelesaikan burst-nya, yaitu pada waktu 80.

Gambar 49.2. Gambar ??



Sumber: Silberschatz

Selain menjadi optimal, penjadual rate-monotonic memiliki batasan utilisasi CPU yang terbatas dan tidak selalu mungkin untuk memaksimalkan secara penuh sumber daya CPU. Kemungkinan buruk utilisasi CPU untuk penjadualan N proses adalah

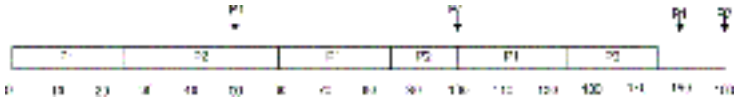
$$\frac{1/n}{2(2^{1/n} - 1)}$$

49.3.2. Penjadualan Earliest-Deadline-First (EDF)

Penjadualan dilakukan berdasarkan deadline, yaitu semakin dekat deadline-nya maka semakin tinggi prioritasnya, dan demikian pula sebaliknya.Ketentuan yang berlaku, ketika proses akan mulai jalan, maka proses akan mengumumkan syarat deadline-nya pada sistem. Prioritas harus ditentukan untuk merefleksikan deadline dari proses yang baru dapat berjalan.

Sebagai contoh adalah kasus yang tidak dapat diselesaikan dengan penjadual rate-monotonic. Pada kasus ini, dengan EDF maka P1 akan mendapat prioritas awal lebih tinggi dari P2 karena P1 lebih mendekati deadline dari P2. Kemudian dilanjutkan oleh P2 sampai dengan akhir burst time dari P1. Apabila rate-monotonic membiarkan P1 untuk melanjutkan kembali, maka pada EDF, P2 (dateline pada 80) yang melanjutkan karena lebih dekat dengan deadline daripada P1 (pada 100). Pada waktu 85 kedua proses sudah menyelesaikan deadline-nya masing-masing. Kemudian berlanjut lagi dengan P2, sampai dengan 100 maka P1 didahulukan kembali karena dateline-nya lebih awal dari P2.

Gambar 49.3. Gambar ??



Berbeda dengan algoritma rate-monotonic adalah, penjadual EDF tidak membutuhkan proses untuk periodik, dan tidak juga harus membutuhkan jumlah waktu CPU per burst yang konstan. Syarat satu-satunya adalah proses mengumumkan deadline-nya pada penjadual ketika dapat jalan. Secara teoritis, algoritma ini optimal, yaitu dapat memenuhi semua dateline dari proses dan juga dapat menjadikan utilisasi CPU menjadi 100%. Namun dalam kenyataan hal tersebut sulit terjadi karena cost dari context switching antara proses dan interrupt handler.

49.3.3. Penjadualan Proportional Share

Penjadual ini akan mengalokasikan T bagian di antara semua aplikasi. Sebuah aplikasi dapat menerima N bagian waktu, yang menjamin bahwa aplikasi akan memiliki N/T dari total waktu prosesor. Sebagai contoh, diasumsikan ada total dari T=100 bagian untuk dibagi diantara tiga proses, yaitu A, B, dan C. A mendapatkan 50 bagian, B mendapat 15 bagian dan C mendapat 20 bagian. Hal ini menjamin bahwa A akan mendapat 50% dari total proses, B mendapar 15% dari total proses dan C mendapat 20% dari total proses.

Penjadualan proportional share harus bekerja dengan memasukkan ketentuan admission control

untuk menjamin bahwa aplikasi mendapatkan alokasi pembagian waktunya. Ketentuan admission control hanya akan menerima permintaan client terhadap sejumlah bagian apabila bagian yang diinginkan tersedia.

49.4. Penjadual Disk

Penjadualan disk yang telah kita pelajari pada bab sebelumnya memfokuskan untuk menangani data yang konvensional, yang sasarannya adalah fairness dan troughput. Sedangkan pada penjadualan waktu nyata dan multimedia yang menjadi tujuan adalah lebih kepada untuk mengatasi hambatan yang tidak dimiliki oleh data konvensional, yaitu: timing deadline dan rate requirement, sehingga dapat memiliki jaminan QoS. Namun sayangnya, kedua hambatan tersebut sering berkonflik. Berkas yang continous-media tipikalnya membutuhkan disk bandwith rate yang sangat tinggi untuk memenuhi requirement data rate mereka. Karena disk memiliko transfer rate yang relatif rendah dan latency rate yang relatif tinggi maka penjadual disk harus mengurangi waktu latency untuk menjamin bandwith yang tinggi. Bagaimanapun, mengurangi waktu latency dapat menyebabkan polisi dari penjadualan yang tidak memberikan prioritas pada deadline.

Berikut merupakan algoritma penjadualan yang memenuhi persyaratan QoS untuk sistem continous-media:

49.4.1. Penjadualan Earliest Deadline first (EDF)

Penjadualan EDF yang digunakan pada penjadualan proses pada subbab sebelumnya, dapat digunakan pula untuk melakukan penjadualan disk. EDF mirip dengan shortest-seek-time-first (SSTF), kecuali melayani permintaan terdekat dengan silinder saat itu karena EDF melayani permintaan yang terdekat dengan deadline. Algoritma ini cocok untuk karakter multimedia yang butuh respon secepat mungkin.

Masalah yang dihadapi dari pendekatan ini adalah pelayanan permintaan yang kaku berdasarkan deadline akan memiliki seek time yang tinggi, karena head dari disk harus secara random mencari posisi yang tepat tanpa memperhatikan posisinya saat ini.

Sebagai contoh adalah disk head pada silinder 75 dan antrian dari silinder (diurutkan berdasarkan deadline) adalah 98, 183, 105. Dengan EDF, maka head akan bergerak dari 75, ke 98, ke 183, dan balik lagi ke 105 (head melewati silinder 105 ketika berjalan dari 98 ke 183). Hal ini memungkinkan penjadual disk telah dapat melayani permintaan silinder 105 selama perjalanan ke silinder 183 dan masuh dapat menjaga persyaratan deadline dari silinder 183.

49.4.2. Penjadualan SCAN-EDF

Masalah dasar dari penjadualan 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 penjadualan FCFS dimana akhirnya dimunculkan penjadualan SCAN, yang menjadi solusi.

SCAN-EDF merupakan algoritma hibrida dari kombinasi penjadualan EDF dengan penjadualan SCAN. SCAN-EDF dimulai dengan EDF ordering tetapi permintaan pelayanan dengan deadline yang sama menggunakan SCAN order. Apa yang terjadi apabila beberapa permintaan memiliki deadline yang berbeda yang relatif saling tertutup? Pada kasus ini, SCAN-EDF akan menumpuk permintaan, menggunakan SCAN ordering untuk melayani permintaan pelayanan yang ada dalam satu tumpukan. Ada banyak cara menumpuk permintaan dengan deadline yang mirip;satu-satunya syarat adalah reorder permintaan pada sebuah tumpukan tidak boleh menghalangi sebuah permintaan untuk dilayani berdasarkan deadlinenya. Apabila deadline tersebar merata, tumpukan dapat diatur pada grup pada ukuran tertentu. Pendekatan yang lain adalah dengan menumpuk permintaan yang deadlinenya jatuh pada treshold waktu yang diberikan, misalnya 10 permintaan pertumpukan.

Pendekatan lain adalah dengan menumpuk permintaan yang deadline-nya jatuh pada threshold waktu yang diberikan, misalnya 100 ms.

49.4.3. Managemen Berkas

Managemen berkas merupakan salah satu komponen dalam sebuah sistem operasi. Sebuah komputer dapat menyimpan informasi di dalam media yang berbeda-beda, seperti magnetic tape, disk, dan drum. Setiap perangkat tersebut memiliki karakteristik yang berbeda-beda.

Untuk kenyamanan dalam penggunaan sistem komputer, sistem operasi menyeragamkan penyajian suatu informasi kepada penggunanya karena seperti yang kita ketahui bahwa informasi yang disimpan dalam media penyimpanan sebenarnya tidaklah berwujud seperti apa yang kita lihat. Adalah tugas sistem operasi untuk melakukan mapping atas informasi yang tersimpan di dalam sebuah media penyimpanan ke dalam perangkat fisik sebagai perangkat akhir dimana pengguna dapat memperoleh informasi tersebut.

Berkas merupakan kumpulan informasi yang berhubungan (sesuai dengan tujuan pembuatan berkas tersebut). Sebuah berkas juga dapat memiliki struktur yang bersifat hirarkis (direktori, volume, dan lain-lain).

Sebuah sistem operasi memiliki tanggung jawab pada berkas sebagai berikut:

1. Pembuatan dan penghapusan sebuah berkas
2. Pembuatan dan penghapusan sebuah direktori
3. Pemanipulasian sebuah berkas atau direktori
4. Mapping berkas ke secondary storage
5. Melakukan back-up sebuah berkas ke media penyimpanan yang bersifat permanen (nonvolatile)

Karakteristik sistem multimedia:

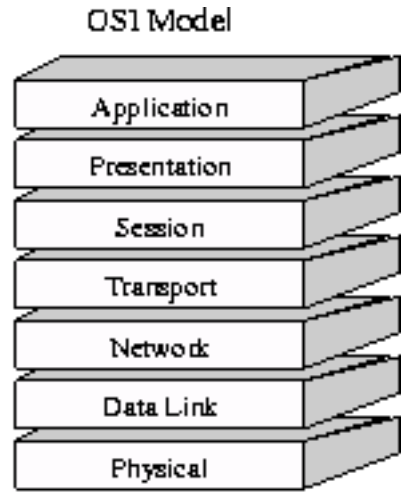
1. Berkas multimedia biasanya memiliki karakteristik memiliki ukuran yang besar, untuk itu dalam mengatur atau memanagemen berkas mulitimedia diperlukan alokasi tempat dalam storage yang cukup besar pula. Selain itu untuk menjamin sebuah berkas multimedia dapat diakses dan dieksekusi dengan baik, diperlukan ukuran memori yang cukup serta spesifikasi lain dari komputer.
2. Memerlukan data rates yang sangat tinggi. Misalnya dalam sebuah video dijital, dimana frame dari video yang ingin ditampilkan beresolusi 800 x 600. Apabila kita menggunakan 24 bits untuk merepresentasikan warna pada tiap piksel, tiap frame berarti membutuhkan 800 x 600 x 24 = 11.520.000 bits data. Jika frame- frame tersebut ditampilkan pada kecepatan 30 frame/detik, maka bandwidth yang diperlukan adalah lebih dari 345 Mbps.
3. Aplikasi mutlimedia sensitif terhadap timing delay selama pemutaran ulang. Setiap kali berkas countinous-media dikirim kepada klien, pengiriman harus kontinu pada kecepatan tertentu selama pemutaran media tersebut. Jika tidak demikian, pendengar atau penonton dari berkas multimedia tersebut akan terganggu dan cenderung untuk melakukan pause dari presentasi berkas multimedia tersebut.

49.5. Managemen Jaringan

Sistem terdistribusi adalah sekumpulan prosesor yang tidak berbagi memori atau clock. Setiap prosesor mempunyai memori dan clock sendiri. Prosesor-prosesor tersebut terhubung melalui

jaringan komunikasi sistem terdistribusi yang menyediakan akses pengguna ke bermacam- macam sumber daya sistem. Akses tersebut menyebabkan peningkatan kecepatan komputasi dan meningkatkan kemampuan penyediaan data.

Gambar 49.4. Gambar



Fungsi utama dari jaringan itu sendiri merupakan komunikasi. Model acuan dalam mempelajari jaringan komputer pada buku ini adalah dengan OSI. Gambar model OSI ini dapat dilihat pada Gambar 49.6.-1 dimana dapat kita lihat adanya lapisan network/jaringan yang mengatur pengiriman paket data pada subnet. Fungsi utama dari lapisan jaringan ini adalah:

1. Menyediakan service terhadap lapisan transport
2. Menentukan rute pengiriman paket, apakah connection atau connectionless.
3. Mengontrol dan memanagemen jaringan, dengan menjaga:
    a. kestabilan, tidak macet dalam pengiriman sebuah paket.
    b. mengontrol arus paket (flow control) dalam jaringan.

Ketika suatu data dikirim melalui jaringan, prosesi transmisi yang berlangsung pasti mengalami hambatan atau keterlambatan yang disebabkan oleh lalu lintas jaringan yang begitu padat. Dalam kaitannya dengan multimedia, pengiriman data dalam sebuah jaringan harus memperhatikan masalah waktu. Yakni penyampaian data kepada klien harus tepat waktu atau paling tidak dalam batas waktu yang masih bisa ditoleransi.

Sebuah protokol yang dapat memperhatikan masalah waktu tersebut adalah real-time transport protocol (RTP). RTP adalah sebuah standar internet untuk melakukan pengiriman data secara real-time, yang meliputi audio dan video.

49.6. Unicasting dan Multicasting

Secara umum, terdapat tiga metode untuk melakukan pengiriman suatu data dari server ke klien dalam sebuah jaringan. Ketiga metode tersebut adalah:

1. Unicasting

Server mengirim data ke klien tunggal. Apabila data ingin dikirim ke lebih dari satu klien, maka server harus membangun sebuah unicast yang terpisah untuk masing-masing klien.

2. Broadcasting

   Server mengirim data ke semua klien yang ada meskipun tidak semua klien meminta/membutuhkan data yang dikirim oleh server.

3. Multicasting

   Server mengirim data ke suatu grup penerima data (klien) yang menginginkan data tersebut. Metode ini merupakan metode yang berada di pertengahan metode unicasting dan broadcasting.

49.7. Real-Time Streaming Protocol

Cara kerja dari media yang di-stream dapat dikirim oleh seorang klien dari sebuah standar web server adalah dengan pendekatan yang menggunakan hypertext transport protocaol atau HTTP yang merupakan protokol yang biasa digunakan untuk mengirim dokumen dari web browser. Biasanya, seorang klien menggunakan media player, seperti QuickTime, RealPlayer, atau Windows Media Player untuk memutar kembali media yang di-stream dari sebuah web server. Biasanya, pertama-tama klien akan meminta metafile, yang meliputi keterangan alamat letak data yang di-stream berada (URL), dari data yang di-stream. Metafile ini nantinya akan dikirim ke web browser klien, kemudian browser akan membuka berkas yang dimaksud dengan memilah media player yang sesuai dengan jenis media yang dispesifikasikan oleh metafile.

Masalah yang dihadapi dari pengiriman data yang di-stream dari standar web server adalah bahwa HTTP merupakan protokol yang tidak memiliki status. Sehingga, web server tidak dapat menjaga status dari koneksinya dengan klien. Keadaan ini mengakibatkan kesulitan yang dialami klien manakala ia ingin melakukan pause saat pengiriman data yang distream. Hal ini dikarenakan pelaksanaan pause membutuhkan pengetahuan (biasanya status) dari web browser, sehingga ketika klien ingin memulai kembali mengirim data melalui streaming web server dapat dengan mudah memutar kembali berdasarkan status yang disimpan tersebut.

Strategi alternatif yang dapat dilakukan untuk menanggulangi hal diatas adalah dengan menggunakan server streaming khusus yang didesain untuk men-streaming media, yaitu real-time streaming protocol (RTSP). RTSP didesain untuk melakukan komunikasi antara server yang melakukan streaming dengan media player. Keuntungan RTSP adalah bahwa protokol ini menyediakan koneksi yang memiliki status antara server dan klien, yang dapat mempermudah klien ketika ingin melakukan pause atau mencari posisi random dalam stream ketika memutar kembali data.

RTSP memiliki empat buah perintah. Perintah ini dikirim dari klien ke sebuah server streaming RTSP. Keempat perintah tersebut adalah:

1. SETUP

   server mengalokasikan sumber daya kepada client session

2. PLAY

   server mengirim sebuah stream ke client session yang telah dibangun dari perintah SETUP sebelumnya

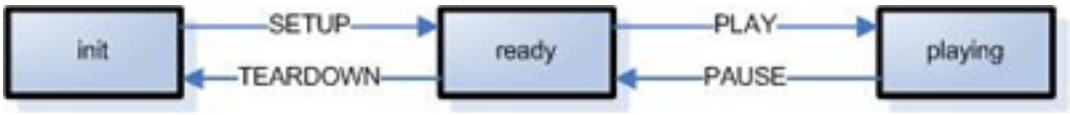
3. PAUSE

   server menunda pengiriman stream namun tetap menjaga sumber daya yang telah dialokasikan

4. TEARDOWN

server memutuskan koneksi dan membebastugaskan sumber daya yang sebelumnya telah digunakan

Gambar 49.5. Mesin finite-state yang merepresentasikan RSTP



Sumber: Silberschatz

49.8. Kompresi

Kompresi merupakan pengurangan ukuran suatu berkas menjadi ukuran yang lebih kecil dari aslinya. Pengompresian berkas ini sangat menguntungkan manakala terdapat suatu berkas yang berukuran besar dan data di dalamnya mengandung banyak pengulangan karakter. Adapun teknik dari kompresi ini adalah dengan mengganti karakter yang berulang-ulang tersebut dengan suatu pola tertentu sehingga berkas tersebut dapat meminimalisasi ukurannya.

Misalnya terdapat kata "Hari ini adalah hari Jum'at. Hari Jum'at adalah hari yang menyenangkan". Jika kita telaah lagi, kalimat tersebut memiliki pengulangan karakter seperti karaktek pembentuk kata hari, hari Jum'at, dan adalah. Dalam teknik sederhana kompresi pada perangkat lunak, kalimat di atas dapat diubah menjadi pola sebagai berikut

ini $ %. % $ # ya@ menyena@kan.

dimana dalam kalimat diatas, karakter pembentuk hari diubah menjadi karakter #, hari Jum'at menjadi %, adalah menjadi $, ng menjadi @. Saat berkas ini akan dibaca kembali, maka perangkat lunak akan mengembalikan karakter tersebut menjadi karakter awal dalam kalimat. Pengubangan karakter menjadi lebih singkat hanya digunakan agar penyimpanan kalimat tersebut dalam memory komputer tidak memakan tempat yang banyak.

Implementasi kompresi dalam personal computer (PC) dibagi menjadi tiga cara, yaitu:

1. Pengkompresian berkas berdasarkan kegunaannya

   (Utility-based file compression) Merupakan jenis kompresi yang melakukan kompresi per berkas dengan menggunakan utilitas kompresi. Untuk melihat berkas yang telah dikompresi, berkas tersebut harus didekompres dengan menggunakan utilitas dekompresi. Dalam jenis ini, sistem operasi tidak tahu menahu mengenai aktivitas kompresian atau dekompresi sebuah berkas. Contoh dari sistem kompresi yang cukup terkenal adalah PKZIP, WinZip, WinRar, dan lain-lain.

2. Pengkompresian berkas pada sistem operasi

   (Operating system file compression) Beberapa sistem operasi memiliki sistem kompresi di dalamnya. Contoh dari sistem operasi yang memiliki sistem kompresi di dalamnya adalah Windows NT yang menggunakan sistem berkas NTFS. Dengan menggunakan sistem kompresi seperti ini, sistem operasi secara otomatis dapat mendekompres berkas yang telah dikompresi manakala berkas tersebut ingin digunakan oleh sebuah program.

3. Pengkompresian Isi (Volume compression)

   dengan pengkompresian ini, kita dapat menghemat penggunaan space pada disk tanpa harus mengkompresi tiap berkas di dalamnya secara individual. Setiap berkas yang dikopi ke dalam volume compression akan dikompresi secara otomatis dan akan didekompresi secara otomatis

apabila berkas tersebut dibutuhkan oleh sebuah program.

Dalam kaitannya dengan mutltimedia, kompresi sangat menguntungkan karena dapat menghemat tempat penyimpanan serta dapat mempercepat proses pengiriman data kepada klien, sebab pengiriman berkas dengan ukuran yang lebih kecil lebih cepat daripada berkas yang memiliki ukuran besar. Kompresi juga penting manakala suatu data di-stream melalui sebuah jaringan.

Algoritma kompresi diklasifikasikan menjadi dua buah, yaitu:

1. Algoritma kompresi lossy

   Keuntungan dari algoritma ini adalah bahwa rasio kompresi (perbandingan antara ukuran berkas yang telah dikompresi dengan berkas sebelum dikompresi) cukup tinggi. Namun algoritma ini dapat menyebabkan data pada suatu berkas yang dikompresi hilang ketika didekompresi. Hal ini dikarenakan cara kerja algoritma lossy adalah dengan mengeliminasikan beberapa data dari suatu berkas. Namun data yang dieliminasikan biasanya adalah data yang kurang diperhatikan atau diluar jangkauan manusia, sehingga pengeliminasian data tersebut kemungkinan besar tidak akan mempengaruhi manusia yang berinteraksi dengan berkas tersebut. Contohnya pada pengkompresian berkas audio, kompresi lossy akan mengeleminasi data dari berkas audio yang memiliki frekuensi sangat tinggi/rendah yang berada diluar jangkauan manusia. Beberapa jenis data yang biasanya masih dapat mentoleransi algoritma lossy adalah gambar, audio, dan video.

2. Algoritma kompresi lossless

   Berbeda dengan algoritma kompresi lossy, pada algoritma kompresi lossless, tidak terdapat perubahan data ketika mendekompresi berkas yang telah dikompresi dengan kompresi lossless ini. Algoritma ini biasanya diimplementasikan pada kompresi berkas teks, seperti program komputer (berkas zip, rar, gzip, dan lain-lain).

Contoh lain dari penerapan algoritma kompresi lossy dalam kehidupan sehari-hari adalah pada berkas berformat MPEG (Moving Picture Experts Group). MPEG merupakan sebuah set dari format berkas dan standar kompresi untuk video digital.

49.9. Rangkuman

Waktu nyata dapat dibagi dua, yaitu waktu nyata keras dan waktu nyata lembut.Sistem waktu nyata keras menjamin pekerjaan yang kritis akan diselesaikan dengan tepat waktu. Sedangkan sistem yang dikatakan seba- gai sistem waktu nyata lembut adalah sistem yang batasannya kurang, dimana pekerjaan yang kritis mendapat prioritas setelah pekerjaan yang lain.

Algoritma Penjadual Waktu Nyata diantaranya adalah Earliest Deadline first (EDF) dan SCAN-EDF Schedulling

Managemen berkas merupakan salah satu komponen dalam sebuah sistem operasi. Sebuah sistem operasi memiliki tanggung jawab pada berkas sebagai seperti dalam pembuatan dan penghapusan sebuah berkas, pembuatan dan penghapusan sebuah direktori, pemanipulasian terhadap sebuah berkas atau direktori, memetakan berkas ke secondary storage, serta melakukan back-up sebuah berkas ke media penyimpanan yang bersifat permanen (non-volitile).

Fungsi utama dari jaringan adalah untuk komunikasi. Secara umum, terdapat tiga metode untuk melakukan pengiriman suatu data dari server ke klien dalam sebuah jaringan. Ketiga metode tersebut adalah unicasting, broadcasting, dan multicasting. Real-stream streaming protocol memiliki empat perintah, yaitu setup, play, pause dan teardown.

Kompresi merupakan pengurangan ukuran suatu berkas menjadi ukuran yang lebih kecil dari aslinya. Implementasi kompresi dalam personal computer (PC) dibagi menjadi tiga cara, yaitu pengkompresian berkas berdasarkan kegunaannya (Utility-based file compression), pengkompresian

berkas pada sistem operasi (Operating system file compression), dam pengkompresian isi (Volume compression). Algoritma kompresi yang cukup dikenal adalah algoritma lossy dan lossless

49.10. Latihan

1. Apa yang membedakan antara sistem waktu nyata lembut dan sistem waktu nyata keras?
2. Kenapa memori virtual tidak dipakai pada sistem waktu nyata?
3. Misalkan ada proses, P1 dan P2 dimana p1=50, t1=25, p2=75 dan t2=30.
    a. Dapatkah 2 proses ini dijadualkan menggunakan rate-monotonic? Ilustrasikan jawabanmu dengan menggunakan Gantt chart.
    b. Ilustrasikan penjadualan kedua proses tersebut dengan menggunakan EDF.
4. Jenis fitur yang bagaimana yang membedakan multitasking yang konvensional dengan yang multimedia?
5. Sebutkan dan jelaskan algoritma yang sesuai untuk kasus waktu nyata?
6. Sebut dan jelaskan apa yang menjadi tanggung jawab sebuah sistem operasi terhadap berkas!
7. Sebut dan jelaskan tiga metode untuk melakukan pengiriman suatu data dari server ke klien dalam sebuah jaringan!
8. Apakah yang dimaksud dengan kompresi dan apakah tujuan serta keuntungan penggunaan kompresi?
9. Sebutkan tiga jenis implementasi kompresi dalam sebuah personal computer (PC)!
10. Sebut dan jelaskan dua buah algoritma kompresi yang cukup dikenal!

49.11. Rujukan

Silberschatz, Galvin, dan Gagne. 2002. Applied Operating System, Edisi pertama. John Wiley & Sons.

Silberschatz, Galvin, dan Gagne. 2006. Applied Operating System, Edisi ketujuh. John Wiley & Sons.

http://www.netnam.vn/unescocourse/os/13.htm

http://www.storagereview.com/guide2000/ref/hdd/file/comprTypes.html

http://www.storagereview.com/guide2000/ref/hdd/file/compr.html

http://www.storagereview.com/guide2000/ref/hdd/file/comprWhy.html

Bab 50. Sistem Terdistribusi

50.1. Pendahuluan

Pengguna tidak sadar akan multiplisitas perangkat. Akses terhadap sumber daya jarak jauh sama dengan sumber daya lokal.

1. Remote loging ke dalam perangkat jarak jauh yang sesuai.
2. Migrasi Data dengan memindahkan seluruh data atau hanya mentransfer file yang diperlukan untuk task langsung.
3. Computation Migration, memindahkan komputasi bukannya data, melalui sistem.

Process Migration: Mengeksekusi seluruh proses arat bagian daripadanya pada situs/tempat yang berbeda.

1. Load balancing
2. Computation Speedup
3. Hardware preference
4. Software Preference
5. Data Access

Tipe-tipe koneksi

1. LAN=di desain untuk mencakup area yang kecil.
2. WAN=menghubungkan situs-situs yang terpisah secara geografi

Aplikasi

1. Transmisi dari sebuah paket network antar host dalam sebuah jaringan ethernet.
2. Setiap host mempunyai sebuah IP unik dan sebuah alamat ethernet yang berkorespondensi
3. Komunikasi membutuhkan alamat keduanya.
4. DNS yang dapat digunakan untuk mendapatkan alamat IP.
5.

Sistem berkas

Sebuah implementasi dari model klasik time sharing dari sebuah sistem file, dimana banyak user dapat berbagi file dan penyimpanan resource.

Struktur sistem berkas

1. Servis: Software entity yang berjalan pada sebuah atau lebih hardware dan menyediakan sebuah tipe fungsi partikular pada client yang tidak dikenal
2. Server: Software servis yang berjalan pada sebuah mesin
3. Client: Proses yang dapat mengivoke sebuah servis mengunakan set operasi yang membentuk client interfacenya

Replikasi

1. Replika dari file yang sama di dalam mesin yang failure-independent
2. Memperbaiki ketersediaan dan dapat mempersingkat waktu servis
3. Schema penamaan memetakan sebuah nama file yang di replikasi pada sebuah replika yang partikulir.
4. Update, sebuah update terhadap replika apapun harus direfleksikan pada replika-replika yang lain.

Mutex

Asumsi

Sistem terdiri dari n proses, tiap proses berada di dalam processor yang berbeda Setiap proses mempunyai critical section yang membutuhkan mutual exclusion

Requirement

Jika sebuah proses sedang mengeksekusi critical section, maka tidak ada proses lain yang mengeksekusi critical sectionnya.

Terdapat dua pendekatan yaitu:

1. Centralized Approach
    a. Satu dari proses dalam sistem dipilih untuk mengkoordinasikan entry terhadap critical section.
    b. Sebuah proses yang ingin memasuki critical section meminta coordinator dengan mengirimkan pesan.
    c. Koordinator memutuskan proses mana yang dapat memasuki critical section, dengan mengirimkan pesan balasan.
    d. Ketika sebuah proses menerima pesan dari koordinator maka ia memasuki critical section
    e. Setelah mengakhiri critical sectionnya, sebuah proses mengirimkan pesan lagi kepada koordinator. Lalu keluar dari C. S.
2. Fully Distributed Approach
    a. Ketika sebuah proses ingin memasuki critical sectionya,ia mengenerate sebuah timestamp baru dan mengirimkan pesan permohonan kepada seluruh proses lain
    b. Ketika proses menerima pesan permohonan, ia dapat langsung membalas atau mengirim

reply back

c. Ketika pesan tersebut menerima reply dari seluruh proese lain, ia dapat memasuki critical section.

d. Ketika ia mengakhiri C.S. proses mengirimkan reply message.

50.2. Variasi Sistem

FIXME

50.3. Topologi Jaringan

Situs dalam sistem dapat terkoneksiscara fisik, dalam berbagai cara, yang dibandingkan terhadap berbagai kriteria tertentu.

1. Biaya dasar, yaitu berapa biaya untuk menghubungkan berbagai situs dalam sistem.
2. Biaya komunikasi, yiatu berapa waktu yang dibutuhkan untuk membawa sebuah pesan dari situs A ke situs B.
3. Relibilitas, jika sebuah link dalam sebuah situs terputus, apakan situs-situs yang lain dapat berfungsi dengan normal.

50.4. Sistem Berkas

FIXME

50.5. Replikasi Berkas

FIXME

50.6. Mutex

FIXME

50.7. *Middleware*

FIXME

50.8. Aplikasi

FIXME

50.9. Kluster

FIXME

50.10. Rangkuman

FIXME

50.11. Latihan

1. FIXME

50.12. Rujukan

FIXME

Bab 51. Keamanan Sistem

51.1. Pendahuluan

Pertama-tama kita harus mengetahui perbedaan antara keamanan dan proteksi? Proteksi menyangkut mengenai faktor-faktor internal suatu sistem komputer. Sedangkan keamanan mempertimbangkan faktor-faktor eksternal (lingkungan) di luar sistem dan faktor proteksi terhadap sumber daya sistem. Melihat perbedaan ini, terlihat jelas bahwa keamanan mencakup hal yang lebih luas dibanding dengan proteksi.

Bagaimana suatu sistem dapat dikatakan aman? Suatu sistem baru dapat dikatakan aman apabila resource yang digunakan dan diakses sesuai dengan kehendak user dalam berbagai keadaan. Sayangnya, tidak ada satu sistem komputer pun yang memiliki sistem keamanan yang sempurna. Data atau informasi penting yang seharusnya tidak dapat diakses oleh orang lain mungkin dapat diakses, dibaca ataupun diubah oleh orang lain.

Oleh karena itu dibutuhkan suatu keamanan sistem untuk menanggulangi kemungkinan dimana informasi penting dapat diakses oleh orang lain. Diatas dijelaskan bahawa tidak ada satu sistem komputer yang memiliki sistem keamanan yang sempurna. Akan tetapi, setidaknya kita harus mempunyai suatu mekanisme yang membuat pelanggaran semacam itu jarang terjadi.

Dalam bab ini kita akan membahas hal-hal yang menyangkut mengenai suatu keamanan sistem, dengan mempelajarinya diharapkan akan membantu kita mengurangi pelanggaran-pelanggaran yang dapat terjadi.

51.2. Manusia dan Etika

Berbicara mengenai manusia dan etika, kita mengetahui bahwa di muka bumi ini terdapat bermacam-macam karakter orang yang berbeda-beda. Sebagian besar orang memiliki hati yang baik dan selalu mencoba untuk menaati peraturan. Akan tetapi, ada beberapa orang jahat yang ingin menyebabkan kekacauan. Dalam konteks keamanan, orang-orang yang membuat kekacauan di tempat yang tidak berhubungan dengan mereka disebut intruder. Ada dua macam intruder, yaitu:

1. Passive intruder

   Intruder yang hanya ingin membaca file yang tidak boleh mereka baca.

2. Active intruder

   Lebih berbahaya dari passive intruder. Mereka ingin membuat perubahan yang tidak diijinkan (unauthorized) pada data.

Ketika merancang sebuah sistem yang aman terhadap intruder, penting untuk mengetahui sistem tersebut akan dilindungi dari intruder macam apa. Empat contoh kategori:

1. Keingintahuan seseorang tentang hal-hal pribadi orang lain.

   Banyak orang mempunyai PC yang terhubung ke suatu jaringan dan beberapa orang dalam jaringan tersebut akan dapat membaca e-mail dan file-file orang lain jika tidak ada 'penghalang' yang ditempatkan. Sebagai contoh, sebagian besar sistem UNIX mempunyai default bahwa semua file yang baru diciptakan dapat dibaca oleh orang lain.

2. Penyusupan oleh orang-orang dalam

   Pelajar, system programmer, operator, dan teknisi menganggap bahwa mematahkan sistem keamanan komputer lokal adalah suatu tantangan. Mereka biasanya sangat ahli dan bersedia mengorbankan banyak waktu untuk usaha tersebut.

3. Keinginan untuk mendapatkan uang.

   Beberapa programmer bank mencoba mencuri uang dari bank tempat mereka bekerja dengan cara-cara seperti mengubah software untuk memotong bunga daripada membulatkannya, menyimpan uang kecil untuk mereka sendiri, menarik uang dari account yang sudah tidak digunakan selama bertahun-tahun, untuk memeras ("Bayar saya, atau saya akan menghancurkan semua record bank anda").

4. Espionase komersial atau militer.

   Espionase adalah usaha serius yang diberi dana besar oleh saingan atau negara lain untuk mencuri program, rahasia dagang, ide-ide paten, teknologi, rencana bisnis, dan sebagainya. Seringkali usaha ini melibatkan wiretaping atau antena yang diarahkan pada suatu komputer untuk menangkap radiasi elektromagnetisnya.

Perlindungan terhadap rahasia militer negara dari pencurian oleh negara lain sangat berbeda dengan perlindungan terhadap pelajar yang mencoba memasukkan message-of-the-day pada suatu sistem. Terlihat jelas bahwa jumlah usaha yang berhubungan dengan keamanan dan proteksi tergantung pada siapa "musuh"nya.

51.3. Kebijaksanaan Pengamanan

Kebijaksanaan pengamanan yang biasa digunakan yaitu yang bersifat sederhana dan umum. Dalam hal ini berarti tiap pengguna dalam sistem dapat mengerti dan mengikuti kebijaksanaan yang telah ditetapkan. Isi dari kebijaksanaan itu sendiri berupa tingkatan keamanan yang dapat melindungi data-data penting yang tersimpan dalam sistem. Data-data tersebut harus dilindungi dari tiap pengguna yang menggunakan sistem tersebut.

Beberapa hal yang perlu dipertimbangkan dalam menentukan kebijaksanaan pengamanan adalah: siapa sajakah yang memiliki akses ke sistem, siapa sajakah yang diizinkan untuk menginstall program ke dalam sistem, siapa sajakah memiliki data-data tertentu, perbaikan terhadap kerusakan yang mungkin terjadi, dan penggunaan yang wajar dari sistem.

51.4. Keamanan Fisik

Lapisan keamanan pertama yang harus diperhitungkan adalah keamanan secara fisik dalam sistem komputer. Keamanan fisik menyangkut tindakan mengamankan lokasi adanya sistem komputer terhadap intruder yang bersenjata atau yang mencoba menyusup ke dalam sistem komputer.

Pertanyaan yang harus dijawab dalam menjamin keamanan fisik antara lain:

1. Siapa saja yang memiliki akses langsung ke dalam sistem?
2. Apakah mereka memang berhak?
3. Dapatkah sistem terlindung dari maksud dan tujuan mereka?
4. Apakah hal tersebut perlu dilakukan?

Banyak keamanan fisik yang berada dalam sistem memiliki ketergantungan terhadap anggaran dan situasi yang dihadapi. Apabila pengguna adalah pengguna rumahan, maka kemungkinan keamanan fisik tidak banyak dibutuhkan. Akan tetapi, jika pengguna bekerja di laboratorium atau jaringan komputer, banyak yang harus dipikirkan.

Saat ini, banyak komputer pribadi memiliki kemampuan mengunci. Biasanya kunci ini berupa socket pada bagian depan casing yang bisa dimasukkan kunci untuk mengunci ataupun membukanya. Kunci casing dapat mencegah seseorang untuk mencuri dari komputer, membukanya secara langsung untuk memanipulasi ataupun mencuri perangkat keras yang ada.

51.5. Keamanan Perangkat Lunak

Contoh dari keamanan perangkat lunak yaitu BIOS. BIOS merupakan perangkat lunak tingkat rendah yang mengkonfigurasi atau memanipulasi perangkat keras tertentu. BIOS dapat digunakan untuk mencegah penyerang mereboot ulang mesin dan memanipulasi sistem LINUX.

Contoh dari keamanan BIOS dapat dilihat pada LINUX, dimana banyak PC BIOS mengizinkan untuk mengeset password boot. Namun, hal ini tidak banyak memberikan keamanan karena BIOS dapat direset, atau dihapus jika seseorang dapat masuk ke case. Namun, mungkin BIOS dapat sedikit berguna. Karena jika ada yang ingin menyerang sistem, untuk dapat masuk ke case dan mereset ataupun menghapus BIOS akan memerlukan waktu yang cukup lama dan akan meninggalkan bekas. Hal ini akan memperlambat tindakan seseorang yang mencoba menyerang sistem.

51.6. Keamanan Jaringan

Pada dasarnya, jaringan komputer adalah sumber daya (resource) yang dishare dan dapat digunakan oleh banyak aplikasi dengan tujuan berbeda. Kadang-kadang, data yang ditransmisikan antara aplikasi- aplikasi merupakan rahasia, dan aplikasi tersebut tentu tidak mau sembarang orang membaca data tersebut.

Sebagai contoh, ketika membeli suatu produk melalui internet, pengguna (user) memasukkan nomor kartu kredit ke dalam jaringan. Hal ini berbahaya karena orang lain dapat dengan mudah menyadap dan membaca data tsb pada jaringan. Oleh karena itu, user biasanya ingin mengenkripsi (encrypt) pesan yang mereka kirim, dengan tujuan mencegah orang-orang yang tidak diizinkan membaca pesan tersebut.

51.7. Kriptografi

Dasar enkripsi cukup sederhana. Pengirim menjalankan fungsi enkripsi pada pesan plaintext, ciphertext yang dihasilkan kemudian dikirimkan lewat jaringan, dan penerima menjalankan fungsi dekripsi (decryption) untuk mendapatkan plaintext semula. Proses enkripsi/dekripsi tergantung pada kunci (key) rahasia yang hanya diketahui oleh pengirim dan penerima. Ketika kunci dan enkripsi ini digunakan, sulit bagi penyadap untuk mematahkan ciphertext, sehingga komunikasi data antara pengirim dan penerima aman.

Kriptografi macam ini dirancang untuk menjamin privacy: mencegah informasi menyebar luas tanpa ijin. Akan tetapi, privacy bukan satu-satunya layanan yang disediakan kriptografi. Kriptografi dapat juga digunakan untuk mendukung authentication (memverifikasi identitas user) dan integritas (memastikan bahwa pesan belum diubah).

Kriptografi digunakan untuk mencegah orang yang tidak berhak untuk memasuki komunikasi, sehingga kerahasiaan data dapat dilindungi. Secara garis besar, kriptografi digunakan untuk mengirim dan menerima pesan. Kriptografi pada dasarnya berpatokan pada key yang secara selektif telah disebar pada komputer-komputer yang berada dalam satu jaringan dan digunakan untuk memproses suatu pesan.

51.8. Operasional

Keamanan operasional (operations security) adalah tindakan apa pun yang menjadikan sistem beroperasi secara aman, terkendali, dan terlindung.

Yang dimaksud dengan sistem adalah jaringan, komputer, lingkungan. Suatu sistem dinyatakan operasional apabila sistem telah dinyatakan berfungsi dan dapat dijalankan dengan durasi yang berkesinambungan, yaitu dari hari ke hari, 24 jam sehari, 7 hari seminggu.

Manajemen Administratif (Administrative Management) adalah penugasan individu untuk mengelola fungsi-fungsi keamanan sistem. Beberapa hal yang terkait:

1. Separation of Duties (pemisahan kewajiban)

   Menugaskan hal-hal yang menyangkut keamanan kepada beberapa orang saja. Misalnya, yang berhak menginstall program ke dalam sistem komputer hanya admin, user tidak diberi hak tersebut.

2. Least Privilege (hak akses minimum)

   Setiap orang hanya diberikan hak akses minimum yang dibutuhkan dalam pelaksanaan tugas mereka.

3. Need to Know (keingintahuan)

   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. Preventative Control (kontrol pencegahan)

   Untuk mencegah error dan intruder memasuki sistem. Misal, kontrol pencegahan untuk mencegah virus memasuki sistem adalah dengan menginstall antivirus.

2. Detective Control (kontrol pendeteksian)

   Untuk mendeteksi error yang memasuki sistem. Misal, mencari virus yang berhasil memasuki sistem.

3. Corrective/Recovery Control (kontrol perbaikan)

   Membantu mengembalikan data yang hilang melalui prosedur recovery data. Misal, memperbaiki data yang terkena virus.

Kategori lainnya mencakup:

1. Deterrent Control

   Untuk menganjurkan pemenuhan (compliance) dengan kontrol eksternal.

2. Application Control (kontrol aplikasi)

   Untuk memperkecil dan mendeteksi operasi-operasi perangkat lunak yang tidak biasa.

3. Transaction Control (kontrol transaksi)

   Untuk menyediakan kontrol di berbagai tahap transaksi (dari inisiasi sampai output, melalui kontrol testing dan kontrol perubahan).

51.9. BCP/DRP

Berdasarkan pengertian, BCP atau Business Continuity Plan adalah rencana bisnis yang berkesinambungan, sedangkan DRP atau Disaster Recovery Plan adalah rencana pemulihan dari kemungkinan kerusakan-kerusakan yang terjadi.

Aspek yang terkandung di dalam suatu rencana bisnis yang berkesinambungan yaitu rencana pemulihan dari kemungkinan kerusakan-kerusakan yang terjadi. Dengan kata lain, DRP terkandung di dalam BCP.

Rencana untuk pemulihan dari kerusakan, baik yang disebabkan oleh alam maupun manusia, tidak hanya berdampak pada kemampuan proses komputer suatu perusahaan, tetapi juga akan berdampak pada operasi bisnis perusahaan tersebut. Kerusakan-kerusakan tersebut dapat mematikan seluruh sistem operasi. Semakin lama operasi sebuah perusahaan mati, maka akan semakin sulit untuk membangun kembali bisnis dari perusahaan tersebut.

Konsep dasar pemulihan dari kemungkinan kerusakan-kerusakan yang terjadi yaitu harus dapat diterapkan pada semua perusahaan, baik perusahaan kecil maupun perusahaan besar. Hal ini tergantung dari ukuran atau jenis prosesnya, baik yang menggunakan proses manual, proses dengan menggunakan komputer, atau kombinasi dari keduanya.

Pada perusahaan kecil, biasanya proses perencanaannya kurang formal dan kurang lengkap. Sedangkan pada perusahaan besar, proses perencanaannya formal dan lengkap. Apabila rencana tersebut diikuti maka akan memberikan petunjuk yang dapat mengurangi kerusakan yang sedang atau yang akan terjadi.

51.10. Proses Audit

Audit dalam konteks teknologi informasi adalah memeriksa apakah sistem komputer berjalan semestinya.

Tujuh langkah proses audit:

1. Implementasikan sebuah strategi audit berbasis risk management 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 risk management 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 top-management, dan metode audit.

Metodologi audit:

1. Audit subject: menentukan apa yang akan diaudit.

2. Audit objective: menentukan tujuan dari audit.

3. Audit scope: menentukan sistem, fungsi, dan bagian dari organisasi yang secara spesifik/khusus akan diaudit.

4. Preaudit planning: mengidentifikasi sumber daya dan SDM yang dibutuhkan, menentukan dokumen-dokumen apa yang diperlukan untuk menunjang audit, menentukan lokasi audit.

5. Audit procedures dan steps for data gathering: menentukan cara melakukan audit untuk memeriksa dan menguji kontrol, 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 mereview hasil audit): evaluasi kesahihan dari dokumen-dokumen, prosedur, dan kebijakan dari organisasi yang diaudit.

Struktur dan isi laporan audit tidak baku, tapi umumnya terdiri atas:

- Pendahuluan: tujuan, ruang lingkup, lamanya audit, prosedur audit.

- Kesimpulan umum dari auditor.

- Hasil audit: apa yang ditemukan dalam audit, apakah prosedur dan kontrol layak atau tidak.

- Rekomendasi.

- Tanggapan dari manajemen (bila perlu).

- Exit interview: interview terakhir antara auditor dengan pihak manajemen untuk membicarakan temuan-temuan dan rekomendasi tindak lanjut. Sekaligus meyakinkan tim manajemen bahwa hasil audit sahih.

51.11. Rangkuman

Data atau informasi penting yang seharusnya tidak dapat diakses oleh orang lain mungkin dapat diakses, baik dibaca ataupun diubah oleh orang lain. Kita harus mempunyai suatu mekanisme yang membuat pelanggaran jarang terjadi.

Ketika merancang sebuah sistem yang aman terhadap intruder, penting untuk mengetahui sistem tersebut akan dilindungi dari intruder macam apa.

Untuk menjaga sistem keamanan sebuah komputer dapat dicapai dengan berbagai cara, antara lain:

- keamanan fisik

hal ini tergantung oleh anggaran dan situasi yang dihadapi.

- keamanan perangkat lunak

contoh dari keamanan perangkat lunak yaitu BIOS.

- keamanan jaringan

yaitu dengan cara kriptografi

DRP (Disaster Recovery Plan) terkandung di dalam BCP (Business Continuity Plan). Konsep dasar DRP harus dapat diterapkan pada semua perusahaan.

Proses audit bertujuan untuk memeriksa apakah sistem komputer berjalan dengan semestinya.

51.12. Latihan

1. Sebutkan dua keuntungan dari enkripsi data pada sistem komputer!
2. Buatlah suatu daftar berisi enam hal yang menjadi perhatian keamanan dari suatu sistem komputer bank. Untuk setiap hal tsb, sebutkan apakah hal tsb berhubungan dengan keamanan fisik, manusia, atau perangkat lunak!
3. Sebutkan 2 macam intruder yang anda ketahui dan sebutkan perbedaannya.
4. Apa yang dimaksud dengan Kriptografi? Jelaskan!
5. Mengapa keamanan perangkat lunak dengan menggunakan BIOS tidak terlalu aman?
6. Apakah hubungan antara BCP dan DRP?
7. Sebutkan langkah-langkah yang diperlukan dalam menjalankan proses audit!

8. Apa saja yang harus direncanakan dalam proses audit?

9. Sebutkan metodologi-metodologi audit!

51.13. Rujukan

Abraham Silberschatz, Peter Baer Galvin dan Greg Gagne: Operating System Concepts with Java -- Sixth Edition, John Wiley & Sons, 2004.

Andrew S. Tanenbaum: Modern Operating Systems -- Second Edition, Prentice Hall, 2001.

Larry L. Peterson, Bruce S. Davie: Computer Networks A Systems Approach -- Second Edition, Morgan Kaufmann, 2000.

Ronald L. Krutz, Russell Dean Vines: The CISSP Prep Guide Mastering the Ten Domains of Computer Security, John Wiley & Sons, 2001.

Bab 52. Perancangan dan Pemeliharaan

52.1. Pendahuluan

Merancang sebuah sistem operasi merupakan hal yang sulit. Merancang sebuah sistem sangat berbeda dengan merancang sebuah algorithma. Hal tersebut disebabkan karena keperluan yang dibutuhkan oleh sebuah sistem sulit untuk didefinisikan secara tepat, lebih kompleks dan sebuah sistem memiliki struktur internal dan antarmuka internal yang lebih banyak serta ukuran dari kesuksesan dari sebuah sistem sangat abstrak.

Masalah pertama dalam mendesain sistem operasi adalah mendefinisikan tujuan dan spesifikasi sistem. pada level tertinggi, desain sistem akan dipengaruhi oleh pemilihan hardware dan tipe sistem seperti batch, time shared, single user, multiuser, distributed, real time atau tujuan umum.

Berdasarkan level desain tertinggi, kebutuhan sistem akan lebih sulit untuk dispesifikasi. Kebutuhan sistem dapat dibagi menjadi dua kelompok utama, yaitu user goal dan sistem goal. User menginginkan properti sistem yang pasti seperti: sistem harus nyaman dan mudah digunakan, mudah dipelajari, reliable, aman dan cepat.

Sekumpulan kebutuhan dapat juga didefinisikan oleh orang-orang yang harus mendesain, membuat, memelihara dan mengoperasikan sistem operasi seperti: sistem operasi harus mudah didesain, diimplementasikan dan dipelihara, sistm harus fleksibel, reliable, bebas eror dan efisien.

Yang harus diperhatikan ketika merancang sebuah sistem yang baik adalah apakah sistem tersebut memenuhi tiga kebutuhan: fungsionalitas: apakah sistem tersebut bekerja dengan baik?, kecepatan: apakah sistem tersebut cukup cepat?, dan fault-tolerance: apakah sistem tersebut dapat terus bekerja?.

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 subsystem pada sistem operasi dapat ditambahkan tanpa mempengaruhi subsystem 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 eror, termasuk kemampuan sistem operasi untuk memproteksi diri sendiri dan pengggunanya dari software yang cacat. Sistem operasi modern menahan diri dari serangan dan cacat dengan menggunakan proteksi perangkat keras untuk memori virtual dan mekanisme proteksi perangkat lunak untuk sumber daya sistem operasi.

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 file.

52.2. Perancangan Antarmuka

Merancang antarmuka merupakan bagian yang paling penting dari merancang sistem. Biasanya hal tersebut juga merupakan bagian yang paling sulit, karena dalam merancang antarmuka harus memenuhi tiga persyartan: sebuah antarmuka harus sederhana, sebuah antarmuka harus lengkap, dan sebuah antarmuka harus memilki kinerja yang cepat.

Alasan utama mengapa antarmuka sulit untuk dirancang adalah karena setiap antarmuka adalah sebuah bahasa pemrograman yang kecil: antarmuka menjelaskan sekumpulan obyek-obyek dan operasi-operasi yang bisa digunakan untuk memanipulasi obyek.

52.3. Implementasi

Rancangan Sistem

Desain sistem memiliki masalah dalam menentukan tujuan dan spesifikasi sistem. Pada level paling tinggi, desain sistem akan dipengaruhi oleh pilihan perangkat keras dan jenis sistem. Kebutuhannya akan lebih sulit untuk dispesifikasikan. Kebutuhan terdiri dari target user dan target sistem. User menginginkan sistem yang nyaman digunakan, mudah dipelajari, dapat dipercaya, aman, dan cepat. Namun itu semua tidaklah signifikan untuk desain sistem. Orang yang mendesain ingin sistem yang mudah didesain, diimplementasikan, fleksibel, dapat dipercaya, bebas eror, efisien. Sampai saat ini belum ada solusi yang pas untuk menentukan kebutuhan dari sistem operasi. Lain lingkungan, lain pula kebutuhannya.

Mekanisme dan Kebijakan

Mekanisme menentukan bagaimana melakukan sesuatu. Kebijakan menentukan apa yang akan dilakukan.Pemisahan antara mekanisme dan kebijakan sangatlah penting untuk fleksibilitas. Perubahan kebijakan akan membutuhkan definisi ulang pada beberapa parameter sistem, bahkan bisa mengubah mekanisme yang telah ada. Sistem operasi Microkernel-based menggunakan pemisahan mekanisme dan kebijakan secara ekstrim dengan mengimplementasikan perangkat dari primitive building blocks. Semua aplikasi mempunyai antarmuka yang sama karena antarmuka dibangun dalam kernel. Kebijakan penting untuk semua alokasi sumber daya dan penjadualan problem. Perlu atau tidaknya sistem mengalokasikan sumber daya, kebijakan yang menentukan. Tapi bagaimana dan apa, mekanismelah yang menentukan.

ZCZCOLD

Rancangan Sistem

Desain sistem memiliki masalah dalam menentukan tujuan dan spesifikasi sistem. Pada level paling tinggi, desain sistem akan dipengaruhi oleh pilihan perangkat keras dan jenis sistem. Kebutuhannya akan lebih sulit untuk dispesifikasikan. Kebutuhan terdiri dari target user dan target sistem. User menginginkan sistem yang nyaman digunakan, mudah dipelajari, dapat dipercaya, aman, dan cepat. Namun itu semua tidaklah signifikan untuk desain sistem. Orang yang mendesain ingin sistem yang mudah didesain, diimplementasikan, fleksibel, dapat dipercaya, bebas eror, efisien. Sampai saat ini belum ada solusi yang pas untuk menentukan kebutuhan dari sistem operasi. Lain lingkungan, lain pula kebutuhannya.

Mekanisme dan Kebijakan

Mekanisme menentukan bagaimana melakukan sesuatu. Kebijakan menentukan apa yang akan dilakukan.Pemisahan antara mekanisme dan kebijakan sangatlah penting untuk fleksibilitas. Perubahan kebijakan akan membutuhkan definisi ulang pada beberapa parameter sistem, bahkan

bisa mengubah mekanisme yang telah ada. Sistem operasi *Microkernel-based* menggunakan pemisahan mekanisme dan kebijakan secara ekstrim dengan mengimplementasikan perangkat dari *primitive building blocks*. Semua aplikasi mempunyai antarmuka yang sama karena antarmuka dibangun dalam kernel.

Kebijakan penting untuk semua alokasi sumber daya dan penjadualan problem. Perlu atau tidaknya sistem mengalokasikan sumber daya, kebijakan yang menentukan. Tapi bagaimana dan apa, mekanismelah yang menentukan.

52.4. Implementasi Sistem

Umumnya sistem operasi ditulis dalam bahasa rakitan, tapi sekarang ini sering ditulis dalam bahasa tingkat tinggi. Keuntungannya adalah kodenya bisa ditulis lebih cepat, lebih padat, mudah dimengerti dan di-debug. Sistem operasi mudah diport (dipindahkan ke perangkat keras lain). Kerugiannya adalah mengurangi kecepatan dan membutuhkan tempat penyimpanan yang lebih banyak.

Pemberian Alamat

Sebelum masuk ke memori, suatu proses harus menunggu. Hal ini disebut antrian masukan. Proses-proses ini akan berada dalam beberapa tahapan sebelum dieksekusi. Alamat-alamat yang dibutuhkan mungkin saja direpresentasikan dalam cara yang berbeda dalam tahapan-tahapan ini. Alamat dalam kode program masih berupa simbolik. Alamat ini akan diikat oleh kompilator ke alamat memori yang dapat diakses. Kemudian linkage editor dan loader, akan mengikat alamat fisiknya. Setiap pengikatan akan memetakan suatu ruang alamat ke lainnya. Penjilidan alamat dapat terjadi pada 3 saat, yaitu:

Waktu Kompilasi: Jika diketahui pada waktu kompilasi, dimana proses ditempatkan di memori. Untuk kemudian kode absolutnya dapat dibuat. Jika kemudian alamat awalnya berubah, maka harus dikompilasi ulang.

Waktu pemanggilan: Jika tidak diketahui dimana poses ditempatkan di memori, maka kompilator harus membuat kode yang dapat dialokasikan. Dalam kasus pengikatan akan ditunda sampai waktu pemanggilan. Jika alamat awalnya berubah, kita hanya perlu menempatkan ulang kode, untuk menyesuaikan dengan perubahan.

Waktu eksekusi: Jika proses dapat dipindahkan dari suatu segmen memori ke lainnya selama dieksekusi. Pengikatan akan ditunda sampai run-time.

ZCZCOLD

Umumnya sistem operasi ditulis dalam bahasa rakitan, tapi sekarang ini sering ditulis dalam bahasa tingkat tinggi. Keuntungannya adalah kodenya bisa ditulis lebih cepat, lebih padat, mudah dimengerti dan di-*debug*. Sistem operasi mudah diport (dipindahkan ke perangkat keras lain). Kerugiannya adalah mengurangi kecepatan dan membutuhkan tempat penyimpanan yang lebih banyak.

Pemberian Alamat

Sebelum masuk ke memori, suatu proses harus menunggu. Hal ini disebut antrian masukan. Proses-proses ini akan berada dalam beberapa tahapan sebelum dieksekusi. Alamat-alamat yang dibutuhkan mungkin saja direpresentasikan dalam cara yang berbeda dalam tahapan-tahapan ini. Alamat dalam kode program masih berupa simbolik. Alamat ini akan diikat oleh kompilator ke alamat memori yang dapat diakses. Kemudian *linkage editor* dan *loader*, akan mengikat alamat fisiknya. Setiap pengikatan akan memetakan suatu ruang alamat ke lainnya.

Penjilidan alamat dapat terjadi pada 3 saat, yaitu:

- **Waktu Kompilasi**: Jika diketahui pada waktu kompilasi, dimana proses ditempatkan di memori. Untuk kemudian kode absolutnya dapat dibuat. Jika kemudian alamat awalnya berubah, maka

harus dikompilasi ulang.

- **Waktu pemanggilan**: Jika tidak diketahui dimana poses ditempatkan di memori, maka kompilator harus membuat kode yang dapat dialokasikan. Dalam kasus pengikatan akan ditunda sampai waktu pemanggilan. Jika alamat awalnya berubah, kita hanya perlu menempatkan ulang kode, untuk menyesuaikan dengan perubahan.

- **Waktu eksekusi**: Jika proses dapat dipindahkan dari suatu segmen memori ke lainnya selama dieksekusi. Pengikatan akan ditunda sampai *run-time*.

52.5. Kinerja (FM)

Kinerja sebuah sistem ditentukan oleh komponen-komponen yang membangun sistem tersebut. Kinerja yang paling diinginkan ada pada sebuah sistem adalah bebas eror, cepat dan fault-tolerance.

52.6. Pemeliharaan Sistem

Pemeliharaan sistem operasi berkaitan erat dengan pemeliharaan komputer, karena bagaimanapun sebuah komputer tidak akan berguna tanpa adanya sebuah sistem operasi di dalamnya. Sistem operasi juga bertindak sebagai manajer bagi semua komponen pada arsitektur komputer. Oleh karena itu, pemeliharaan sistem operasi juga dikaitkan dengan pemeliharaan komponen komputer(hardware) dan software lainnya.

Adapun beberapa hal yang dapat dilakukan daalm rangka memelihara sistem operasi antara lain:

1. Pastikan untuk selalu melakukan shutdown pada sistem operasi sebelum power switch dimatikan. Hal ini penting untuk melindungi cacat pada hard drive yang disebabkan oleh kontak antara head pada hard drive dengan permukaan drive disc, dan juga menghindari kehilangan data-data penting. Pengecualian adalah ketika komputer di-lock dan hard-drive tidak berjalan. Pada situasi ini komputer dapat dimatikan tanpa ada efek membahayakan pada hard drive.

2. Usahakan untuk selalu memback-up data yang sangat penting ke dalam sedikitnya dua physical drive yang terpisah. Jadi backup data bisa dilakukan ke floppy, zip disks, Cd-Rw, dll.

3. Jalankan Scandisk dan defragment minimal sekali dalam satu bulan. Hal ini berguna agar hard drive tetap dalam kondisi baik dan menghindari terjadinya crash.

4. Sisakan minimal 100 MB pada drive C: untuk digunakan oleh sistem operasi. Jika tidak, sistem operasi akan men-delete data-data pada hard-drive, atau sistem operasi menjadi sangat lambat. gunakan Add/delete untuk mendelete program yang tidak lagi digunakan.

5. Jangan membiarkan banyak program di-load saat men-start komputer. program-program tersebut menggunakan memori yang banyak sehingga membuat komputer menjadi lambat.

6. Lakukan pengecekan virus secara rutin.

52.7. Trend

Trend sistem operasi pada tahun-tahun mendatang akan berkaitan dengan sistem operasi yang bersifat Open Source yang lebih dikenal sebagai Linux. Hal ini sesuai dengan kemampuan yang ada pada Linux, yang sangat diharapkan oleh para pengguna di kalangan bisnis, yaitu unjuk kerja yang tinggi, sekuriti, kestabilan yang tinggi, handal dan murah. Ditambah dengan dukungan aplikasi dan vendor kelas enterprise yang kini telah tersedia. Sehingga dapat dikatakan bahwa Linux telah siap untuk digunakan sebagai solusi bisnis yang bisa diandalkan. Para vendor inipun telah siap menyediakan dukungan teknis untuk Linux sehingga hal ini diharapkan dapat menghapus keragu-raguan dunia bisnis untuk memanfaatkan Linux sebagai platform operasi mereka.

Baru-baru ini(2005) Microsoft telah mengeluarkan versi terbaru dari sistem operasi windows, yaitu Longhorn. Dibandingkan dengan versi windows sebelumnya, Longhorn lebih tahan lama, lebih kuat

dan lebih memberikan keamanan dalam hal penyimpanan data. Longhorn mengorganisasikan secara otomatis, langsung, dan berkesinambungan dengan menggunakan Virtual Folders. Virtual Folders selalu dinamis dan uptodate. Ketika user menyalin berkas baru ke komputer, Longhorn membuat semua properti dari berkas baru. Kita tidak perlu lagi untuk menyimpan berkas dalam sebuah lokasi spesifik.

52.8. Rangkuman

Merancang sebuah sistem sangat berbeda dengan merancang sebuah algorithma. Yang harus diperhatikan ketika merancang sebuah sistem yang baik adalah apakah sistem tersebut memenuhi tiga kebutuhan, yaitu fungsionalitas, kecepatan dan fault-tolerance. Orang yang mendesain ingin sistem yang mudah didesain, diimplementasikan, fleksibel, dapat dipercaya, bebas eror, efisien.

Trend perancangan sistem operasi di masa depan lebih kepada penggunaan sistem operasi yang berbasis open source, dalam hal ini adalah Linux. Pada umumnya trend demikian berkembang karena Linux menawarkan kemampuan-kemampuan dan performance yang lebih baik dari sistem operasi lainnya. Ditambah dengan stausnya yang Open Source.

52.9. Latihan

1. Kebutuhan apa saja yang harus diperhatikan ketika merancang sebuah sistem?
2. Sebutkan kelebihan sistem operasi Linux dibandingkan sistem operasi lainnya dalam bidang keamanan/sekuriti sistem, mengingat Linux berbasis Open Source.

52.10. Rujukan

Silberschatz, Galvin, Gagne. 2002. Operating System Concepts: 6th ed. John Wiley & Sons

www.stanford.edu/class/cs240/readings/lampson-hints.pdf

http://www.microsoft.com/windows/longhorn/

http://wiryana.pandu.org/artikel/CEBIT-1999/SAP.html Prof. Peter Ladkin, Ph.D

http://cquirke.mvps.org/whatmos.htm

http://www.infohq.com/Computer/computer_maintenance_tip.htm

Rujukan

[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.

[KennethRosen1999] Kenneth H. Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[Kusuma2000] Sri Kusumadewi . 2000. *Sistem Operasi* . Edisi Dua. Graha Ilmu. Yogyakarta.

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

[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.

[WEBAmirSch2000] Yair Theo Amir Schlossnagle . 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/ [http://www.cs.jhu.edu/~yairamir/cs418/os5/]* . Diakses 10 Agustus 2005.

[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 [http://www.javaworld.com/javaworld/jw-04-1996/jw-04-threads.html]* . Diakses 10 Agustus 2005.

[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 [http://www.cs.utah.edu/classes/cs5460/lectures/lecture19.pdf]* . Diakses 10 Agustus 2005.

[WEBFasilkom2003] Fakultas Ilmu Komputer Universitas Indonesia . 2003. *Sistem Terdistribusi – http://telaga.cs.ui.ac.id/ WebKuliah/ sisdis2003/ [http://telaga.cs.ui.ac.id/WebKuliah/sisdis2003/]* . Diakses 10 Agustus 2005.

[WEBFSF1991a] Free Software Foundation . 1991. *GNU General Public License – http://gnui.vLSM.org/licenses/gpl.txt [http://gnui.vLSM.org/licenses/gpl.txt]* . Diakses 16 Agustus 2005.

[WEBFSF2001a] Free Software Foundation . 2001. *Definisi Perangkat Lunak Bebas – http://gnui.vlsm.org/philosophy/free-sw.id.html [http://gnui.vlsm.org/philosophy/free-sw.id.html]* . Diakses 16 Agustus 2005.

[WEBFSF2001b] Free Software Foundation . 2001. *Frequently Asked Questions about the GNU GPL – http://gnui.vlsm.org/licenses/gpl-faq.html [http://gnui.vlsm.org/licenses/gpl-faq.html]* . Diakses 16 Agustus 2005.

[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 [http://www.cs.princeton.edu/courses/archive/spring02/cs217/lectures/paging.pdf]* . Diakses 10 Agustus 2005.

[WEBGooch1999] Richard Gooch . 1999. *Overview of the Virtual File System – http://www.atnf.csiro.au/ people/ rgooch/ linux/ docs/ vfs.txt [http://www.atnf.csiro.au/people/rgooch/linux/docs/vfs.txt]* . Diakses 10 Agustus 2005.

[WEBGottlieb2000] Allan Gottlieb . 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html [http://allan.ultra.nyu.edu/~gottlieb/courses/1999-00-spring/os/lecture-11.html]* . Diakses 10 Agustus 2005.

[WEBHuham2005] Departemen Hukum dan Hak Asasi Manusia Republik Indonesia . 2005. *Kekayaan Intelektual – http://www.dgip.go.id/article/archive/2 [http://www.dgip.go.id/article/archive/2]* . Diakses 17 Agustus 2005.

[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 [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 10 Agustus 2005.

[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 [http://zenii.linux.org.uk/~davej/docs/post-halloween-2.6.txt]* . Diakses 10 Agustus 2005.

[WEBMassey2000] Massey University . May 2000. *Monitors & Critical Regions – http://www-ist.massey.ac.nz/ csnotes/ 355/ lectures/ monitors.pdf [http://www-ist.massey.ac.nz/csnotes/355/lectures/monitors.pdf]* . Diakses 10 Agustus 2005.

[WEBRamelan1996] Rahardi Ramelan . 1996. *Hak Atas Kekayaan Intelektual Dalam Era Globalisasi http://leapidea.com/presentation?id=6 [http://leapidea.com/presentation?id=6]* . Diakses 16 Agustus 2005.

[WEBRobbins2003] Steven Robbins . 2003. *Starving Philosophers: Experimentation with Monitor Synchronization – http://vip.cs.utsa.edu/nsf/pubs/starving/starving.pdf* . Diakses 10 Agustus 2005.

[WEBSamik2003a] Rahmat M Samik-Ibrahim . 2003. *Pengenalan Lisensi Perangkat Lunak Bebas – http://rms46.vlsm.org/ 1/ 70.pdf [http://rms46.vlsm.org/1/70.pdf]* . vLSM.org. Pamulang. Diakses 17 Agustus 2005.

[WEBSamik2005a] Rahmat M Samik-Ibrahim . 2005. *IKI-20230 Sistem Operasi - Kumpulan Soal Ujian 2002-2005 – http://rms46.vlsm.org/ 1/ 94.pdf [http://rms46.vlsm.org/1/94.pdf]* . vLSM.org. Pamulang. Diakses 10 Agustus 2005.

[WEBSchaklette2004] Mark Shacklette . 2004. *CSPP 51081 Unix Systems Programming: IPC – http://people.cs.uchicago.edu/ ~mark/ 51081/ LabFAQ/ lab5/ IPC.html [http://people.cs.uchicago.edu/~mark/51081/LabFAQ/lab5/IPC.html]* . Diakses 10 Agustus 2005.

[WEBSolomon2004] Marvin Solomon . 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/~solomon/cs537/paging.html [http://www.cs.wisc.edu/~solomon/cs537/paging.html]* . Diakses 10 Agustus 2005.

[WEBStallman1994a] Richard M Stallman . 1994. *Mengapa Perangkat Lunak Seharusnya Tanpa Pemilik – http://gnui.vlsm.org/philosophy/why-free.id.html [http://gnui.vlsm.org/philosophy/why-free.id.html]* . Diakses 16 Agustus 2005.

[WEBWalton1996] Sean Walton . 1996. *Linux Threads Frequently Asked Questions (FAQ) – http://linas.org/ linux/ threads-faq.html [http://linas.org/linux/threads-faq.html]* . Diakses 10 Agustus 2005.

[WEBWiki2005a] From Wikipedia, the free encyclopedia . 2005. *Intellectual property – http://en.wikipedia.org/wiki/Intellectual_property [http://en.wikipedia.org/wiki/Intellectual_property]* . Diakses 16 Agustus 2005.

[WEBWIPO2005] World Intellectual Property Organization . 2005. *About Intellectual Property – http://www.wipo.int/about-ip/en/ [http://www.wipo.int/about-ip/en/]* . Diakses 17 Agustus 2005.

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.

Indeks

L

M

N

O

P

T

V

W