Pengantar Sistem Operasi Komputer

Jilid Pertama

Masyarakat Digital Gotong Royong (MDGR)

Pengantar Sistem Operasi Komputer: Jilid Pertama

oleh Masyarakat Digital Gotong Royong (MDGR)

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

Catatan Revisi

Revisi 4.59	29-Agustus-2008	RMS46
Fop 0.9X, perbaiki ukuran gambar.		
Revisi 4.57	04-Agustus-2008	RMS46
Merapihkan ulang; memperbaiki sana-sini; gabung kembali.		
Revisi 4.52	04-Februari-2008	RMS46
Mengedit ulang+membenah jilid 1 dan 2. Menambah Soal Ujian.		
Revisi 4.27	31-Agustus-2007	RMS46
Merapihkan dan memecah menjadi dua jilid.		
Revisi 4.22	07-Agustus-2007	RMS46
Daftar dalam Pengantar, Mengisi Kerangka, Soal Ap. B., Urut ulang.		
Revisi 4.16	03-Februari-2007	RMS46
Kerangka Baru.		
Revisi 4.8	28-Desember-2006	RMS46
Reset, start mengerjakan kerangka bab.		
Revisi 4.7	18-November-2006	RMS46
Pemulaian persiapan revisi 5.0 (kapan?).		
Revisi 4.00	28-Agustus-2006	RMS46
Menganggap selesai revisi 4.0.		
Revisi 3.64	14-Agustus-2006	RMS46
Mei-Agustus 2006: Pemolesan		
Revisi 3.42	04-Mei-2006	RMS46
April-Mei 2006: Mengosongkan Appendix C: (UPDATE).		
Revisi 3.37	06-April-2006	RMS46
Start Feb2006: Gusur Appendix B: Soal Latihan.		
Revisi 3.27	22-Februari-2006	RMS46
Full XML (was SGML), start update kelompok hingga bab 47.		
Revisi 3.00	26-Agustus-2005	RMS46
Selesai tidak selesai, ini revisi 3.00!		
Revisi 2.34	26-Agustus-2005	RMS46
Memperbaiki sana-sini.		
Revisi 2.24	5-Agustus-2005	RMS46
Mempersiapkan seadanya versi 3.0		
Revisi 2.17	27-Juli-2005	RMS46
Mengubah dari SGML DocBook ke XML DocBook.		
Revisi 2.10	03-Mar-2005	RMS46
Membereskan dan memilah 52 bab.		
Revisi 2.4	02-Dec-2004	RMS46
Update 2.0+. Ubah sub-bab menjadi bab.		
Revisi 2.0	09-09-2004	RMS46
Menganggap selesai revisi 2.0.		

Revisi 1.10	09-09-2004	RMS46
Pesiapan ke revisi 2.0		
Revisi 1.9.2.10	24-08-2004	RMS46
Ambil alih kelompok 51, perbaikan isi buku.		
Revisi 1.9.1.2	15-03-2004	RMS46
Revisi lanjutan: perbaikan sana-sini, ejaan, indeks, dst.		
Revisi 1.9.1.0	11-03-2004	RMS46
Revisi ini diedit ulang serta perbaikan sana-sini.		
Revisi 1.9	24-12-2003	Kelompok 49
Versi rilis final buku OS.		
Revisi 1.8	08-12-2003	Kelompok 49
Versi rilis beta buku OS.		
Revisi 1.7	17-11-2003	Kelompok 49
Versi rilis alfa buku OS.		
Revisi 1.5	17-11-2003	Kelompok 49
Penggabungan pertama (kel 41-49), tanpa indeks dan rujukan utama. ada.		
Revisi 1.4	08-11-2003	Kelompok 49
Pengubahan template versi 1.3 dengan template yang baru yang akan digunakan dalam versi 1.4-2.0		
Revisi 1.3.0.5	12-11-2003	RMS46
Dipilah sesuai dengan sub-pokok bahasan yang ada.		
Revisi 1.3	30-09-2003	RMS46
Melanjutkan perbaikan tata letak dan pengindeksan.		
Revisi 1.2	17-09-2003	RMS46
Melakukan perbaikan struktur SGML, tanpa banyak mengubah isi buku.		
Revisi 1.1	01-09-2003	RMS46
Kompilasi ulang, serta melakukan sedikit perapihan.		
Revisi 1.0	27-05-2003	RMS46
Revisi ini diedit oleh Rahmat M. Samik-Ibrahim (RMS46).		
Revisi 0.21.4	05-05-2003	Kelompok 21
Perapihan berkas dan penambahan entity.		
Revisi 0.21.3	29-04-2003	Kelompok 21
Perubahan dengan menyempurnakan nama file.		
Revisi 0.21.2	24-04-2003	Kelompok 21
Merubah Kata Pengantar.		
Revisi 0.21.1	21-04-2003	Kelompok 21
Menambahkan Daftar Pustaka dan Index.		
Revisi 0.21.0	26-03-2003	Kelompok 21
Memulai membuat tugas kelompok kuliah Sistem Operasi.		

Persembahan

Buku "Kunyuk" ini dipersembahkan *dari* Masyarakat Digital Gotong Royong (MDGR), *oleh* MDGR, *untuk* siapa saja yang ingin mempelajari Sistem Operasi dari sebuah komputer. Buku ini **bukan** merupakan karya individual, melainkan merupakan hasil keringat dari **ratusan** jemaah MDGR! MDGR ini merupakan Gabungan Kelompok Kerja 21–28 Semester Genap 2002/2003, 41–49 Semester Ganjil 2003/2004, 51 Semester Genap 2003/2004, 53–58 Semester Ganjil 2004/2005, 81–89 Semester Genap 2004/2005, 111–120 Semester Ganjil 2005/2006, 150 Semester Genap 2005/2006, 152–157 dan 181–186 Semester Ganjil 2006/2007, 192–198 Semester Genap 2006/2007, 217 Semester Ganjil 2007/2008, Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia (http://rms46.vlsm.org/2/150.html -- http://www.cs.ui.ac.id/) yang namanya tercantum berikut ini:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Kelompok 49 (2003). Kelompok ini merupakan koordinator kelompok 41-48 pada tahap kedua pengembangan buku ini. Kelompok ini selain kompak, juga sangat kreatif dan inovatif. Anggota dari kelompok ini ialah: Fajran Iman Rusadi (Ketua), Carroline D Puspa.

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

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

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

Kelompok 55 (2004). Kelompok ini bertugas untuk me-*review* bagian 5 versi 3.0 yang berupakan gabungan bab 5 dan bab 8 versi 2.0, yang dipecah ke beberapa bab baru. Bagian 5 ini berisi pokok bahasan Managemen Memori. Anggota dari kelompok ini ialah: Nilam Fitriah, Nurmaya, Nova Eka Diana, Okky HTF, Tirza Varananda, Yoanna W, Aria WN, Yudi Ariawan, Hendrik Gandawijaya,

Johanes, Dania Tigarani S, Desiana NM, Annas Firdausi, Hario Adit W, Kartika Anindya P. Fajar Muharandy, Yudhi M Hamzah K, Binsar Tampahan HS, Risvan Ardiansyah, Budi Irawan, Deny Martan, Prastudy Mungkas F, Abdurrasyid Mujahid, Adri Octavianus, Rahmatri Mardiko.

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

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

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

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

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

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

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

Kelompok 85 (2005). Kelompok ini bertugas untuk menulis Bab 25 (Masalah Bounded Buffer) serta Bab 10.2, 16.7, 22.2, 47.5 versi 3.0. Kelompok ini hanya beranggotakan: Fahrurrozi Rahman dan Randy S.P.

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

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

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

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

Kelompok 111 (2005). Sub-kelompok 111-10 bertugas menulis ulang Bab 10 (Konsep Proses) versi 4.0. Sub-kelompok ini beranggotakan: Richard Lokasamita, Rado Yanu, Phyllisia Angelia. Sub-kelompok 111-11 bertugas menulis ulang Bab 11 (Konsep Thread) versi 4.0. Sub-kelompok ini beranggotakan: Ario Santoso, Wahyu Mirza, Daniel Cahyadi. Sub-kelompok 111-12 bertugas menulis ulang Bab 12 (Thread Java) versi 4.0. Sub-kelompok ini beranggotakan: Moh. Ibrahim, Hafiz Arraja,

Sutanto Sugii Joji. Sub-kelompok 111-13 bertugas menulis ulang Bab 13 (Konsep Penjadwalan) versi 4.0. Sub-kelompok ini beranggotakan: Kresna D.S., Rama Rizki, Wisnu LW.

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

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

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

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

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

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

Kelompok 118 (2005). Sub-kelompok 118-40 bertugas menulis ulang Bab 40 (Implementasi Sistem Berkas) versi 4.0. Sub-kelompok ini beranggotakan: Gita Lystia, Rahmawati. Sub-kelompok

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Daftar Isi

Daftar Gambar

Daftar Tabel

Daftar Contoh

Kata Pengantar

1. Calon Revisi 5.0 (Kapan?)

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

– DRAFT – BELUM TERBIT – DRAFT –

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

Bagian I. Konsep Dasar Perangkat Komputer

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

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

Bab 1. Hari Gini Belajar SO?

1.1. Pendahuluan

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

1.2. Mengapa Mempelajari Sistem Operasi?

Setelah lebih dari 60 tahun sejarah perkomputeran, telah terjadi pergeseran yang signifikan dari peranan sebuah Sistem Operasi. Perhatikan tabel 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*	Kumpulan Komputer dalam Jaringan
Memori	Beberapa Kbytes	Beberapa Gbytes
Disk	Beberapa Mbytes	Beberapa ratus Gbytes
Peraga	Terminal Teks	Grafik beresolusi Tinggi
Arsitektur	Aneka ragam arsitektur	Beberapa arsitektur dominan
Sistem Operasi	Setiap arsitekur komputer menggunakan Sistem Operasi yang berbeda	Dominasi *Microsoft* dengan beberapa pengecualian

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

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

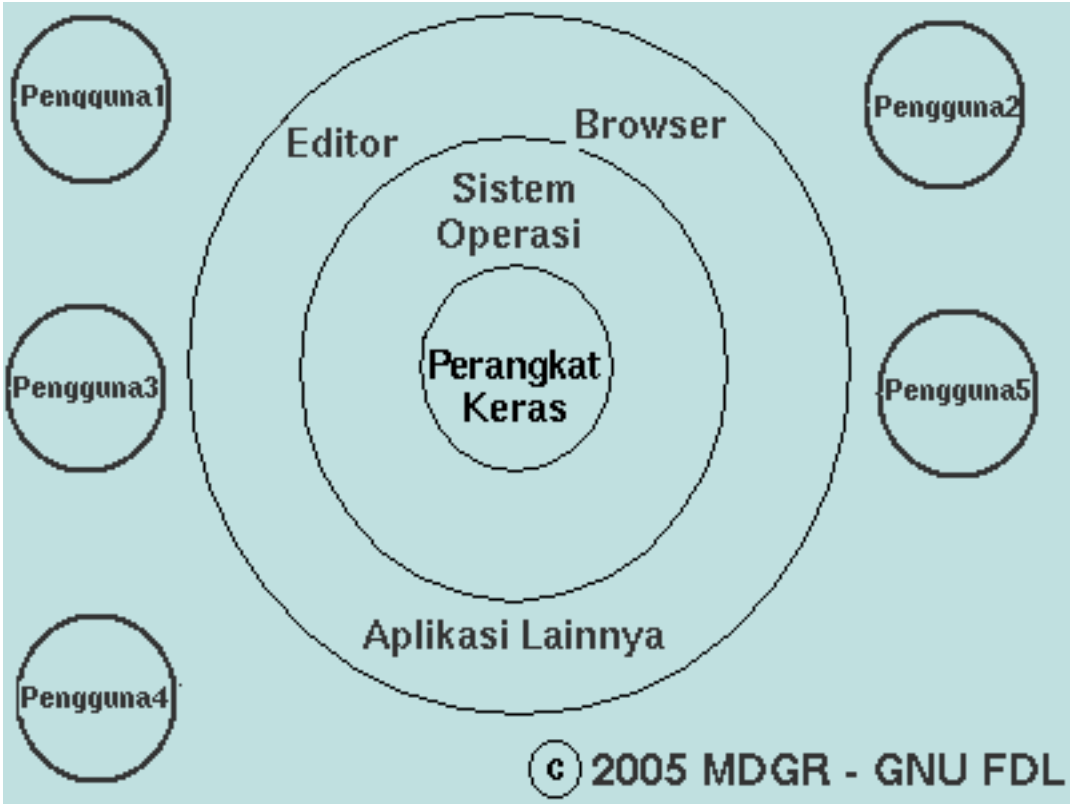
1.3. Definisi Sementara

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

Mendefinisikan istilah "Sistem Operasi" mungkin merupakan hal yang mudah, namun mungkin juga merupakan hal yang sangat ribet! Para pembaca sepertinya pernah mendengar istilah "Sistem Operasi".

Mungkin pula pernah berhubungan secara langsung ataupun tidak langsung dengan istilah tersebut. Namun, belum tentu dapat menjabarkan perihal apa yang sebetulnya dimaksud dengan kata "Sistem Operasi". Sebaliknya, banyak pula yang pernah mendengar merek dagang "*Windows* [TM] [1])" ataupun istilah "GNU/Linux [2])", lalu mengidentikkan nama *Windows* [TM]atau GNU/Linux dengan istilah "Sistem Operasi" tersebut.

Gambar 1.1. Abstraksi Komponen Sistem Komputer



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

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

Sistem Operasi berada di antara perangkat keras komputer dan perangkat aplikasinya. Namun, bagaimana caranya menentukan secara pasti, letak perbatasan antara "perangkat keras komputer" dan "Sistem Operasi", dan terutama antara "perangkat lunak aplikasi" dan "Sistem Operasi"? Umpamanya,

[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.

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

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

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

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

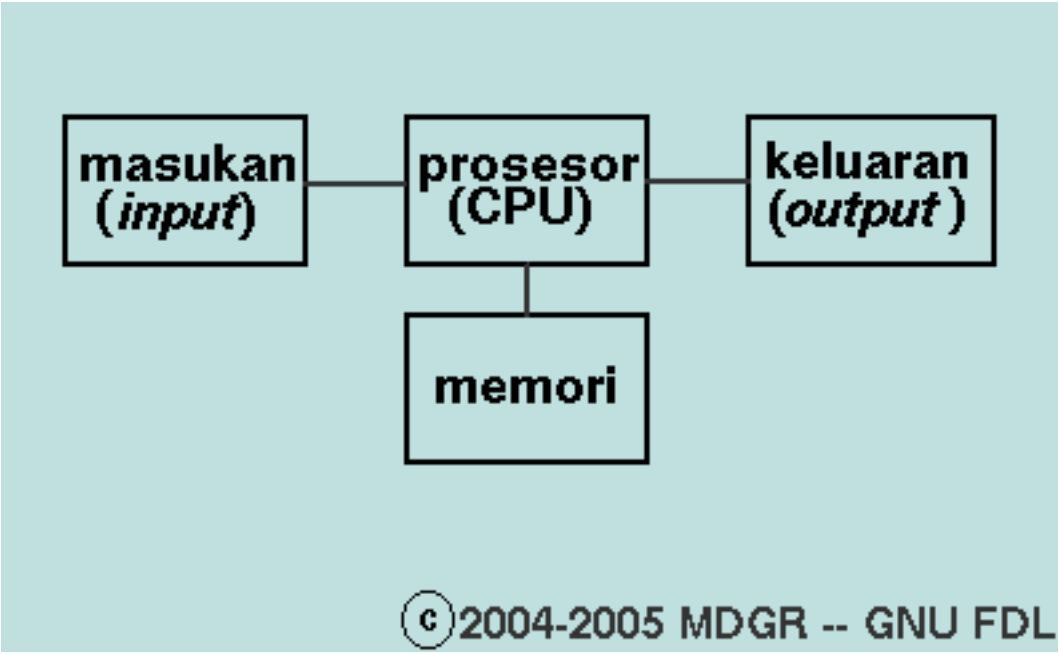
1.4. Sejarah Perkembangan

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

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

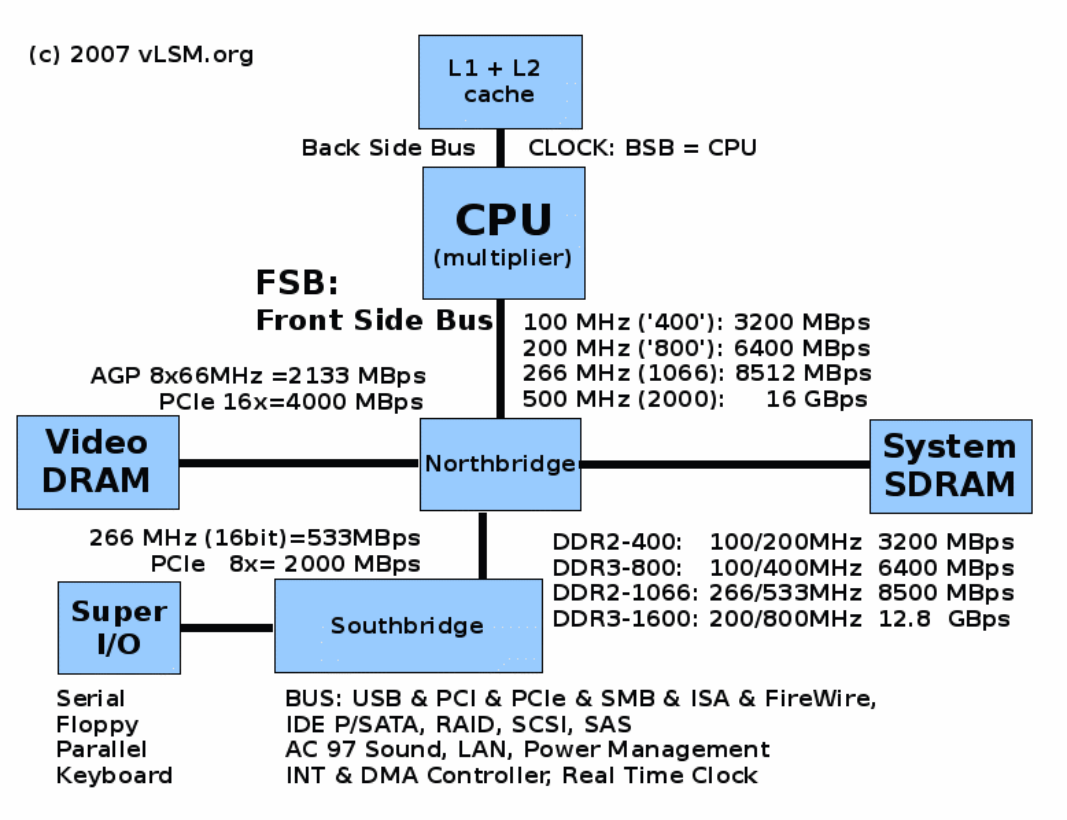
Walaupun berukuran besar, sistem tersebut dikategorikan sebagai "komputer pribadi" (PC). Siapa saja yang ingin melakukan komputasi; harus memesan/antri untuk mendapatkan alokasi waktu (rata-rata 30-120 menit). Jika ingin melakukan kompilasi Fortran, maka pengguna pertama kali akan me-*load*kompilator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

Gambar 1.2. Arsitektur Komputer von-Neumann



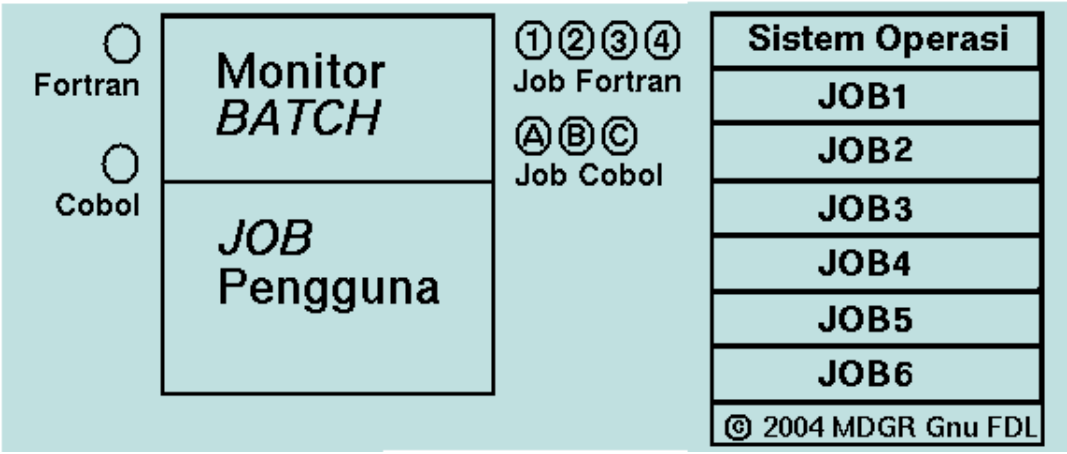
Selanjutnya, terjadi pemisahan tugas antara programer dan operator. Para operator biasanya secara eksklusif menjadi penghuni "ruang kaca" seberang ruang komputer. Para programer yang merupakan pengguna (*users*), mengakses komputer secara tidak langsung melalui bantuan para operator. Para pengguna mempersiapkan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

Gambar 1.3. Bagan Sebuah Komputer Personal



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

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



Pada perkembangan berikutnya, diperkenalkan konsep *Multiprogrammed System*. Dengan sistem ini *job-job* disimpan di memori utama di waktu yang sama dan *CPU* dipergunakan bergantian. Hal ini membutuhkan beberapa kemampuan tambahan yaitu: penyediaan *I/O routine* oleh sistem, pengaturan memori untuk mengalokasikan memori pada beberapa *Job*, penjadwalan *CPU* untuk memilih *job* mana yang akan dijalankan, serta pengalokasian perangkat keras lain (Gambar 1.4, “Bagan Memori Untuk Sistem *Monitor Batch* Sederhana”).

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

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

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

1.5. Bahan Pembahasan

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

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

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

1.6. Tantangan

Lazimnya, Sistem Operasi bukan merupakan mata ajar favorit. Merupakan sebuah tantangan tersendiri untuk membuat mata ajar ini menjadi menarik.

1.7. 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.

1.8. 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.

1.9. Rangkuman

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

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

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

Rujukan

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

Bab 2. HaKI Perangkat Lunak

2.1. Pendahuluan

Sebelum membahas aspek teknis secara mendalam, sebaiknya kita memantapkan terlebih dahulu sebuah pengertian aspek non teknis dari sebuah sistem operasi yaitu Hak atas Kekayaan Intelektual Perangkat Lunak (HaKI PL) Pembahasan dimulai dengan menerangkan konsep HaKI secara umum, serta HaKI PL secara lebih dalam. Secara khusus akan dibahas konsep Perangkat Lunak Bebas/Sumber Terbuka – **PLB/ST** (*Free/Open Source Software – F/OSS*). Pembahasan ini bukan bertujuan sebagai indoktrinasi faham tersebut! Justru yang diharapkan:

- Pelurusan atas persepsi keliru PLB dan ST, serta penjelasan perbedaan dan persamaan dari kedua konsep tersebut.
- Apa yang boleh dan apa yang tidak boleh dilakukan dengan PLB/ST.
- Pelurusan atas persepsi bahwa para penulis program komputer tidak berhak digaji layak.
- Pelurusan atas persepsi bahwa PLB tidak boleh dijual/dikomersialkan.
- Pelurusan atas persepsi bahwa PLB wajib disebarluaskan.
- Pelurusan atas persepsi bahwa saat distribusi tidak wajib menyertakan kode sumber.

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

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

``Hak" itu sendiri dapat dibagi menjadi dua. Pertama, ``Hak Dasar (Azasi)", yang merupakan hak mutlak yang tidak dapat diganggu-gugat. Umpama: hak untuk hidup, hak untuk mendapatkan keadilan, dan sebagainya. Kedua, ``Hak Amanat/ Peraturan" yaitu hak karena diberikan oleh masyarakat melalui peraturan/perundangan. Di berbagai negara, termasuk Amrik dan Indonesia, HaKI merupakan "Hak Amanat/Pengaturan", sehingga masyarakatlah yang menentukan, seberapa besar HaKI yang diberikan kepada individu dan kelompok. Sesuai dengan hakekatnya pula, HaKI dikelompokkan sebagai hak milik perorangan yang sifatnya tidak berwujud (intangible). Terlihat bahwa HaKI merupakan Hak Pemberian dari Umum (Publik) yang dijamin oleh Undang-undang. HaKI bukan merupakan Hak Azazi, sehingga kriteria pemberian HaKI merupakan hal yang dapat diperdebatkan oleh publik. Apa kriteria untuk memberikan HaKI? Berapa lama pemegang HaKI memperoleh hak eksklusif? Apakah HaKI dapat dicabut demi kepentingan umum? Bagaimana dengan HaKI atas formula obat untuk para penderita HIV/AIDs?

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

2.2. Perangkat Lunak Bebas

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.

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

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

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

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

2.3. Aneka Ragam HaKI

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

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

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

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

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

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

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

Contoh: Kacang Atom cap Ayam Jantan.

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

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

  Contoh: rahasia dari formula Parfum.

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

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

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

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

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

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

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

2.4. Lisensi Perangkat Lunak

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

lunak sangat merugikan perusahaan-perusahaan kecil yang cenderung tidak memiliki paten. Tetapi ada juga perusahaan kecil yang menyalahgunakan hal ini.

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

Perangkat Lunak Berpemilik (*Propriety*)

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

Perangkat Lunak Komersial

Perangkat lunak komersial adalah perangkat lunak yang dikembangkan oleh kalangan bisnis untuk memperoleh keuntungan dari penggunaannya. ``Komersial'' dan ``kepemilikan'' adalah dua hal yang berbeda! Kebanyakan perangkat lunak komersial adalah berpemilik, tapi ada perangkat lunak bebas komersial, dan ada perangkat lunak tidak bebas dan tidak komersial. Sebaiknya, istilah ini tidak digunakan.

Perangkat Lunak Semi-Bebas

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

Public Domain

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

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

Freeware

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

Shareware

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

GNU General Public License (GNU/GPL)

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

2.5. Sumber Terbuka (*Open Source*)

Walau pun PL memegang peranan yang penting, pengertian publik terhadap Hak atas Kekayaan Intelektual Perangkat Lunak (HaKI PL) masih relatif minim. Kebinggungan ini bertambah dengan peningkatan pemanfaatan dari Perangkat Lunak Bebas (PLB) – Free Software – dan Perangkat Lunak Sumber Terbuka (PLST) – Open Source Software (OSS). PLB ini sering disalahkaprahkan sebagai PLST, walau pun sebetulnya terdapat beberapa berbedaan yang mendasar diantara kedua pendekatan tersebut. Pada dasarnya, PLB lebih mengutamakan hal fundamental kebebasan, sedangkan PLST lebih mengutamakan kepraktisan pemanfaatan PL itu sendiri.

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

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

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.

2.6. *Copyleft*

Copyleft adalah pelesatan dari *copyright* (Hak Cipta). *Copyleft* merupakan PLB yang turunannya tetap merupakan PLB. Contoh lisensi *copyleft* ialah adalah GNU/GPL *General Public License*. Perangkat lunak *copylefted* merupakan perangkat lunak bebas yang ketentuan pendistribusinya tidak memperbolehkan untuk menambah batasan-batasan tambahan – jika mendistribusikan atau memodifikasi perangkat lunak tersebut. Artinya, setiap salinan dari perangkat lunak, walaupun telah dimodifikasi, haruslah merupakan perangkat lunak bebas.

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

2.7. Ilustrasi Lisensi

GNU/GPL bukan merupakan satu-satunya lisensi *copyleft*. Terdapat banyak lisensi lainnya seperti:
- Lisensi Apache
- Lisensi Artistic
- Lisensi FreeBSD
- Lisensi OpenLDAP

serta masih banyak lisensi lainnya.

2.8. Tantangan

Perangkat Keras Rahasia

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

Pustaka Tidak Bebas

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

Paten Perangkat Lunak

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

Dokumentasi Bebas

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

2.9. 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 kode-sumber-nya, perangkat lunak bebas dapat disesuaikan secara khusus untuk kebutuhan pemakai. Sesuatu yang tidak mudah untuk terselesaikan dengan perangkat lunak berpemilik. Selain itu, perangkat lunak bebas didukung oleh milis-milis pengguna yang dapat menjawab pertanyaan yang timbul karena permasalahan pada penggunaan perangkat lunak bebas.

Rujukan

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Bab 3. Organisasi Sistem Komputer

3.1. Pendahuluan

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

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

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

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

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

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

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;

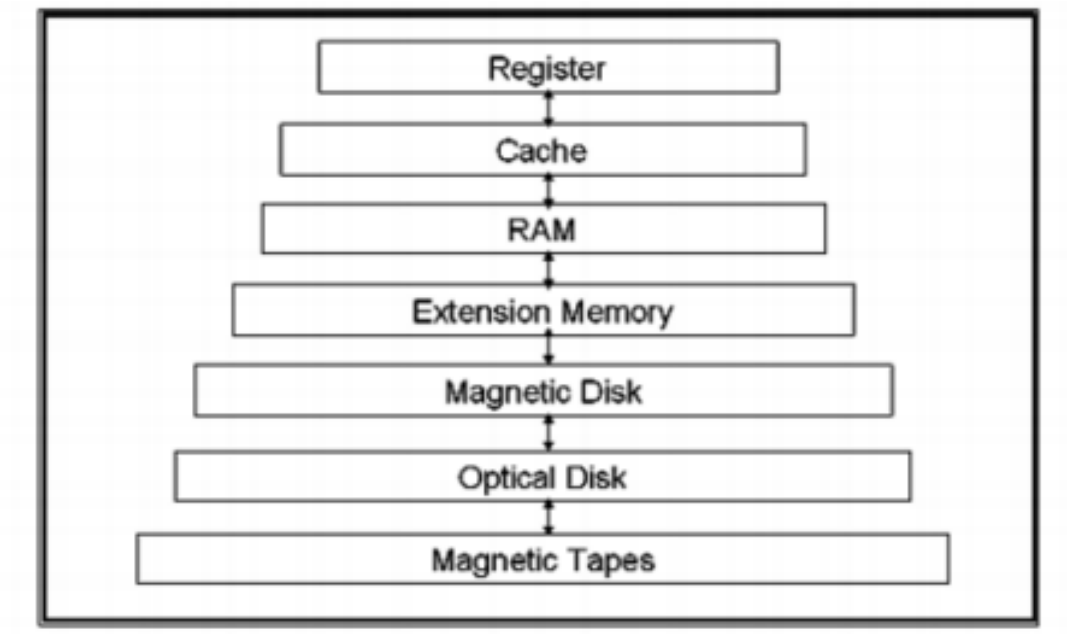
3.2. Prosesor

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

3.3. Penyimpan Data

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

Gambar 3.1. Penyimpanan Hirarkis



Register

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

Cache Memory

Tempat penyimpanan sementara (*volatile*) sejumlah kecil data untuk meningkatkan kecepatan pengambilan atau penyimpanan data di memori oleh prosesor yang berkecepatan tinggi. Dahulu *cache*

disimpan di luar prosesor dan dapat ditambahkan. Misalnya *pipeline burst* cache yang biasa ada di komputer awal tahun 90-an. Akan tetapi seiring menurunnya biaya produksi *die* atau *wafer* dan untuk meningkatkan kinerja, *cache* ditanamkan di prosesor. Memori ini biasanya dibuat berdasarkan desain memori statik.

Random Access Memory

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

Memori Ekstensi

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

Direct Memory Access

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

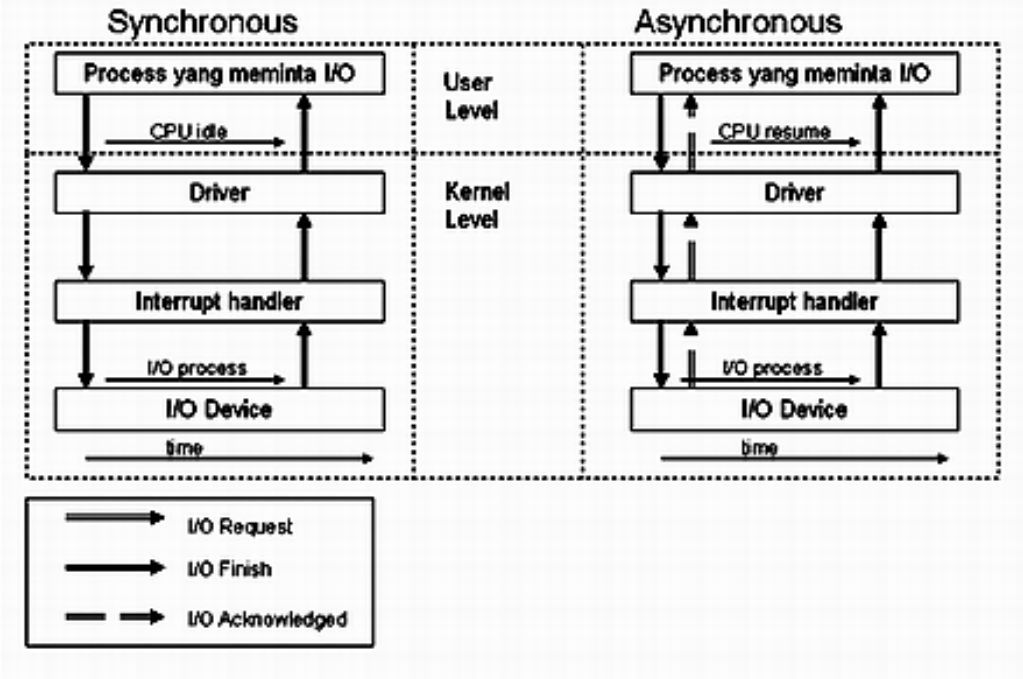
Media penyimpanan data yang non-volatile yang dapat berupa *Flash Drive*, *Optical Disc*, *Magnetic Disk*, *Magnetic Tape*. Media ini biasanya daya tampungnya cukup besar dengan harga yang relatif murah. *Portability*-nya juga relatif lebih tinggi.

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

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

3.4. Masukan/Keluaran

Gambar 3.2. Struktur M/K



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

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

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

3.5. Bus

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

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

Suatu jalur transfer data yang menghubungkan setiap *device* pada komputer. Hanya ada satu buah *device* yang boleh mengirimkan data melewati sebuah bus, akan tetapi boleh lebih dari satu *device*

yang membaca data bus tersebut. Terdiri dari dua buah model: *Synchronous bus* di mana digunakan dengan bantuan clock tetapi berkecepatan tinggi, tapi hanya untuk device berkecepatan tinggi juga; *Asynchronous bus* digunakan dengan sistem *handshake* tetapi berkecepatan rendah, dapat digunakan untuk berbagai macam *device*.

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.

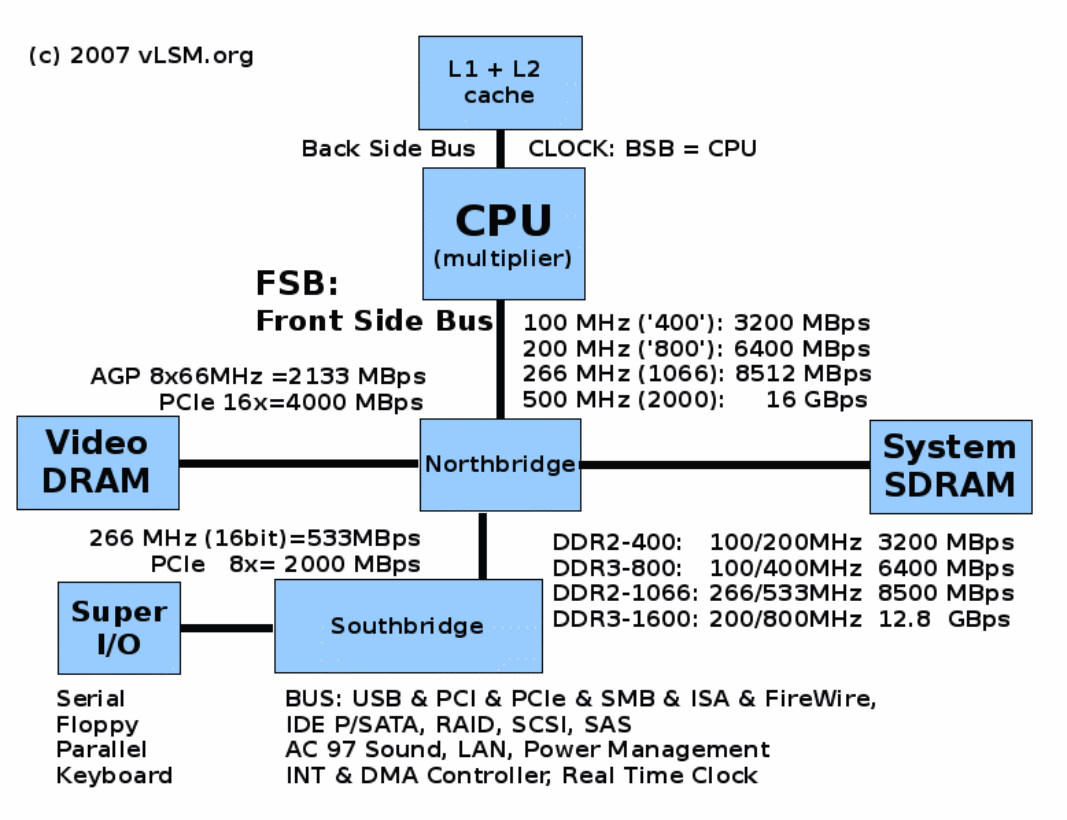
3.6. Boot

Pada saat pertama CPU aktif, program pertama yang di eksekusi berada di ROM. Langkah berikutnya dijalankan sebuah program untuk memasukkan sistem operasi ke dalam komputer. Proses ini disebut Boot Strap.

3.7. Komputer Personal

Berikut merupakan bagan sebuah komputer personal.

Gambar 3.3. Bagan Sebuah Komputer Personal



3.8. 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.

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

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

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

Rujukan

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

Bab 4. Bahasa Java

4.1. Pendahuluan

Java adalah sebuah teknologi yang diperkenalkan oleh Sun Microsystems pada pertengahan tahun 1990. Menurut definisi dari Sun, Java adalah nama untuk sekumpulan teknologi untuk membuat dan menjalankan perangkat lunak pada komputer *standalone* ataupun pada lingkungan jaringan. Kita lebih menyukai menyebut Java sebagai sebuah teknologi dibanding hanya sebuah bahasa pemrograman, karena Java lebih lengkap dibanding sebuah bahasa pemrograman konvensional. Teknologi Java memiliki tiga komponen penting, yaitu:

- *Programming-language specification*
- *Application-programming interface*
- *Virtual-machine specification*

4.2. Bahasa Pemrograman Java

Bahasa Java dapat dikategorikan sebagai sebuah bahasa pemrograman berorientasi objek, pemrograman terdistribusi dan bahasa pemrograman *multithrreaded*. Objek Java dispesifikasi dengan membentuk kelas. Untuk masing-masing kelas Java, kompiler Java memproduksi sebuah file keluaran arsitektur netral yang akan jalan pada berbagai implementasi dari *Java Virtual Machine (JVM)*. Awalnya Java sangat digemari oleh komunitas pemrograman internet, karena Java mendukung untuk *applets*, dimana program dengan akses sumber daya terbatas yang jalan dalam sebuah *web browser*. Java juga menyediakan dukungan level tinggi untuk *networking* dan objek terdistribusi.

Java juga dianggap sebagai sebuah bahasa yang aman. Tampilan ini pada khususnya penting menganggap bahwa sebuah program Java boleh mengeksekusi silang sebuah jaringan terdistribusi.

4.3. Java API

Java API terdiri dari tiga bagian utama:

- Java Standard Edition (SE), sebuah standar API untuk merancang aplikasi desktop dan *applets* dengan bahasa dasar yang mendukung grafis, M/K, keamanan, konektivitas basis data dan jaringan.
- Java Enterprose Edition (EE), sebuah inisiatif API untuk merancang aplikasi server dengan mendukung untuk basis data.
- Java Macro Edition (ME), sebuah API untuk merancang aplikasi yang jalan pada alat kecil seperti telepon genggam, komputer genggam dan pager.

4.4. Java Virtual Machine

Java Virtual Machine (JVM) adalah sebuah spesifikasi untuk sebuah komputer abstrak. JVM terdiri dari sebuah kelas pemanggil dan sebuah interpreter Java yang mengeksekusi kode arsitektur netral. Kelas pemanggil memanggil file .class dari kedua program Java dan Java API untuk dieksekusi oleh interpreter Java. Interpreter Java mungkin sebuah perangkat lunak interpreter yang menterjemahkan satu kode byte pada satu waktu, atau mungkin sebuah just-intime (JIT) kompiler yang menurunkan *bytecode* arsitektur netral kedalam bahasa mesin untuk *host computer*.

4.5. Sistem Operasi Java

Sistem operasi biasanya ditulis dalam sebuah kombinasi dari kode bahssa C dan assembly, terutama disebabkan oleh kelebihan performa dari bahasa tersebut dan memudahkan komunikasi dengan perangkat keras.

Satu kesulitan dalam merancang sistem basis bahasa adalah dalam hal proteksi memori, yaitu memproteksi sistem opeasi dari pemakai program yang sengaja memproteksi pemakai program lainnya. Sistem operasi tradisional mengaharapkan pada tampilan perangkat keras untuk menyediakan proteksi memori. Sistem basis bahasa mengandalkan pada tampilan keamanan dari bahasa. Sebagai hasilnya, sistem basis bahasa menginginkan pada alat perangkat keras kecil, yang mungkin kekurangan tampilan perangkat keras yang menyediakan proteksi memori.

4.6. Dasar Pemrograman

Java2 adalah generasi kedua dari Java *platform* (generasi awalnya adalah Java Development Kit). Java berdiri di atas sebuah mesin interpreter yang diberi nama JVM. JVM inilah yang akan membaca *bytecode* dalam file *.class* dari suatu program sebagai representasi langsung program yang berisi bahasa mesin. Oleh karena itu, bahasa Java disebut sebagai bahasa pemrograman yang *portable* karena dapat dijalankan pada berbagai sistem operasi, asalkan pada sistem operasi tersebut terdapat JVM.

Platform Java terdiri dari kumpulan *library*, JVM, kelas- kelas *loader* yang dipaket dalam sebuah lingkungan rutin Java, dan sebuah *compiler*, *debuger*, dan perangkat lain yang dipaket dalam Java Development Kit (JDK). Java2 adalah generasi yang sekarang sedang berkembang dari *platform*Java. Agar sebuah program Java dapat dijalankan, maka file dengan ekstensi ".java" harus dikompilasi menjadi file *bytecode*. Untuk menjalankan *bytecode* tersebut dibutuhkan JRE (*Java Runtime Environment*) yang memungkinkan pemakai untuk menjalankan program Java, hanya menjalankan, tidak untuk membuat kode baru lagi. JRE berisi JVM dan *library*Java yang digunakan.

*Platform*Java memiliki tiga buah edisi yang berbeda, yaitu J2EE (*Java2 Enterprose Edition*), J2ME (*Java2 Micro Edition*) dan J2SE (*Java2 Second Edition*). J2EE adalah kelompok dari beberapa API (*Application Programming Interface*) dari Java dan teknologi selain Java. J2EE sering dianggap sebagai *middleware* atau teknologi yang berjalan di *server*, namun sebenarnya J2EE tidak hanya terbatas untuk itu. Faktanya J2EE juga mencakup teknologi yang dapat digunakan di semua lapisan dari sebuah sistem informasi. Implementasi J2EE menyediakan kelas dasar dan API dari Java yang mendukung pengembangan dari rutin standar untuk aplikasi klien maupun *server*, termasuk aplikasi yang berjalan di *web browser*. J2SE adalah lingkungan dasar dari Java, sedangkan J2ME merupakan edisi *library* yang dirancang untuk digunakan pada *device* tertentu seperti *pagers* dan *mobile phone*.

Java merupakan bahasa pemrogrman yang bersifat *case sensitive* yang berarti penulisan menggunakan huruf besar ataupun huruf kecil pada kode program dapat berarti lain. Misalnya penulisan "System" akan diartikan berbeda dengan "system" oleh interpreter. Java tidak seperti C++, Java tidak mendukung pemrograman prosedural, tapi mendukung pemrograman berorientasi objek sehingga ada sintaks *class* pada kode programnya.

4.7. Objek dan Kelas

Sebuah kelas menyerupai sebuah struktur yang merupakan tipe data sendiri, misalkan tipe data titik yang terdiri dari koordinat x dan y. Bahasa Java telah menggunakan sebuah kelas untuk menyatakan tipe data titik karena bahasa pemrograman Java merupakan bahasa pemrograman berorientasi objek murni sehingga tidak mengenal struktur tapi mengenal apa yang disebut dengan kelas.

Perbedaan sebuah kelas dengan sebuah struktur adalah sebuah kelas dapat berdiri sendiri dan dapat digunakan untuk berbagai keperluan kelas-kelas yang lain, sedangkan sebuah struktur tidak dapat berdiri sendiri. Sebuah kelas lebih fleksibel untuk digunakan oleh kelas lain tanpa harus membongkar kode program utama, sedangkan jika digunakan struktur maka kode program harus dibongkar unuk disalin bagian strukturnya ke kode program utama yang lain. Sebuah *file* dapat terdiri dari berbagai kelas, namun biasanya pada bahasa pemrograman Java sebuah *file* hanya terdiri dari satu kelas yang disimpan dengan nama kelas, misal *file* List.java berisi kelas *List*. Namun jika kelas yang dibuat misalnya public class nama_kelas, maka kelas itu harus disimpan dalam satu *file* hanya untuk satu kelas. Setelah dilakukan kompilasi maka pada Java akan ada sebuah *file* ".class" yang berisi *bytecode* dari setiap kelas. Jika sebuah *file* terdiri dari dua kelas maka setelah dikompilasi akan dihasilkan dua buah *file* ".class" yang nantinya akan dibaca oleh interpreter Java saat program dieksekusi.

Sebuah kelas saat program dieksekusi dan perintah new dijalankan, maka akan dibuat sebuah objek. Objek adalah elemen pada saat *runtime* yang akan diciptakan, dimanipulasi dan dihancurkan saat eksekusi sehingga sebuah objek hanya ada saat sebuah program dieksekusi, jika masih dalam bentuk kode, disebut sebagai kelas jadi pada saat *runtime* (saat sebuah program dieksekusi), yang kita punya adalah objek, di dalam teks program yang kita lihat hanyalah kelas.

4.8. Atribut

Atribut dari sebuah kelas adalah variabel global yang dimilki sebuah kelas, misalkan pada kelas sebagai berikut:

```
class Elemen

   NilaiMatKul elmt
   Elemen next

   Elemen()
   {end constructor}
{end class}
```

maka elmt dan next adalah atribut dari kelas Elemen. Atribut pada sebuah kelas memiliki izin akses jika kelas digunakan oleh kelas lain, izin akses itu seperti *private*, *public* dan *protected*.

4.9. Atribut *Private*

Izin akses *private* pada sebuah atribut biasanya digunakan oleh sebuah kelas untuk melindungi atribut-atributnya agar tidak dapat diakses oleh kelas lain. Sebuah atribut yang dinyatakan sebagai *private* hanya dapat diakses secara langsung oleh kelas yang membungkusnya, sedangakan kelas lainnya tidak dapat mengakses atribut ini secara langsung, misalkan kelas sebagai berikut:

```
class Elemen
   private NilaiMatKul elmt
   private Elemen next

   Elemen()
   {end constructor}

{end class}
```

maka yang dapat mengakses elmt dan next hanyalah kelas Elemen sehingga jika sebuah kelas *List* di dalamnya mempunyai kode sebagai berikut:

```
Elemen e <- new Elemen
```

maka pengaksesan e.next tidak dizinkan pada kelas *List*. Agar isi dari sebuah atribut *private* dapat diakses oleh kelas lain dapat dibuat sebuah metode yang mengembalikan nilai atribut itu, misalnya sebagai berikut:

```
public getNext() -> Elemen
   -> next
{end getNext}
```

sehingga kelas lain akan mengakses atribut next pada kelas Elemen dengan kode Elemen n < e.getNext().

4.10. Atribut *Public*

Izin akses *public* sebuah kelas, jika sebuah atribut diperbolehkan diakses secara langsung oleh kelas lain. Sebuah atribut yang dinyatakan sebagai *public* dapat diakses secara langsung oleh kelas lain di luar kelas yang membungkusnya, misalkan pada kelas Elemen sebagai berikut:

```
class Elemen
   public NilaiMatKul elmt
   Elemen next

   Elemen ()
   {end constructor}
{end class}
```

maka aatribut elmt dan next dapat diakses secara langsung oleh kelas lain, misalkan dengan kode:

```
Elemen e <- new Elemen()
e.next <- NULL
```

Jika sebuah atribut tidak ditulis izin aksesnya misalkan hanya ditulis Element next, maka interpreter Java akan menganggap atribut next mempunyai izin akses *public*.

4.11. Atribut *Protected*

Izin akses *protected* sebuah atribut biasanya digunakan oleh sebuah kelas, jika sebuah atribut diperbolehkan diakses secara langsung oleh kelas lain yang merupakan kelas turunannya (*inheritance*). Sebuah atribut yang dinyatakan sebagai *protected* tidak dapat diakses secara langsung oleh kelas lain di luar kelas yang membungkusnya, kecuali kelas yang mengaksesnya adalah kelas turunan dari kelas yang membungkusnya, misalkan pada kelas Elemen sebagai berikut:

```
class Elemen
   protected NilaiMatKul elmt
   protected Elemen next

   Elemen()
   {end constructor}
{end class}
```

maka atribut elmt dan next dapat diakses secara langsung oleh kelas lain yang merupakan turunan kelas Elemen. Izin akses *protected*dimaksudkan untuk melindungi atribut agar tidak diakses secara langsung oleh sembarang kelas lain, namun diizinkan diakses secara langsung oleh kelas turunannya.

4.12. Konstruktor

Sebuah kelas harus memiliki sebuah metode yang disebut sebagai konstruktor. nama sebuah konstruktor harus sama dengan nama dari sebuah kelas, misalkan kelas Elemen maka konstruktornya adalah Elemen(). Sebuah konstruktor juga dapat menerima sebuah masukan seperti halnya prosedur pada pemrograman prosedural. Fungsi dari sebuah konstruktor adalah: mengalokasikan sebuah objek saat program dieksekusi, memberikan nilai awal sebagai inisialisasi dari semua atribut yang perlu diinisialisasi dan mengerjakan proses- proses yang diperlukan saat sebuah objek dibuat.

Namun pada kenyataannya sebuah konstruktor dapat tidak berisi apa-apa, hal ini jika memang tidak diperlukan adanya inisialisasi atau proses yang dikerjakan ketika sebuah objek dibuat. Konstruktor

harus bersifat *public*karena sebuah konstruktor akan diakses oleh kelas lain untuk membuat objek suatu kelas.

Sebuah kelas dapat memiliki konstruktor lebih dari satu. Pada saat eksekusi program, kompiler atau interpreter akan mencari konstruktor mana yang sesuai dengan konstruktor yang dipanggil, hal ini disebut sebagai *overloading*.

4.13. Metode

Metode pada sebuah kelas hampir sama dengan fungsi atau prosedur pada pemrograman prosedural. Pada sebuah metode di dalam sebuah kelas juga memiliki izin akses seperti halnya atribut pada kelas, izin akses itu antara lain *private*, *public*dan *protected*yang memiliki arti sama pada izin akses atribut yang telah dibahas sebelumnya. Sebuah kelas boleh memiliki lebih dari satu metode dengan nama yang sama asalkan memiliki parameter masukan yang berbeda sehingga kompiler atau interpreter dapat mengenali metode mana yang dipanggil.

Di dalam sebuah kelas, terdapat juga yang disebut sebagai metode atau atribut statis yang memiliki kata kunci static. Maksud dari statis di sini adalah metode yang dapat diakses secara berbagi dengan semua objek lain tanpa harus membuat objek yang memiliki metode statis tadi (tanpa proses new), tapi sebuah metode statis mempunyai keterbatasan yaitu hanya dapat mengakses atribut atau metode lain di dalam kelas yang membungkusnya yang juga bersifat statis. Metode statis biasanya diimplementasikan untuk metode main.

4.14. *Inheritance*

*Inheritance*atau pewarisan pada pemrograman berorientasi objek merupakan suatu hubungan dua buah kelas atau lebih. Dalam hal ini ada kelas yang memiliki atribut dan metode yang sama dengan kelas lainnya beserta atribut dan metode tambahan yang merupakan sifat khusus kelas yang menjadi turunannya. Sebagai contoh, misalkan ada sebuah kelas Titik yang mempunyai kelas turunan Titik3D:

```
class Titik
   private integer x
   private integer y
      Titik()
         x < 0
         y < 0
      {end Titik}

      public getX() -> integer
         -> x
      {end getX}

      public getY() -> integer
         -> y
      {end getY}
{end class}
class Titik3D: Titik
   private integer z

   Titik3D()
      z <- 0
   {end Titik3D}

      public getZ() -> integer
         -> z
      {end getZ}
```

```
{end class}
```

Keterkaitan antara kelas Titik dan Titik3D adalah kelas Titik3D merupakan kelas turunan dari kelas Titik. Dalam hal ini kelas Titik disebut dengan kelas dasar atau *super class*atau *base class*sedangkan kelas Titik3D disebut sebagai kelas turunan atau *derived class*atau *subclass*.

Pada contoh di atas, ketika kelas Titik3D dibuat objeknya maka objek tersebut dapat menggunakan metode yang ada pada kelas Titik walau pada kode programnya metode itu tidak dituliskan, misalkan sebagai berikut:

```
Titik3D p <-  new Titik3D()
integer x <-  p.getX()
integer y <-  p.getY()
integer z <-  p.getZ()
```

Keuntungan dari pewarisan adalah tidak perlu mengutak atik kode kelas yang membutuhkan tambahan atribut atau metode saja, karena tinggal membuat kelas turunannya tanpa harus mengubah kode kelas dasarnya. Kelas dasar akan mewariskan semua atribut dan kodenya kecuali konstruktor dan destruktor yang memiliki izin akses *public*dan *protected*ke kelas turunannya dengan izin akses yang sama dengan pada kelas dasar.

Ketika sebuah kelas turunan dibuat objeknya saat eksekusi, maka secara implisit konstruktor kelas dasar dipanggil terlebih dahulu baru kemudian konstruktor kelas turunan dijalankan. Begitu juga saat objek dimusnahkan maka secara destruktor kelas turunan akan dijalankan baru kemudian destruktor kelas dasar dijalankan.

4.15. *Abstract*

Pada bahasa pemrograman Java juga ada sebuah kata kunci *abstract*yang dapat digunakan pada sebuah metode, namun jika digunakan pada sebuah metode, maka metode tersebut harus berada di dalam sebuah kelas yang juga menggunakan kata kunci *abstract*. Metode *abstract*tidak boleh memiliki badan program, badan program metode ini dapat diimplementasikan pada kelas turunannya.

Fungsi dari kelas atau metode *abstract*pada bahasa pemrograman Java adalah menyediakan sebuah abstraksi kelas atau metode sehingga dapat dilihat metode apa saja yang ada di dalam kelas tanpa harus melihat isi badan program dari metode-metode itu. Prinsipnya sama dengan fungsi sebuah daftar isi pada sebuah buku, dengan melihat daftar isi bisa diketahui isi sebuah buku tanpa harus membaca semua isi buku terlebih dahulu.

4.16. *Package*

*Package*adalah sebuah kontainer atau kemasan yang dapat digunakan untuk mengelompokkan kelas-kelas sehingga memungkinkan beberapa kelas yang bernama sama disimpan dalam *package*yang berbeda. Sebuah *package*pada Java dapat digunakan oleh *package*yang lain ataupun kelas-kelas di luar *Package*. Jika dalam bahasa pemrograman Java terdapat kode import example.animal.Mamalia; maka program tersebut memakai kelas mamalia yang ada pada *package*example.animal. Jika terdapat kode import example.animal.*; maka program tersebut memakai semua kelas yang ada pada *package*example.animal.

*Package*pada bahasa pemrograman Java dinyatakan dengan kode: package nama_package;

Misalnya: package example.animal;

yang ditulis pada bagian atas kode program kelas anggota *package*. Misal sebuah kelas dengan nama Mamalia ada di dalam *package*dengan nama example.animal maka file yang menyimpan kode program kelas Mamalia dimasukkan dalam direktori.

4.17. *Interface*

*Interface*atau antar muka pada bahasa pemrograman Java sangat mirip dengan kelas, tapi tanpa atribut kelas dan memiliki metode yang dideklarasikan tanpa isi. Deklarasi metode pada sebuah *interface*dapat diimplementasikan oleh kelas lain. Sebuah kelas dapat mengimplementasikan lebih dari satu *interface*bahwa kelas ini akan mendeklarasikan metode pada *interface*yang dibutuhkan kelas itu sekaligus mendefiniskan isinya pada kode program kelas itu. Metode pada *interface*yang diimplementasikan pada suatu kelas harus sama persis dengan yang ada pada *interface*. Misalnya pada *interface*terdapat deklarasi void printAnimal(); maka pada kelas yang mengimplementasikan metode itu harus ditulis sama yaitu:

```
void printAnimal(){ .....
   }
```

Sebuah *interface*dideklarasikan dengan kode:

```
interface nama_antarmuka{ metode_1 metode_2
   ..... metode_n }
```

misalnya:

```
interface Animal{ void printAnimal();
   }
```

Adapun deklarasi kelas yang mengimplementasikan interface sebagai berikut:

```
class nama_kelas implements interface_1,
   interface_2, ..., interface_n{ metode_1 metode_2
   ..... metode_n }
```

misalnya:

```
class Mamalia implements Animal{ Mamalia (){ }
   void prontAnimal(){ system.out.println("printAnimal dalam kelas
   Mamalia"); } }
```

4.18. Rangkuman

Java adalah sebuah teknologi yang diperkenalkan oleh Sun Microsystems pada pertengahan tahun 1990. Menurut definisi dari Sun, Java adalah nama untuk sekumpulan teknologi untuk membuat dan menjalankan perangkat luinak pada komputer *standalone*ataupun pada lingkungan jaringan. Teknologi Java memiliki tiga komponen penting, yaitu: *Programming-language specification*, *Application- programming interface*, *Application-programming interface*.

Java2 adalah generasi kedua dari Java *platform*(generasi awalnya adalah Java Development Kit). Java berdiri di atas sebuah mesin interpreter yang diberi nama JVM. JVM inilah yang akan membaca *bytecode*dalam file *.class*dari suatu program sebagai representasi langsung program yang berisi bahasa mesin. Oleh karena itu, bahasa Java disebut sebagai bahasa pemrograman yang *portable*karena dapat dijalankan pada berbagai sistem operasi, asalkan pada sistem operasi tersebut terdapat JVM.

Rujukan

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

Bagian II. Konsep Dasar Sistem Operasi

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

Bab 5. Komponen Sistem Operasi

5.1. Pendahuluan

Sistem operasi dapat dikatakan adalah perangkat lunak yang sangat kompleks. Hal-hal yang ditangani oleh sistem operasi bukan hanya satu atau dua saja, melainkan banyak hal. Dari menangani perangkat keras, perangkat lunak atau program yang berjalan, sampai menangani pengguna. Hal tersebut menyebabkan sebuah sistem operasi memiliki banyak sekali komponen-komponen tersendiri yang memiliki fungsinya masing-masing. Seluruh komponen yang menyusun sistem operasi tersebut saling bekerjasama untuk satu tujuan, yaitu efisiensi kerja seluruh perangkat komputer dan kenyamanan dalam penggunaan sistem operasi.

Oleh karena itu, penting bagi kita untuk mengetahui komponen-komponen apa saja yang ada di dalam sebuah sistem operasi, agar kita bisa mempelajari sistem operasi secara menyeluruh. Bab ini menceritakan secara umum apa saja komponen-komponen yang ada di sistem operasi. Detail tentang setiap komponen tersebut ada di bab-bab selanjutnya dalam buku ini.

Tanpa satu saja dari komponen-komponen tersebut, bisa dipastikan sebuah sistem operasi tidak akan berjalan dengan maksimal. Bayangkan jika kita memiliki sistem operasi yang tidak memiliki kemampuan untuk menangani program-program yang berjalan sekaligus. Kita tak akan bisa mengetik sambil mendengarkan lagu sambil berselancar di internet seperti yang biasa kita lakukan saat ini.

Contoh sebelumnya hanya sedikit gambaran bagaimana komponen-komponen sistem operasi tersebut saling terkait satu sama lainnya. Mempelajari komponen sistem operasi secara umum dapat mempermudah pemahaman untuk mengetahui hal-hal yang lebih detail lagi tentang sistem operasi.

Dari berbagai macam sistem operasi yang ada, tidak semuanya memiliki komponen-komponen penyusun yang sama. Pada umumnya sebuah sistem operasi modern akan terdiri dari komponen sebagai berikut:

- Manajemen Proses.
- Manajemen Memori Utama.
- Manajemen Sistem Berkas.
- Manajemen Sistem M/K.
- Manajemen Penyimpanan Sekunder.
- Proteksi dan Keamanan.

5.2. Kegiatan Sistem Operasi

Dalam kegiatannya sehari-hari, sistem operasi memiliki sebuah mekanisme proteksi untuk memastikan dirinya, semua program yang berjalan, dan data-data penggunanya berjalan dengan baik. Untuk melakukan hal tersebut, sistem operasi memiliki dua jenis (*mode*) operasi yang saling terpisah. Dua operasi tersebut, yaitu *user mode*, eksekusi program dikendalikan oleh pengguna, dan *kernel mode*, eksekusi program dikendaikan oleh sistem operasi, dinamakan *dual-mode operation*.

Dual-mode operation diimplementasikan pada arsitektur perangkat keras. Sebuah bit yang disebut *mode bit* ditambahkan ke perangkat keras untuk menunjukkan mode operasi saat itu: 0 untuk *kernel mode* dan 1 untuk *user mode*.

Dengan adanya *dual-mode operation*, eksekusi sebuah program/proses bisa dibedakan sumbernya, apakah dieksekusi oleh sistem operasi atau dieksekusi oleh pengguna. Hal ini akan sangat berguna dalam berjalannya sistem operasi.

Selain itu, sistem operasi memiliki sebuah mekanisme untuk melindungi prosesor dari berbagai macam program yang berjalan. Bayangkan jika ada sebuah proses mengalami *infinite loop*. Tentu saja prosesor akan terus menerus melayani program itu dan menghambat proses lainnya yang akan dieksekusi prosesor, dan hal ini bisa dipastikanakan mengurangi kinerja dari komputer.

Perlindungan prosesor tersebut dilakukan dengan *timer*. *Timer* diset untuk melakukan interupsi prosesor setelah beberapa periode waktu. Dengan adanya *timer*, sebuah program bisa dicegah dari berjalan terlalu lama. Misalkan sebuah program memiliki *time limit* 7 menit. Setelah 7 menit tersebut terlewati, sistem operasi akan menginterupsi prosesor dan menghentikan eksekusi program tersebut.

5.3. Manajemen Proses

Proses adalah sebuah program yang sedang dieksekusi. Sedangkan program adalah kumpulan instruksi yang ditulis ke dalam bahasa yang dimengerti sistem operasi. Sebuah proses membutuhkan sejumlah sumber daya untuk menyelesaikan tugasnya. Sumber daya tersebut dapat berupa *CPU time*, alamat memori, berkas-berkas, dan perangkat-perangkat M/K. Sistem operasi mengalokasikan sumber daya-sumber daya tersebut saat proses itu diciptakan atau sedang diproses/dijalankan. Ketika proses tersebut berhenti dijalankan, sistem operasi akan mengambil kembali semua sumber daya agar bisa digunakan kembali oleh proses lainnya.

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

- **Membuat dan menghapus proses pengguna dan sistem proses.** Sistem operasi bertugas mengalokasikan sumber daya yang dibutuhkan oleh sebuah proses dan kemudian mengambil sumber daya itu kembali setelah proses tersebut selesai agar dapat digunakan untuk proses lainnya.
- **Menunda atau melanjutkan proses.** Sistem operasi akan mengatur proses apa yang harus dijalankan terlebih dahulu berdasarkan berdasarkan prioritas dari proses-proses yang ada. Apa bila terjadi 2 atau lebih proses yang mengantri untuk dijalankan, sistem operasi akan mendahulukan proses yang memiliki prioritas paling besar.
- **Menyediakan mekanisme untuk proses sinkronisasi.** Sistem operasi akan mengatur jalannya beberapa proses yang dieksekusi bersamaan. Tujuannya adalah menghindarkan terjadinya inkonsistensi data karena pengaksesan data yang sama, juga untuk mengatur urutan jalannya proses agar setiap proses berjalan dengan lancar
- **Menyediakan mekanisme untuk proses komunikasi.** Sistem operasi menyediakan mekanisme agar beberapa proses dapat saling berinteraksi dan berkomunikasi (contohnya berbagi sumber daya antar proses) satu sama lain tanpa menyebabkan terganggunya proses lainnya.
- **Menyediakan mekanisme untuk penanganan *deadlock*.** *Deadlock* adalah suatu keadaan dimana sistem seperti terhenti karena setiap proses memiliki sumber daya yang tidak bisa dibagi dan menunggu untuk mendapatkan sumber daya yang sedang dimiliki oleh proses lain. Saling menunggu inilah yang disebut *deadlock*(kebuntuan). Sistem operasi harus bisa mencegah, menghindari, dan mendeteksi adanya *deadlock*. Jika *deadlock* terjadi, sistem operasi juga harus dapat memulihkan kondisi sistemnya.

5.4. Manajemen Memori Utama

Sistem operasi memiliki tugas untuk mengatur bagian memori yang sedang digunakan dan mengalokasikan jumlah dan alamat memori yang diperlukan, baik untuk program yang akan berjalan maupun untuk sistem operasi itu sendiri. Tujuan dari manajemen memori utama adalah agar utilitas CPU meningkat dan untuk meningkatkan efisiensi pemakaian memori.

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 M/K. Memori utama termasuk tempat penyimpanan data yang yang bersifat *volatile*(tidak permanen), yaitu data akan hilang kalau komputer dimatikan.

Sistem komputer modern memiliki sistem hirarki memori, artinya memori yang ada di komputer disusun dengan tingkatan kecepatan dan kapasitas yang berbeda. Memori yang memiliki kecepatan sama dengan kecepatan prosesor memiliki kapasitas yang kecil, berkisar hanya dari ratusan KB hingga 4 MB dengan harga yang sangat mahal. Sedangkan memori utama yang kecepatannya jauh di bawah kecepatan prosesor memiliki kapasitas yang lebih besar, berkisar dari 128 MB hingga 4 GB dengan

harga yang jauh lebih murah. Sistem hirarki memori ini memiliki tujuan agar kinerja komputer yang maksimal bisa didapat dengan harga yang terjangkau.

5.5. Manajemen Sistem Berkas

File atau berkas adalah representasi program dan data yang berupa kumpulan informasi yang saling berhubungan dan disimpan di perangkat penyimpanan. Sistem berkas ini sangatlah penting, karena informasi atau data yang disimpan dalam berkas adalah sesuatu yang sangat berharga bagi pengguna. Sistem operasi harus dapat melakukan operasi-operasi pada berkas, seperti membuka, membaca, menulis, dan menyimpan berkas tersebut pada sarana penyimpanan sekunder. Oleh karena itu, sistem operasi harus dapat melakukan operasi berkas dengan baik.

Sistem operasi melakukan manajemen sistem berkas dalam beberapa hal:

- **Pembuatan berkas atau direktori.** Berkas yang dibuat nantinya akan diletakkan pada direktori-direktori yang diinginkan pada sistem berkas. Sistem operasi akan menunjukkan tempat dimana lokasi berkas atau direktori tersebut akan diletakkan. Setelah itu, sistem operasi akan membuat entri yang berisi nama berkas dan lokasinya pada sistem berkas.
- **Penghapusan berkas atau direktori.** Sistem operasi akan mencari letak berkas atau direktori yang hendak dihapus dari sistem berkas, lalu menghapus seluruh entri berkas tersebut, agar tempat dari berkas tersebut dapat digunakan oleh berkas lainnya.
- **Pembacaan dan menulis berkas.** Proses pembacaan dan penulisan berkas melibatkan *pointer* yang menunjukkan posisi dimana sebuah informasi akan dituliskan di dalam sebuah berkas.
- **Meletakkan berkas pada sistem penyimpanan sekunder.** Sistem operasi mengatur lokasi fisik tempat penyimpanan berkas pada sarana penyimpanan sekunder

5.6. Manajemen Sistem M/K (*I/O*)

Pekerjaan utama yang paling sering dilakukan oleh sistem komputer selain melakukan komputasi adalah Masukan/Keluaran (M/K). Dalam kenyataannya, waktu yang digunakan untuk komputasi lebih sedikit dibandingkan waktu untuk M/K. Ditambah lagi dengan banyaknya variasi perangkat M/K sehingga membuat manajemen M/K menjadi komponen yang penting bagi sebuah sistem operasi. Sistem operasi juga sering disebut *device manager*, karena sistem operasi mengatur berbagai macam perangkat (*device*).

Fungsi-fungsi sistem operasi untuk sistem M/K:

- **Penyanggaan (*buffering*).** Menampung data sementara dari/ke perangkat M/K
- **Penjadwalan (*scheduling*).** Melakukan penjadualan pemakaian M/K sistem supaya lebih efisien.
- **Spooling.** Meletakkan suatu pekerjaan program pada penyangga, agar setiap perangkat dapat mengaksesnya saat perangkat tersebut siap.
- **Menyediakan *driver* perangkat yang umum.** *Driver* digunakan agar sistem operasi dapat memberi perintah untuk melakukan operasi pada perangkat keras M/K yang umum, seperti *optical drive*, media penyimpanan sekunder, dan layar monitor.
- **Menyediakan *driver* perangkat yang khusus.** *Driver* digunakan agar sistem operasi dapat memberi perintah untuk melakukan operasi pada perangkat keras M/K tertentu, seperti kartu suara, kartu grafis, dan *motherboard*

5.7. Manajemen Penyimpanan Sekunder

Penyimpanan sekunder (*secondary storage*) adalah sarana penyimpanan yang berada satu tingkat di bawah memori utama sebuah komputer dalam hirarki memori. Tidak seperti memori utama komputer, penyimpanan sekunder tidak memiliki hubungan langsung dengan prosesor melalui bus, sehingga harus melewati M/K.

Sarana penyimpanan sekunder memiliki ciri-ciri umum sebagai berikut:

1. ***Non volatile*(tahan lama).** Walaupun komputer dimatikan, data-data yang disimpan di sarana penyimpanan sekunder tidak hilang. Data disimpan dalam piringan-piringan magnetik.
2. **Tidak berhubungan langsung dengan bus CPU.** Dalam struktur organisasi komputer modern, sarana penyimpanan sekunder terhubung dengan *northbridge*. *Northbridge* yang menghubungkan sarana penyimpanan sekunder pada M/K dengan bus CPU.
3. **Lambat.** Data yang berada di sarana penyimpanan sekunder memiliki waktu yang lebih lama untuk diakses (*read/write*) dibandingkan dengan mengakses di memori utama. Selain disebabkan oleh *bandwidth* bus yang lebih rendah, hal ini juga dikarenakan adanya mekanisme perputaran *head* dan piringan magnetik yang memakan waktu.
4. **Harganya murah.** Perbandingan harga yang dibayar oleh pengguna per *byte* data jauh lebih murah dibandingkan dengan harga memori utama.

Sarana penyimpanan sekunder memiliki fungsi-fungsi sebagai berikut:
1. **Menyimpan berkas secara permanen.** Data atau berkas diletakkan secara fisik pada piringan magnet dari disk, yang tidak hilang walaupun komputer dimatikan (*non volatile*)
2. **Menyimpan program yang belum dieksekusi prosesor.** Jika sebuah program ingin dieksekusi oleh prosesor, program tersebut dibaca dari disk, lalu diletakkan di memori utama komputer untuk selanjutnya dieksekusi oleh prosesor menjadi proses.
3. **Memori virtual.** Adalah mekanisme sistem operasi untuk menjadikan beberapa ruang kosong dari disk menjadi alamat-alamat memori virtual, sehingga prosesor bisa menggunakan memorivirtual ini seolah-olah sebagai memori utama. Akan tetapi, karena letaknya di penyimpanan sekunder, akses prosesor ke memori virtual menjadi jauh lebih lambat dan menghambat kinerja komputer.

Sistem operasi memiliki peran penting dalam manajemen penyimpanan sekunder. Tujuan penting dari manajemen ini adalah untuk keamanan, efisiensi, dan optimalisasi penggunaan sarana penyimpanan sekunder.

5.8. Proteksi dan Keamaman

Seringkali, istilah keamanan dan proteksi membingungkan dalam penggunaannya. Untuk mengurangi kebingungan itu, istilah keamanan digunakan untuk penggambaran secara umum, sedangkan proteksi digunakan untuk menggambarkan secara teknis mekanisme perlindungan sistem operasi.

Proteksi

Proteksi adalah mekanisme sistem operasi untuk mengontrol akses terhadap beberapa objek yang diproteksi dalam sistem operasi. Objek-objek tersebut bisa berupa perangkat keras (seperti CPU, memori, disk, printer, dll) atau perangkat lunak (seperti program, proses, berkas, basis data, dll). Di beberapa sistem, proteksi dilakukan oleh sebuah program yang bernama *reference monitor*. Setiap kali ada pengaksesan sumber daya PC yang diproteksi, sistem pertama kali akan menanyakan *reference monitor* tentang keabsahan akses tersebut. *Reference monitor* kemudian akan menentukan keputusan apakah akses tersebut diperbolehkan atau ditolak.

Secara sederhana, mekanisme proteksi dapat digambarkan dengan konsep *domain*. *Domain* adalah himpunan yang berisi pasangan objek dan hak akses. Masing-masing pasangan *domain* berisi sebuah objek dan beberapa akses operasi (seperti *read, write, execute*) yang dapat dilakukan terhadap objek tersebut. Dalam setiap waktu, setiap proses berjalan dalam beberapa *domain* proteksi. Hal itu berarti terdapat beberapa objek yang dapat diakses oleh proses tersebut, dan operasi-operasi apa yang boleh dilakukan oleh proses terhadap objek tersebut. Proses juga bisa berpindah dari *domain* ke *domain* lain dalam eksekusi.

Keamanan

Pengguna sistem komputer sudah tentu memiliki data-data dan informasi yang berharga baginya. Melindungi data-data ini dari pihak-pihak yang tidak berhak merupakan hal penting bagi sistem operasi. Inilah yang disebut keamanan (*security*).

Sebuah sistem operasi memiliki beberapa aspek tentang keamanan. Aspek-aspek ini berhubungan terutama dengan hilangnya data-data. Sistem komputer dan data-data di dalamnya terancam dari aspek ancaman (*threats*), aspek penyusup (*intruders*), dan aspek musibah.

Dari aspek ancaman, secara umum sistem komputer menghadapi ancaman terbukanya data-data rahasia, pengubahan data-data oleh orang yang tidak berhak, juga pelumpuhan sistem dengan adanya *Denial of Service*(DoS).

Dari aspek penyusup, saat ini banyak orang mencoba masuk ke dalam sistem operasi dengan berbagai macam tujuan. Ada yang hanya sekedar mencoba menjebol sistem operasi (*hacking*), ada yang mencoba mengambil keuntungan dari tindakan penjebolah itu (*cracking*).

Tidak hanya disusupi oleh manusia, sistem operasi juga menghadapi ancaman keamanan dari program-program penyusup, yang disebut *malicious program* atau *malware*. *Malware* adalah program yang menyusup ke dalam sistem operasi dan memiliki tujuan-tujuan tertentu seperti mengambil data-data pribadi, mengambil alih komputer, dan seringkali bertujuan merusak. Yang termasuk kategori *malware* adalah virus, *keylogger*, *worm*, *trojan*, dan *sypware*.

Yang terakhir, sistem operasi dan data-data di dalamnya terancam justru dari hal-hal non teknis, yaitu dari musibah. Sistem operasi terancam akibat adanya bencana alam (banjir, lumpur panas, gempa bumi, dan lain-lain), kerusakan perangkat keras atau lunak, bahkan kelalaian dari penggunanya.

Perkembangan dunia internet saat ini membawa konsekuensi meningkatnya resiko keamanan terhadap sistem operasi. Oleh karena itu, sistem operasi harus memiliki ketahanan keamanan. Bagi kebanyakan pengembang sistem operasi saat ini, keamanan adalah salah satu permasalahan utama.

5.9. Rangkuman

Sistem operasi memiliki beberapa komponen, seperti manajemen proses, manajemen memori utama, manajemen sistem berkas, manajemen sistem M/K, manajemen penyimpanan sekunder, proteksi dan keamanan, dan antarmuka. Semua komponen tersebut saling berkaitan satu sama lain. Sebuah sistem operasi tidak dapat bekerja apabila salah satu saja dari komponen-komponen tersebut hilang.

Memahami komponen-komponen sistem operasi dalam bab ini akan memudahkan pemahaman tentang sistem operasi dalam bab-bab selanjutnya dalam buku ini. Dalam bab-bab selanjutnya, hanya beberapa komponen saja yang akan dibahas lebih lanjut, yaitu manajemen proses, manajemen memori utama, manajemen sistem berkas, dan manajemen sistem M/K.

Rujukan

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

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

[WikiOS2007] Wikipedia, The Free Encyclopedia. 2007 . *Operating System – http:// en.wikipedia.org/ wiki/ Operating_system*. Diakses 8 Februari 2007.

[WikiGUI2007] Wikipedia, The Free Encyclopedia. 2007 . *Graphical User Interface – http:// en.wikipedia.org / wiki/ GUI*. Diakses 13 Februari 2007.

[WikiGUIHistory2007] Wikipedia, The Free Encyclopedia. 2007 . *History of the graphical user interface – http:// en.wikipedia.org/ wiki/ History_of_the_ graphical_user _interface*. Diakses 13 Februari 2007.

[WikiCLI2007] Wikipedia, The Free Encyclopedia. 2007 . *Command Line Interface – http:// en.wikipedia.org/ wiki/Command_line_ interface*. Diakses 13 Februari 2007.

[WikiSpooling2007] Wikipedia, The Free Encyclopedia. 2007 . *Spooling – http:// en.wikipedia.org/ wiki/ Spooling*. Diakses 7 Juni 2007.

[McGillOS2007] McGill University. 2007 . *Operating System – http:// www.cs.mcgill. ca/~cs310/ lect_notes/ cs310_lecture02 .pdf*. Diakses 10 Februari 2007.

[MattBlaze2004] Matt Blaze. 2004 . *Operating System – http:// www.crypto.com/ courses/fall04/ cse380/20040921. pdf*. Diakses 6 Juni 2007.

Bab 6. Layanan dan Antarmuka

6.1. Pendahuluan

Di bab sebelumnya telah dibahas sistem operasi berdasarkan komponen-komponennya. Selanjutnya yang akan dibahas adalah bagaimana proses di dalam sistem komputer itu sendiri sehingga layanan-layanan tersebut dapat diimplementasikan ke user.

Kita akan membahas tentang *system call* sebagai sarana atau bentuk komunikasi di dalam sistem komputer, *system programs* dan juga *application programs* beserta contoh-contohnya. Pertama-tama, akan dijelaskan terlebih dahulu mengenai layanan-layanan sistem operasi, kali ini menurut sudut pandang proses di dalamnya.

6.2. Jenis Layanan

Seperti yang telah kita ketahui bersama, tujuan dari sebuah sistem operasi adalah sebagai penghubung antara *user* dan *hardware*, dimana sistem operasi memberikan kemudahan-kemudahan agar *user* tidak harus mengakses *hardware* secara langsung dalam bahasa mesin, tetapi dalam bentuk layanan-layanan yang diberikan oleh sistem operasi.

Berikut ini adalah kategori-kategori layanan yang diberikan oleh sistem operasi:

- **Antarmuka.** Sistem operasi menyediakan berbagai fasilitas yang membantu *programmer* dalam membuat program seperti *editor*. Walaupun bukan bagian dari sistem operasi, tapi layanan ini diakses melalui sistem operasi.
- **Eksekusi Program.** Sistem harus bisa me- *load* program ke memori, dan menjalankan program tersebut. Program harus bisa menghentikan pengeksekusian baik secara normal maupun tidak (ada *error*).
- **Operasi Masukan/Keluaran.** Program yang sedang dijalankan kadang kala membutuhkan Masukan/Keluaran. Untuk efisiensi dan keamanan, pengguna biasanya tidak bisa mengatur piranti masukan/keluaran secara langsung, untuk itulah sistem operasi harus menyediakan mekanisme dalam melakukan operasi masukan/keluaran.
- **Manipulasi Sistem Berkas.** Program harus membaca dan menulis berkas, dan kadang kala juga harus membuat dan menghapus berkas.
- **Komunikasi.** Kadang kala sebuah proses memerlukan informasi dari proses 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.
- **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 mempertahanan 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, terdapat juga layanan-layanan lain yang bertujuan untuk mempertahankan efisiensi sistem itu sendiri. Layanan tambahan itu yaitu:

- **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.
- **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.

- **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 mengindentifikasi pengguna. Ini bisa dilakukan dengan meminta *password* bila ingin menggunakan sumber daya.

6.3. Antarmuka

Pengertian antarmuka (*interface*) adalah salah satu layanan yang disediakan sistem operasi sebagai sarana interaksi antara pengguna dengan sistem operasi. Antarmuka adalah komponen sistem operasi yang bersentuhan langsung dengan pengguna. Terdapat dua jenis antarmuka, yaitu *Command Line Interface*(CLI) dan *Graphical User Interface*(GUI).

Command Line Interface(CLI)

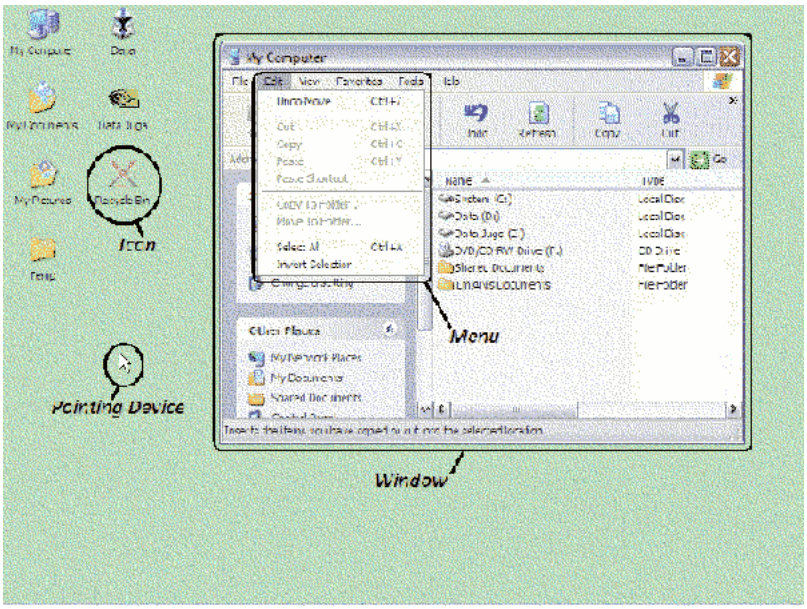
CLI adalah tipe antarmuka dimana pengguna berinteraksi dengan sistem operasi melalui *text-terminal*. Pengguna menjalankan perintah dan program di sistem operasi tersebut dengan cara mengetikkan baris-baris tertentu.

Meskipun konsepnya sama, tiap-tiap sistem operasi memiliki nama atau istilah yang berbeda untuk CLI-nya. UNIX memberi nama CLI-nya sebagai bash, ash, ksh, dan lain sebagainya. Microsoft Disk Operating System (MS-DOS) memberi nama command.com atau Command Prompt. Sedangkan pada Windows Vista, Microsoft menamakannya PowerShell. Pengguna Linux mengenal CLI pada Linux sebagai *terminal*, sedangkan pada Apple namanya adalah commandshell.

Graphical User Interface(GUI)

GUI adalah tipe antarmuka yang digunakan oleh pengguna untuk berinteraksi dengan sistem operasi melalui gambar-gambar grafik, ikon, menu, dan menggunakan perangkat penunjuk (*pointing device*) seperti *mouse* atau *track ball*. Elemen-elemen utama dari GUI bisa diringkas dalam konsep WIMP (*window, icon, menu, pointing device*).

Gambar 6.1. Contoh GUI



Pengguna komputer yang awam seringkali menilai sebuah sistem operasi dari GUI-nya. Sebuah sistem operasi dianggap bagus jika tampilan luarnya (GUI-nya) bagus. Padahal, seperti telah dijelaskan sebelumnya, komponen sistem operasi tidak hanya GUI, sehingga penilaian terhadap sebuah sistem

operasi tidak bisa hanya dari satu komponen saja. Karena GUI adalah kesan pertama pengguna dengan sistem operasi itu, setiap pengembang sistem operasi berlomba-lomba mengembangkan GUI-nya dengan keunggulannya masing-masing.

Sejarah mencatat bahwa Xerox PARC (Palo Alto Research Center) yang pertama kali meriset tentang GUI. Pada tahun 1984, Apple merilis Macintosh yang menggunakan GUI hasil riset Xerox PARC. Beberapa tahun kemudian, Microsoft merilis sistem operasi Windows-nya yang juga menggunakan GUI. Apple mengklaim bahwa Microsoft mencuri ide dari Apple.

Seperti halnya CLI, tiap-tiap sistem operasi juga memiliki nama tersendiri untuk komponen GUI-nya. Pada Apple Mac OS X, GUI-nya disebutAqua. Microsoft memberi nama GUI Windows XP sebagai Lunar dan GUI Windows Vista sebagai Aero. Pada Linux, ada dua pengembang utama *desktop environment* pada Linux, yang masing-masing menghasilkan produk KDE (K *Desktop Environment*) dan GNOME. KDE digunakan pada beberapa distro seperti SuSE dan Mandrake, sedangkan GNOME dipakai pada beberapa distro seperti Fedora Core dan Ubuntu.

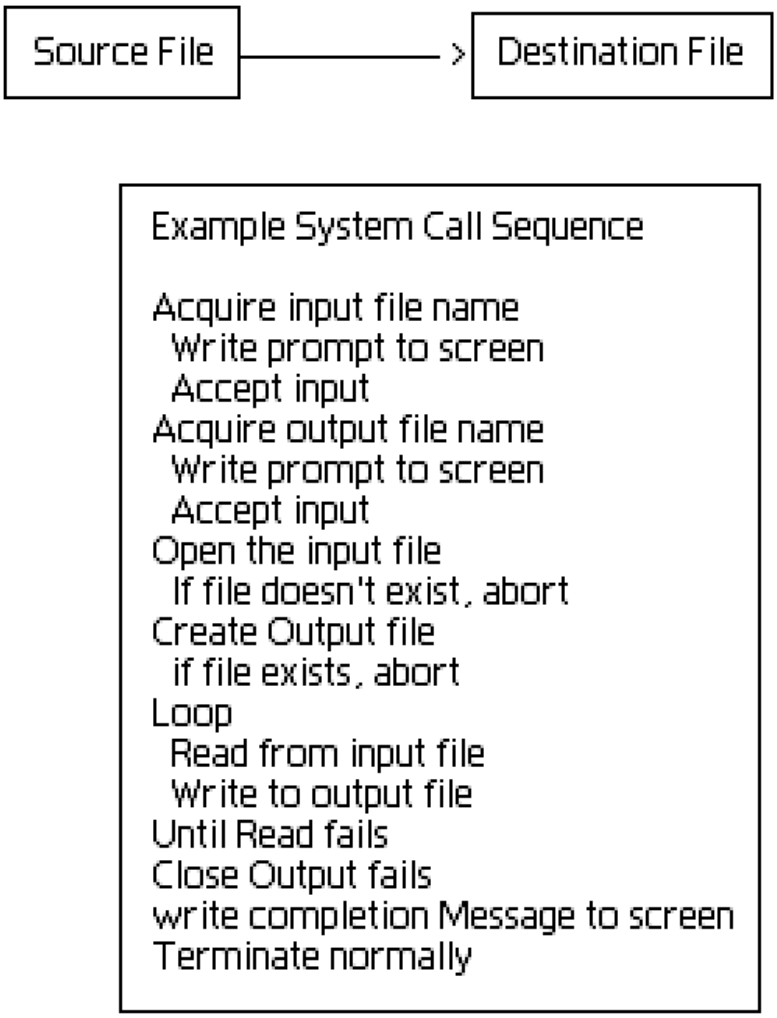
6.4. *System Calls*

Komputer digunakan untuk melakukan suatu proses yang dikehendaki *user*. Oleh karena itu harus ada suatu bentuk komunikasi antara *user* dan *hardware*. Komunikasi itu terjadi dalam bentuk *system calls*. SO melalui *shell*-nya akan menangkap perintah dari *user* yang kemudian akan dikomunikasikan melalui *system calls*. Disinilah peran SO sebagai jembatan komunikasi antara *user* dan *hardware* itu terjadi. *System calls* itu sendiri umumnya ditulis dalam bahasa C dan C++.

Mengenai *shell*, *shell* itu sendiri secara umum adalah *layer* yang berfungsi sebagai *interface* antara user dan inti dalam sistem operasi (kernel). Melalui *shell*, user dapat memberi perintah-perintah yang akan dikirim ke sistem operasi, sehingga *shell* ini merupakan *layer* yang menerima interaksi dari user secara langsung. *Shell* dalam SO secara umum dibagi menjadi 2, *Command Line*(CLI) dan *Graphical*(GUI). Jadi dengan kata lain, *system calls* berperan sebagai *interface* dalam layanan-layanan yang disediakan oleh sistem operasi.

Untuk lebih jelasnya lihat gambar berikut. Contoh di atas adalah *sytem calls* di dalam program yang membaca data dari satu *file* lalu meng- *copy*-nya ke *file* lain.

Gambar 6.2. Contoh System Call



6.5. API (Application Program Interface)

Dalam contoh program sederhana di atas, dibutuhkan setidaknya ribuan *system calls* per detik. Oleh karena itu Kebanyakan *programmer* membuat aplikasi dengan menggunakan *Application Programming Interface*(API). Dalam API itu terdapat fungsi-fungsi/perintah-perintah untuk menggantikan bahasa yang digunakan dalam *system calls* dengan bahasa yang lebih terstruktur dan mudah dimengerti oleh *programmer*. Fungsi yang dibuat dengan menggunakan API tersebut kemudian akan memanggil *system calls* sesuai dengan sistem operasinya. Tidak tertutup kemungkinan nama dari *system calls* sama dengan nama di API.

Keuntungan memprogram dengan menggunakan API adalah:

- **Portabilitas.** Programmer yang menggunakan API dapat menjalankan programnya dalam sistem operasi mana saja asalkan sudah ter- *install* API tersebut. Sedangkan *system call* berbeda antar sistem operasi, dengan catatan dalam implementasinya mungkin saja berbeda.
- **Lebih Mudah Dimengerti.** API menggunakan bahasa yang lebih terstruktur dan mudah dimengerti daripada bahasa *system call*. Hal ini sangat penting dalam hal editing dan pengembangan.

System call interface ini berfungsi sebagai penghubung antara API dan *system call* yang dimengerti oleh sistem operasi. *System call interface* ini akan menerjemahkan perintah dalam API dan kemudian akan memanggil *system calls* yang diperlukan.

Untuk membuka suatu *file* tersebut *user* menggunakan program yang telah dibuat dengan menggunakan bantuan API, maka perintah dari *user* tersebut diterjemahkan dulu oleh program

menjadi perintah *open()*. Perintah *open()* ini merupakan perintah dari API dan bukan perintah yang langsung dimengerti oleh kernel sistem operasi. Oleh karena itu, agar keinginan *user* dapat dimengerti oleh sistem operasi, maka perintah *open()* tadi diterjemahkan ke dalam bentuk *system call* oleh *system call interface*. Implementasi perintah *open()* tadi bisa bermacam-macam tergantung dari sistem operasi yang kita gunakan.

6.6. Jenis *System Calls*

Berikut ini adalah tipe system call:

- **Manajemen Proses.** *System call* untuk manajemen proses diperlukan untuk mengatur proses-proses yang sedang berjalan. Kita dapat melihat penggunaan *system calls* untuk manajemen proses pada Sistem Operasi Unix. Contoh yang paling baik untuk melihat bagaimana *system call* bekerja untuk manajemen proses adalah *Fork*. *Fork* adalah satu satunya cara untuk membuat sebuah proses baru pada sistem Unix.
- **Manajemen Berkas.** System calls yang berhubungan dengan berkas sangat diperlukan. Seperti ketika kita ingin membuat atau menghapus suatu berkas, atau ketika ingin membuka atau menutup suatu berkas yang telah ada, membaca berkas tersebut, dan menulis berkas itu. *System calls* juga diperlukan ketika kita ingin mengetahui atribut dari suatu berkas atau ketika kita juga ingin merubah atribut tersebut. Yang termasuk atribut berkas adalah nama berkas, jenis berkas, dan lain-lain. Ada juga *system calls* yang menyediakan mekanisme lain yang berhubungan dengan direktori atau sistem berkas secara keseluruhan. Jadi bukan hanya berhubungan dengan satu spesifik berkas. Contohnya membuat atau menghapus suatu direktori, dan lain-lain.
- **Manajemen Piranti.** Program yang sedang dijalankan kadang kala memerlukan tambahan sumber daya. Jika banyak pengguna yang menggunakan sistem dan memerlukan tambahan sumber daya maka harus meminta peranti terlebih dahulu. Lalu setelah selesai, penggunaannnya harus dilepaskan kembali dan ketika sebuah peranti telah diminta dan dialokasikan maka peranti tersebut bisa dibaca, ditulis, atau direposisi.
- **System Call Informasi/Pemeliharaan.** Beberapa *system calls* disediakan untuk membantu pertukaran informasi antara pengguna dan sistem operasi, contohnya adalah *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 informasi tersebut.
- **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. 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.

6.7. *System Programs*

Seperti yang sudah kita pelajari di bab-bab awal bahwa terdapat empat komponen utama dalam sistem komputer, apabila kita jabarkan, dari yang paling bawah adalah perangkat keras (*Hardware*), lalu di atasnya adalah sistem operasi, kemudian di atasnya dimana yang berhubungan langsung dengan para pengguna adalah sistem program dan program aplikasi. Di dalam sistem komputer, sistem program berguna untuk menyediakan kemudahan-kemudahan bagi pengembangan progran serta eksekusi. sistem program yang sering kita gunakan contohnya adalah *format* dan *login*. Sistem program dibagi dalam beberapa kategori yaitu:

- **Manajemen/manipulasi Berkas.** Membuat, menghapus, menyalin, mengganti nama, mencetak, memanipulasi berkas dan direktori.
- **Informasi Status.** Beberapa program meminta informasi tentang tanggal, jam, jumlah memori dan disk yang tersedia, jumlah pengguna dan informasi yang sejenis.

- **Modifikasi Berkas.** Membuat berkas dan memodifikasi isi berkas yang disimpan pada *disk* atau *tape*.
- **Pendukung Bahasa Pemograman.** Kadang kala kompilator, *assembler*, *interpreter*, dari bahasa pemograman diberikan kepada pengguna dengan bantuan sistem operasi.
- **Loading dan Eksekusi Program.** Ketika program di *assembly* atau dikompilasi, program tersebut harus di *load* ke dalam memori untuk dieksekusi. Untuk itu sistem harus menyediakan *absolute loaders*, *relocatable loaders*, *linkage editors*, dan *overlay loaders*.
- **Komunikasi.** 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.

6.8. *Application Programs*

Program aplikasi atau yang juga sering disebut aplikasi adalah setiap program yang dirancang untuk melakukan fungsi yang khusus atau spesifik untuk pengguna atau, untuk kasus-kasus tertentu, untuk program aplikasi lainnya. Contoh-contoh dari program aplikasi meliputi *word processors*, *database programs*, *Web browsers*, *development tools*, *drawing*, *paint*, *image editing programs*, dan *communication programs*. Dalam menjalankan tugas-tugasnya program aplikasi menggunakan layanan-layanan sistem operasi komputer dan program-program penunjang lainnya. Seperti yang sudah dibahas.dalam subbab 3.1 (*Aplication Program Interface*) bahwa para *programmer* menggunakan API untuk memudahkan mereka dalam membuat program aplikasi.

6.9. Rangkuman

Proses-proses yang terjadi di dalam sistem komputer dimulai dari pengguna komputer yang menggunakan program aplikasi dan sistem program, dimana nanti perintah-perintah dari user dibawa oleh sistem call ke kernel yang berada pada sistem operasi untuk diterjemahkan ke dalam bahasa mesin.

Rujukan

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

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

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

Bab 7. Struktur Sistem Operasi

7.1. Pendahuluan

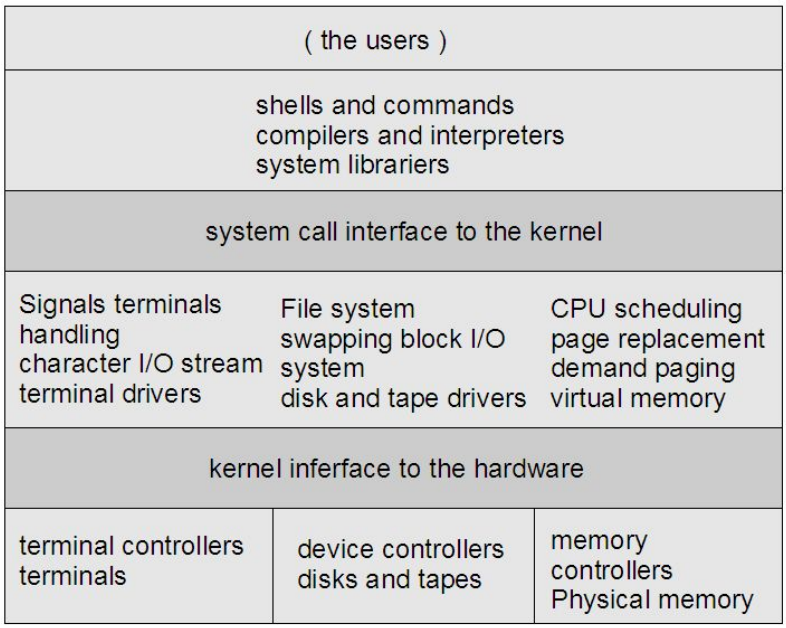
Sistem operasi modern adalah suatu sistem yang besar dan kompleks. Dan tentu saja proses mendesain sistem operasi bukanlah pekerjaan mudah. Karena itu, didalam desain sistem operasi digunakan suatu struktur agar sistem tersebut bisa dipelajari dengan mudah. digunakan, dan dikembangkan ebih lanjut. Jika pada bab sebelumnya, kita memandang sistem operasi dari luar, yaitu dengan *system call* yang bisa digunakan, maka dalam bab ini kita akan melihat dari dalam, yaitu bagaimana sistem operasi disusun. Ternyata, ada beberapa pendekatan/model yang digunakan, seperti struktur sederhana, struktur berlapis, dan mikro kernel.

7.2. Struktur Sederhana

Pada awalnya, sistem operasi dimulai sebagai sistem yang kecil, sederhana, dan terbatas. Lama kelamaan, sistem operasi semakin berkembang menjadi suatu sistem yang lebih besar dari awalnya. Dalam perkembangannya, ada sistem yang terstruktur dengan kurang baik, dan ada juga yang baik. Contoh sistem yang terstruktur kurang baik adalah MS-DOS. Sistem operasi ini dirancang sedemikian rupa agar mampu berjalan pada *hardware* yang terbatas. Memang memiliki struktur, tapi belum terbagi-bagi dalam modul-modul, dan *interface* serta fungsionalitas tidak begitu jelas batasannya.

Begitu pula dengan UNIX, yang pada awalnya juga terbatas oleh *hardware* yang ada. Sistem ini dapat dibagi menjadi dua bagian, yaitu kernel dan program sistem. Kernel sendiri dapat dibagi menjadi dua bagian, yaitu *device driver* dan *interface*, yang kemudian terus berkembang seiring dengan perkembangan UNIX. Berikut ini adalah skema struktur UNIX.

Gambar 7.1. Struktur UNIX



Versi-versi UNIX selanjutnya dirancang agar mampu bekerja dengan hardware yang lebih baik. Begitu pula dengan strukturnya, yang dibuat makin modular.

7.3. Struktur Berlapis

Rasanya susah membayangkan sekian banyak fungsi yang disediakan oleh sistem operasi diimplementasikan dalam satu program saja. Karena itu, lebih mudah untuk membaginya dalam sejumlah layer/lapisan. Tentu setiap lapisan memiliki fungsinya sendiri-sendiri, dan juga bisa menambah fungsi-fungsi lain, berdasarkan fungsi-fungsi yang tersedia pada lapisan-lapisan lain yang dibawahnya.

Lapisan-lapisan sistem operasi adalah suatu abstraksi dari enkapsulasi sekumpulan struktur data dalam sistem operasi. Lapisan-lapisan yang berada di atas bisa mengakses operasi-operasi yang tersedia di lapisan-lapisan bawahnya. Stallings memberi model yang lebih detail, sebagai berikut:

- **Lapisan 1.** Berisi berbagai sirkuit elektronik, misal register, *memory cells*, dan *logic gate*.
- **Lapisan 2.** Berisi instruksi prosesor, misal instruksi aritmatika, instruksi transfer data, dsb.
- **Lapisan 3.** Penambahan konsep seperti prosedur/subrutin, maupun fungsi yang me-return nilai tertentu.
- **Lapisan 4.** Penambahan interrupt.
- **Lapisan 5.** Program sebagai sekumpulan instruksi yang dijalankan oleh prosesor.
- **Lapisan 6.** Berhubungan dengan *secondary storage device*, yaitu membaca/menulis *head*, *track*, dan *sektor*.
- **Lapisan 7.** Menciptakan alamat logika untuk proses. Mengatur hubungan antara *main memory*, *virtual memory*, dan *secondary memory*.
- **Lapisan 8.** Program sebagai sekumpulan instruksi yang dijalankan oleh prosesor.
- **Lapisan 9.** Berhubungan dengan secondary storage device, yaitu membaca/menulis head,track, dan sektor.
- **Lapisan 10.** Menciptakan alamat logika untuk proses. Mengatur hubungan antara *main memory*, *virtual memory*, dan *secondary memory*.
- **Lapisan 11.** Program sebagai sekumpulan instruksi yang dijalankan oleh prosesor.
- **Lapisan 12.** File adalah objek yang memiliki nama dan ukuran. Abstraksi dari lapisan 9.
- **Lapisan 13.** Menyediakan *interface* agar bisa berinteraksi dengan pengguna.

Lapisan-lapisan dari 1-4 bukanlah bagian dari sistem operasi dan masih menjadi bagian dari prosesor secara ekslusif.

Lapisan ke-5 hingga ke-7, sistem operasi sudah berhubungan dengan prosesor. Selanjutnya dari lapisan ke-8 hingga 13, sistem operasi berhubungan dengan media penyimpanan maupun perlatan-peralatan lain yang ditancapkan, misalnya peralatan jaringan.

7.4. Mikro Kernel

Pada pembahasan "Struktur Sederhana", sempat disinggung istilah "kernel". Apakah kernel itu? Kernel adalah komponen sentral dari sistem operasi. Ia mengatur hal-hal seperti *interrupt handler*(untuk menyediakan layanan interupsi), *process scheduler*(membagi-bagi proses dalam prosesor), *memory management*, *I/O*, dan sebagainya. Atau dengan kata lain, ia adalah jembatan antara *hardware* dengan *software*.

Cara tradisional untuk membangun sistem operasi adalah dengan membuat kernel monolitis, yaitu semua fungsi disediakan oleh kernel, dan ini menjadikan kernel suatu program yang besar dan kompleks.

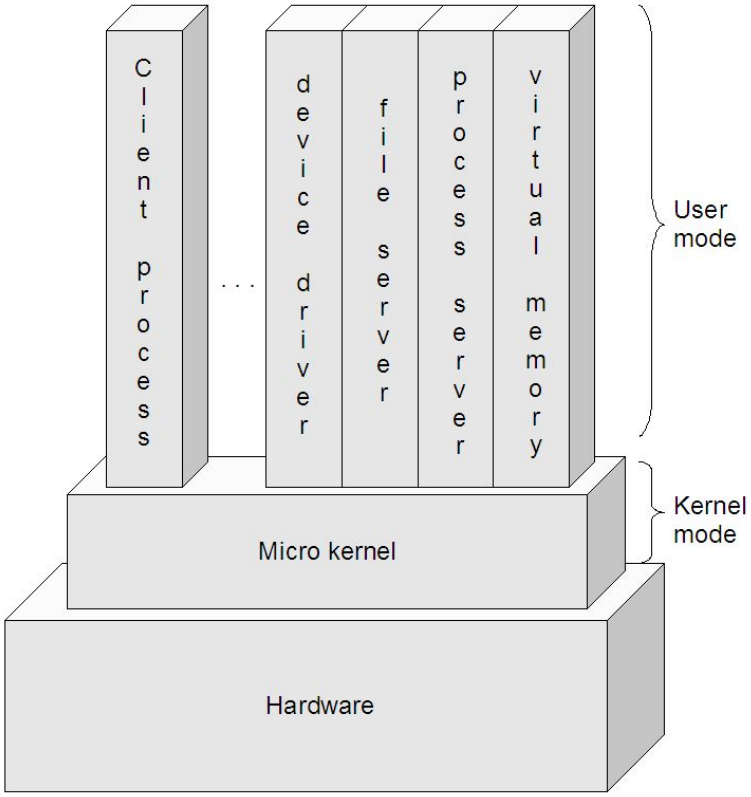
Cara yang lebih modern, adalah dengan menggunakan kernel mikro. Pada awalnya, konsep mikro kernel dikembangkan pada sistem operasi Mach. Ide dasar dari pengembangan kernel mikro adalah bahwa hanya fitur-fitur yang perlu saja yang diimplementasikan dalam kernel (mengenai fitur-fitur apa saja yang perlu diimplementasikan, ini bisa berbeda tergantung desain sistem operasi).

Walaupun garis pembatas mengenai apa saja yang berada di dalam dan luar kernel mikro bisa berbeda antara desain yang satu dengan yang lain, namun ada karakteristik yang umum, yaitu servis-servis yang

umumnya menjadi bagian sistem operasi menjadi subsistem eksternal yang bisa berinteraksi satu sama lain dan dengan kernel tentunya. Ini mencakup *device driver*, *file system*, *virtual memory manager*, *windowing system*, *dan security devices*. Pendekatan kernel mikro menggantikan pendekatan berlapis yang vertikal tradisional.

Komponen-komponen sistem operasi yang berada di luar kernel mikro diimplementasikan sebagai *server process* dan berkomunikasi dengan *message passing* via kernel mikro. Misalnya jika user ingin membuat berkas baru, dia mengirim pesan ke *file system server*, atau jika ingin membuat proses baru, dia mengirimkan pesan ke *process server*.

Gambar 7.2. Struktur kernel mikro



Beberapa kelebihan kernel mikro:

a. ***Interface* yang seragam.** Proses tidak lagi dibedakan, baik antara *kernel-level* maupun *user-level*, karena semuanya berkomunikasi via *message passing*.
b. ***Extensibility* .** Bisa menambahkan fitur-fitur baru tanpa perlu melakukan kompilasi ulang.
c. ***Flexibility* .** Fitur-fitur yang sudah ada bisa dikurangi, atau dimodifikasi sesuai dengan kebutuhan sehingga menjadi lebih efisien. Misalnya tidak semua pengguna membutuhkan security yang sangat ketat, atau kemampuan untuk melakukan *distributed computing*.
d. ***Portability* .** Pada kernel mikro, semua atau sebagian besar kode yang prosesor-spesifik berada di dalamnya. Jadi, proses porting ke prosesor lain bisa dilakukan dengan relatif sedikit usaha. Pada kelompok desktop misalnya, tampaknya dominasi Intel makin kuat. Tapi, sampai seberapa lama itu bisa bertahan? Karena itulah, *portability* adalah salah satu isu yang sangat penting.
e. ***Reliability* .** Semakin besar suatu *software*, maka tentulah semakin sulit untuk menjamin *reliability*-nya. Desain dengan pendekatan berlapis sangatlah membantu, dan dengan pendekatan kernel mikro bisa lebih lagi. Kernel mikro dapat dites secara ekstensif .Karena dia menggunakan API yang sedikit, maka bisa meningkatkan kualitas code di luar kernel.
f. ***Support for object-oriendted OS* .** Model kernel mikro sangat sesuai untuk mengembangkan sistem operasi yang berbasis *object-oriented*. Contoh sistem operasi yang menggunakan kernel mikro adalah MacOS X dan QNX.

7.5. Proses *Boot*

Booting adalah istilah untuk menghidupkan komputer. Secara umum, gambaran yang terjadi pada proses boot adalah sebagai berikut.

- Saat komputer dihidupkan, memorinya masih kosong. Belum ada instruksi yang dapat dieksekusi oleh prosesor. Karena itu, prosesor dirancang untuk selalu mencari alamat tertentu di BIOS ROM. Pada alamat tersebut, terdapat sebuah instruksi jump yang menuju ke alamat eksekusi awal BIOS. Setelah itu, prosesor menjalankan *power-on-self test*(POST), yaitu memeriksa kondisi hardware yang ada.
- Sesudah itu, BIOS mencari *video card*. Secara khusus, dia mencari program BIOS milik video card. Kemudian *system BIOS* menjalankan video card BIOS. Barulah setelah itu, video card diinisalisasi.
- Kemudian BIOS memeriksa ROM pada hardware yang lain, apakah memiliki BIOS tersendiri apakah tidak. Jika ya, maka akan dieksekusi juga.
- BIOS melakukan pemeriksaan lagi, misal memeriksa besar memori dan jenis memori. Lebih lanjut lagi, dia memeriksa hardware yang lain, seperti disk. Lalu dia mencari disk dimana proses boot bisa dilakukan, yaitu mencari *boot sector*. *Boot sector ini* bisa berada di *hard disk*, atau *floppy disk*.

7.6. Kompilasi Kernel

Pada dasarnya Linux hanyalah sebuah kernel. Sedangkan program-program lain seperti teks editor, *browser*, kompilator, dan seterusnya melengkapi kernel menjadi suatu paket sistem operasi. Tentunya agar kernel dapat bekerja dengan optimal, perlu dilakukan konfigurasi sesuai dengan *hardware* yang ada. Biasanya, kompilasi kernel dilakukan saat hendak menambahkan *device* baru yang belum dikenali sebelumnya atau jika hendak mengaktifkan fitur tertentu pada sistem operasi. Pada proses kompilasi kernel, sangat mungkin terjadi kesalahan. Karena itu, jangan lupa membackup kernel yang lama, dan menyiapkan *emergency boot disk*. Pada penjelasan berikut, diasumsikan kernel yang digunakan adalah versi 2.6.20 dan komputer menggunakan prosesor Intel.

Beberapa tahapan dalam kompilasi kernel:

a. **Mendownload kernel.** Ada banyak situs di internet tempat mendownload kernel. Tapi ada baiknya jika anda mengunjungi situs resminya, yaitu "kernel.org". Anda bisa melihat beraneka versi kernel dan patchnya disana.Format penamaan kernel Linux adalah linux-X.YY.ZZ.tar.gz atau linux-X.YY.ZZ.tar.bz2, dimana: X = major number; Y = minor number; ZZ = revision number. Contoh: linux-2.6.20. Angka 2 adalah major number (angka 2 ini jarang berubah dan baru berubah jika sudah terjadi perubahan besar) Angka 6 adalah minor number (karena 6 adalah bilangan genap, berarti kernel ini versi stabil) Angka 20 menunjukkan nomor revisi.
b. **Mengekstrak kernel.** Source code kernel Anda yang lama bisa dilihat di direktori /usr/src/linux. Supaya source code kernel sebelumnya tidak hilang, ekstraklah kernel yang baru di direktori yang berbeda, misal /usr/src/linux-2.6.20 (tentunya sesuaikan angka-angka tersebut dengan versi kernel yang anda pakai).
c. **Buat *symbolic link*.**

```
ln -s linux-2.6.20 linux
```

d. **Konfigurasi kernel.** Sebelum proses kompilasi, anda memiliki 2 pilihan, yaitu membuat konfigurasi baru, atau menggunakan konfigurasi kernel sebelumnya. Jika anda ingin membuat konfigurasi baru, maka jalankan perintah: make xconfig (atau make menuconfig). Pada tahap ini, anda mengkonfigurasi kernel sesuai dengan hardware yang ada di komputer anda. Isinya antara lain mengatur jenis prosesor, memory, networking, USB, dsb. Dengan ini, kernel bisa bekerja optimal pada hardware yang ada Setelah berkas configurasi (.config) terbentuk, anda bisa memulai proses kompilasi. Sedangkan jika anda ingin menggunakan konfigurasi kernel yang lama, anda bisa mengcopy berkas .config dari direktori kernel yang lama ke direktori kernel yang baru, lalu menjalankan perintah: make oldconfig
e. **Kompilasi.** Jalankan perintah "make bzImage". Proses kompilasi kernel bisa memakan waktu cukup lama, dan sangat mungkin terjadi kesalahan disitu. Jika ada kesalahan, coba lakukan lagi konfigurasi kernel. Setelah itu, coba lakukan kompilasi lagi. Jika sukses, terbentuk berkas bzImage di/usr/src/linux-2.6.20/arch/i386/boot. *Copy* ke direktori /boot dengan perintah:

```
cp arch/i386/boot/bzImage /boot/vmlinuz-2.6.20
```

Selanjutnya, kompilasi modul. Jalankan perintah:

```
make modules
```

diikuti oleh

```
make modules_install
```

Terbentuk berkas System.map. Copylah ke /boot dengan perintah:

```
cp System.map /boot/System.map-2.6.20
```

Supaya kernel yang baru bisa digunakan, ubahlah konfigurasi bootloader anda supaya ia mengetahui dimana kernel yang baru berada. Misal jika anda menggunakan Lilo, modifikasi berkas lilo.conf (ada di /etc), atau jika anda menggunakan grub, modifikasi berkas menu.lst (ada di /boot/grub). Khusus jika anda menggunakan lilo, jalankan perintah lilo. Setelah itu, reboot komputer anda.

7.7. Komputer Meja

Sesuai dengan namanya, komputer meja memang komputer yang dirancang digunakan di meja. Komputer seperti ini mulai populer di tahun 1970an. Sebelum itu, jenis komputer yang populer adalah mainframe. Saat ini, komputer meja adalah komputer yang harganya relatif paling terjangkau dan begitu banyak dijumpai di rumah-rumah,sekolah-sekolah,kantor-kantor. Selain harganya yang relatif terjangkau, hal lain yang membuat komputer meja populer adalah komponennya bisa di- *upgrade* dengan mudah.

Secara umum, komponen komputer meja adalah:

- **Fan.** Untuk mendinginkan komputer
- **Motherboard.** Untuk mengintegrasikan komponen-komponen komputer yang ada.
- **Hard disk.** Tempat penyimpanan data.
- **Optical disc drive.** Untuk membaca kepingan CD/DVD.
- **Floppy disk drive.** Untuk membaca *floppy disk*.
- **Prosesor.** Untuk mengeksekusi program.
- **CPU cooler.** Untuk mendinginkan prosesor
- **RAM.** Menyimpan program yang sedang berjalan atau untuk transfer data.
- **Sound card.** Memproses suara dari prosesor, kemudian dikeluarkan melalui speaker.
- **Modem.** Memproses sinyal informasi, misal untuk akses internet.
- **Network card.** Memungkinkan komputer berkomunikasi dengan komputer lain dalam jaringan.

7.8. Sistem Prosesor Jamak

Secara tradisional, komputer dipandang sebagai suatu mesin sekuensial, yaitu mereka menjalankan sekumpulan instruksi yang tersusun dalam urutan tertentu. Prosesor menjalankan program dengan cara mengeksekusi instruksi mesin satu demi satu dalam suatu waktu. Tapi tentunya ini tidak selalu benar. Dengan *pipelining* misalnya, prosesor tidak perlu menunggu suatu instruksi selesai dan bisa mengerjakan instruksi lainnya. Seiring dengan perkembangan teknologi, para perancang komputer terus berusaha mengembangkan teknik untuk meningkatkan performa dan tentu juga *reliability*. Salah satunya adalah *multiprocessing*, yaitu menggunakan prosesor jamak.

Menurut Silberschatsz dkk, keuntungan sistem prosesor jamak adalah:

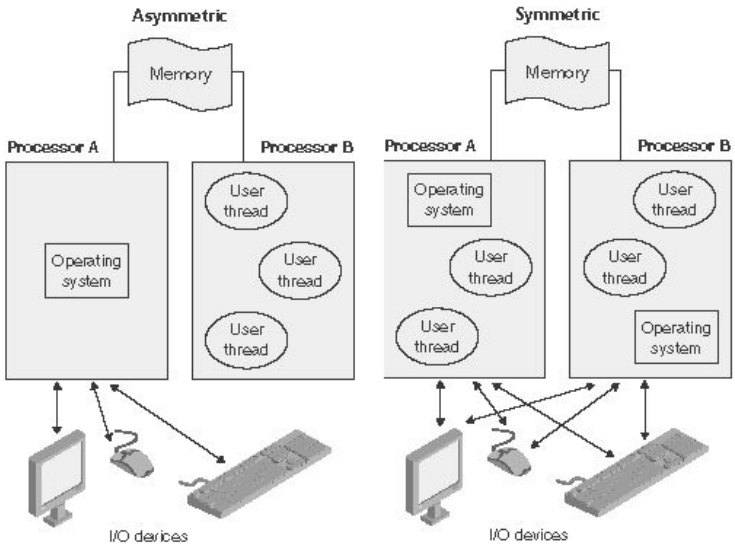
a. **Peningkatan *throughput*.** Karena lebih banyak proses/thread yang dapat dijalankan. Ini bukan berarti kemampuan komputasi bertambah seiring dengan bertambahnya jumlah prosesor. Yang meningkat adalah peningkatan jumlah pekerjaan yang dapat dilakukan dalam waktu tertentu.
b. **Lebih ekonomis.** Daripada sistem dengan banyak prosesor tunggal, karena bisa berbagi memori, storage, dan power supply. Misalnya jika beberapa program memproses data yang sama, maka adalah lebih murah untuk menyimpan data tersebut pada satu disk dan membaginya diantara prosesor-prosesor tersebut, daripada menggunakan banyak komputer dengan disk lokal yang berisi salinan data tersebut.
c. **Peningkatan kehandalan.** Jika pekerjaan terbagi rata, maka kegagalan salah satu prosesor bisa ditanggulangi oleh prosesor-prosesor yang lain. Memang performa menurun (menjadi lebih lambat), tetapi sistem tetap berjalan. Fenomena ini disebut *graceful gradation*. Sementara sistem yang memiliki sifat *graceful gradation* disebut sebagai sistem yang *fault-tolerant*.

Ada 2 model dalam sistem prosesor jamak, yaitu ASMP (Asymmetric Multi Processing) dan SMP (Symmetric Multi Processing). Pada model ASMP, ide dasarnya adalah master/slave, yaitu kernel selalu berjalan di prosesor tertentu, sedangkan prosesor-prosesor lainnya menjalankan utiliti yang ada di sistem operasi atau mengerjakan tugas-tugas tertentu. Prosesor master bertugas menjadwal proses atau thread. Ketika suatu proses/thread aktif, dan prosesor slave membutuhkan layanan (misal untuk I/O), maka dia harus mengirim permintaan ke prosesor master dan menunggu hingga permintaanya dilaksanakan. Model ini adalah sederhana, karena hanya satu prosesor yang mengatur sumber daya memori dan I/O.

Sayangnya, model ini memiliki beberapa kelemahan seperti:

- Kegagalan prosesor utama bisa menyebabkan kegagalan keseluruhan sistem.
- Bisa terjadi penururan performa, yaitu terjadi *bottleneck* di prosesor utama karena dialah yang bertanggung jawab atas penjadwalan dan manajenem proses.

Gambar 7.3. Model ASMP dan SMP



Kekurangan itulah menyebabkan model ASMP kurang disukai.

Model lainnya adalah SMP. Pada model ini, kernel bisa dijalankan di prosesor mana saja, dan tiap prosesor bisa melakukan penjadwalan proses/thread secara mandiri. Model seperti ini membuat desain sistem operasi menjadi lebih rumit, karena proses-proses bisa berjalan secara paralel. Karena itu, haruslah dijamin agar hanya 1 prosesor yang mengerjakan tugas tertentu dan proses-proses itu tidak mengalami *starvation*.

Keuntungan SMP:

a. ***Performance .*** Jika komputer yg menggunakan 1 prosesor bisa diatur sedemikian rupa sehingga sebagian pekerjaan bisa dilakukan secara paralel, maka komputer SMP bisa melakukannya dengan lebih baik lagi.
b. ***Availability .*** Karena semua prosesor menjalankan tugas yang sama, maka kegagalan pada salah satu prosesor tidak membuat sistem berhenti. Sistem tetap berjalan (fungsional), walaupun performa menurun.
c. ***Incremental growth .*** Performa bisa ditingkatkan dengan menambah prosesor lagi.

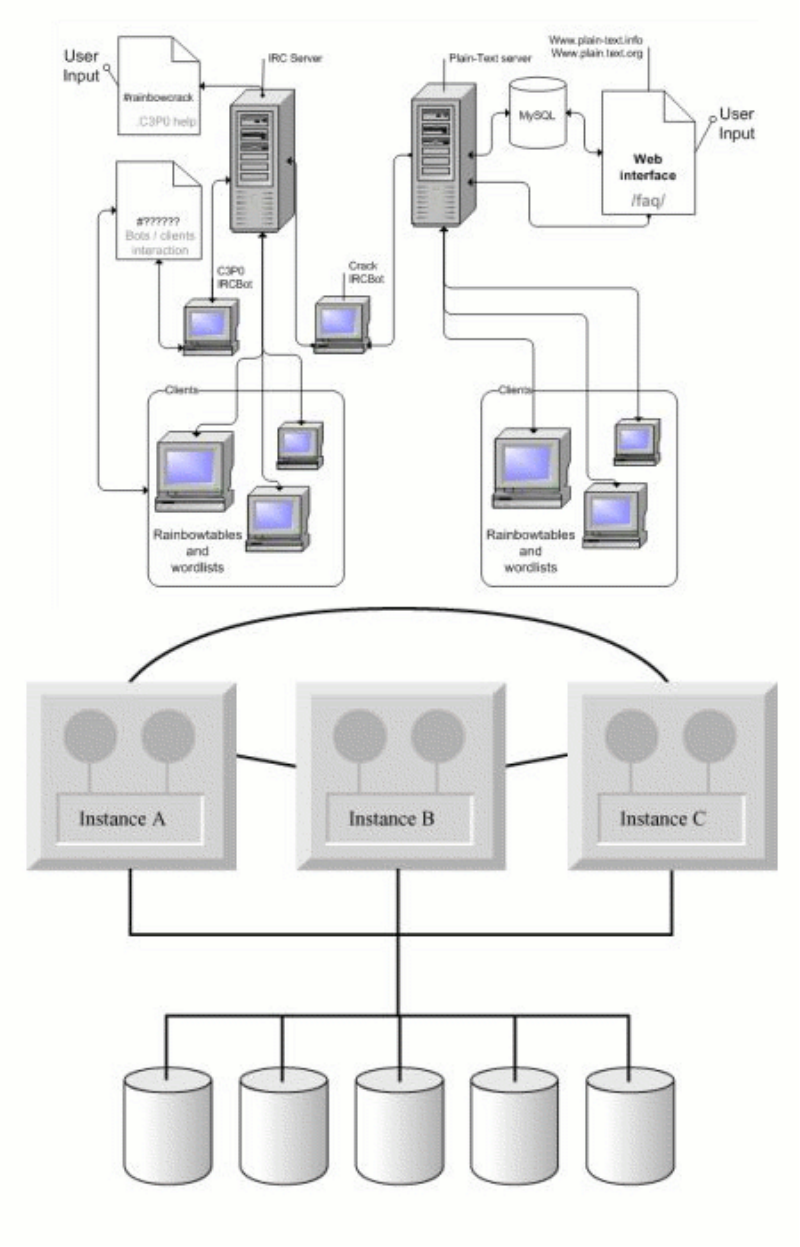
7.9. Sistem Terdistribusi dan Terkluster

Seiring dengan perkembangan teknologi komputer dan server, muncullah trend DDP (*distributed data processing*), yaitu prosesor, data, dan aspek-aspek lainnya bisa tersebar dalam lingkup tertentu. Sistem seperti ini melibatkan adanya pembagian proses komputasi, pengendali, dan interaksi dalam jaringan. Dalam perusahaan-perushaan besar misalnya, sering digunakan kombinasi antara komputer dan server. Komputer untuk menjalankan aplikasi-aplikasi seperti pengolah grafis, *word processing*, *spreadsheet*, sementara *server* sebagai *back-end* mengendalikan *database* dan sistem informasi perusahaan. Hal seperti ini adalah dampak dari perkembangan sistem terdistribusi. Tetapi, apakah sistem terdistribusi itu? Sistem terdistribusi adalah koleksi prosesor yang terhubung dalam jaringan serta tidak berbagi memori, yaitu memiliki memori masing-masing. Prosesor-prosesor itu bisa berkomunikasi melalui banyak cara, misalnya melalui jalur telepon atau high speed bus.

Keuntungan:

a. ***Resource sharing .*** Suatu komputer bisa mengakses sumber daya yang ada di komputer lain. Misalkan, komputer A bisa mengakses database yang ada di komputer B. Sebaliknya, komputer B bisa mencetak dokumen dengan menggunakan printer yang terpasang di komputer A.
b. ***Computation speedup .*** Jika suatu proses komputasi bisa dipecah-pecah menjadi sejumlah bagian yang berjalan secara konkuren, dalam sistem terdistribusi bagian-bagian komputasi ini bisa terbagi dalam komputer-komputer yang ada. Inilah yang menimbulkan adanya speedup. Lebih jauh lagi, bisa terjadi load sharing, yaitu jika suatu komputer mengerjakan tugas terlalu banyak, sebagian dari tugasnya itu bisa dialihkan ke komputer lain.
c. ***Reliability .*** Jika satu komputer mengalami kegagalan, maka secara keseluruhan sistem masih tetap dapat berjalan. Contoh: jika sistem terdiri atas komputer-komputer yang tersusun secara independen, kegagalan salah satu komputer seharusnya tidak mempengaruhi keseluruhan sistem. Tapi jika sistem terdiri atas komputer-komputer yang mengatur tugas spesifik seperti terminal I/O atau filesystem, maka kerusakan satu komputer saja bisa menyebabkan keseluruhan sistem mati. Tentunya, perlu mekanisme untuk mendeteksi kegagalan seperti ini, sehingga jika ada komputer yang rusak, sumber daya yang ada padanya tidak digunakan dan sebagai gantinya komputer yang lain bisa menangani itu.
d. ***Communication .*** Karena satu komputer terhubung dengan komputer-komputer laiinya, sangat dimungkinkan terjadi pertukaran informasi. Dengan adanya message passing, fungsi fungsi yang ada di suatu komputer misal file transfer, login, web browsing, bisa diperluas dalam sistem terdistribusi. Ini menyebabkan fungsi-fungsi ini bisa diakses secara jarak jauh. Misalnya, sejumlah orang yang terlibat dalam satu proyek, walaupun terpisah secara geografis, tetap bisa berkolaborasi dalam proyek itu. Dalam dunia industri, terjadi *downsizing*. Downsizing adalah mengganti *mainframe* dengan komputer atau *workstation* yang terhubung via jaringan. Dengan itu, mereka bisa mendapatkan fungsionalitas yang sesuai dengan biaya, kemudahan mengatur sumber daya, kemudahan maintenance/perawatan, dan lain-lain.

Gambar 7.4. Sistem Terdistribusi dan Terkluster



Disamping memiliki beberapa keuntungan, sistem terdistribusi juga memiliki beberapa kelemahan, misalnya:

- Jika tidak direncanakan dengan tepat, sistem terdistribusi bisa menurunkan proses komputasi, misalnya jika kegagalan salah satu komputer mempengaruhi komputer-komputer yang lain.
- Troubleshooting menjadi lebih rumit, karena bisa memerlukan koneksi ke komputer lain yang terhubung secara remote, atau menganalisis komunikasi antar komputer.
- Tidak semua proses komputasi cocok untuk dilakukan dalam sistem terdistribusi, karena besarnya keperluan komunikasi dan sinkronisasi antar komputer. Jika *bandwith*, *latency*, atau kebutuhan komunikasi terlalu besar, maka performanya bisa menjadi lebih jelek daripada sistem yang tidak terdistribusi sama sekali. Karena itu, lebih baik komputasi dilakukan di sistem yang tidak terdistribusi.

Salah satu topik yang sedang hangat dibicarakan dalam dunia komputer adalah sistem terkluster. Sistem terkluster menjadi alternatif SMP untuk memperoleh performa dan ketersediaan yang tinggi. Saat ini, sistem terkluster populer untuk aplikasi-aplikasi server. Sistem terkluster pada dasarnya

adalah sekumpulan komputer independen (bisa berjalan sendiri) yang terhubung satu sama lain untuk menyatukan sumber daya yang ada sehingga seolah-olah menjadi satu komputer saja.

Keuntungan:

a. ***Absolute scalability .*** Adalah mungkin untuk menciptakan sistem terkluster yang jauh lebih *powerful* daripada satu komputer *standalone* yang terbesar sekalipun. Satu kluster bisa terdiri atas puluhan, bahkan ratusan komputer, dan masing-masing adalah multiprosesor.
b. ***Incremental scalability .*** Kluster diatur sedemikian rupa sehingga bisa dupgrade sedikit demi sedikit sesuai dengan kebutuhan, tanpa harus mengupgrade keseluruhan sistem sekaligus secara besar-besaran.
c. ***High availability .*** Karena setiap komputer yang tergabung adalah standalone (mandiri), maka kegagalan salah satu komputer tidak menyebabkan kegagalan sistem.
d. ***Superior price/performance .*** Dengan konfigurasi yang tepat, dimungkinkan untuk membangun sistem yang jauh lebih powerful atau sama dengan komputer standalone, dengan biaya yang lebih rendah.

7.10. Sistem Waktu Nyata

Pada awalnya, istilah *real time* digunakan dalam simulasi. Memang sekarang lazim dimengerti bahwa real time adalah "cepat", namun sebenarnya yang dimaksud adalah simulasi yang bisa menyamai dengan proses sebenarnya (di dunia nyata) yang sedang disimulasikan.

Suatu sistem dikatakan *real time* jika dia tidak hanya mengutamakan ketepatan pelaksanaan instruksi/ tugas, tapi juga interval waktu tugas tersebut dilakukan. Dengan kata lain, sistem *real time* adalah sistem yang menggunakan *deadline*, yaitu pekerjaan harus selesai jangka waktu tertentu. Sementara itu, sistem yang tidak *real time* adalah sistem dimana tidak ada *deadline*, walaupun tentunya respons yang cepat atau performa yang tinggi tetap diharapkan.

Pada sistem waktu nyata, digunakan batasan waktu. Sistem dinyatakan gagal jika melewati batasan yang ada. Misal pada sistem perakitan mobil yang dibantu oleh robot. Tentulah tidak ada gunanya memerintahkan robot untuk berhenti, jika robot sudah menabrak mobil.

Sistem waktu nyata bisa dijumpai pada tugas-tugas yang mission critical, misal sistem untuk sistem pengendali reaktor nuklir atau sistem pengendali rem mobil. Juga sering dijumpai pada peralatan medis, peralatan pabrik, peralatan untuk riset ilmiah, dan sebagainya.

Ada dua model sistem *real time*, yaitu *hard real time* dan *soft real time*.

Hard real time mewajibkan proses selesai dalam kurun waktu tertentu. Jika tidak, maka gagal. Misalnya adalah alat pacu jantung. Sistem harus bisa memacu detak jantung jika detak jantung sudah terdeteksi lemah.

Sementara *soft real time* menerapkan adanya prioritas dalam pelaksanaan tugas dan toleransi waktu. Misalnya adalah transmisi video. Gambar bisa sampai dalam keadaan terpatah-patah, tetapi itu bisa ditolerir karena informasi yang disampaikan masih bisa dimengerti.

7.11. Aspek Lainnya

- **Sistem multimedia.** Sistem multimedia adalah sistem yang menyajikan berbagai bentuk informasi dengan barbagai tehnik seperti gambar, suara, teks, animasi, video yang disajikan secara interaktif untuk memberi informasi atau menghibur.
  - **Handal.** Para pengguna tentulah tidak akan gembira jika sistem terlalu sering *crash*.
  - **Sistem berkas.** Ukuran berkas multimedia cenderung sangat besar. Sebagai contoh , 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 .*** Piperlukan bandwidth (ukuran saluran data) yang besar untuk multimedia, misalnya video.

  - **Waktu nyata.** Selain memerlukan bandwidth yang besar, berkas multimedia harus disampaikan secara lancar berkesinambungan, serta tidak terputus-putus. Walaupun demikian, tentu ada toleransi tertentu terhadap kualitas gambar/suara (soft real time). Misal pada aplikasi video conference. Pengguna masih bisa mentolerir jika gambar sedikit terpatah-patah atau suara sedikit lebih lambat dari video.
- ***Embedded System .*** *Embedded system* pada dasarnya adalah komputer khusus yang tugasnya menjalankan tugas spesifik. Tidak seperti PC, yang bisa digunakan untuk banyak hal seperti browsing, menonton video, membuat program dan sebagainya, *embedded system* hanya melakukan satu atau beberapa tugas tertentu saja, tentunya masing-masing memiliki kebutuhan yang spesifik dan seringkali diperlengkapi dengan *hardware* khusus yang tidak lazim ditemui pada PC biasa. Biasanya *embedded system* menggunakan *hardware* yang terbatas, misal memori yang kecil atau tidak memiliki *harddisk*, tidak memiliki fasilitas canggih seperti *virtual memory* yang lazim ditemui di PC biasa, dan lain-lain. Karena embedded system hanya melakukan tugas tertentu, maka sistem bisa dioptimasi sedemikian rupa sehingga bisa memperkecil ukuran fisiknya dan menekan biaya produksi. Secara fisik, *embedded system* bisa dijumpai mulai dari yang berukuran kecil, seperti PDA,MP3 player atau jam digital, kemudian ke yang lebih besar seperti TV, video game console, router sehingga yang kompleks seperti sistem pengendali pabrik,sistem pengatur lampu lalu lintas, atau sistem pemandu pesawat. *Emdedded system* melakukan komputasi secara *real-time* dan mereka bisa saja berjalan dengan sedikit interaksi dari manusia (atau tidak sama sekali).
- **Komputasi Berbasis Jaringan.** Pada awalnya, komputasi tradisional hanya meliputi penggunaan komputer meja (desktop) untuk pemakaian pribadi di kantor atau di rumah. Dengan adanya perkembangan WWW, proses pertukaran informasi antar komputer semakin mudah saja. Bahkan lebih dari itu, proses komputasi pun dilakukan dalam jaringan. Dengan demikian, batas antara komputasi tradisional dan komputasi berbasis jaringan sudah tidak jelas lagi. Peralatan yang dulu tidak terhubung ke jaringan, kini terhubung ke jaringan. Sementara peralatan yang sudah terhubung ke jaringan, kini menggunakan teknologi yang lebih baik lagi, misal dengan peningkatan *hardware* atau penggunaan protokol komunikasi yang baru. Sekarang misalnya, adalah lazim menemui *hotspot* di tempat-tempat umum sehingga akses internet makin mudah saja.
- **PDA dan Telepon Seluler.** Sistem genggam ialah sebutan untuk komputer-komputer dengan kemampuan tertentu, serta berukuran kecil sehingga dapat digenggam. 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. Beberapa contoh dari sistem ini ialah Palm Pilot, 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 berukuran kartu nama yang memiliki kemampuan mengoleh informasi. 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. Sistem ini juga bisa digunakan sebagai sarana identifikasi pemiliknya

7.12. Rangkuman

Ada beberapa cara untuk menghubungkan komponen-komponen yang ada didalam suatu sistem operasi. Pertama, dengan menggunakan struktur sederhana. Kedua, dengan pendekatan terlapis atau level. Lapisan yang lebih rendah menyediakan layanan untuk lapisan yang lebih tinggi. Ketiga, dengan pendekatan kernel mikro, yaitu sistem operasi disusun dalam bentuk kernel yang lebih kecil.

Sistem berprosesor jamak 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*/ spesifik tugas tertentu. Sistem operasi yang *hard real-time* mempunyai batasan waktu yang tetap yang sudah didefinisikan dengan baik.Pemrosesan harus selesai dalam batasan-batasan yang sudah didefinisikan, atau sistem akan gagal.

Sistem *soft real-time* mempunyai lebih sedikit batasan waktu yang keras, dan tidak mendukung penjadwalan dengan menggunakan batas akhir. Sistem ini menggunakan prioritas untuk memilih tugas-tugas mana yang harus terlebih dahulu dikerjakan

Sistem terdistribusi adalah sistem yang menjalankan bagian-bagian program di beberapa komputer yang berbeda. Komputer-komputer itu terhubung dalam suatu jaringan. Sistem terdistribusi adalah salah satu bentuk dari paralelisme

Sistem terkluster adalah sistem yang terdiri dari banyak komputer yang disusun sedemikian rupa untuk menyatukan sumber daya yang ada dan seolah-olah dapat dipandang sebagai satu komputer saja. Model seperti ini sering digunakan untuk membangun super komputer.

Pengaruh dari internet dan WWW 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 Agar komputer dapat menjalankan suatu program, maka program tersebut harus berada di memori utama.

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.

[WEBWiki2007a] . 2007 . *Kernel - http://en.wikipedia.org/wiki/Kernel_(computer_science)* . Diakses 22 Februari 2007.

[WEBWiki2007b] . 2007 . *Symmetric multiprocessing - http://en.wikipedia.org/wiki/Symmetric_multiprocessing* . Diakses 23 Februari 2007.

[WEBWiki2007c] . 2007 . *Microkernel - http://en.wikipedia.org/wiki/Microkernel* . Diakses 23 Februari 2007.

[WEBWiki2007d] . 2007 . *Real time system - http://en.wikipedia.org/wiki/Real-time_system* . Diakses 23 Februari 2007.

Bab 8. *Virtual Machine*(VM)

8.1. Pendahuluan

Virtual Machine(VM) adalah sebuah mesin yang mempunyai dasar logika yang menggunakan pendekatan lapisan-lapisan (*layers*)dari sistem komputer.Sehingga sistem komputer dengan tersendiri dibangun atas lapisan-lapisan tersebut, dengan urutan lapisannya mulai dari lapisan terendah sampai lapisan teratas adalah sebagai berikut:

- Perangkat keras (semua bagian fisik komputer)
- Kernel (program untuk mengontrol disk dan sistem file, *multi-tasking*, *load-balancing*, *networking* dan *security*)
- Sistem program (program yang membantu *general user*)

Kernel yang berada pada lapisan kedua ini, 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 lainnya seolah-olah pada level yang sama. Meskipun sistem program berada di level tertinggi , namun program aplikasi bisa melihat segala sesuatu pada tingkatan dibawahnya seakan-akan mereka adalah bagian dari mesin. Pendekatan dengan lapisan-lapisan inilah yang kemudian menjadi kesimpulan logis pada konsep *Virtual Machine*(VM) atau *virtual machine*(VM).

Kekurangan *Virtual Machine*(VM)

Ada beberapa kesulitan utama dari konsep VM, diantaranya adalah:

- **Dalam sistem penyimpanan.** 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*, dimana ukuran daya penyimpanannya identik dengan ukuran sebenarnya. Dengan demikian, pendekatan VM juga menyediakan sebuah antarmuka yang identik dengan *underlying bare hardware*.
- **Dalam hal pengimplementasian.** Meski konsep VM cukup baik, namun VM sulit diimplementasikan.

Kelebihan *Virtual Machine*(VM)

Terlepas dari segala kekurangannya, VM memiliki beberapa keunggulan, antara lain:

- **Dalam 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.
- **Memungkinkan untuk mendefinisikan suatu jaringan dari *Virtual Machine*(VM).** Tiap-tiap bagian mengirim informasi melalui jaringan komunikasi virtual. Sekali lagi, jaringan dimodelkan setelah komunikasi fisik jaringan diimplementasikan pada perangkat lunak.

8.2. Virtualisasi Penuh

Virtualisasi penuh dalam ilmu komputer ialah teknik virtualisasi yang digunakan untuk implementasi pada berbagai macam lingkungan *virtual machine*: Salah satunya menyediakan simulasi lengkap yang mendasari suatu hardware. Hasilnya adalah sebuah system yang mampu mengeksekusi semua perangkat lunak pada perangkat keras yang bias dijalankan pada *Virtual Machine*(VM), termasuk semua sistem operasi.

Setiap user CP/CMS telah disediakan sebuah simulasi, komputer yang berdiri sendiri (stand-alone computer). Setiap mesin virtual serupa telah mempunyai kemampuan lengkap mesin yang mendasar, dan untuk *user Virtual Machine (VM)*telah tak dapat dibedakan dengan sistem privasi. Simulasi ini sangat luas, dan didasarkan pada prinsip operasi manual untuk perangkat keras. Jadi termasuk setiap elemen sebagai set instruksi, main memory, intrupsi, exceptions, dan akses peralatan. Hasilnya ialah sebuah mesin tunggal yang dapat menjadi multiplexed diantara banyak user. Virtualisasi penuh hanya mungkin diberikan pada kombinasi yang benar dari elemen hardware dan software. Sebagai contoh, tidak mungkin dengan kebanyakan system IBM pada seri 360, hanya dengan IBM sistem 360-67.

Tantangan utama pada virtualisasi penuh ada pada intersepsi dan simulasi dari operasi yang memiliki hak istimewa seperti instruksi I/O. Efek dari setiap operasi yang terbentuk dengan penggunaan *Virtual Machine*(VM) haruslah dirawat dalam *Virtual Machine*(VM) itu operasi virtual tidak diijinkan untuk diubah *state* dari virtual mesin lainnya, control program atau hardware. Beberapa instruksi mesin dapat di eksekusi secara langsung oleh hardware, semenjak itu efek sepenuhnya terkandung di dalam elemen yang dimanage oleh program kontrol, seperti lokasi memori dan register aritmatik. Tetapi instruksi lain yang dikenal dapat menembus mesin virtual tidak diijinkan untuk langsung di eksekusi, haruslah sebagai gantinya dikurung dan disimulasi.

Beberapa instruksi baik akses atau pengaruh *state* informasi berada di luar *Virtual Machine*(VM). Virtualisasi penuh telah terbukti sukses untuk sharing sistem diantara banyak user dan mengisolasi user dari user yang lainnya untuk reabilitas (kepercayaan) dan keamanan.

8.3. Virtualisasi Paruh

Virtualisasi paruh dalam ilmu komputer ialah teknik virtualisasi yang digunakan untuk pengimplementasian pada berbagai macam lingkungan *virtual machine*, salah satunya dengan menyediakan sebagian besar hal yang mendasari suatu *hardware*. Sebenarnya tidak semua fitur yang dimiliki *hardware* tersebut tersimulasi, tapi ada beberapa kemudahan *Virtual Machine*(VM) yang mana tidak semua *software* dapat berjalan tanpa modifikasi. Biasanya untuk mengartikan bahwa seluruh sistem operasi tidak dapat berjalan pada *Virtual Machine*(VM) akan mengisyaratkan virtualisasi penuh, namun banyak aplikasi dapat berjalan pada mesin tersebut.

Virtualisasi paruh memiliki pertanda lebih mudah diterapkan daripada virtualisasi penuh. Seiring dengan syarat kegunaan, *Virtual Machine*(VM) yang kuat mampu untuk mendukung aplikasi penting. Kekurangan virtualisasi paruh dibandingkan dengan virtualisasi penuh adalah keterbelakangan kesesuaian (*compatibility*) atau mudah dibawa (*portability*). Jika tampilan hardware tertentu tidak disimulasikan, suatu software yang menggunakan tampilan itu akan fail. Terlebih lagi, akan sulit untuk mengantisipasi nilai suatu tampilan yang telah akan digunakan oleh pemberian aplikasi. Virtualisasi paruh telah terbukti secara sukses untuk pertukaran sumberdaya antar banyak pengguna.

8.4. IBM VM

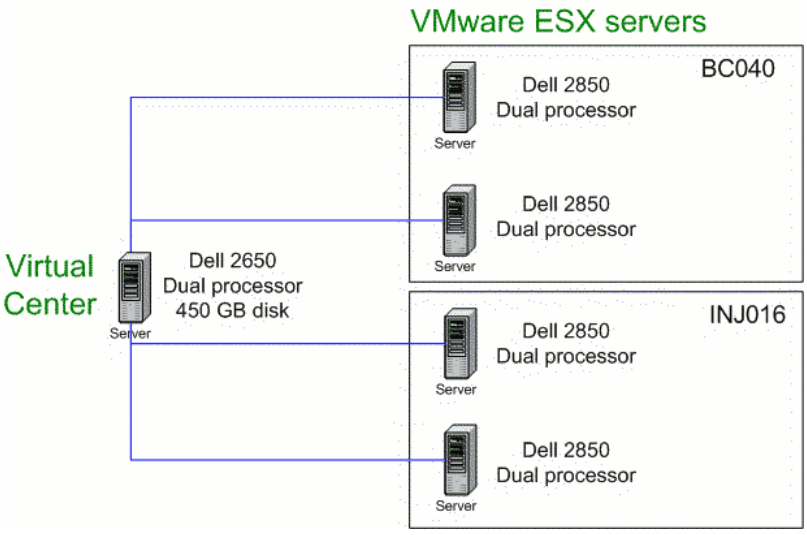
Istilah *Virtual Machine*(VM) sendiri mulai dikenalkan oleh IBM ketika meluncurkan sistem operasi mainframenya pada tahun 1965-an. Diperkenalkan untuk sistem S/370 dan S/390 dan disebut sebagai sistem operasi VM/ESA (*Enterprise System Architecture*). Sehingga sering menimbulkan kebingungan antara penamaan produk atau penamaan mekanisme. Banyak orang yang menyebut, walau memiliki mekanisme *Virtual Machine*(VM) tetapi bila bukan dari sistem IBM tersebut, maka tidak disebut dengan *Virtual Machine*.

Pada penjelasan ini, istilah *Virtual Machine*(VM) adalah suatu jenis mekanisme virtualisasi suatu mesin di atas mesin lainnya. Jadi bukan jenis produk dari salah satu vendor dengan nama *Virtual Machine*. Terdapat beberapa kegunaan dari *Virtual Machine*(VM), pada umumnya tampak untuk menggambarkan program yang bertindak selayaknya mesin.

8.5. VMware

Pada GNU/Linux salah satu *virtual machine* yang terkenal adalah VMware http://www.vmware.com. VMware memungkinkan beberapa sistem operasi dijalankan pada satu mesin PC tunggal secara bersamaan. Hal ini dapat dilakukan tanpa melakukan partisi ulang dan boot ulang. Pada *Virtual Machine*(VM) yang disediakan akan dijalankan sistem operasi sesuai dengan yang diinginkan. Dengan cara ini maka pengguna dapat memboot suatu sistem operasi (misal Linux) sebagai sistem operasi tuan rumah (*host*) dan lalu menjalankan sistem operasi lainnya misal MS Windows. Sistem operasi yang dijalankan di dalam sistem operasi tuan rumah dikenal dengan istilah sistem operasi tamu (*guest*).

Gambar 8.1. Contoh skema penggunaan pada VMware versi ESX Servers



Kebanyakan orang berpikir bahwa secara logisnya VMware diibaratkan sebagai *software* yang sering digunakan untuk keperluan percobaan *game*, aplikasi, untuk meng-install dua sistem operasi dan menjalankannya (misalnya Windows maupun Linux) pada *harddisk* yang sama tanpa memerlukan logout dari sistem operasi yang lainnya, secara gampang kita hanya tinggal menekan Alt + Tab untuk mengganti SO.

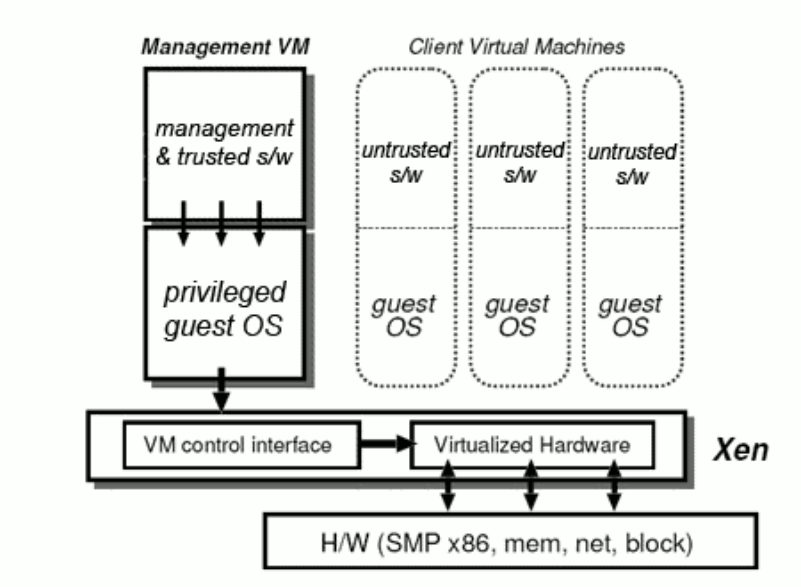
Akan tetapi pada dasarnya VMware bukanlah emulator, karena tidak mengemulasikan CPU dan perangkat keras di dalam suatu *Virtual Machine*(VM), tetapi hanya membolehkan sistem operasi lainnya dijalankan secara paralel dengan sistem operasi yang telah berjalan. Setiap *Virtual Machine*(VM) dapat memiliki alamat IP sendiri (jika mesin tersebut di suatu jaringan), dan pengguna dapat menganggapnya sebagai mesin terpisah.

8.6. Xen VMM

Xen adalah open source *virtual machine monitor*, dikembangkan di University of Cambridge. Dibuat dengan tujuan untuk menjalankan sampai dengan seratus sistem operasi ber-fitur penuh (*full featured OSs*) di hanya satu komputer. Virtualisasi Xen menggunakan teknologi paravirtualisasi menyediakan isolasi yang aman, pengatur sumberdaya, garansi untuk *quality-of-services* dan *live migration* untuk sebuah mesin virtual.

Untuk menjalankan Xen, sistem operasi dasar harus dimodifikasi secara khusus untuk kebutuhan tersendiri dan dengan cara ini dicapai kinerja virtualisasi sangat tinggi tanpa hardware khusus.

Gambar 8.2. Contoh dari penggunaan Xen VMM



8.7. Java VM

Program Java yang telah dikompilasi adalah *platform-neutral bytecodes* yang dieksekusi oleh *Java Virtual Machine*(JVM). JVM sendiri terdiri dari: *class loader*, *class verification*, *runtime interpreter*, *Just In-Time*(JIT) untuk meningkatkan kinerja kompilator.

Bahasa mesin terdiri dari sekumpulan instruksi yang sangat sederhana dan dapat dijalankan secara langsung oleh CPU dari suatu komputer. Sebuah program yang dibuat dengan bahasa tingkat tinggi tidak dapat dijalankan secara langsung pada komputer. Untuk dapat dijalankan, program tersebut harus ditranslasikan kedalam bahasa mesin. Proses translasi dilakukan oleh sebuah program yang disebut *compiler*.

Setelah proses translasi selesai, program bahasa-mesin tersebut dapat dijalankan, tetapi hanya dapat dijalankan pada satu jenis komputer. Hal ini disebabkan oleh setiap jenis komputer memiliki bahasa mesin yang berbeda-beda. Alternatif lain untuk mengkompilasi program bahasa tingkat tinggi selain menggunakan compiler, yaitu menggunakan *interpreter*. Perbedaan antara *compiler* dan *interpreter* adalah *compiler* mentranslasi program secara keseluruhan sekaligus, sedangkan *interpreter* menstranslasi program secara instruksi per instruksi. Java dibuat dengan mengkombinasikan antara *compiler* dan *interpreter*.

Program yang ditulis dengan java di- *compile* menjadi bahasa mesin. Tetapi bahasa mesin untuk komputer tersebut tidak benar-benar ada. Oleh karena itu disebut *"Virtual"* komputer, yang dikenal dengan *Java Virtual Machine (JVM)*. Bahasa mesin untuk JVM disebut Java *bytecode*. Salah satu keunggulan dari Java adalah dapat digunakan atau dijalankan pada semua jenis komputer. Untuk menjalankan program Java, komputer membutuhkan sebuah *interpreter* untuk Java *bytecode*.

Interpreter berfungsi untuk mensimulasikan JVM sama seperti *virtual computer* mensimulasikan PC komputer. Java *bytecode* yang dihasilkan oleh setiap jenis komputer berbeda-beda, sehingga diperlukan interpreter yang berbeda pula untuk setiap jenis komputer. Tetapi program Java *bytecode* yang sama dapat dijalankan pada semua jenis komputer yang memiliki Java *bytecode*.

8.8. *.NET Framework*

.NET Framework merupakan suatu komponen Windows yang terintegrasi yang dibuat dengan tujuan pengembangan berbagai macam aplikasi serta menjalankan aplikasi generasi mendatang termasuk pengembangan aplikasi XML Web Services.

Keuntungan Menggunakan *.NET Framework*

1. **Mudah.** Yang dimaksud mudah di sini adalah kemudahan developer untuk membuat aplikasi yang dijalankan di .NET Framework. Mendukung lebih dari 20 bahasa pemrograman : *VB.NET, C#, J#, C++, Pascal, Phyton (IronPhyton), PHP (PhLager).*
2. **Efisien.** Kemudahan pada saat proses pembuatan aplikasi, akan berimplikasi terhadap efisiensi dari suatu proses produktivitas, baik efisien dalam hal waktu pembuatan aplikasi atau juga efisien dalam hal lain, seperti biaya (cost).
3. **Konsisten.** Kemudahan-kemudahan pada saat proses pembuatan aplikasi, juga bisa berimplikasi terhadap konsistensi pada aplikasi yang kita buat. Misalnya, dengan adanya *Base Class Library*, maka kita bisa menggunakan objek atau *Class* yang dibuat untuk aplikasi berbasis windows pada aplikasi berbasis web. Dengan adanya kode yang bisa dintegrasikan ke dalam berbagai macam aplikasi ini, maka konsistensi kode-kode aplikasi kita dapat terjaga.
4. **Produktivitas.** Semua kemudahan-kemudahan di atas, pada akhirnya akan membuat produktivitas menjadi lebih baik. Produktivitas naik, terutama produktivitas para *developer*, akan berdampak pada meningkatnya produktivitas suatu perusahaan atau project.

8.9. Rangkuman

Konsep *Virtual Machine*(VM) adalah dengan menggunakan pendekatan lapisan-lapisan (layers) dari sistem komputer. Adapun beberapa hal yang berhubungan dan termasuk dalam *vitual machine* antara lain vitualisasi penuh dan paruh, IBM VM, VMware, Xen VMM, Java VM dan *.NET Framework.*

Virtualisasi adalah metode untuk membuat sesuatu menjadi lepas dari ketergantungan secara fisik. Contoh; virtual machine adalah komputer, yang sebetulnya hanya berupa sebuah file di hard disk kita. Dengan virtualisasi, maka sebuah komputer (fisik) bisa menjalankan banyak komputer virtual sekaligus pada saat yang bersamaan. *Virtual Machine*(VM) sendiri mulai dikenalkan oleh IBM ketika meluncurkan sistem operasi mainframenya pada tahun 1965-an. Diperkenalkan untuk sistem S/370 dan S/390 dan disebut sebagai sistem operasi VM/ESA (*Enterprise System Architecture*). VMware adalah suatu aplikasi yang memungkinkan kita untuk meng- *install* dua sistem operasi dan menjalakan aplikasinya (misalnya Windows and Linux) pada hardisk yang sama tanpa perlu logout dari SO yg lain. Xen adalah open source *Virtual Machine Monitor*, dikembangkan di University of Cambridge, untuk menjalakannya harus melalui dengan sebuah proses yakni pemodifikasian sistem operasi untuk lebih *full featured OSs*. Bahasa mesin untuk JVM disebut *Java bytecode* dengan keunggulannya yang bisa dijalankan pada berbagai jenis komputer atau *platform*. *.NET Framework* merupakan salah satu komponen yang tersedia dan terintegrasi pada sistem operasi Windows untuk kepentingan berbagai pengembangan aplikasi.

Rujukan

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

[WEBWiki2007] From Wikipedia, the free encyclopedia. 2007 . *Full Virtualization – http://en.wikipedia.org/wiki/Full_virtualization*. Diakses 15 Februari 2007.

[WEBWiki2007] From Wikipedia, the free encyclopedia. 2007 . *Partial Virtualization – http://en.wikipedia.org/wiki/Partial_virtualization.*. Diakses 15 Februari 2007.

[WEBCap-lore2007] Cap-lore. 2007 . *Short History of IBMs Virtual Machines – http://cap-lore.com/Software/CP.html.*. Diakses 15 Februari 2007.

[WEBWiryana2007] Wiryana Pandu. 2007 . *Komputer di dalam komputer – http://wiryana.pandu.org/indexe371.html*. Diakses 15 Februari 2007.

[WEBHarry2007] Harry Sufehmi. 2007 . *Virtualisasi – http://harry.sufehmi.com/archives/2006-07-29-1222/*. Diakses 28 Februari 2007.

[WEBFaculte2007] Sourythep Samoutphonh. 2007 . *VMware in the I and C School – http://ic-it.epfl.ch/bc2004/serveurs/vmware/index_en.php*. Diakses 12 Maret 2007.

Bab 9. GNU/Linux

9.1. Pendahuluan

Linux adalah sebuah sistem operasi yang dikembangkan oleh Linus Benedict Torvalds dari Universitas Helsinki Finlandia sebagai proyek hobi mulai tahun 1991. Ia menulis Linux, sebuah kernel untuk prosesor 80386, prosesor 32-bit pertama dalam kumpulan CPU Intel yang cocok untuk PC. Baru pada tanggal 14 Maret 1994 versi 1.0 mulai diluncurkan, dan hal ini menjadi tonggak sejarah Linux.

Linux merupakan *clone* dari UNIX yang telah di- *port* ke beragam platform, antara lain: Intel 80x86, AlphaAXP, MIPS, Sparch, Power PC, dsb. Sekitar 95% kode sumber kernel sama untuk semua *platform* perangkat keras.

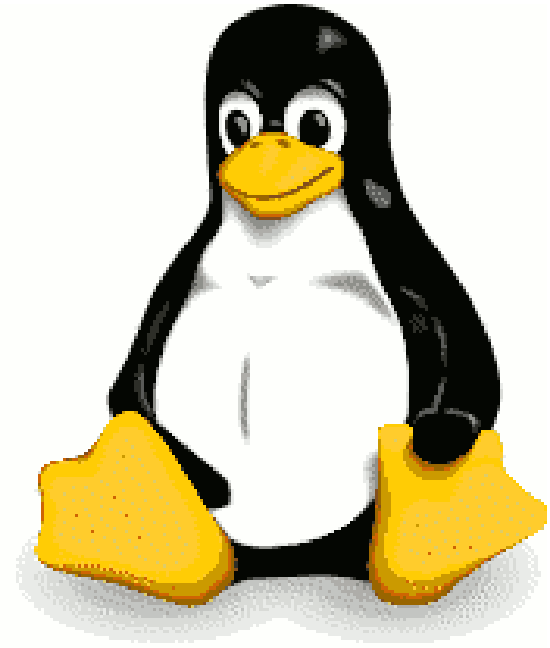
Linux termasuk sistem operasi yang didistribusikan secara *open source*, artinya kode sumber Linux diikutsertakan sehingga dapat dipelajari dan dikembangkan dengan mudah. Selain itu Linux dikembangkan oleh GNU (*General Public License*). Linux dapat digunakan untuk berbagai keperluan, seperti: jaringan, pengembangan *software*, dan sebagai *end-user platform*. Selama ini Linux menjadi sistem operasi yang menjadi banyak perhatian karena kecanggihan dan harganya yang relatif murah dibanding dengan sistem operasi yang lain. Macam-macam distribusi Linux antara lain: Stackware, Debian, RedHat, S.u.s.e., Caldera, dan Turbo Linux. Macam-macam distribusi Linux ini akan dibahas lebih mendalam pada Bagian 9.3, “Distro”.

Istilah Linux atau GNU/Linux (GNU) juga digunakan sebagai rujukan kepada keseluruhan distro Linux (Linux distribution), yang didalamnya selalu disertakan program-program lain yang mendukung sistem operasi ini. Contoh program-program tersebut adalah *Web Server*, Bahasa Pemrograman, Basis Data, Tampilan Desktop (Desktop Environment) (seperti GNOME dan KDE), dan aplikasi/ *software* perkantoran (office suite) seperti OpenOffice.org, KOffice, Abiword, Gnumeric, dan lainnya. Distro Linux telah mengalami pertumbuhan yang pesat dari segi popularitas, sehingga lebih populer dari versi UNIX yang menganut sistem lisensi dan berbayar (*proprietary*) maupun versi UNIX bebas lain yang pada awalnya menandingi dominasi Microsoft Windows dalam beberapa sisi.

Linux mendukung banyak Perangkat keras Komputer, dan telah digunakan di dalam berbagai peralatan dari Komputer pribadi, Superkomputer dan Sistem Benam (Embedded System) (seperti Telepon Seluler Ponsel dan Perekam Video pribadi Tivo).

Pada mulanya, Linux dibuat, dikembangkan dan digunakan oleh peminatnya saja. Kini Linux telah mendapat dukungan dari perusahaan besar seperti IBM, dan Hewlett-Packard dan perusahaan besar lain. Para pengamat teknologi informatika beranggapan kesuksesan ini dikarenakan Linux tidak bergantung kepada *vendor* (*vendor-independence*), biaya operasional yang rendah, dan kompatibilitas yang tinggi dibandingkan versi UNIX proprietari, serta faktor keamanan dan kestabilannya dibandingkan dengan Microsoft Windows. Ciri-ciri ini juga menjadi bukti atas keunggulan model pengembangan perangkat lunak sumber terbuka opensource software.

Gambar 9.1. Logo Linux



Tux, seekor Pinguin, merupakan logo dan maskot bagi Linux. Linux adalah *trademark* (SN: 1916230) yang dimiliki oleh Linus Torvalds. Linux terdaftar sebagai "Program sistem operasi komputer bagi penggunaan komputer dan operasi". *Trademark* ini didaftarkan setelah ada suatu kejadian di mana seorang pemalsu bernama William R Della Croce Jr mulai mengirim surat kepada para distributor Linux dan mengklaim *trademark* Linux adalah hak miliknya serta meminta royalti sebanyak 10% dari mereka. Para distributor Linux mulai mendorong agar *trademark* yang asli diberikan kepada Linus Torvalds. Pemberian lisensi *trademark* Linux sekarang dibawah pengawasan Linux Mark Institute.

9.2. Kernel

Bagian ini akan menjelaskan kernel secara umum dan sejarah perkembangan Kernel Linux. Kernel adalah suatu perangkat lunak yang menjadi bagian utama dari sebuah sistem operasi. Tugasnya melayani bermacam program aplikasi untuk mengakses perangkat keras komputer secara aman.

Karena akses terhadap perangkat keras terbatas, sedangkan ada lebih dari satu program yang harus dilayani dalam waktu yang bersamaan, maka kernel juga bertugas untuk mengatur kapan dan berapa lama suatu program dapat menggunakan satu bagian perangkat keras tersebut. Hal tersebut dinamakan sebagai *multiplexing*.

Akses kepada perangkat keras secara langsung merupakan masalah yang kompleks, oleh karena itu kernel biasanya mengimplementasikan sekumpulan abstraksi *hardware*. Abstraksi-abstraksi tersebut merupakan sebuah cara untuk menyembunyikan kompleksitas, dan memungkinkan akses kepada perangkat keras menjadi mudah dan seragam. Sehingga abstraksi pada akhirnya memudahkan pekerjaan programer.

Untuk menjalankan sebuah komputer kita tidak harus menggunakan kernel sistem operasi. Sebuah program dapat saja langsung di- *load* dan dijalankan diatas mesin 'telanjang' komputer, yaitu bilamana pembuat program ingin melakukan pekerjaannya tanpa bantuan abstraksi perangkat keras atau bantuan sistem operasi. Teknik ini digunakan oleh komputer generasi awal, sehingga bila kita ingin berpindah dari satu program ke program lain, kita harus mereset dan meload kembali program-program tersebut.

Ada 4 kategori kernel:
1. **Monolithic kernel.** Kernel yang menyediakan abstraksi perangkat keras yang kaya dan tangguh.
2. **Microkernel.** Kernel yang menyediakan hanya sekumpulan kecil abstraksi perangkat keras sederhana, dan menggunakan aplikasi-aplikasi yang disebut sebagai *server* untuk menyediakan fungsi-fungsi lainnya.

3. **Hybrid (modifikasi dari microkernel).** Kernel yang mirip microkernel, tetapi ia juga memasukkan beberapa kode tambahan di kernel agar ia menjadi lebih cepat
4. **Exokernel.** Kernel yang tidak menyediakan sama sekali abstraksi hardware, tapi ia menyediakan sekumpulan pustaka yang menyediakan fungsi-fungsi akses ke perangkat keras secara langsung atau hampir-hampir langsung.

Dari keempat kategori kernel yang disebutkan diatas, kernel Linux termasuk kategori *monolithic* kernel. Kernel Linux berbeda dengan sistem Linux. Kernel Linux merupakan sebuah perangkat lunak orisinil yang dibuat oleh komunitas Linux, sedangkan sistem Linux, yang dikenal saat ini, mengandung banyak komponen yang dibuat sendiri atau dipinjam dari proyek pengembangan lain.

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

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

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

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

Semenjak Desember 2003, telah diluncurkan Kernel versi 2.6, yang dewasa ini (2008) telah mencapai *patch* versi 2.6.26.1 (http://kambing.ui.edu/kernel-linux/v2.6/). Hal-hal yang berubah dari versi 2.6 ini ialah:

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

9.3. Distro

Distro Linux (singkatan dari distribusi Linux) adalah sebutan untuk sistem operasi komputer mirip Unix yang menggunakan kernel Linux. Distribusi Linux bisa berupa perangkat lunak bebas dan bisa juga berupa perangkat lunak komersial seperti Red Hat Enterprise, SuSE, dan lain-lain.

Ada banyak distribusi atau distro Linux yang telah muncul. Beberapa bertahan dan besar, bahkan sampai menghasilkan distro turunan, contohnya adalah :

- **Debian GNU/Linux (http://www.debian.org/).**
  Debian GNU/Linux adalah distro non-komersial yang dihasilkan oleh para sukarelawan dari seluruh dunia yang saling bekerjasama melalui Internet. Distro ini menginginkan adanya semangat *open-source* yang harus tetap ada pada Debian. Kedinamisan distro ini membuat setiap rilis paket-paketnya di-update setiap waktu dan dapat diakses melalui utilitas apt-get. Apt-get adalah sebuah utilitas baris-perintah yang dapat digunakan secara dinamis untuk meng- *upgrade* sistem Debian GNU/Linux melalui apt-repository jaringan archive Debian yang luas. Milis dan forum debian selalu penuh dengan pesan-pesan baik mengenai *bug*, masalah, *sharing*, dll. Dengan adanya sistem komunikasi ini *bug* dan masalah keamanan pada tiap paket dapat dilaporkan oleh para pengguna dan pengembang Debian dengan cepat.Keuntungan dari Debian adalah *upgradability*, ketergantungan antar paket didefinisikan dengan baik, dan pengembangannya secara terbuka.Beberapa proyek dan turunan Debian GNU/Linux:
  1. De2, http://de2.vlsm.org/
  2. Knoppix, http://www.knoppix.org/
  3. Debian JP, http://www.debian.linux.or.jp/
  4. Libranet.
  5. dan lain-lain
- **Red Hat Linux (http://www.redhat.com/).**
  Red Hat adalah distro yang cukup populer di kalangan pengembang dan perusahaan Linux. Dukungan-dukungan secara teknis, pelatihan, sertifikasi, aplikasi pengembangan, dan bergabungnya para *hacker* kernel dan *free-software* seperti Alan Cox, Michael Johnson, Stephen Tweedie menjadikan Red Hat berkembang cepat dan digunakan pada perusahaan. Poin terbesar dari distro ini adalah Red Hat Package Manager (RPM). RPM adalah sebuah perangkat lunak untuk memanajemen paket-paket pada sistem Linux kita dan dianggap sebagai standar *de-facto* dalam pemaketan pada distro-distro turunannya dan yang mendukung distro ini secara luas.
- **Slackware (http://www.slackware.com/).**
  Distronya Patrick Volkerding yang terkenal pertama kali setelah SLS. Slackware dikenal lebih dekat dengan gaya UNIX, sederhana, stabil, mudah di- *custom*, dan didesain untuk komputer 386/486 atau lebih tinggi. Distro ini termasuk distro yang *cryptic* dan manual sekali bagi pemula Linux, tapi dengan menggunakan distro ini beberapa penggunanya dapat mengetahui banyak cara kerja sistem dan distro tersebut. Debian adalah salah satu distro selain Slackware yang masuk dalam kategori ini. Sebagian besar aktivitas konfigurasi di Slackware dilakukan secara manual (tidak ada tool seperti Yast pada S.U.S.E ataupun Linuxconf pada RedHat).
- **S.u.S.E. (http://www.suse.com/).**
  S.u.S.E. adalah distro yang populer di Jerman dan Eropa, terkenal akan dukungan *driver* VGA-nya dan YasT. S.u.S.E tersedia secara komersial dan untuk versi GPL-nya dapat diinstal melalui ftp di situs S.u.S.E. Instalasi berbasis menu grafis dari CD-ROM, disket boot modular, 400-halaman buku referensi, dukungan teknis, dukungan driver-driver terutama VGA dan *tool* administrasi sistem S.u.S.E., YaST, membuat beberapa pengguna memilih distro ini.S.u.S.E. juga terlibat dalam pembuatan X server (video driver) untuk proyek XFree86 sehingga X server distro ini mendukung kartu grafis baru. S.U.S.E. menggunakan dua sistem pemaketan yaitu RPM (versi lama) dan SPM, S.U.S.E. Package Manager (versi baru).
- **Turbo Linux (http://www.turbolinux.com/).**
  TurboLinux menargetkan pada produk berbasis Linux dengan kinerja tinggi untuk pasar workstation dan server terutama untuk penggunaan clustering dan orientasinya ke perusahaan. Beberapa produk-produknya: TurboLinux *Workstation* untuk dekstopnya, TurboLinux Server untuk *backend server* dengan kinerja tinggi terutama untuk penggunaan bisnis di perusahaan, *e-commerce* dan transaksi B2B (*Business-to-Business*).Salah satu produknya *TurboCluster Server* ditargetkan untuk pembuatan *server cluster* yang berskala luas dan dapat digunakan 25 cluster node atau lebih. *TurboCluster server* ini pernah memenangkan poling Best Web Solution dari editor Linux Journal.enFuzion, satu lagi produk yang berbasis pada konsep sederhana dan powerful yang dinamakan ' *parametric execution*'. enFuzion akan merubah jaringan komputer perusahaan

menjadi super computer dengan kecepatan tinggi dan ' *fault tolerant*'. Pengguna produk dan layanan TurboLinux terbanyak adalah perusahaan dan perorangan di Jepang dan Asia.

Untuk mendapatkan distro linux, anda dapat mendownloadnya langsung dari situs distributor distro bersangkutan, atau membelinya dari penjual lokal.

9.4. Lisensi

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.

9.5. Prinsip Rancangan Linux

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

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

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

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

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

Linux saat ini mendukung ekstensi thread POSIX dan subset dari ekstensi untuk kontrol proses *real-time* POSIX.

Sistem Linux terdiri dari tiga bagian kode penting:

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

Kernel

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

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

Pustaka Sistem

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

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

Utilitas Sistem

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

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

9.6. Modul Kernel Linux

Pengertian Modul Kernel Linux

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

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

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

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

Manajemen Modul Kernel Linux

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

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

Komponen manajemen modul yang lain adalah peminta modul. Kernel mendefinisikan antarmuka komunikasi yang dapat dihubungi oleh program manajemen modul. Saat hubungan tercipta, kernel akan menginformasikan proses manajemen kapan pun sebuah proses meminta *device driver*, sistem berkas, atau layanan jaringan yang belum terpanggil dan memberikan manajer kesempatan

untuk memanggil layanan tersebut. Permintaan layanan akan selesai saat modul telah terpanggil. Manajer proses akan memeriksa secara berkala apakah modul tersebut masih digunakan, dan akan menghapusnya saat tidak diperlukan lagi.

Registrasi Driver

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

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

Resolusi Konflik

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

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

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

9.7. Rangkuman

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

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

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

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

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

Rujukan

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

[WEBWiki2007] Wikipedia. 2007 . *Distribusi Linux – http://id.wikipedia.Org/wiki/ Distribusi_Linux.html*. Diakses 8 Februari 2007.

[WEBWiki2007] Wikipedia. 2007 . *Kernel (ilmu komputer) – http://id.wikipedia.Org /wiki/Kernel*. Diakses 8 Februari 2007.

[WEBIwan2007] Iwan. 2007 . *Distro Linux – http://www.Duniasemu.org/writings/DistroLinux.html*. Diakses 8 Februari 2007.

Bagian III. Proses dan Penjadwalan

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

Bab 10. Konsep Proses

10.1. Pendahuluan

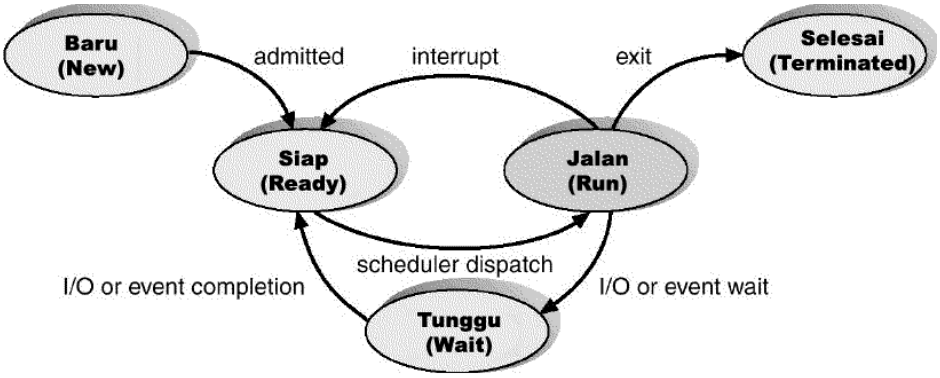
Proses didefinisikan sebagai program yang sedang dieksekusi. Menurut Silberschatz proses tidak hanya sekedar suatu kode program (*text section*), melainkan meliputi beberapa aktivitas yang bersangkutan seperti *program counter* dan stack. Sebuah proses juga melibatkan stack yang berisi data sementara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

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

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

10.2. Diagram Status Proses

Gambar 10.1. Status Proses



Sebuah proses dapat memiliki tiga status utama yaitu:
1. ***Running*.** Status yang dimiliki pada saat instruksi-instruksi dari sebuah proses dieksekusi
2. ***Waiting*.** Status yang dimiliki pada saat proses menunggu suatu sebuah event seperti proses M/K.
3. ***Ready*.** Status yang dimiliki pada saat proses siap untuk dieksekusi oleh prosesor

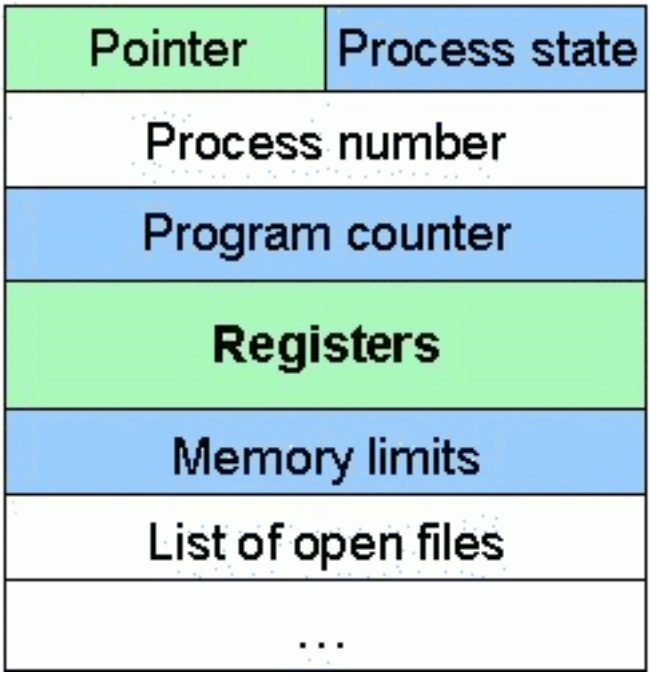
Terdapat dua status tambahan, yaitu saat pembentukan dan terminasi:
1. ***New*.** Status yang dimiliki pada saat proses baru saja dibuat
2. ***Terminated*.** Status yang dimiliki pada saat proses telah selesai dieksekusi.

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

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

10.3. *Process Control Block*

Gambar 10.2. *Process Control Block*



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

1. **Status Proses.** Status *new*, *ready*, *running*, *waiting*, *halted*, dan juga banyak lagi.
2. ***Program Counter* .** Suatu stack yang berisi alamat dari instruksi selanjutnya untuk dieksekusi untuk proses ini.
3. **CPU register.** Register bervariasi dalam jumlah dan jenis, tergantung pada rancangan komputer. Register tersebut termasuk *accumulator*, register indeks, *stack pointer*, *general-purposes register*, ditambah *code information* pada kondisi apa pun. Beserta dengan *program counter*, keadaan/status informasi harus disimpan ketika gangguan terjadi, untuk memungkinkan proses tersebut berjalan/ bekerja dengan benar setelahnya (lihat Gambar 10.3, Status Proses).
4. **Informasi manajemen memori.** Informasi ini dapat termasuk suatu informasi sebagai nilai dari dasar dan batas register, tabel halaman, atau tabel segmen tergantung pada sistem memori yang digunakan oleh sistem operasi (lihat Bagian V, Memori).
5. **Informasi pencatatan.** Informasi ini termasuk jumlah dari CPU dan waktu riil yang digunakan, batas waktu, jumlah akun, jumlah *job* atau proses, dan banyak lagi.
6. **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.

10.4. Pembentukan Proses

Saat komputer berjalan, terdapat banyak proses yang berjalan secara bersamaan. Sebuah proses dibuat melalui *system call create-process* yang membentuk proses turunan (*child process*) yang dilakukan oleh proses induk (*parent process*). Proses turunan tersebut juga mampu membuat proses baru sehingga semua proses ini pada akhirnya membentuk pohon proses. Ketika sebuah proses dibuat maka proses tersebut dapat memperoleh sumber-daya seperti waktu CPU, memori, berkas, atau perangkat M/K. Sumber daya ini dapat diperoleh langsung dari sistem operasi, dari proses induk yang membagi-bagikan sumber daya kepada setiap proses turunannnya, atau proses turunan dan proses induk berbagi sumber-daya yang diberikan sistem operasi.

Di dalam UNIX daftar dari proses yang sedang aktif berjalan bisa didapatkan dengan menggunakan perintah ps, contoh ps -el. Ada dua kemungkinan bagaimana jalannya (*running*) proses induk dan turunan. Proses-proses tersebut berjalan secara konkuren atau proses induk menunggu sampai beberapa/seluruh proses turunannya selesai berjalan. Juga terdapat dua kemungkinan dalam pemberian ruang alamat (*address space*) proses yang baru. Proses turunan dapat merupakan duplikasi.

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

10.5. Fungsi fork()

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

Contoh 10.1. Contoh Penggunaan fork()

```
#include <stdio.h>          /*standard M/K*/
#include <unistd.h>          /*fork()*/
#include <sys/types.h>       /*pid_t*/

int main()
{
   pid_t pid;
   pid = fork();

   if (pid < 0) {
      //terjadi error
      fprintf(stderr,"Fork Gagal");
      exit(-1);
   } else if (pid == 0) {
      //proses anak
      execlp("/bin/ls","ls",NULL);
   } else {
      //proses induk
      wait(NULL);
      printf("Proses anak selesai");
      exit(0);
   }
}
```

Berikut adalah contoh penggunaan fork() dengan menggunakan bahasa C. Tipe data pid_t merupakan *signed integer* yang sebenernya dalam pustaka GNU. Tipe ini adalah int, fungsinya adalah merepresentasikan PID. Program C diatas menggambarkan atau mengilustrasikan UNIX *system call*. Didalam UNIX Shell akan membaca perintah dari terminal, kemudian perintah tersebut di fork(), dan menghasilkan proses anak, proses anak inilah yang akan mengeksekusi perintah dari *shell* tersebut, sementara proses induk hanya menunggu dengan menggunakan *system call* wait() dan mengeksekusi perintah lain saat proses anak terminasi.

10.6. Terminasi Proses

Suatu proses diterminasi ketika proses tersebut telah selesai mengeksekusi perintah terakhir atau diterminasi dengan sengaja oleh proses lain, biasanya proses induk yang melakukan hal ini. Pada saat terminasi. Semua sumber-daya yang digunakan oleh proses akan dialokasikan kembali oleh sistem operasi agar dapat dimanfaatkan oleh proses lain. Suatu proses yang diterminasi karena selesai melakukan tugasnya, sistem operasi akan memanggil *system call* exit() sedangkan proses yang diterminasi dengan sengaja oleh proses lain melalui *system call abort*. Biasanya proses induk melakukan terminasi sengaja pada turunannya. Alasan terminasi tersebut seperti:

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

10.7. Proses Linux

Sebuah proses adalah konteks dasar dimana semua permintaan user dilayani sistem operasi. Agar menjadi kompatibel dengan sistem UNIX lainnya, Linux harus menggunakan model proses yang sama dengan sistem UNIX lainnya.

Prinsip dasar dari manajemen proses UNIX adalah memisahkan dua operasi untuk membuat proses dan menjalankan program baru. Proses baru dibuat dengan fungsi fork(), sedangkan program baru dijalankan setelah memanggil fungsi exec(). Model seperti ini memiliki kelebihan yaitu kesederhanaan dibanding harus menetapkan setiap detail dari lingkungan program baru dalam *system call* yang menjalankan program tersebut. Program baru dengan mudah berjalan dalam lingkungannya sendiri. Jika proses induk mengharapkan untuk memodifikasi lingkungan dimana program baru berjalan, dia bisa melakukan fork dan tetap menjalankan program asli dalam proses anak. Membuat beberapa *system call* membutuhkan modifikasi proses anak sebelum akhirnya mengeksekusi program baru. Setiap proses memiliki identitas proses yang isinya berupa:

1. **PID.** PIDs digunakan untuk menetapkan proses ke sistem operasi ketika sebuah aplikasi membuat *System call* untuk sinyal, modifikasi, atau menunggu proses lain.
2. ***Credentials* .** Setiap proses harus memiliki hubungan antara *user* ID dengan *group* ID yang menentukan hak sebuah proses untuk mengakses sumberdaya sistem dan file.
3. ***Personality* .** Dapat sedikit memodifikasi *semantics of system calls*.

10.8. Rangkuman

1. Sebuah proses adalah suatu program yang sedang dieksekusi.
2. Proses lebih dari sebuah kode program tetapi juga mencakup *program counter*, stack, dan sebuah *data section*.
3. Dalam pengeksekusiannya sebuah proses juga memiliki status yang mencerminkan keadaan dari proses tersebut.
4. Status tersebut mungkin menjadi satu dari lima status berikut: *new*, *ready*, *running*, *waiting*, atau *terminated*.
5. Proses direpresentasikan dengan PCB yang menyimpan segala informasi yang berkaitan dengan proses tersebut.

Rujukan

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

[WEBFSF1991a] Free Software Foundation. 2007 . *proses linux – http://www.ilmukomputer.com/ 2006/ 08/ 28/ sistem-operasi-proses-pada-linux/ fajar-proses-linux.zip*. Diakses 14 Feb 2007.

Bab 11. Konsep *Thread*

11.1. Pendahuluan

Pada bab sebelumnya kita telah mempelajari tentang proses, namun seiring berjalannya waktu dan tuntutan teknologi ternyata ditemukan kelemahan yang sebenarnya bisa diminimalisir pada proses. Untuk itulah diciptakan *thread* yang merupakan cara dari komputer untuk menjalankan dua atau lebih *task* dalam waktu bersamaan, sedangkan *multithreading* adalah cara komputer untuk membagi-bagi pekerjaan yang dikerjakan sebagian-sebagian dengan cepat sehingga menimbulkan efek seperti menjalakan beberapa *task* secara bersamaan walaupun otaknya hanya satu. Di sini kita akan belajar mengapa harus ada *thread*, perbedaan *thread* dengan proses, keuntungan pengimplementasian *thread*, model-model *multithreading*, pengimplementasian pustaka *thread*, pembatalan *thread*, *thread pools*, penjadwalan *thread* dan *thread* di Linux.

11.2. Keuntungan *MultiThreading*

Multiprocessing merupakan penggunaan dua atau lebih CPU dalam sebuah sistem komputer. *Multitasking* merupakan metode untuk menjalankan lebih dari satu proses dimana terjadi pembagian sumberdaya seperti CPU. *Multithreading* adalah cara pengeksekusian yang mengizinkan beberapa *thread* terjadi dalam sebuah proses, saling berbagi sumber daya tetapi dapat dijalankan secara independen.

Keuntungan dari sistem yang menerapkan *multithreading* dapat kita kategorikan menjadi 4 bagian:

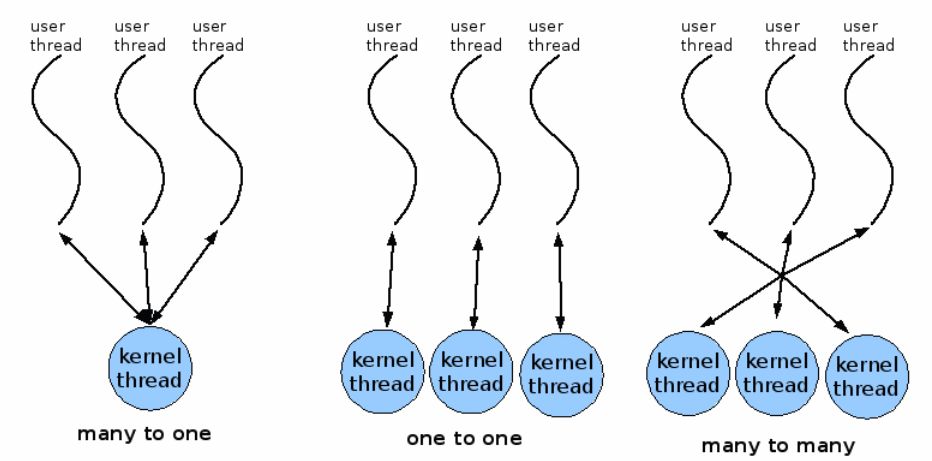
a. **Responsif.** Aplikasi interaktif menjadi tetap responsif meskipun sebagian dari program sedang diblok atau melakukan operasi lain yang panjang. Umpamanya, sebuah *thread* dari *web browser* dapat melayani permintaan pengguna sementara *thread* yang lain berusaha menampilkan gambar.
b. **Berbagi sumber daya.** Beberapa *thread* yang melakukan proses yang sama akan berbagi sumber daya. Keuntungannya adalah mengizinkan sebuah aplikasi untuk mempunyai beberapa *thread* yang berbeda dalam lokasi memori yang sama.
c. **Ekonomis.** Pembuatan sebuah proses memerlukan pengalokasian memori dan sumber daya. Alternatifnya adalah dengan menggunakan *thread*, karena *thread* membagi memori dan sumber daya yang dimilikinya sehingga lebih ekonomis untuk membuat *thread* dan *context switching thread*. Akan susah mengukur perbedaan waktu antara *thread* dan *switch*, tetapi secara umum pembuatan dan pengaturan proses akan memakan waktu lebih lama dibandingkan dengan *thread*. Pada Solaris, pembuatan proses memakan waktu 30 kali lebih lama dibandingkan pembuatan *thread* sedangkan proses *context switch* 5 kali lebih lama dibandingkan *context switching thread*.
d. **Utilisasi arsitektur multiprosesor.** Keuntungan dari *multithreading* dapat sangat meningkat pada arsitektur multiprosesor, dimana setiap *thread* dapat berjalan secara paralel di atas procesor yang berbeda. Pada arsitektur processor tunggal, CPU menjalankan setiap *thread* secara bergantian tetapi hal ini berlangsung sangat cepat sehingga menciptakan ilusi paralel, tetapi pada kenyataanya hanya satu *thread* yang dijalankan CPU pada satu-satuan waktu.

11.3. Model *MultiThreading*

Beberapa terminologi yang akan dibahas:

a. ***Thread* pengguna:** *Thread* yang pengaturannya dilakukan oleh pustaka *thread* pada tingkatan pengguna. Karena pustaka yang menyediakan fasilitas untuk pembuatan dan penjadwalan *thread*, *thread* pengguna cepat dibuat dan dikendalikan.
b. ***Thread* Kernel:** *Thread* yang didukung langsung oleh kernel. Pembuatan, penjadwalan dan manajemen *thread* dilakukan oleh kernel pada *kernel space*. Karena dilakukan oleh sistem operasi, proses pembuatannya akan lebih lambat jika dibandingkan dengan *thread* pengguna.

Gambar 11.1. Model-Model *MultiThreading*



Model-Model *MultiThreading*:

a. **Model *Many-to-One* .** Model ini memetakan beberapa *thread* tingkatan pengguna ke sebuah *thread*. tingkatan kernel. Pengaturan *thread* dilakukan dalam ruang pengguna sehingga efisien. Hanya satu *thread* pengguna yang dapat mengakses *thread* kernel pada satu saat. Jadi *Multiple thread* tidak dapat berjalan secara paralel pada multiprosesor. Contoh: Solaris *Green Threads* dan GNU *Portable Threads*.
b. **Model *One-to-One* .** Model ini memetakan setiap *thread* tingkatan pengguna ke setiap *thread*. Ia menyediakan lebih banyak *concurrency* dibandingkan model *Many-to-One*. Keuntungannya sama dengan keuntungan *thread* kernel. Kelemahan model ini ialah setiap pembuatan *thread* pengguna memerlukan tambahan *thread* kernel. Karena itu, jika mengimplementasikan sistem ini maka akan menurunkan kinerja dari sebuah aplikasi sehingga biasanya jumlah *thread* dibatasi dalam sistem. Contoh: Windows NT/XP/2000 , Linux, Solaris 9.
c. **Model *Many-to-Many* .** Model ini memultipleks banyak *thread* tingkatan pengguna ke *thread* kernel yang jumlahnya sedikit atau sama dengan tingkatan pengguna. Model ini mengizinkan *developer* membuat *thread* sebanyak yang ia mau tetapi *concurrency* tidak dapat diperoleh karena hanya satu *thread* yang dapat dijadwalkan oleh kernel pada suatu waktu. Keuntungan dari sistem ini ialah kernel *thread* yang bersangkutan dapat berjalan secara paralel pada multiprosessor.

11.4. Pustaka *Thread*

Pustaka *Thread* atau yang lebih familiar dikenal dengan *Thread Library* bertugas untuk menyediakan API untuk *programmer* dalam menciptakan dan memanage *thread*. Ada dua cara dalam mengimplementasikan pustaka *thread*:

a. Menyediakan API dalam level pengguna tanpa dukungan dari kernel sehingga pemanggilan fungsi tidak melalui *system call*. Jadi, jika kita memanggil fungsi yang sudah ada di pustaka, maka akan menghasilkan pemanggilan fungsi *call* yang sifatnya lokal dan bukan *system call*.
b. Menyediakan API di level kernel yang didukung secara langsung oleh sistem operasi. Pemanggilan fungsi *call* akan melibatkan *system call* ke kernel.

Ada tiga pustaka *thread* yang sering digunakan saat ini, yaitu: POSIX Pthreads, Java, dan Win32. Implementasi POSIX standard dapat dengan cara *user* level dan kernel level, sedangkan Win32 adalah kernel level. Java API *thread* dapat diimplementasikan oleh Pthreads atau Win32.

11.5. Pembatalan *Thread*

Thread Cancellation ialah pembatalan *thread* sebelum tugasnya selesai. Umpamanya, jika dalam program Java hendak mematikan Java Virtual Machine (JVM). Sebelum JVM dimatikan, maka seluruh *thread* yang berjalan harus dibatalkan terlebih dahulu. Contoh lain adalah di masalah *search*.

Apabila sebuah *thread* mencari sesuatu dalam *database* dan menemukan serta mengembalikan hasilnya, *thread* sisanya akan dibatalkan. *Thread* yang akan diberhentikan biasa disebut *target thread*.

Pemberhentian *target Thread* dapat dilakukan dengan 2 cara:
a. ***Asynchronous cancellation.*** Suatu *thread* seketika itu juga membatalkan *target thread*.
b. ***Deferred cancellation .*** Suatu *thread* secara periodik memeriksa apakah ia harus batal, cara ini memperbolehkan *target thread* untuk membatalkan dirinya secara terurut.

Hal yang sulit dari pembatalan *thread* ini adalah ketika terjadi situasi dimana sumber daya sudah dialokasikan untuk *thread* yang akan dibatalkan. Selain itu kesulitan lain adalah ketika *thread* yang dibatalkan 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 dibatalkan tetapi seringkali sistem operasi tidak mengambil kembali semua sumber daya dari *thread* yang dibatalkan.

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

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

11.6. *Thread Pools*

Pada *web server* yang *multithreading* ada dua masalah yang timbul:
a. Ukuran waktu yang diperlukan untuk menciptakan *thread* yang melayani permintaan yang diajukan pada kenyataannya *thread* dibuang seketika sesudah ia menyelesaikan tugasnya.
b. 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, 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 ada *thread* yang tersedia pada saat dibutuhkan maka *server* menunggu sampai ada satu *thread* yang bebas.

Keuntungan *thread pool* adalah:
a. Biasanya lebih cepat untuk melayani permintaan dengan *thread* yang ada dibandingkan menunggu *thread* baru dibuat.
b. *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*. Jumlah *thread* dalam *pool* dapat tergantung dari jumlah CPU dalam sistem, jumlah memori fisik, dan jumlah permintaan klien yang *concurrent*.

11.7. Penjadwalan *Thread*

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

Untuk menjadwalkan *thread*, sistem dengan model *mulithreading many to many* atau *many to one* menggunakan:
a. **Process Contention Scope (PCS).** Pustaka *thread* menjadwalkan thread pengguna untuk berjalan pada LWP (*lightweight process*) yang tersedia.

b. **System Contention Scope (SCS).** SCS berfungsi untuk memilih satu dari banyak *thread*, kemudian menjadwalkannya ke satu *thread* tertentu(CPU / Kernel).

11.8. *Thread* Linux

Ketika pertama kali dikembangkan, Linux tidak didukung dengan *threading* di dalam kernelnya, tetapi dia mendukung proses-proses sebagai entitas yang dapat dijadwalkan melalui *clone() system calls*. Sekarang Linux mendukung penduplikasian proses menggunakan *system call* clone() dan fork(). Clone() mempunyai sifat mirip dengan fork(), kecuali dalam hal pembuatan *copy* dari proses yang dipanggil dimana ia membuat sebuah proses yang terpisah yang berbagi *address space* dengan proses yang dipanggil. Pembagian *address space* dari *parent process* memungkinkan *cloned task* bersifat mirip dengan *thread* yang terpisah. Pembagian *address space* ini dimungkinkan karena proses direpresentasikan di dalam Kernel Linux. Di dalam Kernel Linux setiap proses direpresentasikan sebagai sebuah struktur data yang unik. Jadi, daripada menciptakan yang baru maka struktur data yang baru mengandung *pointer* yang menunjuk ke tempat dimana data berada. Jadi ketika fork() dipanggil, proses yang baru akan tercipta beserta duplikasi dari segala isi di struktur data di *parent process*, namun ketika clone() dipanggil, ia tidak menduplikasi *parent processnya* tetapi menciptakan *pointer* ke struktur data pada *parent process* yang memungkinkan *child process* untuk berbagi memori dan sumber daya dari *parent processnya*. Project LinuxThread menggunakan *system call* ini untuk mensimulasi *thread* di *user space*. Sayangnya, pendekatan ini mempunyai beberapa kekurangan, khusunya di area *signal handling*, *scheduling*, dan *interprocess synchronization primitive*.

Untuk meningkatkan kemampuan Thread Linux, dukungan kernel dan penulisan ulang pustaka *thread* sangat diperlukan. Dua *project* yang saling bersaing menjawab tantangan ini. Sebuah tim yang terdiri dari pengembang dari IBM membuat NGPT (Next Generation POSIX Threads). Sementara pengembang dari *Red Hat* membuat NPTL (Native POSIX Thread Library).Sebenarnya Linux tidak membedakan antara proses dan *thread*. Dalam kenyataannya, Linux lebih menggunakan istilah *task* dibandingkan proses dan *thread* ketika merujuk kepada pengaturan alur pengontrolan di dalam program.

11.9. Rangkuman

Thread adalah alur kontrol dari suatu proses.

Keuntungan menggunakan *Multithreading*:
a. Meningkatkan respon dari pengguna.
b. Pembagian sumber daya.
c. Ekonomis.
d. Mengambil keuntungan dari arsitektur multiprosessor.

Tiga model *Multithreading*:
a. Model *Many-to-One*.
b. Model *One-to-One*.
c. Model *Many-to-Many*.

Pembatalan *Thread*: Tugas untuk membatalkan *Thread* sebelum menyelesaikan tugasnya.

Pembatalan *Thread* terdiri dari 2 jenis:
1. *Asynchronous cancellation*.
2. *Deffered cancellation.*

Thread Pools menciptakan sejumlah *Thread* yang ditempatkan di dalam *pool* dimana *Thread* menunggu untuk dipanggil.

Thread Schedulling ada 2 macam:
1. *Local Schedulling*.
2. *Global Schedulling*.

Istilah *thread* di Linux adalah *task*.

Pembuatan Thread di Linux menggunakan *System call* clone().

Rujukan

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

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

[WEBWIKI2007] Wikipedia. 2007 . *Thread_(computer_science) – http://en.wikipedia.org/ wiki/ Thread_(computer_science)*. Diakses 16 Februari 2007.

[WEBWIKI2007] Wikipedia.. 2007 . *Multiprocessing – http://en.wikipedia.org/ wiki/ Multiprocessing*. Diakses 16 Februari 2007.

[WEBWIKI2007] Wikipedia.. 2007 . *Computer_multitasking – http://en.wikipedia.org/ wiki/ Computer_mutitasking*. Diakses 16 Februari 2007.

[WEBIBM] IBM.. 2007 . *linux_threading – http://www-128.ibm.com/ developerworks/ java/ library/ j-prodcon*. Diakses 16 Februari 2007.

Bab 12. *Thread* Java

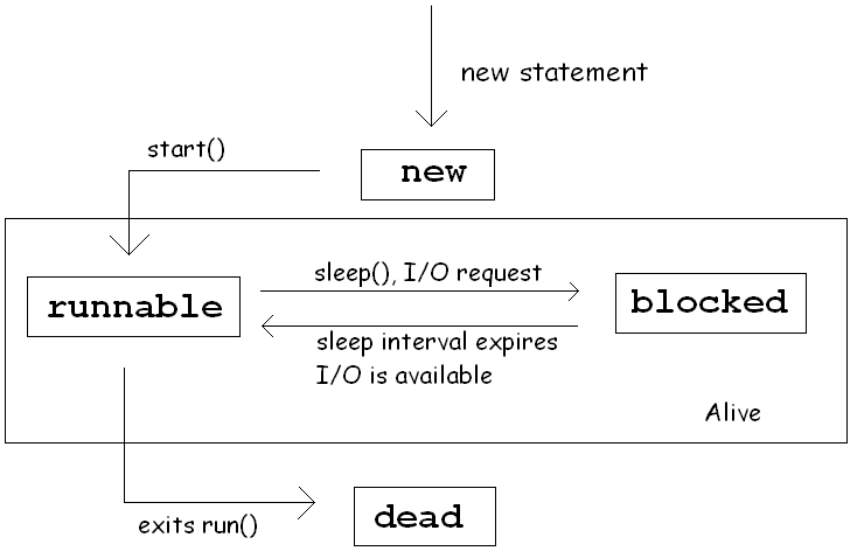
12.1. Pendahuluan

Suatu proses dikontrol oleh paling sedikit satu *thread*. Namun, sebagian besar proses yang ada sekarang biasanya dijalankan oleh beberapa buah *thread*. *Multithreading* adalah sebuah mekanisme di mana dalam suatu proses, ada beberapa *thread* yang mengerjakan tugasnya masing-masing pada waktu yang bersamaan. Contohnya, sebuah *web browser* harus menampilkan sebuah halaman yang memuat banyak gambar. Pada program yang *single-threaded*, hanya ada satu *thread* untuk mengatur suatu gambar, lalu jika gambar itu telah ditampilkan, barulah gambar lain bisa diproses. Dengan *multithreading*, proses bisa dilakukan lebih cepat jika ada *thread* yang menampilkan gambar pertama, lalu *thread* lain untuk menampilkan gambar kedua, dan seterusnya, di mana *thread-thread* tersebut berjalan secara paralel.

Saat sebuah program Java dieksekusi, yaitu saat `main()` dijalankan, ada sebuah *thread* utama yang bekerja. Java adalah bahasa pemrograman yang mendukung adanya pembentukan *thread* tambahan selain *thread* utama tersebut. *Thread* dalam Java diatur oleh *Java Virtual Machine*(JVM) sehingga sulit untuk menentukan apakah *thread* Java berada di *user-level* atau *kernel-level*.

12.2. Status *Thread*

Gambar 12.1. Status Thread



Suatu *thread* bisa berada pada salah satu dari status berikut:

- ***New .*** *Thread* yang berada di status ini adalah objek dari kelas `Thread` yang baru dibuat, yaitu saat instansiasi objek dengan *statement new*. Saat *thread* berada di status `new`, belum ada sumber daya yang dialokasikan, sehingga *thread* belum bisa menjalankan perintah apapun.
- ***Runnable .*** Agar *thread* bisa menjalankan tugasnya, method `start()` dari kelas `Thread` harus dipanggil. Ada dua hal yang terjadi saat pemanggilan method `start()`, yaitu alokasi memori untuk *thread* yang dibuat dan pemanggilan method `run()`. Saat method `run()` dipanggil, status *thread* berubah menjadi *runnable*, artinya *thread* tersebut sudah memenuhi syarat untuk dijalankan oleh JVM. *Thread* yang sedang berjalan juga berada di status *runnable*.

- ***Blocked .*** Sebuah *thread* dikatakan berstatus *blocked* atau terhalang jika terjadi *blocking statement*, misalnya pemanggilan method `sleep()`. `sleep()` adalah suatu method yang menerima argumen bertipe *integer* dalam bentuk milisekon. Argumen tersebut menunjukkan seberapa lama *thread* akan "tidur". Selain `sleep()`, dulunya dikenal method `suspend()`, tetapi sudah disarankan untuk tidak digunakan lagi karena mengakibatkan terjadinya *deadlock*. Di samping *blocking statement*, adanya interupsi M/K juga dapat menyebabkan *thread* menjadi *blocked*. *Thread* akan menjadi *runnable* kembali jika interval method `sleep()`-nya sudah berakhir, atau pemanggilan method `resume()` jika untuk menghalangi *thread* tadi digunakan method `suspend()` atau M/K sudah tersedia lagi.
- ***Dead .*** Sebuah *thread* berada di status *dead* bila telah keluar dari method `run()`. Hal ini bisa terjadi karena *thread* tersebut memang telah menyelesaikan pekerjaannya di method `run()`, maupun karena adanya pembatalan *thread*. Status jelas dari sebuah *thread* tidak dapat diketahui, tetapi method `isAlive()` mengembalikan nilai boolean untuk mengetahui apakah *thread* tersebut *dead* atau tidak.

12.3. Pembentukan *Thread*

Ada dua cara untuk membuat thread di program Java, yaitu:
a. Extends kelas Thread
b. Implements interface Runnable.
Interface Runnable didefinisikan sebagai berikut:

```
public interface Runnable
   {
        public abstract void run();
   }
```

Kelas `Thread` secara implisit juga meng- *implements interface* `Runnable`. Oleh karena itu, setiap kelas yang diturunkan dari kelas `Thread` juga harus mendefinisikan method `run()`. Berikut ini adalah contoh kelas yang menggunakan cara pertama untuk membuat *thread*, yaitu dengan meng-*extends* kelas `Thread`.

```
class CobaThread1 extends Thread
{
   public void run()
   {
         for (int ii = 0; ii&lt;4; ii++){
         System.out.println("Ini CobaThread1");
         Test.istirohat(11);
         }
   }
}
```

Konsep pewarisan dalam Java tidak mendukung *multiple inheritance*. Jika sebuah kelas sudah meng-*extends* suatu kelas lain, maka kelas tersebut tidak lagi bisa meng- *extends* kelas `Thread`. Oleh karena itu, cara kedua, yaitu meng- *implements interface* `Runnable`, lebih umum digunakan, karena kita bisa meng- *implements* dari banyak kelas sekaligus.

```
class CobaThread2 implements Runnable
{
```

```
   public void run()
   {
        for(int ii = 0; ii&lt;4; ii++){
        System.out.println("Ini CobaThread2");
        Test.istirohat(7);
   }
}

public class Test
{
   public static void main (String[] args)
   {
      Thread t1 = new CobaThread1();
      Thread t2 = new Thread (new CobaThread2());
      t1.start();
      t2.start();

      for (int ii = 0; ii&lt;8; ii++){
        System.out.println("Thread UTAMA");
        istirohat(5);
        }
   }

   public static void istirohat(int tunda)
   {
        try{
           Thread.sleep(tunda*100);
           } catch(InterruptedException e) {}
   }
}
```

Pada bagian awal `main()`, terjadi instansiasi objek dari kelas `CobaThread1` dan `CobaThread2`, yaitu `t1` dan `t2`. Perbedaan cara penginstansian objek ini terletak pada perbedaan akses yang dimiliki oleh kelas-kelas tersebut. Supaya *thread* bisa bekerja, method `start()` dari kelas `Thread` harus dipanggil. Kelas `CobaThread1` memiliki akses ke method-method yang ada di kelas `Thread` karena merupakan kelas yang diturunkan langsung dari kelas `Thread`. Namun, tidak demikian halnya dengan kelas `CobaThread2`. Oleh karena itu, kita harus tetap membuat objek dari kelas `Thread` yang menerima argumen objek `CobaThread2` pada *constructor*-nya, barulah `start()` bisa diakses. Hal ini ditunjukkan dengan *statement* `Thread t2 = new Thread (new CobaThread2())`.

Jadi, ketika terjadi pemanggilan method `start()`, *thread* yang dibuat akan langsung mengerjakan baris-baris perintah yang ada di method `run()`. Jika `run()` dipanggil secara langsung tanpa melalui `start()`, perintah yang ada di dalam method `run()` tersebut akan tetap dikerjakan, hanya saja yang mengerjakannya bukanlah *thread* yang dibuat tadi, melainkan *thread* utama.

12.4. Penggabungan *Thread*

Tujuan *multithreading* adalah agar *thread-thread* melakukan pekerjaan secara paralel sehingga program dapat berjalan dengan lebih baik. *Thread* tambahan yang dibuat akan berjalan secara terpisah dari *thread* yang membuatnya. Namun, ada keadaan tertentu di mana *thread* utama perlu menunggu sampai *thread* yang dibuatnya itu menyelesaikan tugasnya. Misalnya saja, untuk bisa mengerjakan instruksi selanjutnya, *thread* utama membutuhkan hasil penghitungan yang dilakukan oleh *thread* anak. Pada keadaan seperti ini, *thread* utama bisa menunggu selesainya pekerjaan *thread* anak dengan pemanggilan method `join()`.

Contohnya, dalam suatu program, *thread* utama membuat sebuah thread tambahan bernama `t1`.

```
try{
   t1.join();
   } catch (InterruptedException ie) {};
```

Kode di atas menunjukkan bahwa *thread* utama akan menunggu sampai *thread* `t1` menyelesaikan tugasnya, yaitu sampai method `run()` dari `t1` *terminate*, baru melanjutkan tugasnya sendiri. Pemanggilan method `join()` harus diletakkan dalam suatu blok `try-catch` karena jika pemanggilan tersebut terjadi ketika *thread* utama sedang diinterupsi oleh *thread* lain, maka `join()` akan melempar `InterruptedException`. `InterruptedException` akan mengakibatkan terminasi *thread* yang sedang berada dalam status *blocked*.

12.5. Pembatalan *Thread*

Pembatalan *thread* adalah menterminasi sebuah *thread* sebelum tugasnya selesai. *Thread* yang akan dibatalkan, atau biasa disebut *target thread*, dapat dibatalkan dengan dua cara, yaitu *asynchronous cancellation* dan *deferred cancellation*. Pada *asynchoronous cancellation*, sebuah *thread* langsung menterminasi *target thread*, sedangkan pada *deferred cancellation*, *target thread* secara berkala memeriksa apakah ia harus *terminate* sehingga dapat memilih saat yang aman untuk *terminate*.

Pada *thread* Java, *asynchronous cancellation* dilakukan dengan pemanggilan method `stop()`. Akan tetapi, method ini sudah di- *deprecated* karena terbukti tidak aman. `Stop()` dapat mengakibatkan terjadinya *exception* `ThreadDeath`, yang mematikan *thread-thread* secara diam-diam, sehingga *user* mungkin saja tidak mendapat peringatan bahwa programnya tidak berjalan dengan benar.

Cara yang lebih aman untuk membatalkan *thread* Java adalah dengan *deferred cancellation*. Pembatalan ini dapat dilakukan dengan pemanggilan method `interrupt()`, yang akan mengeset status interupsi pada *target thread*. Sementara itu, *target thread* dapat memeriksa status interupsi-nya dengan method `isInterrupted()`.

```
class CobaThread3 implements Runnable
{
   public void run(){
      while (true){
         System.out.println("saya thread CobaThread3");
         if (Thread.currentThread.isInterrupted())       //cek status
      break;
      }
   }
}
```

Suatu *thread* dari kelas `CobaThread3` dapat diinterupsi dengan kode berikut:

```
Thread targetThread = new Thread (new CobaThread3());
targetThread.start();
targetThread.interrupt();     //set status interupsi
```

Ketika *thread targetThread* berada pada `start()`, *thread* tersebut akan terus me- *loop* pada method `run()` dan melakukan pengecekan status interupsi melalui method `isInterrupted()`. Status interupsinya sendiri baru di-set ketika pemanggilan method `interrupt()`, yang ditunjukkan dengan *statement* `targetThread.interrupt();`. Setelah status interupsi di-set,

ketika pengecekan status interupsi selanjutnya pada method `run()`, `isInterrupted()` akan mengembalikan nilai boolean true, sehingga targetThread akan keluar dari method `run()`-nya melalui *statement* `break` dan *terminate*.

Selain melalui `isInterrupted()`, pengecekan status interupsi dapat dilakukan dengan method `interrupted()`. Perbedaan kedua method ini adalah `isInterrupted()` akan mempertahankan status interupsi, sedangkan pada `interrupted()`, status interupsi akan di- *clear*.

Thread yang statusnya sedang *blocked* karena melakukan operasi M/K menggunakan *package* `java.io` tidak dapat memeriksa status interupsinya sebelum operasi M/K itu selesai. Namun, melalui Java 1.4 diperkenalkan *package* `java.nio` yang mendukung interupsi *thread* yang sedang melakukan operasi M/K.

12.6. JVM

Setiap program Java dijalankan oleh *Java Virtual Machine*(JVM). Artinya, program Java dapat dijalankan di *platform* manapun selama *platform* tersebut mendukung JVM. Umumnya JVM diimplementasikan di bagian atas suatu *host operating system*, hal ini memungkinkan JVM untuk menyembunyikan detail implementasi dari sistem operasi tersebut.

Pemetaan *thread* Java ke suatu sistem operasi tergantung pada implementasi JVM pada sistem operasi itu. Misalnya, Windows 2000 menggunakan model *one-to-one*, Solaris dulunya menggunakan model *many-to-one*, sedangkan Tru64 UNIX menggunakan model *many-to-many*.

12.7. Aplikasi *Thread* dalam Java

Dalam ilustrasi program yang ada pada subbab Bagian 12.5, “Pembatalan *Thread*”, kita tidak dapat mengetahui *thread* yang mana yang akan terlebih dahulu mengerjakan tugasnya. Hal ini terjadi karena ada dua *thread* yang berjalan secara paralel, yaitu *thread* utama dan *thread* t1. Artinya, keluaran dari program ini bisa bervariasi. Salah satu kemungkinan keluaran program ini adalah sebagai berikut:

```
Thread UTAMA
Ini CobaThread1
Ini CobaThread2
Thread UTAMA
Ini CobaThread2
Thread UTAMA
Ini CobaThread1
Ini CobaThread2
Thread UTAMA
Thread UTAMA
Ini CobaThread2
Ini CobaThread1
Thread UTAMA
Thread UTAMA
Ini CobaThread1
Thread UTAMA
```

12.8. Rangkuman

Setiap program Java memiliki paling sedikit satu *thread*. Bahasa pemrograman Java memungkinkan adanya pembuatan dan manajemen *thread* tambahan oleh JVM. Sebuah *thread* bisa berada di salah satu dari 4 status, yaitu *new*, *runnable*, *blocked*, dan *dead*. Ada dua cara untuk membuat *thread* dalam Java, yaitu dengan meng- *extends* kelas `Thread` dan dengan meng- *implements interface* `Runnable`.

Dalam beberapa kondisi, *thread* yang dibuat dapat digabungkan dengan *parent thread*-nya. Method `join()` berfungsi agar suatu *thread* menunggu *thread* yang dibuatnya menyelesaikan tugasnya terlebih dahulu, baru mulai mengeksekusi perintah selanjutnya.

Pembatalan thread secara *asynchronous* dilakukan dengan pemanggilan method `stop()`. Akan tetapi, cara ini terbukti tidak aman, sehingga untuk menterminasi *thread* digunakanlah *deferred cancellation*. Pembatalan dilakukan dengan pemanggilan method `interrupt()` untuk mengeset status interupsi, serta `isInterrupted()` atau `interrupted()` untuk memeriksa status interupsi tersebut.

Program Java dapat dijalankan di berbagai *platform* selama *platform* tersebut mendukung JVM. Pemetaan *thread* Java ke *host operating system* tergantung pada implementasi JVM di sistem operasi tersebut.

Rujukan

[Lewis1998] John Lewis dan William Loftus. 1998 . *Java Software Solutions Foundation Of Program Design*. First Edition. Addison Wesley.

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

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

[WEBJava2007] Java 2 Platform SE v1.3.1. 2007 . *Java 2 Platform SE v1.3.1: Class Thread – http://java.sun.com/j2se/1.3/docs/api/java/lang/Thread.html*. Diakses 27 Februari 2007.

[WEBJTPD2007] Java Thread Primitive Deprecation. 2007 . *Java Thread Primitive Deprecation – http://java.sun.com/j2se/1.3/docs/guide/misc/threadPrimitiveDeprecatio n.html*. Diakses 27 Februari 2007.

Bab 13. Konsep Penjadwalan

13.1. Pendahuluan

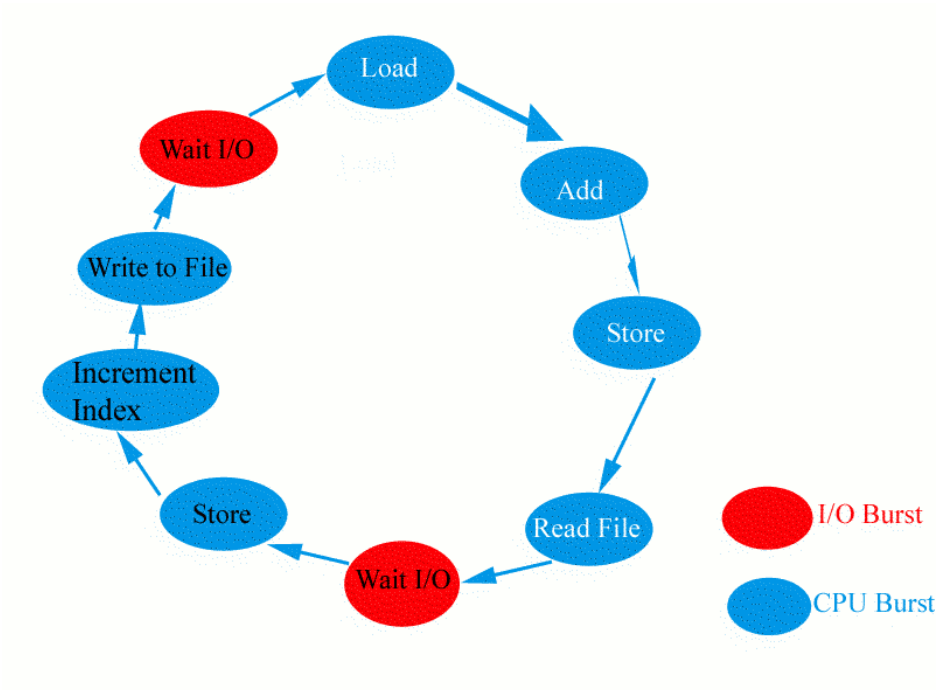
Kita mengenal istilah *multiprograming*, yang bertujuan untuk memaksimalkan penggunaan CPU dengan cara mengatur alokasi waktu yang digunakan oleh CPU, sehingga proses berjalan sepanjang waktu dan memperkecil waktu *idle*. Akibatnya sistem operasi dapat membuat komputer lebih produktif. Oleh karena itu perlu adanya penjadwalan proses-proses yang ada pada sistem.

Penjadwalan CPU adalah suatu proses pengaturan atau penjadwalan proses-proses yang ada di dalam komputer. Dimana proses-proses tersebut berjalan dalam pola yang disebut Siklus *Burst* yang akan dijelaskan pada bab ini. Penjadwalan CPU secara garis besar dibagi menjadi 2, yaitu Penjadwalan *Preemptive* dan Penjadwalan *Non Preemptive*. Bab ini juga akan memaparkan mengenai kriteria yang baik bagi suatu konsep penjadwalan dan penjelasan mengenai *dispatcher*, yaitu suatu komponen yang turut terlibat dalam penjadwalan. Penjadwalan sangat penting dalam menentukan *performance* sebuah komputer karena mengatur alokasi *resource* dari CPU untuk menjalankan proses-proses di dalam komputer. Penjadwalan CPU merupakan suatu konsep dasar dari *multiprograming*, karena dengan adanya penjadwalan dari CPU itu sendiri maka proses-proses tersebut akan mendapatkan alokasi *resource* dari CPU.

13.2. Siklus *Burst CPU*– M/K

Keberhasilan dari penjadwalan CPU tergantung dari beberapa properti prosesor. Pengeksekusian dari proses tersebut terdiri atas siklus CPU ekskusi dan M/K *Wait*. Proses hanya akan bolak-balik dari dua *state* ini, inilah yang disebut Siklus *Burst* CPU-M/K. Pengeksekusian proses dimulai dengan *Burst* CPU, setelah itu diikuti olehi *Burst* M/K, kemudian *Burst* CPU lagi lalu *Burst* M/K lagi, dan seterusnya dilakukan secara bergiliran. *Burst* CPU terakhir akan berakhir dengan permintaan sistem untuk mengakhiri pengeksekusian melalui *Burst* M/K lagi. Kejadian siklus *Burst* akan dijelaskan pada Figure 13.1.

Gambar 13.1. Siklus Burst



Dari gambar dapat kita lihat bahwa *Burst* CPU time yang lama jarang terjadi karena kebanyakan proses akan melakukan antara *output* ke layar atau *file*, maupun meminta *input* dari *user* atau *file* sehingga situasi dimana proses bekerja dengan *memory* dan register dalam jangka waktu lama tidak terlalu banyak ditemukan.

13.3. Penjadwalan *Preemptive*

Penjadwalan CPU mungkin akan dijalankan ketika proses dalam keadaan:
1. Berubah dari *running* ke *waiting state*.
2. Berubah dari *running* ke *ready state*.
3. Berubah dari *waiting* ke *ready state*.
4. *Dihentikan*.

Penjadwalan *Preemptive* mempunyai arti kemampuan sistem operasi untuk memberhentikan sementara proses yang sedang berjalan untuk memberi ruang kepada proses yang prioritasnya lebih tinggi. Penjadwalan ini bisa saja termasuk penjadwalan proses atau M/K. Penjadwalan *Preemptive* memungkinkan sistem untuk lebih bisa menjamin bahwa setiap proses mendapat sebuah *slice* waktu operasi. Dan juga membuat sistem lebih cepat merespon terhadap *event* dari luar (contohnya seperti ada data yang masuk) yang membutuhkan reaksi cepat dari satu atau beberapa proses. Membuat penjadwalan yang *Preemptive* mempunyai keuntungan yaitu sistem lebih responsif daripada sistem yang memakai penjadwalan *Non Preemptive*.

Dalam waktu-waktu tertentu, proses dapat dikelompokkan ke dalam dua kategori: proses yang memiliki *Burst* M/K yang sangat lama disebut *I/O Bound*, dan proses yang memiliki *Burst* CPU yang sangat lama disebut *CPU Bound*. Terkadang juga suatu sistem mengalami kondisi yang disebut *busywait*, yaitu saat dimana sistem menunggu *request input*(seperti *disk*, *keyboard*, atau jaringan). Saat *busywait* tersebut, proses tidak melakukan sesuatu yang produktif, tetapi tetap memakan *resource* dari CPU. Dengan penjadwalan *Preemptive*, hal tersebut dapat dihindari.

Dengan kata lain, penjadwalan *Preemptive* melibatkan mekanisme interupsi yang menyela proses yang sedang berjalan dan memaksa sistem untuk menentukan proses mana yang akan dieksekusi selanjutnya.

Penjadwalan nomor 1 dan 4 bersifat *Non Preemptive* sedangkan lainnya *Preemptive*. Penjadwalan yang biasa digunakan sistem operasi dewasa ini biasanya bersifat *Preemptive*. Bahkan beberapa penjadwalan sistem operasi, contohnya *Linux 2.6*, mempunyai kemampuan *Preemptive* terhadap *system call*-nya (*preemptible kernel*). *Windows 95, Windows XP, Linux, Unix, AmigaOS, MacOS X, dan Windows NT* adalah beberapa contoh sistem operasi yang menerapkan penjadwalan *Preemptive*.

Lama waktu suatu proses diizinkan untuk dieksekusi dalam penjadwalan *Preemptive* disebut *time slice/quantum*. Penjadwalan berjalan setiap satu satuan *time slice* untuk memilih proses mana yang akan berjalan selanjutnya. Bila *time slice* terlalu pendek maka penjadwal akan memakan terlalu banyak waktu proses, tetapi bila *time slice* terlau lama maka memungkinkan proses untuk tidak dapat merespon terhadap *event* dari luar secepat yang diharapkan.

13.4. Penjadwalan *Non Preemptive*

Penjadwalan *Non Preemptive* ialah salah satu jenis penjadwalan dimana sistem operasi tidak pernah melakukan *context switch* dari proses yang sedang berjalan ke proses yang lain. Dengan kata lain, proses yang sedang berjalan tidak bisa di- *interupt*.

Penjadwalan *Non Preemptive* terjadi ketika proses hanya:
1. Berjalan dari *running state* sampai *waiting state*.
2. Dihentikan.

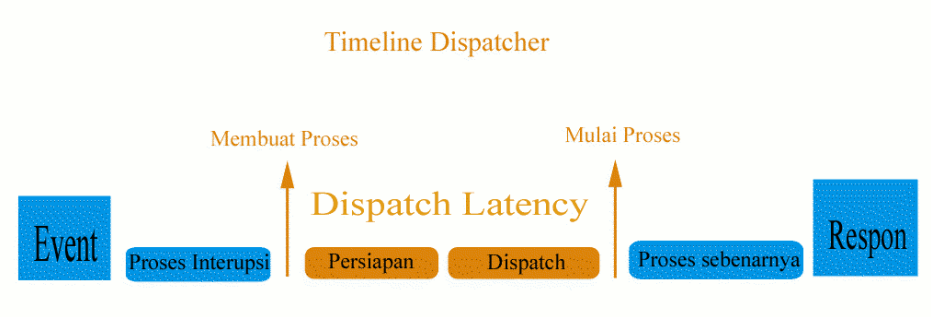
Ini berarti CPU menjaga proses sampai proses itu pindah ke *waiting state* ataupun dihentikan (proses tidak diganggu). Metode ini digunakan oleh *Microsoft Windows 3.1 dan Macintosh*. Ini adalah metode yang dapat digunakan untuk *platforms hardware* tertentu, karena tidak memerlukan perangkat keras khusus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

13.5. *Dispatcher*

Komponen yang lain yang terlibat dalam penjadwalan CPU adalah *dispatcher*. *Dispatcher* adalah modul yang memberikan kontrol CPU kepada proses yang sedang terjadwal. Fungsinya adalah:

1. ***Context switching .*** Mengganti state dari suatu proses dan mengembalikannya untuk menghindari monopoli *CPU time*. *Context switching* dilakukan untuk menangani suatu *interrupt*(misalnya menunggu waktu M/K). Untuk menyimpan *state* dari proses-proses yang terjadwal sebuah *Process Control Block* harus dibuat untuk mengingat proses-proses yang sedang diatur *scheduler*. Selain *state* suatu proses, *PCB* juga menyimpan *process ID*, *program counter*(posisi saat ini pada program), prioritas proses dan data-data tambahan lainnya.
2. ***Switching to user mode* dari *kernel mode.***
3. **Lompat dari suatu bagian di progam *user* untuk mengulang program.**

Gambar 13.2. Dispatch Latency



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

13.6. Kriteria Penjadwalan

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

Kriteria yang digunakan dalam menilai adalah:

1. ***CPU Utilization .*** Kita ingin menjaga CPU sesibuk mungkin. *CPU utilization* akan mempunyai *range* dari 0 sampai 100 persen. Di sistem yang sebenarnya ia mempunyai *range* dari 40 sampai 100 persen.
2. ***Throughput .*** Salah satu ukuran kerja adalah banyaknya proses yang diselesaikan per satuan waktu. Jika kita mempunyai beberapa proses yang sama dan memiliki beberapa algoritma penjadwalan yang berbeda, *throughput* bisa menjadi salah satu kriteria penilaian, dimana algoritma yang menyelesaikan proses terbanyak mungkin yang terbaik.
3. ***Turnaround Time .*** Dari sudut pandang proses tertentu, kriteria yang penting adalah berapa lama untuk mengeksekusi proses tersebut. Memang, lama pengeksekusian sebuah proses sangat tergantung dari hardware yang dipakai, namun kontribusi algoritma penjadwalan tetap ada dalam lama waktu yang dipakai untuk menyelesaikan sebuah proses. Misal kita memiliki sistem komputer yang identik dan proses-proses yang identik pula, namun kita memakai algoritma yang berbeda, algoritma yang mampu menyelesaikan proses yang sama dengan waktu yang lebih singkat mungkin lebih baik dari algoritma yang lain. Interval waktu yang diijinkan dengan waktu yang dibutuhkan untuk menyelesaikan sebuah proses disebut *turnaround time*. *Turnaround time* adalah jumlah periode untuk menunggu untuk dapat ke memori, menunggu di *ready queue*, eksekusi CPU, dan melakukan operasi M/K.
4. ***Waiting Time .*** Algoritma penjadwalan CPU tidak mempengaruhi waktu untuk melaksanakan proses tersebut atau M/K, itu hanya mempengaruhi jumlah waktu yang dibutuhkan proses di antrian *ready*. *Waiting time* adalah jumlah waktu yang dibutuhkan proses di antrian *ready*.
5. ***Response Time .*** Di sistem yang interaktif, *turnaround time* mungkin bukan waktu yang terbaik untuk kriteria. Sering sebuah proses dapat memproduksi *output* di awal, dan dapat meneruskan hasil

yang baru sementara hasil yang sebelumnya telah diberikan ke pengguna. Ukuran lain adalah waktu dari pengiriman permintaan sampai respon yang pertama diberikan. Ini disebut *response time*, yaitu waktu untuk memulai memberikan respon, tetapi bukan waktu yang dipakai *output* untuk respon tersebut.

6. ***Fairness .*** Suatu algoritma harus memperhatikan pengawasan nilai prioritas dari suatu proses (menghindari terjadinya *starvation CPU time*).
7. **Efisiensi.** Rendahnya *overhead* dalam *context switching*, penghitungan prioritas dan sebagainya menentukan apakah suatu algoritma efisien atau tidak.

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

13.7. Rangkuman

Penjadwalan CPU adalah pemilihan proses dari *antrian ready* untuk dapat dieksekusi. Penjadwalan CPU merupakan konsep dari *multiprogramming*, dimana CPU digunakan secara bergantian untuk proses yang berbeda. Suatu proses terdiri dari dua siklus yaitu *Burst* M/K dan *Burst* CPU yang dilakukan bergantian hingga proses selesai. Penjadwalan CPU mungkin dijalankan ketika proses:

1. *running* ke *waiting time*
2. *running* ke *ready state*
3. *waiting* ke *ready state*
4. *terminates*

Proses 1 dan 4 adalah proses *Non Preemptive*, dimana proses tersebut tidak bisa di- *interrupt*, sedangkan 2 dan 3 adalah proses *Preemptive*, dimana proses boleh di *interrupt*.

Komponen yang lain dalam penjadwalan CPU adalah *dispatcher*, *dispatcher* adalah modul yang memberikan kendali CPU kepada proses. Waktu yang diperlukan oleh *dispatcher* untuk menghentikan suatu proses dan memulai proses yang lain disebut dengan *dispatch latency*.

Jika dalam suatu proses *Burst* CPU jauh lebih besar daripada *Burst* M/K maka disebut *CPU Bound*. Demikian juga sebaliknya disebut dengn *M/K Bound*.

Dalam menilai baik atau buruknya suatu algoritma penjadwalan kita bisa memakai beberapa kriteria, diantaranya *CPU utilization*, *throughput*, *turnaround time*, *waiting time*, dan *response time*. Algoritma yang baik adalah yang mampu memaksimalkan *CPU utilization* dan *throughput*, dan meminimalkan *turnaround time*, *waiting time*, dan *response time*.

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 .

[WEBWiki2007a] From Wikipedia, the free encyclopedia. 2007 . *Dispatcher – http://en.wikipedia.org/wiki/Dispatcher*. Diakses 20 Februari 2007.

[WEBWiki2007b] From Wikipedia, the free encyclopedia. 2007 . *Preemptive multitasking – http://en.wikipedia.org/wiki/Pre-emptive_multitasking*. Diakses 20 Februari 2007.

[WEBDCU2007] From Dublin City University homepage. 2007 . *Processes – http://computing.dcu.ie/~HUMPHRYS/Notes/OS/processes.html*. Diakses 20 Februari 2007.

Bab 14. Algoritma Penjadwalan

14.1. Pendahuluan

Penjadwalan berkaitan dengan permasalahan memutuskan proses mana yang akan dilaksanakan dalam suatu sistem. Proses yang belum mendapat jatah alokasi dari CPU akan mengantri di *ready queue*. Algoritma penjadwalan berfungsi untuk menentukan proses manakah yang ada di *ready queue* yang akan dieksekusi oleh CPU. Bagian berikut ini akan memberikan ilustrasi beberapa algoritma penjadwalan.

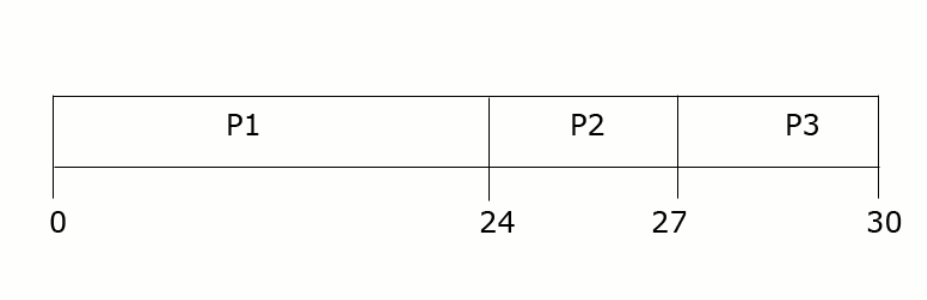
14.2. *FCFS (First Come First Served)*

Algoritma ini merupakan algoritma penjadwalan yang paling sederhana yang digunakan CPU. Dengan menggunakan algoritma ini setiap proses yang berada pada status *ready* dimasukkan kedalam *FIFO queue* atau antrian dengan prinsip *first in first out*, sesuai dengan waktu kedatangannya. Proses yang tiba terlebih dahulu yang akan dieksekusi.

Contoh

Ada tiga buah proses yang datang secara bersamaan yaitu pada 0 ms, P1 memiliki burst time 24 ms, P2 memiliki burst time 3 ms, dan P3 memiliki burst time 3 ms. Hitunglah *waiting time* rata-rata dan *turnaround time*(*burst time + waiting time*) dari ketiga proses tersebut dengan menggunakan algoritma FCFS. *Waiting time* untuk P1 adalah 0 ms (P1 tidak perlu menunggu), sedangkan untuk P2 adalah sebesar 24 ms (menunggu P1 selesai), dan untuk P3 sebesar 27 ms (menunggu P1 dan P2 selesai).

Gambar 14.1. Gantt Chart Kedatangan Proses



Urutan kedatangan adalah P1, P2 , P3; *gantt chart* untuk urutan ini adalah:

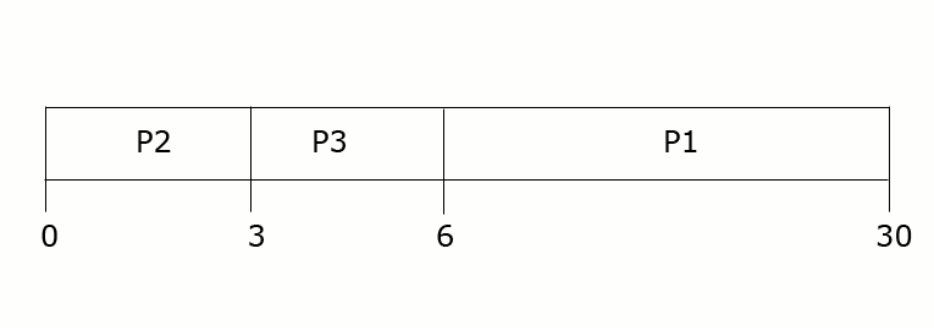
Waiting time rata-ratanya adalah sebesar(0+24+27)/3 = 17ms. *Turnaround time* untuk P1 sebesar 24 ms, sedangkan untuk P2 sebesar 27 ms (dihitung dari awal kedatangan P2 hingga selesai dieksekusi), untuk P3 sebesar 30 ms. *Turnaround time* rata-rata untuk ketiga proses tersebut adalah (24+27+30)/3 = 27 ms.

Kelemahan dari algoritma ini:

1. *Waiting time* rata-ratanya cukup lama.
2. Terjadinya *convoy effect*, yaitu proses-proses menunggu lama untuk menunggu 1 proses besar yang sedang dieksekusi oleh CPU. Algoritma ini juga menerapkan konsep non-preemptive, yaitu setiap proses yang sedang dieksekusi oleh CPU tidak dapat di-interrupt oleh proses yang lain.

Misalkan proses dibalik sehingga urutan kedatangan adalah P3, P2, P1. *Waiting time* adalah P1=6; P2=3; P3=0. *Average waiting time*: (6+3+0)/3=3.

Gambar 14.2. Gantt Chart Kedatangan Proses Sesudah Urutan Kedatangan Dibalik



14.3. *SJF (Shortest Job First)*

Pada algoritma ini setiap proses yang ada di *ready queue* akan dieksekusi berdasarkan *burst time* terkecil. Hal ini mengakibatkan *waiting time* yang pendek untuk setiap proses dan karena hal tersebut maka *waiting time* rata-ratanya juga menjadi pendek, sehingga dapat dikatakan bahwa algoritma ini adalah algoritma yang optimal.

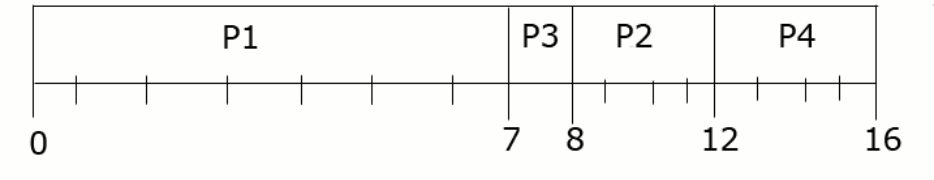
Tabel 14.1. Contoh Shortest Job First

Process	***Arrival Time***	***Burst Time***
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

Contoh: Ada 4 buah proses yang datang berurutan yaitu P1 dengan *arrival time* pada 0.0 ms dan *burst time* 7 ms, P2 dengan *arrival time* pada 2.0 ms dan *burst time* 4 ms, P3 dengan *arrival time* pada 4.0 ms dan *burst time* 1 ms, P4 dengan *arrival time* pada 5.0 ms dan *burst time* 4 ms. Hitunglah *waiting time* rata-rata dan *turnaround time* dari keempat proses tersebut dengan mengunakan algoritma SJF.

Average waiting time rata-rata untuk ketiga proses tersebut adalah sebesar (0 +6+3+7)/4=4 ms.

Gambar 14.3. Shortest Job First (Non-Preemptive)



Average waiting time rata-rata untuk ketiga prses tersebut adalah sebesar (9+1+0+2)/4=3 ms.

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

Algoritma ini dapat dibagi menjadi dua bagian yaitu :

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

14.4. *Priority Scheduling*

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

Prioritas suatu proses dapat ditentukan melalui beberapa karakteristik antara lain:

1. *Time limit*.
2. *Memory requirement*.
3. Akses file.
4. Perbandingan antara *burst* M/K dengan *CPU burst*.
5. Tingkat kepentingan proses.

Priority scheduling juga dapat dijalankan secara *preemptive* maupun *non-preemptive*. Pada *preemptive*, jika ada suatu proses yang baru datang memiliki prioritas yang lebih tinggi daripada proses yang sedang dijalankan, maka proses yang sedang berjalan tersebut dihentikan, lalu CPU dialihkan untuk proses yang baru datang tersebut. Sementara itu, pada *non-preemptive*, proses yang baru datang tidak dapat menganggu proses yang sedang berjalan, tetapi hanya diletakkan di depan *queue*.

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

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

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

14.5. *Round Robin*

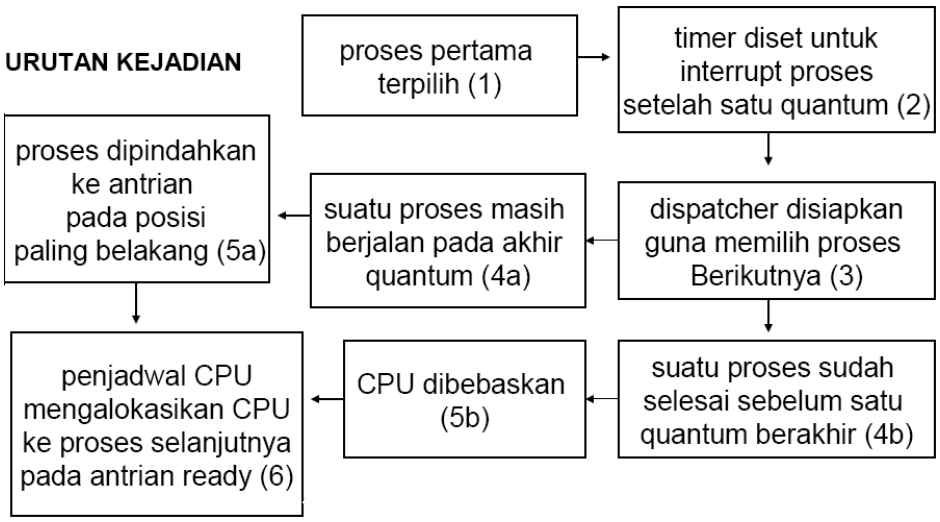
Algoritma ini menggilir proses yang ada di antrian. Proses akan mendapat jatah sebesar *time quantum*. Jika *time quantum*-nya habis atau proses sudah selesai, CPU akan dialokasikan ke proses berikutnya. Tentu proses ini cukup adil karena tak ada proses yang diprioritaskan, semua proses mendapat jatah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

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

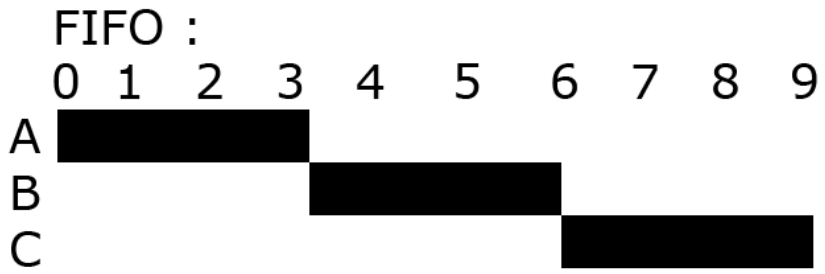
Permasalahan utama pada *Round Robin* adalah menentukan besarnya *time quantum*. Jika time *quantum* yang ditentukan terlalu kecil, maka sebagian besar proses tidak akan selesai dalam 1 *quantum*. Hal ini tidak baik karena akan terjadi banyak *switch*, padahal CPU memerlukan waktu untuk beralih dari suatu proses ke proses lain (disebut dengan context switches time). Sebaliknya, jika time quantum terlalu

besar, algoritma *Round Robin* akan berjalan seperti algoritma *first come first served*. *Time quantum* yang ideal adalah jika 80% dari total proses memiliki CPU burst time yang lebih kecil dari 1 *time quantum*.

Gambar 14.4. Urutan Kejadian Algoritma Round Robin



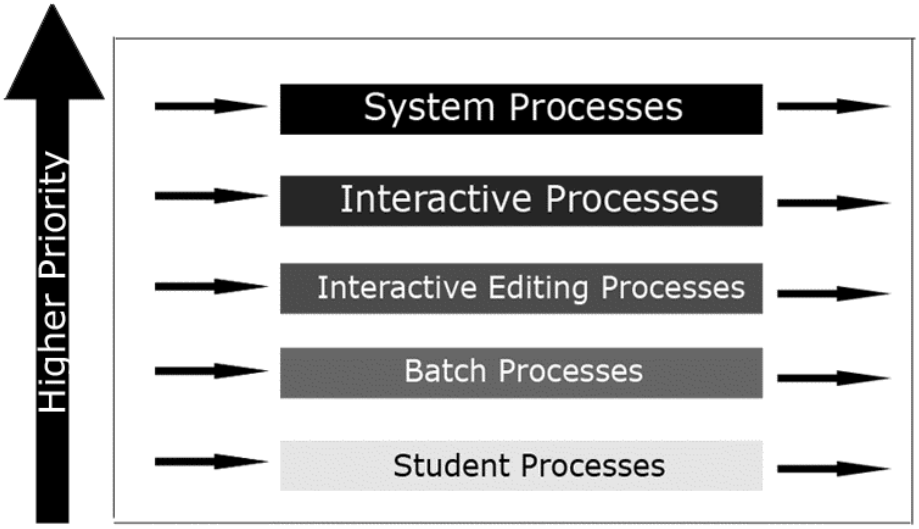
Gambar 14.5. Penggunaan Waktu Quantum



14.6. *Multilevel Queue*

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

Gambar 14.6. Multilevel Queue



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

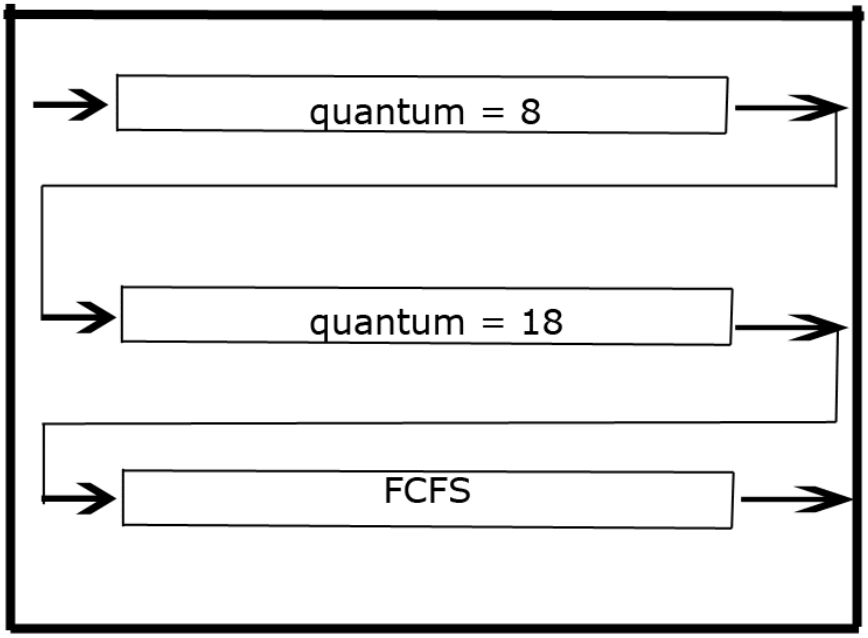
Dalam hal ini, dapat dilihat bahwa seolah-olah algoritma dengan prioritas yang dasar adalah *algoritma multilevel queue* dimana setiap *queue* akan berjalan dengan algoritma FCFS yang memiliki banyak kelemahan. Oleh karena itu, dalam prakteknya, *algoritma multilevel queue* memungkinkan adanya penerapan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

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

14.7. *Multilevel Feedback Queue*

Algoritma ini mirip sekali dengan algoritma *multilevel queue*. Perbedaannya ialah algoritma ini mengizinkan proses untuk pindah antrian. Jika suatu proses menyita CPU terlalu lama, maka proses itu akan dipindahkan ke antrian yang lebih rendah. Hal 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 M/K dapat terus sibuk. Semakin rendah tingkatannya, panjang CPU burst proses juga semakin besar.

Gambar 14.7. Multilevel Feedback Queue



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

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

Multilevel feedback queue adalah salah satu algoritma yang berdasar pada algoritma *multilevel queue*. Perbedaan mendasar yang membedakan *multilevel feedback queue* dengan *multilevel queue* biasa adalah terletak pada adanya kemungkinan suatu proses berpindah dari satu antrian ke antrian lainnya, entah dengan prioritas yang lebih rendah ataupun lebih tinggi, misalnya pada contoh berikut.

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

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

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

Contoh: Terdapat tiga antrian; Q1=10 ms, FCFS Q2=40 ms, FCFS Q3=FCFS proses yang masuk, masuk ke antrian Q1. Jika dalam 10 ms tidak selesai, maka proses tersebut dipindahkan ke Q2. Jika dalam 40 ms tidak selesai, maka dipindahkan lagi ke Q3. Berdasarkan hal-hal di atas maka algoritma

ini dapat digunakan secara fleksibel dan diterapkan sesuai dengan kebutuhan sistem. Pada zaman sekarang ini algoritma *multilevel feedback queue* adalah salah satu yang paling banyak digunakan.

14.8. Rangkuman

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

a. *First Come First Serve*. Algoritma ini mendahulukan proses yang lebih dulu datang. Kelemahannya, *waiting time* rata-rata cukup lama.
b. *Shortest Job First*. Algoritma ini mendahulukan proses dengan *CPU burst* terkecil sehingga akan mengurangi *waiting time* rata-rata.
c. *Priority Scheduling*. Algoritma ini mendahulukan prioritas terbesar. Kelemahannya, prioritas kecil tidak mendapat jatah CPU Hal ini dapat diatasi dengan *aging* yaitu semakin lama lama menunggu, prioritas semakin tinggi.
d. *Round Robin*. Algoritma ini menggilir proses-proses yang ada diantrian dengan jatah *time quantum* yang sama. Jika waktu habis, CPU dialokasikan ke proses selanjutnya.
e. *Multilevel Queue*. Algoritma ini membagi beberapa antrian yang akan diberi prioritas berdasarkan tingkatan. Tingkatan lebih tinggi menjadi prioritas utama.
f. *Multilevel Feedback Queue*. Pada dasarnya sama dengan *Multilevel Queue*, yang membedakannya adalah pada algoritma ini diizinkan untuk pindah antrian.

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

Rujukan

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

[Scheduling] Moonbase. 1991 . *Scheduling– http://moonbase.wwc.edu/~aabyan/352/Scheduling.html*. Diakses 20 Februari 2007.

Bab 15. Penjadwalan Prosesor Jamak

15.1. Pendahuluan

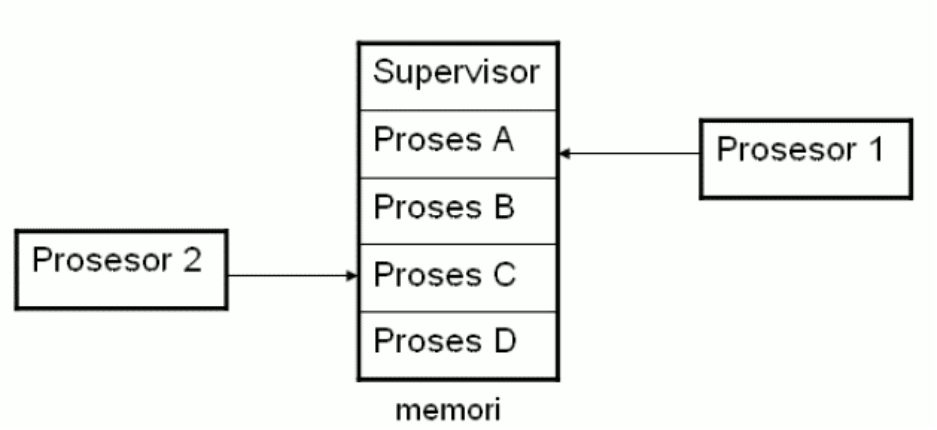
Untuk mempertinggi kinerja, kehandalan, kemampuan komputasi, paralelisme, dan keekonomisan dari suatu sistem, tambahan prosesor dapat diimplementasikan ke dalam sistem tersebut. Sistem seperti ini disebut dengan sistem yang bekerja dengan banyak prosesor (prosesor jamak atau *multiprocessor*). Seperti halnya pada prosesor tunggal, prosesor jamak juga membutuhkan penjadwalan. Namun pada prosesor jamak, penjadwalannya jauh lebih kompleks daripada prosesor tunggal karena pada prosesor jamak memungkinkan adanya *load sharing* antar prosesor yang menyebabkan penjadwalan menjadi lebih kompleks namun kemampuan sistem tersebut menjadi lebih baik. Oleh karena itu, kita perlu mempelajari penjadwalan pada prosesor jamak berhubung sistem dengan prosesor jamak akan semakin banyak digunakan karena kemampuannya yang lebih baik dari sistem dengan prosesor tunggal. Ada beberapa jenis dari sistem prosesor jamak, namun yang akan dibahas dalam bab ini adalah penjadwalan pada sistem prosesor jamak yang memiliki fungsi- fungsi prosesor yang identik (*homogenous*).

15.2. Penjadwalan *Master/Slave*

Pendekatan pertama untuk penjadwalan prosesor jamak adalah penjadwalan *asymmetric multiprocessing* atau bisa disebut juga sebagai penjadwalan *master/slave*. Dimana pada metode ini hanya satu prosesor(*master*) yang menangani semua keputusan penjadwalan, pemrosesan M/K, dan aktivitas sistem lainnya dan prosesor lainnya (*slave*) hanya mengeksekusi proses. Metode ini sederhana karena hanya satu prosesor yang mengakses struktur data sistem dan juga mengurangi data *sharing*.

Dalam teknik penjadwalan *master/slave*, satu prosesor menjaga status dari semua proses dalam sistem dan menjadwalkan kinerja untuk semua prosesor *slave*. Sebagai contoh, prosesor *master* memilih proses yang akan dieksekusi, kemudian mencari prosesor yang *available*, dan memberikan instruksi *Start processor*. Prosesor *slave* memulai eksekusi pada lokasi memori yang dituju. Saat *slave* mengalami sebuah kondisi tertentu seperti meminta M/K, prosesor *slave* memberi interupsi kepada prosesor *master* dan berhenti untuk menunggu perintah selanjutnya. Perlu diketahui bahwa prosesor *slave* yang berbeda dapat ditujukan untuk suatu proses yang sama pada waktu yang berbeda.

Gambar 15.1. *Multiprogramming* dengan *multiprocessor*



Gambar di atas mengilustrasikan perilaku dari *multiprocessor* yang digunakan untuk *multiprogramming* Beberapa proses terpisah dialokasikan di dalam memori. Ruang alamat proses terdiri dari halaman-halaman sehingga hanya sebagian saja dari proses tersebut yang berada dalam memori pada satu waktu. Hal ini memungkinkan banyak proses dapat aktif dalam sistem.

15.3. Penjadwalan SMP

Penjadwalan SMP (*Symmetric multiprocessing*) adalah pendekatan kedua untuk penjadwalan prosesor jamak. Dimana setiap prosesor menjadwalkan dirinya sendiri (*self scheduling*). Semua proses mungkin berada pada antrian *ready* yang biasa, atau mungkin setiap prosesor memiliki antrian *ready* tersendiri. Bagaimanapun juga, penjadwalan terlaksana dengan menjadwalkan setiap prosesor untuk memeriksa antrian *ready* dan memilih suatu proses untuk dieksekusi. Jika suatu sistem prosesor jamak mencoba untuk mengakses dan meng- *update* suatu struktur data, penjadwal dari prosesor-prosesor tersebut harus diprogram dengan hati-hati; kita harus yakin bahwa dua prosesor tidak memilih proses yang sama dan proses tersebut tidak hilang dari antrian. Secara virtual, semua sistem operasi modern mendukung SMP, termasuk Windows XP, Windows 2000, Solaris, Linux, dan Mac OS X.

15.4. *Affinity* dan *Load Ballancing*

Affinity

Data yang paling sering diakses oleh beberapa proses akan memadati *cache* pada prosesor,sehingga akses memori yang sukses biasanya terjadi di memori *cache*. Namun, jika suatu proses . berpindah dari satu prosesor ke prosesor lainnya akan mengakibatkan isi dari *cache* memori yang dituju menjadi tidak valid, sedangkan *cache* memori dari prosesor asal harus disusun kembali populasi datanya. Karena mahalnya *invalidating* dan *re-populating* dari cache, kebanyakan sistem SMP mencoba untuk mencegah migrasi proses antar prosesor sehingga menjaga proses tersebut untuk berjalan di prosesor yang sama. Hal ini disebut afinitas prosesor (*processor affinity*).

Ada dua jenis afinitas prosesor, yakni:
- *Soft affinity* yang memungkinkan proses berpindah dari satu prosesor ke prosesor yang lain, dan
- *Hard affinity* yang menjamin bahwa suatu proses akan berjalan pada prosesor yang sama dan tidak berpindah. Contoh sistem yang menyediakan *system calls* yang mendukung *hard affinity* adalah Linux.

Load Balancing

Dalam sistem SMP, sangat penting untuk menjaga keseimbangan *workload* antara semua prosesor untuk memaksimalkan keuntungan memiliki *multiprocessor*. Jika tidak, mungkin satu atau lebih prosesor *idle* disaat prosesor lain harus bekerja keras dengan *workload* yang tinggi. *Load balancing* adalah usaha untuk menjaga *workload* terdistribusi sama rata untuk semua prosesor dalam sistem SMP. Perlu diperhatikan bahwa *load balancing* hanya perlu dilakukan pada sistem dimana setiap prosesor memiliki antrian tersendiri(*private queue*) untuk proses-proses yang berstatus *ready*. Pada sistem dengan antrian yang biasa (*common queue*), *load balancing* tidak diperlukan karena sekali prosesor menjadi *idle*, prosesor tersebut segera mengerjakan proses yang dapat dilaksanakan dari antrian biasa tersebut. Perlu juga diperhatikan bahwa pada sebagian besar sistem operasi kontemporer mendukung SMP, jadi setiap prosesor bisa memiliki *private queue*.

Ada dua jenis *load balancing*, yakni:
- *Push migration*, pada kondisi ini ada suatu *task* spesifik yang secara berkala memeriksa *load* dari tiap-tiap prosesor. Jika terdapat ketidakseimbangan, maka dilakukan perataan dengan memindahkan(*pushing*) proses dari yang kelebihan muatan ke prosesor yang *idle* atau yang memiliki muatan lebih sedikit.
- *Pull migration*, kondisi ini terjadi saat prosesor yang *idle* menarik(*pulling*) proses yang sedang menunggu dari prosesor yang sibuk.

Kedua pendekatan tersebut tidak harus *mutually exclusive* dan dalam kenyataannya sering diimplementasikan secara paralel pada sistem *load-balancing*.

Keuntungan dari *affinity* berlawanan dengan keuntungan dari *load balancing*, yaitu keuntungan menjaga suatu proses berjalan pada satu prosesor yang sama dimana proses dapat memanfaatkan data

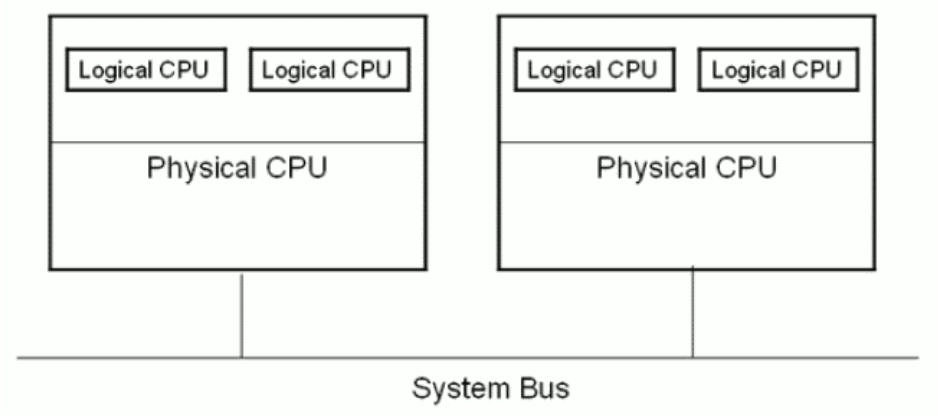
yang sudah ada pada memori *cache* prosesor tersebut berkebalikan dengan keuntungan menarik atau memindahkan proses dari satu prosesor ke prosesor lain. Dalam kasus *system engineering*, tidak ada aturan tetap keuntungan yang mana yang lebih baik. Walaupun pada beberapa sistem, prosesor *idle* selalu menarik proses dari prosesor *non-idle* sedangkan pada sistem yang lain, proses dipindahkan hanya jika terjadi ketidakseimbangan yang besar antara prosesor.

15.5. *Symetric Multithreading*

Sistem SMP mengizinkan beberapa thread untuk berjalan secara bersamaan dengan menyediakan banyak *physical processor*. Ada sebuah strategi alternatif yang lebih cenderung untuk menyediakan *logical processor* daripada *physical processor*. Strategi ini dikenal sebagai SMT (*Symetric Multithreading*). SMT juga biasa disebut teknologi *hyperthreading* dalam prosesor intel.

Ide dari SMT adalah untuk menciptakan banyak *logical processor* dalam suatu *physical processor* yang sama dan mempresentasikan beberapa prosesor kepada sistem operasi. Setiap *logical processor* mempunyai state arsitekturnya sendiri yang mencangkup *general purpose* dan *machine state register*. Lebih jauh lagi, setiap *logical prosesor* bertanggung jawab pada penanganan interupsinya sendiri, yang berarti bahwa interupsi cenderung dikirimkan ke *logical processor* dan ditangani oleh *logical processor* bukan *physical processor*. Dengan kata lain, setiap *logical processor* men- *share resource* dari *physical processor*-nya, seperti *chace* dan bus.

Gambar 15.2. *Symetric Multithreading*



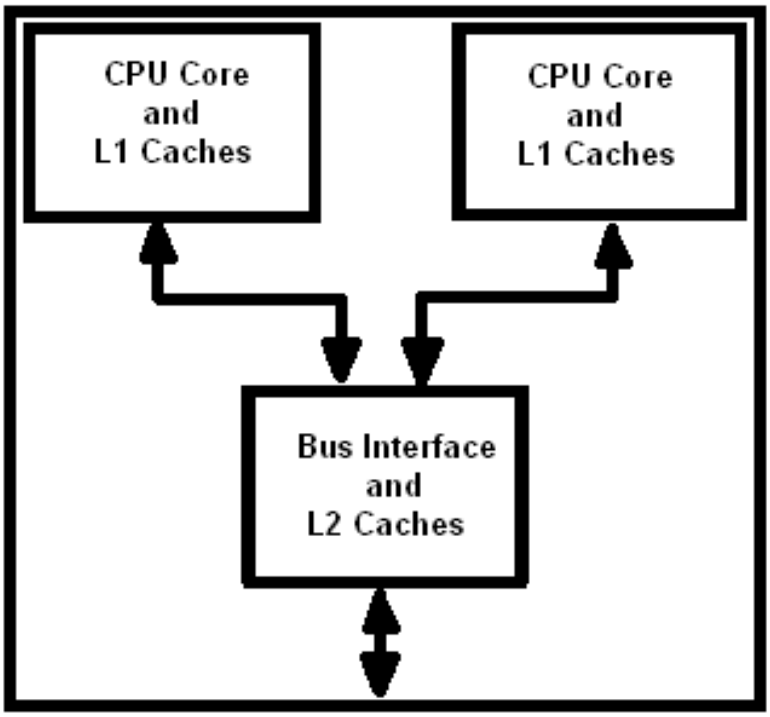
Gambar di atas mengilustrasikan suatu tipe arsitektur SMT dengan dua *physical processor* dengan masing-masing punya dua *logical processor*. Dari sudut pandang sistem operasi, pada sistem ini terdapat empat prosesor.

Perlu diketahui bahwa SMT adalah fitur yang disediakan dalam hardware, bukan software, sehingga hardware harus menyediakan representasi state arsitektur dari setiap *logical processor* sebagaimana representasi dari penanganan interupsinya. Sistem operasi tidak perlu didesain khusus jika berjalan pada sistem SMT, akan tetapi performa yang diharapkan tidak selalu terjadi pada sistem operasi yang berjalan pada SMT. Misalnya, suatu sistem memiliki 2 *physical processor*, keduanya *idle*, penjadwal pertama kali akan lebih memilih untuk membagi thread ke *physical processor* daripada membaginya ke *logical processor* dalam *physical processor* yang sama, sehingga *logical processor* pada satu *physical processor* bisa menjadi sibuk sedangkan *physical processor* yang lain menjadi *idle*.

15.6. *Multicore*

Multicore microprocessor adalah kombinasi dua atau lebih prosesor independen kedalam sebuah *integrated circuit*(IC). Umumnya, *multicore* mengizinkan perangkat komputasi untuk memeragakan suatu bentuk *thread level paralelism*(TLP) tanpa mengikutsertakan banyak prosesor terpisah. TLP lebih dikenal sebagai *chip-level multiprocessing*.

Gambar 15.3. Chip CPU *dual-core*



Keuntungan:

- Meningkatkan performa dari operasi *cache snoop (bus snooping)*. *Bus snooping* adalah suatu teknik yang digunakan dalam sistem pembagian memori terdistribusi dan *multiprocessor* yang ditujukan untuk mendapatkan koherensi pada *cache*. Hal ini dikarenakan sinyal antara CPU yang berbeda mengalir pada jarak yang lebih dekat, sehingga kekuatan sinyal hanya berkurang sedikit. Sinyal dengan kualitas baik ini memungkinkan lebih banyak data yang dikirimkan dalam satu periode waktu dan tidak perlu sering di- *repeat*.
- Secara fisik, desain CPU *multicore* menggunakan ruang yang lebih kecil pada PCB (*Printed Circuit Board*) dibanding dengan desain *multichip* SMP
- Prosesor *dual-core* menggunakan sumber daya lebih kecil dibanding sepasang prosesor *dual-core*
- Desain *multicore* memiliki resiko *design error* yang lebih rendah daripada desain *single-core*

Kerugian:

- Dalam hal sistem operasi, butuh penyesuaian kepada *software* yang ada untuk memaksimalkan kegunaan dari sumberdaya komputasi yang disediakan oleh prosesor *multicore*. Kemampuan prosesor *multicore* untuk meningkatkan performa aplikasi juga bergantung pada penggunaan banyaknya *thread* dalam aplikasi tersebut.
- Dari sudut pandang arsitektur, pemanfaatan daerah permukaan silikon dari desain *single-core* lebih baik daripada desain *multicore*.
- Pengembangan *chip multicore* membuat produksinya menjadi menurun karena semakin sulitnya pengaturan suhu pada chip yang padat.

Pengaruh *multicore* terhadap *software*

Keuntungan software dari arsitektur *multicore* adalah kode-kode dapat dieksekusi secara paralel. Dalam sistem operasi, kode-kode tersebut dieksekusi dalam thread-thread atau proses-proses yang terpisah. Setiap aplikasi pada sistem berjalan pada prosesnya sendiri sehingga aplikasi paralel akan mendapatkan keuntungan dari arsitektur *multicore*. Setiap aplikasi harus tertulis secara spesifik untuk memaksimalkan penggunaan dari banyak thread.

Banyak aplikasi software tidak dituliskan dengan menggunakan thread-thread yang *concurrent* karena kesulitan dalam pembuatannya. *Concurrency* memegang peranan utama dalam aplikasi paralel yang sebenarnya.

Langkah-langkah dalam mendesain aplikasi paralel adalah sebagai berikut:

- ***Partitioning .*** Tahap desain ini dimaksudkan untuk membuka peluang awal pengeksekusian secara paralel. Fokus dari tahap ini adalah mempartisi sejumlah besar tugas dalam ukuran kecil dengan tujuan menguraikan suatu masalah menjadi butiran-butiran kecil.
- ***Communication .*** Tugas-tugas yang telah terpartisi diharapkan dapat langsung dieksekusi secara paralel tapi tidak bisa, karena pada umumnya eksekusi berjalan secara independen. Pelaksanaan komputasi dalam satu tugas membutuhkan asosiasi data antara masing-masing tugas. Data kemudian harus berpindah-pindah antar tugas dalam melangsungkan komputasi. Aliran informasi inilah yang dispesifikasi dalam fase *communication*.
- ***Agglomeration .*** Pada tahap ini kita pindah dari sesuatu yang abstrak ke yang konkret. Kita tinjau kembali kedua tahap diatas dengan tujuan untuk mendapatkan algoritma pengeksekusian yang lebih efisien. Kita pertimbangkan juga apakah perlu untuk menggumpalkan (*agglomerate*) tugas-tugas pada fase *partition* menjadi lebih sedikit, dengan masing-masing tugas berukuran lebih besar.
- ***Mapping .*** Dalam tahap yang keempat dan terakhir ini, kita menspesifikasi dimana setiap tugas akan dieksekusi. Masalah *mapping* ini tidak muncul pada *uniprocessor* yang menyediakan penjadwalan tugas.

Pada sisi server, prosesor *multicore* menjadi ideal karena server mengizinkan banyak user untuk melakukan koneksi ke server secara simultan. Oleh karena itu, Web server dan application server mempunyai *throughput* yang lebih baik.

15.7. Rangkuman

Penjadwalan *asymmetric multiprocessing* atau penjadwalan *master/slave* menangani semua keputusan penjadwalan, pemrosesan M/K, dan aktivitas sistem lainnya hanya dengan satu prosesor (*master*). Dan prosesor lainnya (*slave*) hanya mengeksekusi proses.

SMP (*Symmetric multiprocessing*) adalah pendekatan kedua untuk penjadwalan prosesor jamak. Dimana setiap prosesor menjadwal dirinya sendiri (self scheduling).

Load balancing adalah usaha untuk menjaga *workload* terdistribusi sama rata untuk semua prosesor dalam sistem SMP. *Load balancing* hanya perlu untuk dilakukan pada sistem dimana setiap prosesor memiliki antrian tersendiri(*private queue*) untuk proses-proses yang akan dipilih untuk dieksekusi. Ada dua jenis *load balancing*: *push migration* dan *pull migration*. Pada *push migration*, ada suatu *task* spesifik yang secara berkala memeriksa *load* dari tiap-tiap prosesor, jika terdapat ketidakseimbangan, maka dilakukan perataan dengan memindahkan(*pushing*) proses dari yang kelebihan muatan ke prosesor yang *idle* atau yang memiliki muatan lebih sedikit. *Pull migration* terjadi saat prosesor yang *idle* menarik (*pulling*) proses yang sedang menunggu dari prosesor yang sibuk. Kedua pendekatan tersebut tidak harus *mutually exclusive* dan dalam kenyataannya sering diimplementasikan secara paralel pada sistem *load-balancing*.

Afinitas prosesor (*processor affinity*) adalah pencegahan migrasi proses antar prosesor sehingga menjaga proses tersebut tetap berjalan di prosesor yang sama. Ada dua jenis afinitas prosesor, yakni *soft affinity* dan *hard affinity*. Pada situasi *soft affinity* ada kemungkinan proses berpindah dari satu prosesor ke prosesor yang lain. Sedangkan *hard affinity* menjamin bahwa suatu proses akan berjalan pada prosesor yang sama dan tidak berpindah. Contoh sistem yang menyediakan *system calls* yang mendukung *hard affinity* adalah Linux.

SMT (*Symetric Multithreading*) adalah strategi alternatif untuk menjalankan beberapa *thread* secara bersamaan. SMT juga biasa disebut teknologi *hyperthreading* dalam prosesor intel.

Multicore microprocessor adalah kombinasi dua atau lebih *independent* processor kedalam sebuah *integrated circuit*(IC). *Multicore microprocessosor* mengizinkan perangkat komputasi untuk memeragakan suatu bentuk *thread level parallelism*(TLP) tanpa mengikutsertakan banyak *microprocessor* pada paket terpisah. TLP lebih dikenal sebagai *chip-level multiprocessing*.

Rujukan

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

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

[WEBWiki2007] Wikipedia. 2007 . *Multicore(Computing) – http://en.wikipedia.org/wiki/Multicore*. Diakses 27 Februari 2007.

Bab 16. Evaluasi dan Ilustrasi

16.1. Pendahuluan

Seperti yang sudah dijelaskan di bab sebelumnya, terdapat banyak algoritma penjadwalan CPU. Pertanyaan yang muncul selanjutnya adalah bagaimana memilih algoritma penjadwal untuk suatu sistem. Untuk memilihnya, pertama kita harus mengetahui kriteria algoritma penjadwal yang baik. Kriteria ini bisa mencakup hal berikut:

- Memaksimalkan penggunaan CPU dengan batasan *response time* maksimum adalah satu detik.
- Memaksimalkan *throughput* sehingga rata-rata *turnaround time* berbanding lurus dengan total waktu eksekusi.

Dengan mempertimbangkan kriteria itu, kita bisa mengevaluasi algoritma penjadwalan dengan berbagai metode-metode yang akan dijelaskan dalam bab ini.

16.2. *Deterministic Modelling*

Salah satu metode evaluasi adalah evaluasi analitik, yaitu menggunakan algoritma yang sudah ada dan *workload* dari sistem untuk menghasilkan suatu formula yang menggambarkan *performance* suatu algoritma. Salah satu tipe evaluasi analitik ini adalah *deterministic modelling*. Model ini menggunakan *workload* yang sudah diketahui dan melihat *performance* algoritma dalam menjalankan *workload* tersebut. Untuk dapat memahami *deterministic modelling* ini, kita gunakan sebuah contoh. Misalkan lima proses datang pada waktu yang bersamaan dengan urutan sebagai berikut:

Tabel 16.1. Contoh

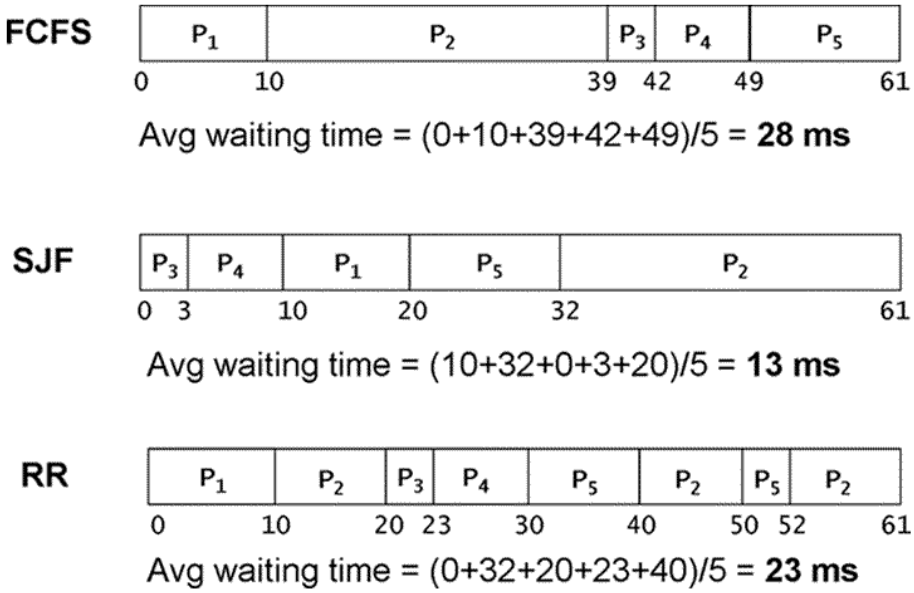
Proses	Burst Time
P1	10
P2	29
P3	3
P4	7
P5	12

Dengan *deterministic modelling* kita membandingkan *performance* algoritma dalam mengekseskusi proses-proses di atas. Eksekusi proses dengan algoritma *First Come First Serve (FCFS), Shortest Job First (SJF),* dan *Round Robin (RR)* digambarkan pada Gambar 16.1, “Perbandingan dengan Deterministic Modelling”.

Dari *average waiting time*-nya, dapat dilihat bahwa untuk kasus ini algoritma SJF memberikan hasil yang paling baik, yaitu *average waiting time*-nya paling kecil.

Deterministic modelling ini memungkinkan kita membandingkan *performance* algoritma dengan cepat dan sederhana karena kita bisa melihat perbandingannya langsung dari angka yang dihasilkan. Namun, model ini hanya berlaku untuk kasus tertentu dimana kita bisa mengetahui angka pasti dari *workload* yang akan dijadikan input dalam perhitungan. Model ini bisa digunakan untuk memilih algoritma terbaik dimana kita menjalankan program yang sama berulang-ulang dan kebutuhan untuk menjalankan program dapat diukur.

Gambar 16.1. Perbandingan dengan Deterministic Modelling



16.3. *Queueing Modelling*

Pada kenyataanya, program atau proses yang dijalankan pada suatu sistem bervariasi dari hari ke hari. Untuk itu, kita tidak bisa menggunakan *deterministic modelling* untuk membandingkan *performance* algoritma dengan menentukan kebutuhan proses atau waktu eksekusinya. Yang bisa ditentukan adalah distribusi CPU dan *I/O burst*. Distribusi ini bisa diukur, lalu kemudian diperkirakan atau diestimasikan, sehingga bisa didapatkan formula yang menjelaskan probabilitas *CPU burst*. Kita juga bisa menentukan distribusi waktu kedatangan proses dalam sistem. Dua distribusi tersebut memungkinkan kita menghitung *throughput* rata-rata, penggunaan CPU, *waiting time*, dan lain-lain.

Salah satu pemanfaatan metode di atas dengan menggunakan *queueing-network analysis* dimana kita menganggap sistem komputer sebagai jaringan *server-server*. Setiap *server* memiliki antrian proses-proses. CPU adalah *server* yang memiliki *ready queue*, yaitu antrian proses-proses yang menunggu untuk diekseskusi dan sistem M/K adalah *server* yang memiliki *device queue*, yaitu antrian M/K *device* yang menunggu untuk dilayani.

Misalnya diketahui n sebagai panjang antrian rata-rata (banyaknya proses yang berada dalam antrian),

W sebagai waktu tungggu rata-rata yang dialami proses dalam antrian, dan λ sebagai kecepatan rata-rata kedatangan proses baru ke antrian (banyaknya proses baru yang datang per detik). Pada sistem yang berada dalam " *steady state*", jumlah proses yang meninggalkan antrian pasti sama dengan yang proses yang datang. Dengan demikian, kita bisa gunakan persamaan yang dikenal dengan ***Formula Little*** berikut ini:

n= λ **x W**

n: panjang antrian

W: waktu tunggu rata-rata dalam antrian

λ: kecepatan rata-rata kedatangan proses baru

Formula Little ini valid untuk semua algoritma penjadwalan dan distribusi kedatangan proses. Sebagai contoh, jika kita mengetahui rata-rata 7 proses datang setiap detik, dan normalnya ada 14 proses dalam

antrian, maka kita bisa menghitung waktu tunggu rata-rata proses dalam antrian (W) sebagai berikut: λ = 7 proses/detik n= 14 proses maka W= n/ λ = 14/7= 2 detik per proses.

Queueing analysis ini bisa digunakan untuk membandingkan algoritma penjadwalan, tapi memiliki beberapa keterbatasan. Jenis algoritma distribusi yang bisa ditangani model ini terbatas dan pada algoritma atau distribusi yang kompleks, akan sulit untuk menggunakan model ini. Karena perhitungannya akan sangat rumit dan tidak realistis. Selain itu, kita juga perlu menggunakan asumsi yang mungkin saja tidak akurat. Model ini seringkali hanya memberi perkiraan dan keakuratan hasil perhitungannya dipertanyakan.

16.4. Simulasi

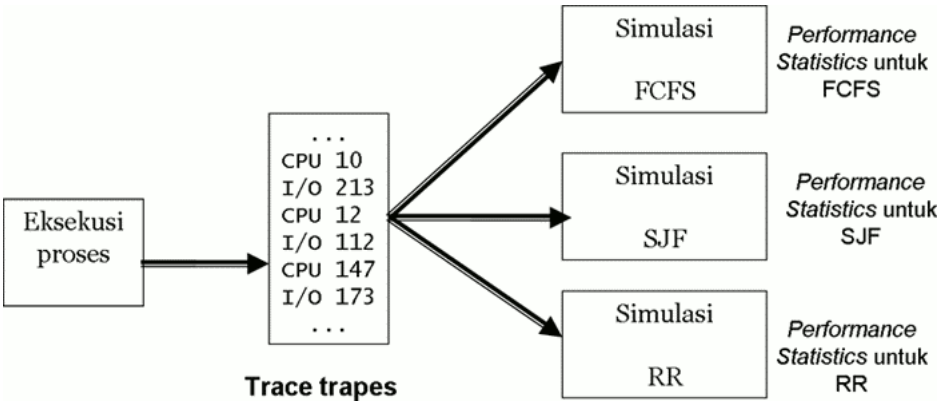
Untuk mendapatkan hasil yang lebih akurat dari evaluasi algoritma penjadwalan, bisa digunakan simulasi. Simulasi ini dibuat dengan memprogram sebuah model dari sistem komputer. Simulatornya memiliki variabel yang merepresentasikan *clock*. Jika nilai dari variabel ini bertambah, simulator mengubah status dari sistem untuk menggambarkan aktivitas proses, *device*, dan penjadwal. Selama simulatornya berjalan, data-data statistik mengenai *performance* algoritma dikumpulkan dan dicetak.

Untuk menjalakan simulator ini diperlukan data yang merepresentasikan aktivitas sistem seperti proses, *CPU burst*, dan lain-lain. Biasanya data ini dibuat dengan *random-number generator* dengan memanfaatkan distribusi probabilitas seperti distribusi Poisson, eksponensial, dan lain-lain. Tapi menjalankan simulasi dengan data yang dihasilkan dari distribusi ini bisa saja tidak akurat, karena dari distribusi hanya diketahui frekuensi atau berapa kali suatu kejadian muncul. Distribusi ini tidak memperhatikan urutan kejadiannya. Untuk mengatasi masalah itu, digunakan ***trace tapes*** .

Cara membuat *trace tapes* adalah dengan mengamati dan merekam aktivitas sistem yang sesungguhnya. Dengan ini, kita bisa menjalankan simulator dengan urutan data dari *events* yang sebenarnya. Cara ini cukup efektif dan bisa memberikan hasil yang akurat.

Berikut ini adalah ilustrasi evaluasi algoritma penjadwalan dengan simulasi:

Gambar 16.2. Evaluasi Algoritma Penjadwalan dengan Simulasi



Urutan eksekusi proses direkam dengan *trace tapes*, kemudian *simulator* menjalankan simulasi penjadwalan proses-proses tersebut dengan berbagai macam algoritma penjadwalan. Simulasi ini kemudian menghasilkan catatan mengenai *performance* dari setiap algoritma penjadwalan tersebut. Dengan membandingkan catatan *performance* itu, pengguna bisa mencari algoritma penjadwalan yang paling baik.

Meskipun memberikan hasil yang akurat, simulasi ini bisa saja memerlukan waktu yang besar dan biaya yang mahal. ***Trace tapes*** juga membutuhkan ruang penyimpanan yang besar di memori. Mendesain, memprogram, dan men- *debug* simulator juga adalah sebuah pekerjaan yang besar.

16.5. Implementasi

Simulasi bahkan memiliki keakuratan yang terbatas. Satu-satunya cara yang paling akurat untuk mengevaluasi algoritma penjadwalan adalah dengan langsung membuat programnya, masukkan ke dalam sistem operasi, dan lihat bagaimana ia bekerja. Dengan ini, algoritma yang akan dievaluasi ditempatkan dan dijalankan di sistem yang sebenarnya.

Implementasi ini membutuhkan biaya yang besar. Selain mahal, kesulitannya antara lain:
- Memprogram algoritmanya.
- Memodifikasi sistem operasi agar bisa mendukung algoritma tersebut.
- Reaksi pengguna akan sistem operasi yang terus berubah secara konstan karena pengguna tidak peduli dengan pengembangan sistem operasi. Pengguna hanya ingin proses yang dijalankannya dapat diselesaikan dengan cepat.
- *Environment* dimana program dijalankan akan berubah.

Perubahan *environment* ini bukan hanya perubahan wajar sebagaimana yang terjadi jika program baru ditulis dan tipe-tipe masalah berubah, tapi juga perubahan *performance* dari penjadwal. Pengguna (dalam hal ini *programmer*) bisa memodifikasi programnya untuk memanipulasi penjadwalan.

Misalnya sebuah komputer dikembangkan dengan pengklasifikasian proses menjadi interaktif dan non-interaktif berdasarkan jumlah terminal M/K yang digunakan proses. Jika dalam interval satu detik suatu proses tidak menggunakan terminal M/K untuk masukan atau keluaran, maka proses itu dikategorikan proses non-interaktif dan dipindahkan ke antrian yang prioritasnya lebih rendah. Dengan sistem seperti ini, bisa saja seorang *programmer* memodifikasi programnya -yang bukan program interaktif- untuk menulis karakter acak ke terminal dalam interval kurang dari satu detik. Dengan demikian, programnya akan dianggap program interaktif dan mendapat prioritas tinggi, meskipun sebenarnya *output* yang diberikan ke terminal tidak memiliki arti.

Bagaimanapun, algoritma penjadwalan yang paling fleksibel adalah yang bisa berganti-ganti atau diatur sesuai kebutuhan. Misalnya komputer yang memiliki kebutuhan grafis tinggi akan memiliki algoritma penjadwalan yang berbeda dengan komputer server. Beberapa sistem operasi seperti UNIX memungkinkan *system manager* untuk mengatur penjadwalan berdasarkan konfigurasi sistem tertentu.

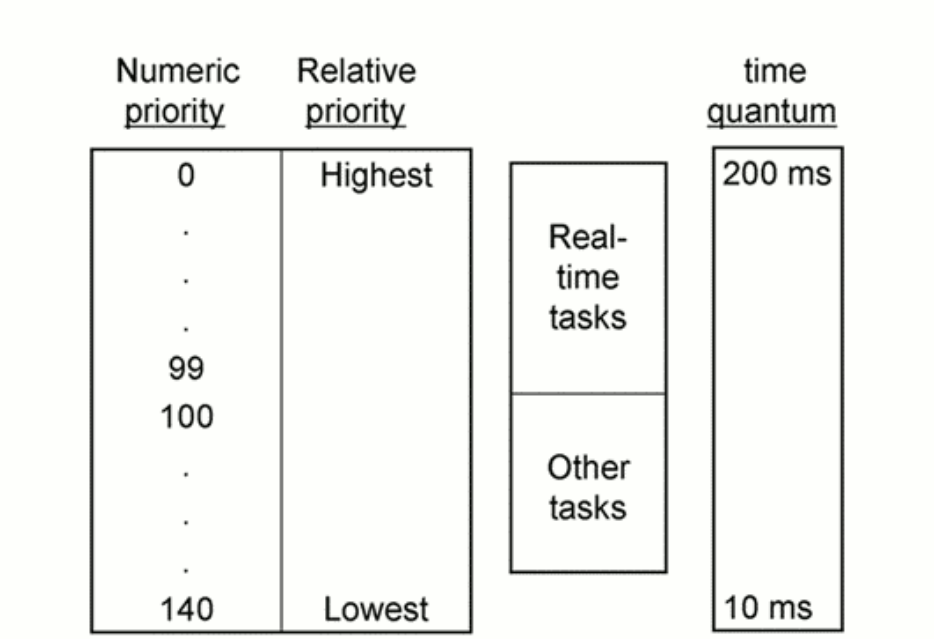
16.6. Ilustrasi: Linux

Mulai di versi 2.5, Kernel linux dapat berjalan di berbagai algoritma penjadwalan UNIX tradisional. Dua masalah dengan penjadwal UNIX tradisional adalah tidak disediakannya dukungan yang cukup untuk SMP *(symmetric multiprocessor)* sistem dan tidak diperhitungkan dengan baik jumlah *tasks* pada sistem yang berkembang. Dalam versi 2.5, penjadwal memeriksa dengan teliti hal tersebut, dan sekarang kernel juga menyajikan algoritma penjadwalan yang dapat *run* dalam waktu yang konstan tidak tergantung dari jumlah *tasks* dalam sistem. Penjadwal yang baru juga menyediakan peningkatan dukungan untuk SMP, termasuk *processor affinity* dan *load balancing*, sebaik dalam menyediakan keadilan dan dukungan terhadap *interactive tasks*.

Penjadwal linux adalah *preemptive*, algoritmanya berdasarkan prioritas dengan dua *range* prioritas yang terpisah: *real-time range* dari 0-99 dan *nice value* berkisar dari 100-140. Dua range ini dipetakan menjadi *global priority scheme* dimana nilai yang lebih rendah memiliki prioritas yang lebih tinggi. Tidak seperti penjadwal yang lain, Linux menetapkan prioritas yang lebih tinggi memiliki waktu kuantum yang lebih panjang dan prioritas yang lebih rendah memiliki waktu kuantum yang lebih pendek.

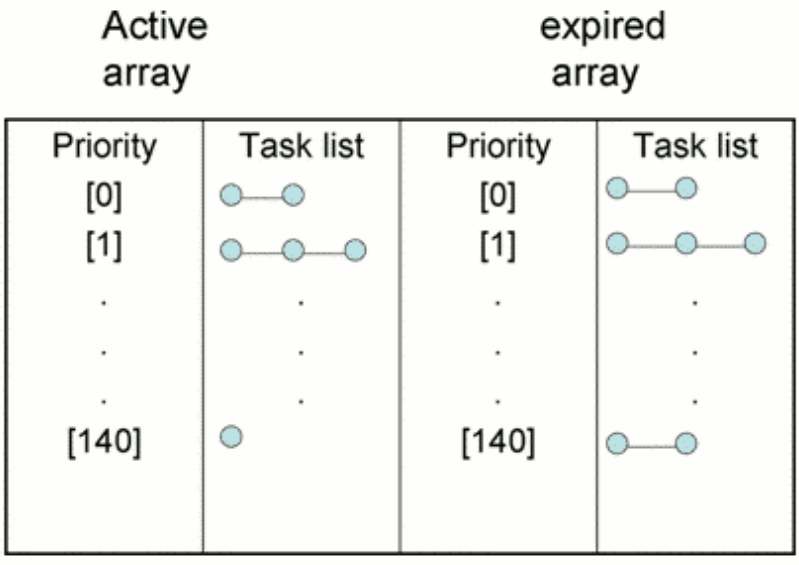
Linux mengimplementasikan *real time scheduling* seperti yang didefinisikan oleh POSIX 1.b: *First Come First Served* dan *Round Robin*. Sistem waktu nyata(*real time*)diberikan untuk *task* yang prioritasnya tetap. Sedangkan *task* yang lainnya memiliki prioritas yang dinamis berdasakan *nice values* ditambah atau dikurangi dengan 5. Interaktifitas sebuah task menentukan apakah nilai 5 tersebut akan ditambah atau dikurangi dari *nice value*. Task yang lebih interaktif mempunyai ciri khas memiliki *sleep times* yang lebih lama dan karena itu maka ditambah dengan -5, karena penjadwal lebih menyukai *interactive task*. Hasil dari pendekatan ini akan membuat prioritas untuk *interactive task* lebih tinggi. Sebaliknya, *task* dengan *sleep time* yang lebih pendek biasanya lebih *CPU-bound* jadi prioritasnya lebih rendah.

Gambar 16.3. Hubungan antara prioritas dan waktu kuantum



Task yang berjalan memenuhi syarat untuk dieksekusi oleh CPU selama *time slice*-nya masih ada. Ketika sebuah *task* telah kehabisan *time slice*-nya, maka task tersebut akan *expired* dan tidak memenuhi syarat untuk dieksekusi lagi sampai semua *task* yang lain sudah habis waktu kuantumnya. Kernel mengatur daftar semua *task* yang berjalan di *runqueue data structure*. Karena dukungan Linux untuk SMP, setiap prossesor mengatur *runqueue* mereka sendiri dan penjadwalan yang bebas. Setiap *runqueue* terdiri dari dua array prioritas - *active* dan *expired*. *Active array* terdiri dari semua *task* yang mempunyai sisa waktu *time slices*, dan *expired array* terdiri dari *task* yang telah berakhir. Setiap array prioritas ini memiliki daftar *task indexed* berdasakan prioritasnya. Penjadwal memilih *task* dengan prioritas paling tinggi di *active array* untuk dieksekusi dalam CPU. Di mesin multiprossesor, ini berarti setiap prossesor menjadwalkan prioritas paling tinggi dalam *runqueue structure* masing-masing. Ketika semua *task* telah habis *time slices*-nya (dimana, *active array*-nya sudah kosong), dua array prioritas bertukar; *expired array* menjadi *active array*, dan sebaliknya.

Gambar 16.4. Daftar *task indexed* berdasarkan prioritas



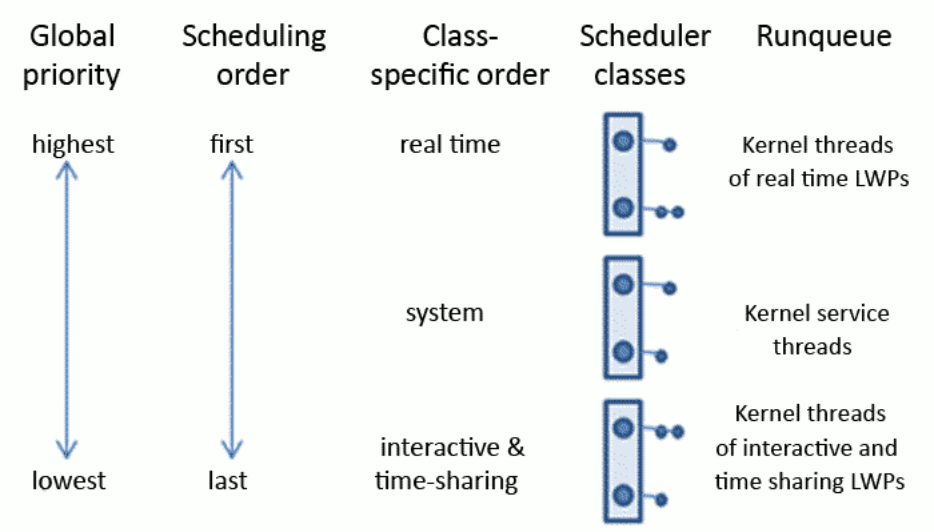
Penghitungan ulang dari *task* yang memiliki prioritas yang dinamis berlangsung ketika *task* telah menyelesaikan waktu kuantumnya dan akan dipindahkan ke *expired array*. Jadi, ketika ada dua larik (*array*) ditukar, semua *task* di *array* aktif yang baru ditentukan prioritasnya yang baru dan disesuaikan juga *time slices*-nya.

16.7. Ilustrasi: Solaris

Solaris menggunakan penjadwalan berdasarkan prioritas dimana yang mempunyai prioritas yang lebih tinggi dijalankan terlebih dahulu. Informasi tentang penjadwalan kernel *thread* dapat dilihat dengan `ps -elcL`.

Kernel Solaris adalah *fully preemtible*, artinya semua *thread*, termasuk *thread* yang mendukung aktifitas kernel itu sendiri dapat ditunda untuk menjalankan *thread* dengan prioritas yang lebih tinggi.

Gambar 16.5. Penjadwalan Solaris



Solaris mengenal 170 prioritas yang berbeda, 0-169. Terbagi dalam 4 kelas penjadwalan yang berbeda:

1. **Real time (RT).** *Thread* di kelas RT memiliki prioritas yang tetap dengan waktu kuantum yang tetap juga. *Thread* ini memiliki prioritas yang tinggi berkisar antara 100-159. Hal inilah yang membuat proses waktu nyata memiliki *response time* yang cepat. Proses waktu nyata akan dijalankan sebelum proses-proses dari kelas yang lain dijalankan sehingga dapat menghentikan proses di *system class*. Pada umumnya, hanya sedikit proses yang merupakan *real time class*.
2. **System (SYS).** Solaris menggunakan *system class* untuk menjalankan kernel proses, seperti penjadwalan dan *paging daemon*. *Threads* di kelas ini adalah *"bound" threads*, berarti bahwa mereka akan dijalankan sampai mereka di blok atau prosesnya sudah selesai. Prioritas untuk SYS *threads* berkisar 60-99. Sekali dibangun, prioritas dari sistem proses tidak dapat dirubah. *System class* dialokasikan untuk kernel *use*(*user* proses berjalan di kernel *mode* bukan di *system class*).
3. **Time Sharing (TS).** *Time sharing class* merupakan *default class* untuk proses dan kernel *thread* yang bersesuaian. *Time slices* masing-masing proses dibagi berdasarkan prioritasnya. Dalam hal ini, prioritas berbanding terbalik dengan *time slices*-nya. Untuk proses yang prioritasnya tinggi mempunyai *time-slices* yang pendek, dan sebaliknya proses dengan prioritas yang rendah mempunyai *time slices* yang lebih panjang. Besar prioritasnya berada antara 0-59. Proses yang interaktif berada di prioritas yang tinggi sedangkan proses *CPU-bound* mempunyai prioritas yang rendah. Aturan penjadwalan seperti ini memberikan *response time* yang baik untuk proses yang interaktif, dan *troughput* yang baik untuk proses *CPU-bound*.
4. **Interactive (IA).** Kelas Interaktif menggunakan aturan yang sama dengan aturan dengan kelas kelas *time sharing*, tetapi kelas ini memberikan prioritas yang tinggi untuk aplikasi jendela (*windowing application*) sehingga menghasilkan *performance* yang lebih baik. Seperti TS, *range* IA berkisar 0-59.

Tabel 16.2. *Solaris dispatch table for interactive and time sharing threads*

Priority	Time quantum	Time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

Keterangan:
- *Priority*: prioritas berdasarkan kelas untuk *time sharing* dan *interactive class*. Nomor yang lebih tinggi menunjukkan prioritas yang lebih tinggi.
- *Time quantum*: waktu kuantum untuk setiap prioritas. Dapat diketahui bahwa fungsi waktu kuantum berbanding terbalik dengan prioritasnya.
- *Time quantum expired*: Prioritas terbaru untuk *thread* yang telah habis *time slices*-nya tanpa diblok. Dapat dilihat dari tabel bahwa *thread* yang *CPU-bound* tetap mempunyai prioritas yang rendah.
- *Return from sleep*: Prioritas *thread* yang kembali dari *sleeping*(misalnya menunggu dari M/K). Seperti yang terlihat dari tabel ketika M/K berada di *waiting thread*, prioritasnya berada antara 50-59, hal ini menyebabkan *response time* yang baik untuk proses yang interaktif.

5. **Fixed Priority (FX).** *Thread* di kelas *fixed priority* memiliki *range* prioritas (0-59) yang sama seperti di *time-sharing class*; tetapi, prioritas mereka tidak akan berubah.
6. **Fair Share Scheduler (FSS).** Thread yang diatur oleh FSS dijadwalkan berdasar pembagian sumber daya dari CPU yang tersedia dan dialokasikan untuk himpunan proses-proses (yang dikenal sebagai *project*). FS juga berkisar 0-59. FSS and FX baru mulai diimplementasikan di Solaris 9.

Seperti yang telah diketahui, setiap kelas penjadwalan mempunyai himpunan dari prioritas-prioritas. Tetapi, penjadwal mengubah *class-specific priorities* menjadi *global priorities* kemudian memilih *thread* dengan prioritas paling tinggi untuk dijalankan. *Thread* yang dipilih tersebut jalan di CPU sampai *thread* tersebut (1) di- *block*, (2) habis *time slices*-nya, atau (3) dihentikan oleh *thread* dengan prioritas yang lebih tinggi. Jika ada beberapa *thread* dengan prioritas yang sama, penjadwal akan menggunakan Round-Robin queue. Seperti yang pernah dijelaskan sebelumnya, Solaris terdahulu menggunakan *many-to-many model* tetapi solaris 9 berubah menggunakan *one-to-one* model.

16.8. Rangkuman

Cara mengevaluasi algoritma:
- *Deterministic Modelling*: menggunakan perhitungan matematika berdasarkan workload sistem. Sederhana, namun pemakaiannya terbatas.
- *Queueing Model*: menggunakan perhitungan matematika berdasarkan waktu antrian dan waktu kedatangan proses.
- Simulasi: membuat model sistem komputer. Cara ini mahal dan membutuhkan kerja besar dalam memprogram.

- Implementasi: langsung mengimplementasi pada sistem sebenarnya. Paling efektif, namun mahal.

Tabel 16.3. *Scheduling Priorities in Linux*

Priorities	Scheduling class
0-99	System Threads, Real time (SCHED_FIFO, SCHED_RR)
100-139	User priorities (SCHED_NORMAL)

Tabel 16.4. *Scheduling Priorities in Solaris*

Priorities	Scheduling class
0-59	Time Shared, Interactive, Fixed, Fair Share Scheduler
60-99	System Class
100-159	Real-Time (note real-time higher than system threads)
160-169	Low level Interrupts

Rujukan

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

[WEBIBMNY] IBM Corporation. NY . *General Programming Concepts – Writing and Debugging Programs – http://publib16.boulder.ibm.com/ pseries/ en_US/ aixprggd/ genprogc/ ls_sched_subr.htm*. Diakses 1 Juni 2006.

[WEBIBM1997] IBM Coorporation. 1997 . *General Programming Concepts: Writing and Debugging Programs – Threads Scheduling http://www.unet.univie.ac.at/ aix/ aixprggd/ genprogc/ threads_sched.htm*. Diakses 1 Juni 2006.

[WEBLindsey2003] Clark S Lindsey. 2003 . *Physics Simulation with Java – Thread Scheduling and Priority – http://www.particle.kth.se/ ~fmi/ kurs/ PhysicsSimulation/ Lectures/ 10A/ schedulePriority.html*. Diakses 1 Juni 2006.

[WEBMooreDrakos1999] Ross Moore dan Nikos Drakos. 1999 . *Converse Programming Manual – Thread Scheduling Hooks – http://charm.cs.uiuc.edu/ manuals/ html/ converse/ 3_3Thread_ Scheduling_Hooks.html*. Diakses 1 Juni 2006.

[WEBVolz2003] Richard A Volz. 2003 . *Real Time Computing – Thread and Scheduling Basics – http:// linserver.cs.tamu.edu/ ~ravolz/ 456/ Chapter-3.pdf*. Diakses 1 Juni 2006.

[WEBPRI2007] Scott Cromar. 2007 . *Princeton University [Unix Systems]– http://www.princeton.edu/~unix/ Solaris/troubleshoot/schedule.html*. Diakses 10 maret 2007.

[WEBOSL2005] Max Bruning. 2005 . *A Comparison of Solaris, Linux, and FreeBSD Kernels– http:// www.opensolaris.org/ os/ article/ 2005-10-14_ a_comparison_of_solaris__linux_ and_freebsd_kernels/*. Diakses 10 maret 2007.

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 17. Konsep Interaksi

17.1. Pendahuluan

Dalam bab ini kita akan mengulas bagaimana hubungan antar proses dapat berlangsung, misalnya bagaimana beberapa proses dapat saling berkomunikasi dan bekerja sama, pengertian sinkronasi dan penyangga, sistem *client-server* dan RPC, serta penyebab *deadlock* dan *starvation*.

Sinkronisasi diperlukan untuk menghindari terjadinya ketidakkonsistenan data akibat adanya akses data secara konkuren. Proses-proses disebut konkuren jika proses-proses itu ada dan berjalan pada waktu yang sama, proses-proses konkuren ini bisa bersifat independen atau bisa juga saling berinteraksi. Proses-proses konkuren yang saling berinteraksi memerlukan sinkronisasi agar terkendali dan juga menghasilkan output yang benar.

17.2. Komunikasi Antar Proses

Sistem Berbagi Memori

Sistem Berbagi Memori atau yang disebut juga sebagai *Shared Memory System* merupakan salah satu cara komunikasi antar proses dengan cara mengalokasikan suatu alamat memori untuk dipakai berkomunikasi antar proses. Alamat dan besar alokasi memori yang digunakan biasanya ditentukan oleh pembuat program. Pada metode ini, sistem akan mengatur proses mana yang akan memakai memori pada waktu tertentu sehingga pekerjaan dapat dilakukan secara efektif.

Sistem Berkirim Pesan

Sistem berkirim pesan adalah proses komunikasi antar bagian sistem untuk membagi variabel yang dibutuhkan. Proses ini menyediakan dua operasi yaitu mengirim pesan dan menerima pesan. Ketika dua bagian sistem ingin berkomunikasi satu sama lain, yang harus dilakukan pertama kali adalah membuat sebuah *link* komunikasi antara keduanya. Setelah itu, kedua bagian itu dapat saling bertukar pesan melalui *link* komunikasi tersebut.

Sistem berkirim pesan sangat penting dalam sistem operasi. Karena dapat diimplementasikan dalam banyak hal seperti pembagian memori, pembagian bus, dan melaksanakan proses yang membutuhkan pengerjaan bersama antara beberapa bagian sistem operasi.

Terdapat dua macam cara berkomunikasi, yaitu:

1. **Komunikasi langsung.** Dalam komunikasi langsung, setiap proses yang ingin berkirim pesan harus mengetahui secara jelas dengan siapa mereka berkirim pesan. Hal ini dapat mencegah pesan salah terkirim ke proses yang lain. Karakteristiknya antara lain:
   a. *Link* dapat otomatis dibuat
   b. Sebuah *link* berhubungan dengan tepat satu proses komunikasi berpasangan
   c. Diantara pasangan itu terdapat tepat satu *link*
   d. *Link* tersebut biasanya merupakan *link* komunikasi dua arah
2. **Komunikasi tidak langsung.** Berbeda dengan komunikasi langsung, jenis komunikasi ini menggunakan sejenis kotak surat atau *port* yang mempunyai ID unik untuk menerima pesan. Proses dapat berhubungan satu sama lain jika mereka membagi *port* mereka. Karakteristik komunikasi ini antara lain:
   a. *Link* hanya terbentuk jika beberapa proses membagi kotak surat mereka
   b. Sebuah *link* dapat terhubung dengan banyak proses
   c. Setiap pasang proses dapat membagi beberapa *link* komunikasi
   d. *Link* yang ada dapat merupakan *link* terarah ataupun *link* yang tidak terarah

17.3. Sinkronisasi

Komunikasi antara proses membutuhkan *subroutine* untuk mengirim dan menerima data primitif. Terdapat desain yang berbeda-beda dalam implementasi setiap primitif. Pengiriman pesan mungkin dapat diblok (*blocking*) atau tidak dapat dibloking (*nonblocking*) - juga dikenal dengan nama sinkron atau asinkron.

Ketika dalam keadaan sinkron, terjadi dua kejadian:
1. ***Blocking send .*** Pemblokiran pengirim sampai pesan sebelumnya diterima.
2. ***Blocking receive .*** Pemblokiran penerima sampai terdapat pesan yang akan dikirim.

Sedangkan untuk keadaan asinkron, yang terjadi adalah:
1. ***Non-blocking send .*** Pengirim dapat terus mengirim pesan tanpa memperdulikan apakah pesan sebelumnya sampai atau tidak.
2. ***Non-blocking receive .*** Penerima menerima semua pesan baik berupa pesan yang valid atau pesan yang salah (null).

17.4. Penyangga

Dalam setiap jenis komunikasi, baik langsung atau tidak langsung, penukaran pesan oleh proses memerlukan antrian sementara. Pada dasarnya, terdapat tiga cara untuk mengimplementasikan antrian tersebut:
1. **Kapasitas Nol.** Antrian mempunyai panjang maksimum nol, sehingga tidak ada penungguan pesan (*message waiting*). Dalam kasus ini, pengirim harus memblok sampai penerima menerima pesan.
2. **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.
3. **Kapasitas Tak Terbatas.** Antrian mempunyai panjang yang tak terhingga, sehingga semua pesan dapat menunggu disini. Pengirim tidak akan pernah di blok.

17.5. *Client/Server*

Dengan makin berkembangnya teknologi jaringan komputer, sekarang ini ada kecenderungan sebuah sistem yang menggunakan jaringan untuk saling berhubungan. Dalam jaringan tersebut, biasanya terdapat sebuah komputer yang disebut *server*, dan beberapa komputer yang disebut *client*. *Server* adalah komputer yang dapat memberikan *service* ke *server*, sedangkan *client* adalah komputer yang mengakses beberapa *service* yang ada di *client*. Ketika *client* membutuhkan suatu *service* yang ada di *server*, dia akan mengirim *request* kepada *server* lewat jaringan. *Jika request* tersebut dapat dilaksanakan, maka *server* akan mengirim balasan berupa *service* yang dibutuhkan untuk saling berhubungan menggunakan *Socket*.
1. Karakteristik *Server*
   a. Pasif
   b. Menunggu *request*
   c. Menerima *request*, memproses mereka dan mengirimkan balasan berupa *service*
2. Karakteristik *Client*
   a. Aktif
   b. Mengirim *request*
   c. Menunggu dan menerima balasan dari *server*

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 menghubungi *server* maka *server* akan menyetujui komunikasi dengan *client* melalui *socket* yang dibangun.

Sebagai contoh sebuah program *web browser* pada host x (IP 146.86.5.4) ingin berkomunikasi dengan *web server* (IP 152.118.25.15) yang sedang menunggu pada *port* 80. Host x akan menunjuk sebuah

port. Dalam hal ini *port* yang digunakan ialah *port* 1655. Sehingga terjadi sebuah hubungan dengan sepasang *socket* (146.86.5.4:1655) dengan (152.118.25.15:80).

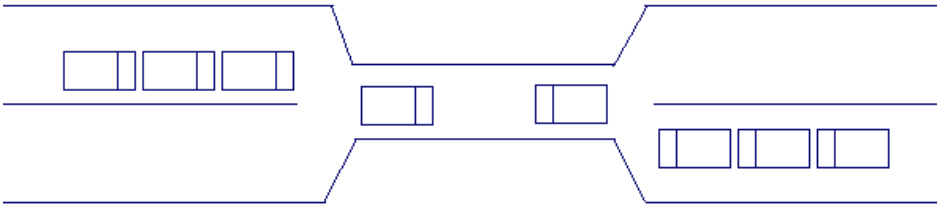
17.6. *RPC*

Remote Procedure Call (RPC) adalah sebuah metode yang memungkinkan kita untuk mengakses sebuah prosedur yang berada di komputer lain. Untuk dapat melakukan ini sebuah *server* harus menyediakan layanan *remote procedure*. Pendekatan yang dilakuan adalah sebuah *server* membuka *socket*, lalu menunggu *client* yang meminta prosedur yang disediakan oleh server. Bila *client* tidak tahu harus menghubungi *port* yang mana, *client* bisa me- *request* kepada sebuah *matchmaker* pada sebuah RPC *port* yang tetap. *Matchmaker* akan memberikan *port* apa yang digunakan oleh prosedur yang diminta *client*.

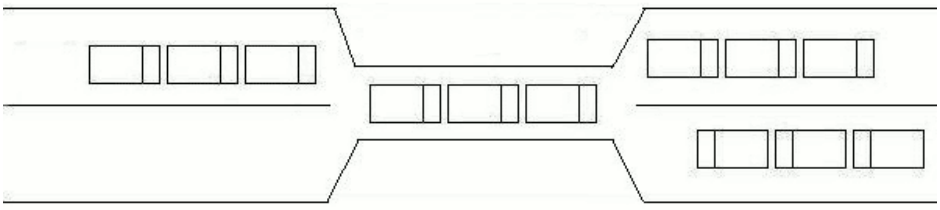
RPC masih menggunakan cara primitif dalam pemrograman, yaitu menggunakan paradigma *procedural programming*. Hal itu membuat kita sulit ketika menyediakan banyak *remote procedure*. RPC menggunakan *socket* untuk berkomunikasi dengan proses lainnya. Pada sistem seperti SUN, RPC secara *default* sudah ter- *install* 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.

17.7. *Deadlock* dan *Starvation*

Gambar 17.1. Dead Lock



Gambar 17.2. Starvation



1. ***Deadlock*** . *Deadlock* adalah suatu kondisi dimana dua proses atau lebih saling menunggu proses yang lain untuk melepaskan *resource* yang sedang dipakai. Karena beberapa proses itu saling menunggu, maka tidak terjadi kemajuan dalam kerja proses-proses tersebut. *Deadlock* adalah masalah yang biasa terjadi ketika banyak proses yang membagi sebuah *resource* yang hanya boleh dirubah oleh satu proses saja dalam satu waktu. Di kehidupan nyata, *deadlock* dapat digambarkan dalam gambar berikut.Pada gambar diatas, *deadlock* dianalogikan sebagai dua antrian mobil yang akan menyeberangi jembatan. Dalam kasus diatas, antrian di sebelah kiri menunggu antrian kanan untuk mengosongkan jembatan (*resource*), begitu juga dengan antrian kanan. Akhirnya tidak terjadi kemajuan dalam kerja dua antrian tersebut.Misal ada proses A mempunyai *resource* X, proses B mempunyai *resource* Y. Kemudian kedua proses ini dijalankan bersama, proses A memerlukan

resource Y dan proses B memerlukan *resource* X, tetapi kedua proses tidak akan memberikan *resource* yang dimiliki sebelum proses dirinya sendiri selesai dilakukan. Sehingga akan terjadi tunggu-menunggu.

2. ***Starvation .*** *Starvation* adalah kondisi yang biasanya terjadi setelah *deadlock*. Proses yang kekurangan *resource* (karena terjadi *deadlock*) tidak akan pernah mendapat *resource* yang dibutuhkan sehingga mengalami *starvation* (kelaparan). Namun, *starvation* juga bisa terjadi tanpa *deadlock*. Hal ini ketika terdapat kesalahan dalam sistem sehingga terjadi ketimpangan dalam pembagian *resouce*. Satu proses selalu mendapat *resource*, sedangkan proses yang lain tidak pernah mendapatkannya. Ilustrasi *starvation* tanpa *deadlock* di dunia nyata dapat dilihat di bawah ini.Pada gambar diatas, pada antrian kanan terjadi *starvation* karena *resource* (jembatan) selalu dipakai oleh antrian kiri, dan antrian kanan tidak mendapatkan giliran.

17.8. Rangkuman

Dalam menjalankan fungsinya dalam sistem operasi, dibutuhkan interaksi antara beberapa proses yang berbeda. Interaksi tersebut bertujuan agar terjadi kesinambungan antar proses yang terjadi sehingga sistem operasi dapat berjalan sebagaimana mestinya. Interaksi tersebut dapat melalui sistem berbagi memori atau dengan cara saling berkirim pesan. Terkadang, beberapa pesan yang dikirim tidak dapat diterima seluruhnya oleh penerima dan menyebabkan informasi yang lain menjadi tidak valid, maka dibutuhkanlah sebuah mekanisme sinkronasi yang akan mengatur penerimaan dan pengiriman pesan sehingga kesalahan penerimaan pesan dapat diperkecil. Pesan yang dikirim dapat ditampung dalam penyangga sebelum diterima oleh penerima.

Interaksi antar proses dapat juga terjadi antara proses yang memiliki sistem berbeda. Dalam interaksi tersebut dikenal sebutan *client* dan *server* yang memungkinkan sistem yang berbeda untuk berinteraksi dengan menggunakan *socket*. Dalam interaksi tersebut dikenal juga RPC (*Remote Procedure Call*) yaitu metode yang memungkinkan sebuah sistem mengakses prosedur sistem lain dalam komputer berbeda.

Dalam interaksi antar proses, terkadang suatu proses saling menunggu proses yang lain sebelum melanjutkan prosesnya, sehingga proses-proses tersebut saling menunggu tanpa akhir, hal ini disebut *deadlock*. Jika *deadlock* terjadi dalam waktu lama, maka terjadilah *starvation*, yaitu suatu proses tidak mendapatkan *resource* yang dibutuhkan.

Rujukan

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

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

[WEBWIKIA] Wikipedia. *http://en.wikipedia.org/wiki/Client_%28computing%29* . Diakses 24 Mei 2007.

[WEBWIKIB] Wikipedia. *http://en.wikipedia.org/wiki/Server_%28computing%29* . Diakses 24 Mei 2007.

[WEBWIKIC] Wikipedia. *http://en.wikipedia.org/wiki/Client_server* . Diakses 24 Mei 2007.

Bab 18. Sinkronisasi

18.1. Pendahuluan

Sinkronisasi diperlukan untuk menghindari terjadinya ketidak-konsistenan data akibat adanya akses data secara konkuren. Proses-proses disebut konkuren jika proses-proses itu ada dan berjalan pada waktu yang sama, proses-proses konkuren ini bisa bersifat independen atau bisa juga saling berinteraksi. Proses-proses konkuren yang saling berinteraksi memerlukan sinkronisasi agar terkendali dan juga menghasilkan output yang benar.

18.2. *Race Condition*

Race condition adalah suatu kondisi dimana dua atau lebih proses mengakses *shared memory*/sumber daya pada saat yang bersamaan dan hasil akhir dari data tersebut tergantung dari proses mana yang terakhir selesai dieksekusi sehingga hasil akhirnya terkadang tidak sesuai dengan yang dikehendaki.

Contoh 18.1. Race Condition

```
 1.  int counter = 0;
 2.  //Proses yang dilakukan oleh produsen
 3.  item nextProduced;
 4.  while (1) {
 5.     while (counter == BUFFER_SIZE) { ... do nothing ... }
 6.     buffer[in] = nextProduced;
 7.     in = (in + 1) % BUFFER_SIZE;
 8.     counter++;
 9.  }
10. //Proses yang dilakukan oleh konsumen
11. item nextConsumed;
12. while (1) {
13.    while (counter == 0)           { ... do nothing ... }
14.    nextConsumed = buffer[out] ;
15.    out = (out + 1) % BUFFER_SIZE;
16.    counter--;
17. }
```

Pada program di atas, terlihat bahwa terdapat variabel counter yang diinisialisasi dengan nilai 0, dan ditambah 1 setiap kali terjadi produksi serta dikurangi 1 setiap kali terjadi konsumsi. Pada bahasa mesin, baris kode `counter++` dan `counter--` diimplementasikan seperti di bawah ini:

Contoh 18.2. *Race Condition* dalam bahasa mesin

```
//counter++
register1 = counter
register1 = register1 + 1
counter   = register1

//counter--
register2 = counter
register2 = register2 - 1
counter   = register2
```

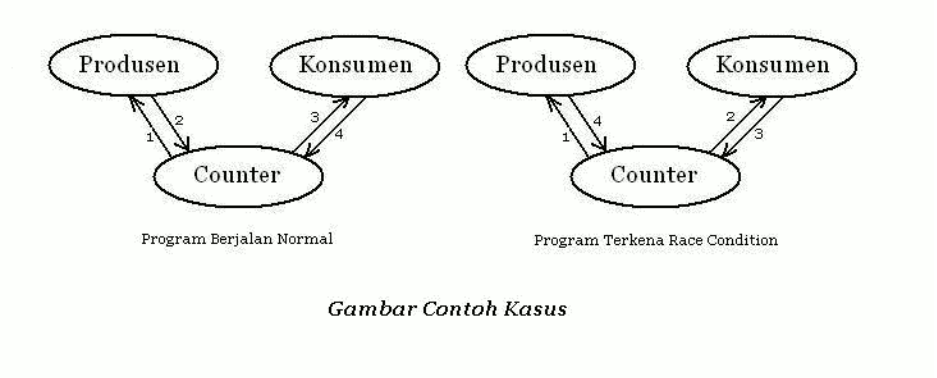
Jika perintah `counter++` dan `counter--` berusaha mengakses nilai counter secara konkuren, maka nilai akhir dari counter bisa salah. Hal ini tidak berarti nilainya pasti salah, tapi ada kemungkinan untuk terjadi kesalahan. Contoh urutan eksekusi baris kode tersebut yang mengakibatkan kesalahan pada nilai akhir counter:

Contoh 18.3. Program yang memperlihatkan *Race Condition*

```
        2//misalkan nilai awal counter adalah 2
1.produsen: register1 = counter       (register1 = 2)
2.produsen: register1 = register1 + 1 (register1 = 3)
3.konsumen: register2 = counter       (register2 = 2)
4.konsumen: register2 = register2 - 1 (register2 = 1)
5.konsumen: counter   = register2     (counter = 1)
6.produsen: counter   = register1     (counter = 3)
```

Status akhir dari counter seharusnya adalah 0, tapi kalau urutan pengeksekusian program berjalan seperti di atas, maka hasil akhirnya menjadi 3. Perhatikan bahwa nilai akhir counter akan mengikuti eksekusi terakhir yang dilakukan oleh komputer. Pada program di atas, pilihannya bisa 1 atau 3. 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 bekerjasama dieksekusi secara beraturan demi mencegah timbulnya suatu keadaan *race condition.*

Gambar 18.1. Ilustrasi program produsen dan konsumen



Gambar Contoh Kasus

Kunci untuk mencegah masalah ini dan di situasi yang lain yang melibatkan memori bersama, berkas bersama, dan sumber daya lain yang digunakan secara bersama-sama adalah menemukan beberapa jalan untuk mencegah lebih dari satu proses melakukan proses tulis dan baca kepada data yang sama pada saat yang sama. Dengan kata lain, kita membutuhkan *mutual exclusion*, sebuah jalan yang

menjamin jika sebuah proses sedang menggunakan variabel atau berkas yang digunakan bersama-sama, proses lain akan dikeluarkan dari pekerjaan yang sama.

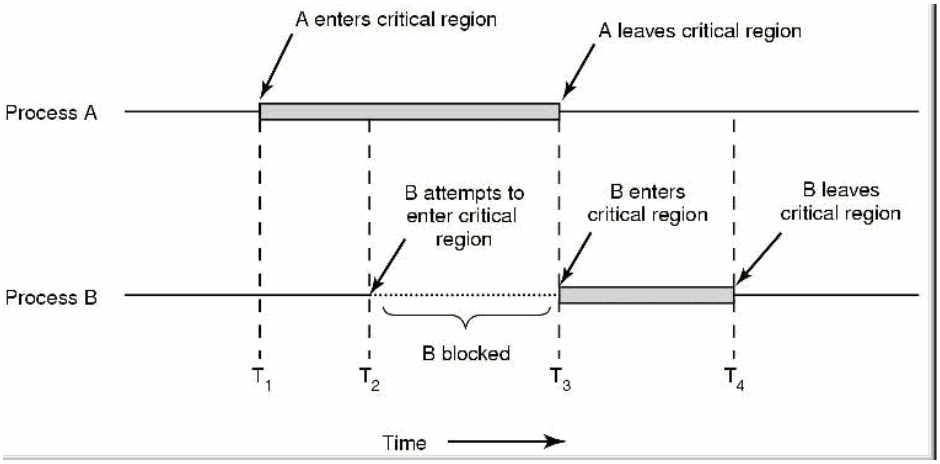
Cara untuk menghindari *race condition* adalah kita harus dapat menjamin bahwa jika suatu proses sedang menjalankan *critical section*, maka proses lain tidak boleh masuk ke dalam *critical section* tersebut. *Critical section* dijelaskan dibawah.

18.3. *Critical Section*

Critical section adalah segmen kode yang mengakses data yang digunakan proses secara bersama-sama yang dapat membawa proses itu ke bahaya *race condition.* Biasanya sebuah proses sibuk melakukan perhitungan internal dan hal-hal lainnya tanpa ada bahaya yang menuju ke *race condition* pada sebagian besar waktu. Akan tetapi, biasanya setiap proses memiliki segmen kode dimana proses itu dapat mengubah variabel, meng- *update* suatu tabel, menulis ke suatu file, dan lain-lainnya, yang dapat membawa proses itu ke bahaya *race condition.*

18.4. Prasyarat Solusi *Critical Section*

Gambar 18.2. Ilustrasi *critical section*



Solusi dari masalah *critical section* harus memenuhi tiga syarat berikut:

1. ***Mutual Exclusion .*** *Mutual Exclusion* merupakan sebuah jalan yang menjamin jika sebuah proses sedang menggunakan variabel atau berkas yang digunakan bersama-sama, proses lain akan dikeluarkan dari pekerjaan yang sama. Misal 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.

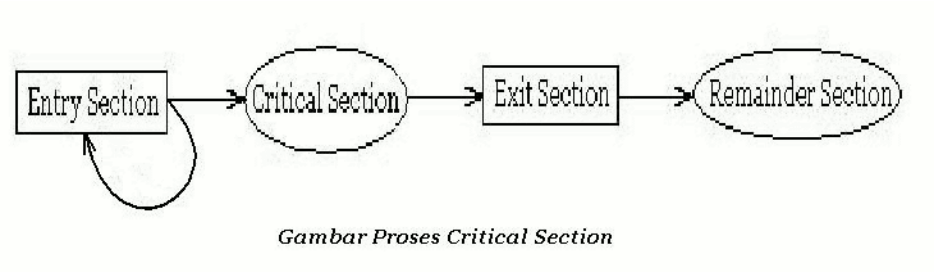
Contoh 18.4. Struktur umum dari proses Pi adalah:

```
do {
        entry section
           critical section
        exit section
           remainder section
        } while (1);
```

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.* Dari kode di atas, dapat kita lihat bahwa untuk bisa memasuki *critical section* sebuah proses harus melalui *entry section.*

Gambar 18.3. ilustrasi proses Pi



Gambar Proses Critical Section

2. **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*).
3. **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).

18.5. *Critical Section* dalam Kernel

Problem untuk kernel muncul karena berbagai *tasks* mungkin mencoba untuk mengakses data yang sama. Jika hanya satu kernel *task* ditengah pengaksesan data ketika *interrupt 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* sebagai hasilnya. Saat sepotong kernel code mulai dijalankan, akan terjamin bahwa itu adalah satu-satunya kernel code yang dijalankan sampai salah satu dari aksi dibawah ini muncul:

1. **Interupsi.** Interupsi adalah suatu masalah bila mengandung critical section-nya sendiri. *Timer interrupt* tidak secara langsung menyebabkan terjadinya penjadwalan ulang suatu proses; hanya meminta suatu jadwal untuk dilakukan kemudian, jadi kedatangan suatu interupsi tidak mempengaruhi urutan eksekusi dari kernel code. Sekali *interrupt service* selesai, eksekusi akan menjadi lebih simpel untuk kembali ke kernel code yang sedang dijalankan ketika interupsi mengambil alih.
2. ***Page Fault* .** 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, dan proses yang berjalan akan di tunda sampai M/K selesai. Pada kasus yang hampir sama, jika *system call service routine* memanggil penjadwalan ketika sedang berada di mode kernel, mungkin secara eksplisit dengan membuat *direct call* pada code penjadwalan atau secara implisit dengan memanggil sebuah fungsi untuk menunggu M/K selesai, setelah itu proses akan menunggu dan penjadwalan ulang akan muncul. Ketika proses jalan kembali, proses tersebut akan melanjutkan untuk mengeksekusi dengan mode kernel, melanjutkan intruksi setelah pemanggilan ke penjadwalan.
3. ***Kernel code* memanggil fungsi penjadwalan sendiri.** setiap waktu banyak proses yang berjalan dalam *kernel mode*,akibatnya sangat mungkin untuk terjadi *race condition*,contoh, dalam *kernel mode* terdapat struktur data yang menyimpan *list file* yang terbuka, *list* tersebut termodifikasi bila ada *data file* yang baru dibuka atau ditutup,dengan menambah atau menghapus dari *list*, *race condition* timbul ketika ada dua *file* yang dibuka dan ditutup bersamaan, untuk mengatasi hal tersebut kernel mempunyai metode yaitu:
   a. **Preemptive kernel.** pada mode ini proses yang sedang dieksekusi dalam kernel diizinkan untuk diinterupsi oleh proses lain yang memenuhi syarat, akibatnya mode ini juga rentan terkena

race condition. Keuntungannya adalah mode ini amat efektif untuk digunakan dalam *real time programming*, namun mode ini lebih sulit diimplementasikan dari pada mode non preemptive kernel. Mode ini diimplementasikan oleh Linux versi 2.6 dan versi komersial Linux lainnya.

b. **Non preemptive kernel.** mode yang tidak memperbolehkan suatu proses yang berjalan dalam kernel mode diinterupsi oleh proses lain, proses lain hanya bisa dijalankan setelah proses yang ada dalam kernel selesai dieksekusi, implementasinya memang lebih mudah dibandingkan dengan preemptive kernel, mode ini diimplementasikan lewat Windows XP dan Windows 2000.

18.6. Rangkuman

Suatu proses yang bekerja bersama-sama dan saling berbagi data dapat mengakibatkan *race condition* atau pengaksesan data secara bersama-sama. *Critical section* adalah suatu segmen kode dari proses-proses itu yang yang memungkinkan terjadinya *race condition*. Untuk mengatasi masalah *critical section* ini, suatu data yang sedang diproses tidak boleh diganggu proses lain.

Solusi prasyarat *critical section*:
1. ***Mutual Exclusion.* .**
2. **Terjadi kemajuan *(progress)*.**
3. **Ada batas waktu tunggu *(bounded waiting)*.**

Critical section dalam kernel:
1. **Interupsi.**
2. ***Page Fault* .**
3. ***Kernel code* memanggil fungsi penjadwalan sendiri.**

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.

[WEBFSF1991a] Free Software Foundation. 1991 . – *http://gnui.vLSM.org/ licenses/ gpl.txt*. Diakses 7 April 2007.

Bab 19. Solusi *Critical Section*

19.1. Pendahuluan

Pada bab sebelumnya telah dijelaskan tentang masalah *critical section* yang dapat menimbulkan *Race Condition*. Oleh karena itu, dibutuhkan solusi yang tepat untuk menghindari munculnya *Race Condition*. Solusi tersebut harus memenuhi ketiga syarat berikut:

1. *Mutual Exclusion*
2. *Progress*
3. *Bounded Waiting*

Ada dua jenis solusi untuk memecahkan masalah *critical section*, yaitu.

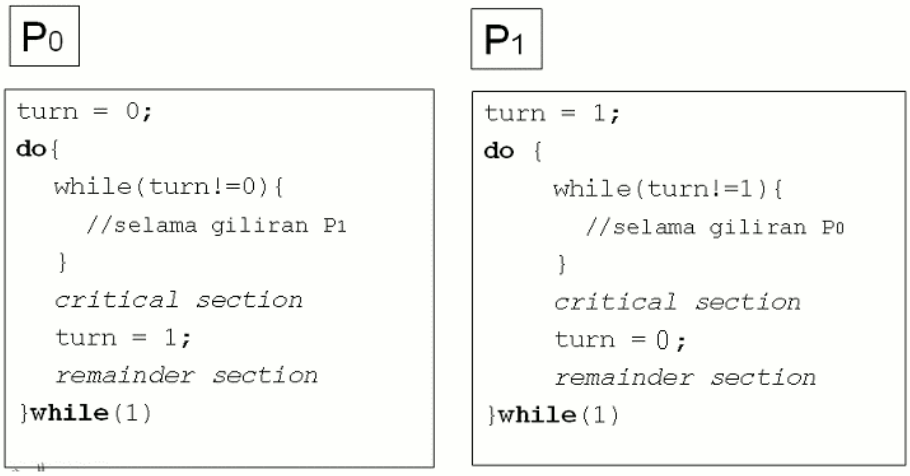
1. **Solusi Perangkat Lunak.** Solusi ini menggunakan algoritma-algoritma untuk mengatasi masalah *critical section*.
2. **Solusi Perangkat Keras.** Solusi ini tergantung pada beberapa instruksi mesin tertentu, misalnya dengan me-non-aktifkan interupsi, mengunci suatu variabel tertentu atau menggunakan instruksi level mesin seperti tes dan set.

Pembahasan selanjutnya adalah mengenai solusi perangkat lunak menggunakan algoritma-algoritma. Algoritma-algoritma yang akan dibahas adalah algoritma untuk memecahkan masalah *critical section* untuk dua proses yaitu Algoritma I, Algoritma II dan Algoritma III. Perlu diingat bahwa Algoritma I dan Algoritma II tidak dapat menyelesaikan masalah *critical section*. Adapun algoritma yang dibahas untuk memecahkan masalah *critical section* untuk n-buah proses adalah Algoritma Tukang Roti.

19.2. Algoritma I

Algoritma I mencoba mengatasi masalah *critical section* untuk dua proses. Algoritma ini menerapkan sistem bergilir kepada kedua proses yang ingin mengeksekusi *critical section*, sehingga kedua proses tersebut harus bergantian menggunakan *critical section*.

Gambar 19.1. Algoritma I



Algoritma ini menggunakan variabel bernama *turn*, nilai *turn* menentukan proses mana yang boleh memasuki *critical section* dan mengakses data yang di- *sharing*. Pada awalnya variabel *turn* diinisialisasi 0, artinya P0 yang boleh mengakses *critical section*. Jika *turn*= 0 dan P0 ingin menggunakan *critical section*, maka ia dapat mengakses *critical section*-nya. Setelah selesai mengeksekusi *critical section*, P0 akan mengubah *turn* menjadi 1, yang artinya giliran P1 tiba dan P1 diperbolehkan mengakses *critical section*. Ketika *turn*= 1 dan P0 ingin menggunakan *critical section*, maka P0 harus menunggu sampai P1 selesai menggunakan *critical section* dan mengubah *turn* menjadi 0.

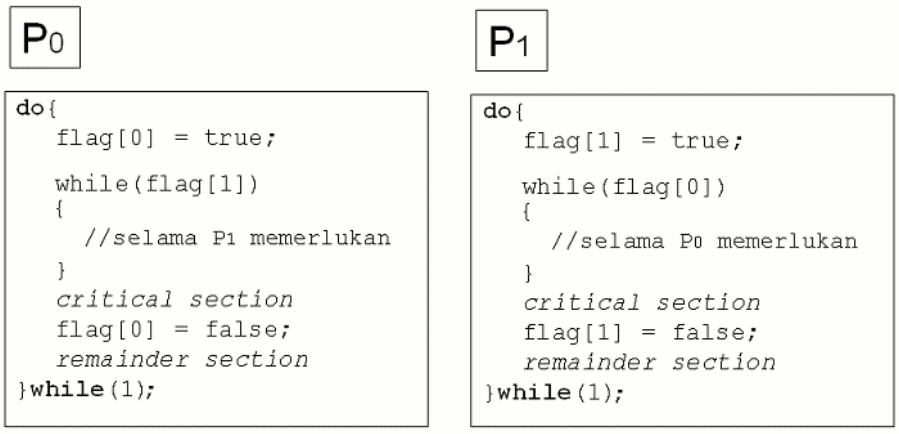
Ketika suatu proses sedang menunggu, proses tersebut masuk ke dalam *loop*, dimana ia harus terus-menerus mengecek variabel *turn* sampai berubah menjadi gilirannya. Proses menunggu ini disebut *busy waiting*. Sebenarnya *busy waiting* mesti dihindari karena proses ini menggunakan *CPU*. Namun untuk kasus ini, penggunaan *busy waiting* diijinkan karena biasanya proses menunggu hanya berlangsung dalam waktu yang singkat.

Pada algoritma ini masalah muncul ketika ada proses yang mendapat giliran memasuki *critical section* tapi tidak menggunakan gilirannya sementara proses yang lain ingin mengakses *critical section*. Misalkan ketika *turn*= 1 dan P1 tidak menggunakan gilirannya maka *turn* tidak berubah dan tetap 1. Kemudian P0 ingin menggunakan *critical section*, maka ia harus menunggu sampai P1 menggunakan *critical section* dan mengubah turn menjadi 0. Kondisi ini tidak memenuhi syarat *progress* karena P0 tidak dapat memasuki *critical section* padahal saat itu tidak ada yang menggunakan *critical section* dan ia harus menunggu P1 mengeksekusi non- *critical section*-nya sampai kembali memasuki *critical section*. Kondisi ini juga tidak memenuhi syarat *bounded waiting* karena jika pada gilirannya P1 mengakses *critical section* tapi P1 selesai mengeksekusi semua kode dan *terminate*, maka tidak ada jaminan P0 dapat mengakses *critical section* dan P0-pun harus menunggu selamanya.

19.3. Algoritma II

Algoritma II juga mencoba memecahkan masalah *critical section* untuk dua proses. Algoritma ini mengantisipasi masalah yang muncul pada algoritma I dengan mengubah penggunaan variabel *turn* dengan variabel *flag*. Variabel *flag* menyimpan kondisi proses mana yang boleh masuk *critical section*. Proses yang membutuhkan akses ke *critical section* akan memberikan nilai *flag*-nya *true*. Sedangkan proses yang tidak membutuhkan *critical section* akan men- *set* nilai flagnya bernilai *false*.

Gambar 19.2. Algoritma II



Suatu proses diperbolehkan mengakses *critical section* apabila proses lain tidak membutuhkan *critical section* atau flag proses lain bernilai *false*. Tetapi apabila proses lain membutuhkan *critical section* (ditunjukkan dengan nilai *flag*-nya *true*), maka proses tersebut harus menunggu dan "mempersilakan" proses lain menggunakan *critical section*-nya. Disini terlihat bahwa sebelum memasuki *critical section* suatu proses melihat proses lain terlebih dahulu (melalui *flag*-nya), apakah proses lain membutuhkan *critical section* atau tidak.

Awalnya *flag* untuk kedua proses diinisialisai bernilai *false*, yang artinya kedua proses tersebut tidak membutuhkan *critical section*. Jika P0 ingin mengakses *critical section*, ia akan mengubah *flag*[0] menjadi *true*. Kemudian P0 akan mengecek apakah P1 juga membutuhkan *critical section*, jika *flag*[1] bernilai *false* maka P0 akan menggunakan *critical section*. Namun jika *flag*[1] bernilai *true* maka P0 harus menunggu P1 menggunakan *critical section* dan mengubah *flag*[1] menjadi *false*.

Pada algoritma ini masalah muncul ketika kedua proses secara bersamaan menginginkan *critical section*, kedua proses tersebut akan men- *set* masing-masing *flag*-nya menjadi *true*. P0 men- *set* *flag*[0] = *true*, P1 men- *set* *flag*[1] = *true*. Kemudian P0 akan mengecek apakah P1 membutuhkan

critical section. P0 akan melihat bahwa *flag*[1] = *true*, maka P0 akan menunggu sampai P1 selesai menggunakan *critical section*. Namun pada saat bersamaan, P1 juga akan mengecek apakah P0 membutuhkan *critical section* atau tidak, ia akan melihat bahwa *flag*[0] = *true*, maka P1 juga akan menunggu P0 selesai menggunakan *critical section*-nya. Kondisi ini menyebabkan kedua proses yang membutuhkan *critical section* tersebut akan saling menunggu dan "saling mempersilahkan" proses lain untuk mengakses *critical section*, akibatnya malah tidak ada yang mengakses *critical section*. Kondisi ini menunjukkan bahwa Algoritma II tidak memenuhi syarat *progress* dan syarat *bounded waiting*, karena kondisi ini akan terus bertahan dan kedua proses harus menunggu selamanya untuk dapat mengakses *critical section*.

19.4. Algoritma III

Algoritma III ditemukan oleh G.L. Petterson pada tahun 1981 dan dikenal juga sebagai Algoritma Petterson. Petterson menemukan cara yang sederhana untuk mengatur proses agar memenuhi *mutual exclusion*. Algoritma ini adalah solusi untuk memecahkan masalah *critical section* pada dua proses. Ide dari algoritma ini adalah menggabungkan variabel yang di- *sharing* pada Algoritma I dan Algoritma II, yaitu variabel *turn* dan variabel *flag*. Sama seperti pada Algoritma I dan II, variabel *turn* menunjukkan giliran proses mana yang diperbolehkan memasuki *critical section* dan variabel *flag* menunjukkan apakah suatu proses membutuhkan akses ke *critical section* atau tidak.

Gambar 19.3. Algoritma III

P0

```
do{
   flag[0] = true;
   turn = 1;

   while(flag[1])&&(turn==1
   ){
     //selama P1 memerlukan
   dan giliran P1
   }
   critical section
   flag[0] = false;
   remainder section
}while(1)
```

P1

```
do{
   flag[1] = true;
   turn = 0;

   while(flag[0])&&(turn==0
   ){
     //selama P0 memerlukan
   dan giliran P0
   }
   critical section
   flag[1] = false;
   remainder section
}while(1)
```

Awalnya *flag* untuk kedua proses diinisialisai bernilai *false*, yang artinya kedua proses tersebut tidak membutuhkan akses ke *critical section*. Kemudian jika suatu proses ingin memasuki *critical section*, ia akan mengubah *flag*-nya menjadi *true* (memberikan tanda bahwa ia butuh *critical section*) lalu proses tersebut memberikan *turn* kepada lawannya. Jika lawannya tidak menginginkan *critical section* (*flag*-nya *false*), maka proses tersebut dapat menggunakan *critical section*, dan setelah selesai menggunakan *critical section* ia akan mengubah *flag*-nya menjadi *false*. Tetapi apabila proses lawannya juga menginginkan *critical section* maka proses lawan-lah yang dapat memasuki *critical section*, dan proses tersebut harus menunggu sampai proses lawan menyelesaikan *critical section* dan mengubah *flag*-nya menjadi *false*.

Misalkan ketika P0 membutuhkan *critical section*, maka P0 akan mengubah *flag*[0] = *true*, lalu P0 mengubah *turn*= *1*. Jika P1 mempunyai *flag*[1] = *false*, (berapapun nilai *turn*) maka P0 yang dapat mengakses *critical section*. Namun apabila P1 juga membutuhkan *critical section*, karena *flag*[1] = *true* dan *turn*= 1, maka P1 yang dapat memasuki *critical section* dan P0 harus menunggu sampai P1 menyelesaikan *critical section* dan mengubah *flag*[1] = *false*, setelah itu barulah P0 dapat mengakses *critical section*.

Bagaimana bila kedua proses membutuhkan *critical section* secara bersamaan? Proses mana yang dapat mengakses *critical section* terlebih dahulu? Apabila kedua proses (P0 dan P1) datang bersamaan,

kedua proses akan menset masing-masing *flag* menjadi *true* (*flag*[0] = *true* dan *flag*[1] = *true*), dalam kondisi ini P0 dapat mengubah turn = 1 dan P1 juga dapat mengubah turn = 0. Proses yang dapat mengakses *critical section* terlebih dahulu adalah proses yang terlebih dahulu mengubah turn menjadi *turn* lawannya. Misalkan P0 terlebih dahulu mengubah *turn*= 1, lalu P1 akan mengubah *turn*= 0, karena *turn* yang terakhir adalah 0 maka P0-lah yang dapat mengakses *critical section* terlebih dahulu dan P1 harus menunggu.

Algoritma III memenuhi ketiga syarat yang dibutuhkan. Syarat *progress* dan *bounded waiting* yang tidak dipenuhi pada Algoritma I dan II dapat dipenuhi oleh algoritma ini karena ketika ada proses yang ingin mengakses *critical section* dan tidak ada yang menggunakan *critical section* maka dapat dipastikan ada proses yang bisa menggunakan *critical section*, dan proses tidak perlu menunggu selamanya untuk dapat masuk ke *critical section*.

19.5. Algoritma Tukang Roti

Algoritma Tukang Roti adalah solusi untuk masalah *critical section* pada n-buah proses. Algoritma ini juga dikenal sebagai *Lamport's Baker Algorithm*. Ide algoritma ini adalah dengan menggunakan prinsip penjadwalan seperti yang ada di tempat penjualan roti. Para pelanggan yang ingin membeli roti sebelumnya harus mengambil nomor urut terlebih dahulu dan urutan orang yang boleh membeli ditentukan oleh nomor urut yang dimiliki masing-masing pelanggan tersebut.

Prinsip algoritma ini untuk menentukan proses yang boleh mengakses *critical section* sama seperti ilustrasi tukang roti diatas. Proses diibaratkan pelanggan yang jumlahnya n-buah dan tiap proses yang membutuhkan *critical section* diberi nomor yang menentukan proses mana yang diperbolehkan untuk masuk kedalam *critical section*. Nomor yang diberikan adalah sekuensial atau terurut, tapi seperti juga nomor urut yang ada di tukang roti, tidak ada jaminan bahwa tiap proses mendapat nomor urut yang berbeda. Untuk mengatasinya digunakan parameter lain yaitu proses ID. Dikarenakan tiap proses memiliki proses ID yang unik dan terurut maka dapat dipastikan hanya satu proses yang dapat mengakses *critical section* dalam satu waktu. Proses yang dapat mengakses *critical section* terlebih dahulu adalah proses yang memiliki nomor urut paling kecil. Apabila ada beberapa proses yang memiliki nomor yang sama maka proses yang mempunyai nomor ID paling kecil yang akan dilayani terlebih dahulu.

19.6. Rangkuman

Solusi *critical section* harus memenuhi ketiga syarat berikut:
1. *Mutual Exclusion*
2. *Progress*
3. *Bounded Waiting*

Algoritma I dan II terbukti tidak dapat memecahkan masalah *critical section* untuk dua proses karena tidak memenuhi syarat *progress* dan *bounded waiting*. Algoritma yang dapat menyelesaikan masalah *critical section* pada dua proses adalah Algoritma III. Sedangkan untuk masalah *critical section* pada n-buah proses dapat diselesaikan dengan menggunakan Algoritma Tukang Roti

Rujukan

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

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

[WikiCS2007] Wikipedia, The Free Encyclopedia. 2007 . *Critical Section http:// en.wikipedia.org/ wiki/ Critical Section*. Diakses 5 Maret 2007.

Bab 20. Perangkat Sinkronisasi

20.1. Pendahuluan

Sinkronisasi adalah proses pengaturan jalannya beberapa proses pada saat yang bersamaan. Tujuan utama sinkronisasi adalah menghindari terjadinya inkonsistensi data karena pengaksesan oleh beberapa proses yang berbeda (*mutual exclusion*) serta untuk mengatur urutan jalannya proses-proses sehingga dapat berjalan dengan lancar dan terhindar dari *deadlock* atau *starvation*.

Sinkronisasi umumnya dilakukan dengan bantuan perangkat sinkronisasi. Dalam bab ini akan dibahas beberapa perangkat sinkronisasi, yaitu : `TestAndSet()`, Semafor, dan Monitor.

20.2. `TestAndSet()`

Sebagian besar sistem komputer saat ini menyediakan instruksi-instruksi perangkat keras khusus yang mengizinkan kita untuk menguji dan memodifikasi nilai dari sebuah variabel. Instruksi ini disebut instruksi `TestAndSet()`. Instruksi ini bersifat atomik, yaitu instruksi yang tidak dapat diinterupsi. Kita dapat menggunakan instruksi khusus ini untuk memecahkan masalah *critical section* yang sederhana. Contoh instruksi `TestAndSet()`:

Contoh 20.1. TestAndSet()

```
boolean TestAndSet (boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Karakteristik penting dari instruksi ini adalah pengeksekusian instruksi secara atomik. Jadi, jika dua instruksi `TestAndSet()` dijalankan secara serentak, kedua instruksi tersebut akan dieksekusi secara sekuensial. Dengan mendeklarasikan variabel `lock`, kita dapat mengimplementasikan *mutual exclusion* seperti pada contoh berikut ini:

Contoh 20.2. TestAndSet() dengan *mutual exclusion*

```
do {
   while (TestAndSetLock (&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Walaupun algoritma di atas memenuhi persyaratan *mutual exclusion*, algoritma tersebut belum memenuhi persyaratan *bounded waiting*.

Instruksi `TestAnd Set()` di bawah ini sudah memenuhi semua persyaratan *critical section*.

Contoh 20.3. TestAndSet() yang memenuhi *critical section*

```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet (&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
} while (TRUE);
```

Sebelumnya kita harus meginisialisasikan dua variabel `boolean`.

```
boolean waiting[n];
boolean lock;
```

Kedua variabel tersebut diinisialisasikan menjadi `false`. Untuk membuktikan bahwa persyaratan *mutual exclusion* terpenuhi, proses P_i dapat memasuki *critical section*-nya hanya jika `waiting[i] == false` atau `key == false`. Nilai dari `key` menjadi `false` hanya jika `TestAndSet()` dieksekusi. Proses pertama yang mengeksekusi `TestAndSet()` akan menemukan `key == false` sehingga proses lain harus menunggu. Variabel `waiting[i]` menjadi `false` hanya jika proses lain meninggalkan *critical section*-nya. Untuk menjaga persyaratan *mutual exclusion* terpenuhi, hanya ada satu `waiting[i]` yang diset `false`.

Untuk membuktikan persyaratan *progress* terpenuhi, kita lihat pembuktian untuk *mutual exclusion*. Sebuah proses keluar dari *critical section*-nya pada saat `lock` diset `false` atau `waiting[j]` diset `false`. Keduanya mengizinkan sebuah proses yang sedang menunggu untuk masuk ke *critical section*.

Untuk membuktikan persyaratan *bounded waiting* terpenuhi, kita perhatikan bahwa ketika sebuah proses meninggalkan *critical section*-nya, proses tersebut melihat *array of waiting* dalam urutan siklus (i + 1, i + 2, ..., n - 1, 0, ..., i - 1). Proses pertama yang berada dalam *enter section* (`waiting[j] == true`) adalah proses selanjutnya yang akan memasuki *critical section*.

20.3. Semafor

Dalam kehidupan nyata, semafor adalah sistem sinyal yang digunakan untuk berkomunikasi secara *visual*. Dalam *software*, semafor adalah sebuah variabel bertipe `integer` yang selain saat inisialisasi,

hanya dapat diakses melalui dua operasi standar, yaitu *increment* dan *decrement*. Semafor digunakan untuk menyelesaikan masalah sinkronisasi secara umum. Berdasarkan jenisnya, semafor hanya bisa memiliki nilai 1 atau 0, atau lebih dari sama dengan 0. Konsep semafor pertama kali diajukan idenya oleh Edsger Dijkstra pada tahun 1967.

Operasi standar pada semafor (dalam bahasa pemrograman C):

```
void kunci(int sem_value) {
   while(sem_value <= 0);
   sem_value--;
}

void buka(int sem_value) {
   sem_value++;
}
```

Nama asli dari operasi tersebut sebenarnya adalah *Proberen (test)* dan *Verhogen (increment)*. Namun, sebutan untuk 2 method ini sangat beragam, antara lain sering dikenal dengan nama : `release` dan `acquire`, `P` dan `V`, serta `kunci` dan `buka`. Dalam buku ini akan digunakan istilah `buka` dan `kunci`. Fungsi `wait` dipanggil ketika *thread* akan memasuki *critical section*-nya atau ketika *thread* akan memakai *resource* yang tersedia. Jika `sem_value` kurang dari sama dengan 0, *thread* tersebut harus menunggu sampai *thread* lain memanggil fungsi `buka`. Fungsi `buka` dipanggil ketika *thread* meningggalkan *critical section*-nya atau ketika melepaskan *resource* yang telah digunakannya. Tentu saja kedua operasi tersebut harus bersifat atomik karena `sem_value` dapat diakses oleh beberapa proses (*shared resource*).

Semafor memiliki dua jenis, yaitu:
1. ***Binary semaphore .*** Semafor ini hanya memiliki nilai 1 atau 0. Sering juga disebut sebagai semafor primitif
2. ***Counting semaphore .*** Semafor ini memiliki nilai 0, 1, serta integer lainnya. Banyak sistem operasi yang tidak secara langsung mengimplementasikan semafor ini, tetapi dengan memanfaatkan *binary semaphore*

20.4. Fungsi Semafor

Seperti telah disebutkan sebelumnya, semafor berfungsi untuk menangani masalah sinkronisasi secara umum, yaitu:
1. ***Mutual Exclusion .*** Sesuai dengan prinsip *mutual exclusion*, jika suatu *thread* sedang berada dalam *critical section*-nya, *thread* lain harus menunggu *thread* tersebut keluar dari *critical section*-nya sebelum dapat memasuki *critical section*-nya sendiri. Di sinilah semafor digunakan, *thread* yang akan memasuki *critical section*-nya akan memanggil fungsi `kunci` terlebih dahulu. Jika tidak ada *thread* lain yang sedang berada dalam *critical section*, *thread* ini akan memasuki *critical section*-nya. Jika terdapat *thread* lain yang sedang berada dalam *critical section*-nya, *thread* ini harus menunggu.Setelah *thread* keluar dari *critical section*-nya, *thread* tersebut akan memanggil fungsi `buka` sehingga sem_value akan naik menjadi lebih dari 0, dan satu (dari beberapa) *thread* yang sedang menunggu akan mendapatkan giliran untuk memasuki *critical section*-nya.

   Sebagai contoh, misalnya terdapat dua buah *thread* yang sedang berjalan bersamaan:

```
            thread A:            thread B:
count = count + 1;   count = count + 1;
```

   Thread A dan B mengakses variabel yang sama, yaitu `count` sehingga *thread* A dan B harus berjalan satu-satu. Untuk itu digunakan semafor mutex yang berupa *binary semaphore* dengan nilai awal 1.

```
           thread A:          thread B:
kunci(mutex);        kunci(mutex);
count = count + 1;  count = count + 1;
buka(mutex);      buka(mutex);
```

Thread manapun yang mengeksekusi `kunci` terlebih dahulu akan jalan terus, sedangkan *thread* yang tiba belakangan akan menunggu sampai *thread* yang sudah berjalan terlebih dahulu mengeksekusi `buka`, setelah itu kedua *thread* berjalan lagi dengan normal.

2. ***Resource Controller* .** Bayangkan sebuah restoran yang setiap malamnya ramai dikunjungi pelanggan. Kapasitas restoran terbatas, tetapi pemilik restoran memiliki kebijakan bahwa semua pengunjung yang datang akan mendapatkan kesempatan untuk makan, dengan konsekuensi yaitu pelanggan harus sabar menunggu gilirannya. Oleh karena itu, dikerahkanlah pegawai restoran untuk menahan tamu di luar jika restoran penuh lalu mempersilahkan tamu masuk jika tempat telah tersedia.Dari analogi di atas, pelanggan adalah *thread*, kapasitas restoran adalah *resource*, dan pegawai restoran adalah semafor. Semafor menyimpan banyaknya *resource* yang tersedia. Saat *thread* ingin memakai *resource* ia akan memanggil fungsi `kunci`. Jika *resource* masih tersedia, *thread* bisa langsung menggunakannya, sebaliknya jika semua *resource* sedang dipakai, *thread* tersebut harus menunggu. Setelah *resource* selesai dipakai *thread* akan memanggil fungsi `buka` sehingga *resource* yang bebas bertambah.

   Contohnya dapat kita lihat pada kasus berikut: Terdapat tiga buah *thread* yang berjalan bersamaan. *Resource* yang tersedia hanya cukup untuk dua buah *thread*.

```
thread A:          thread B:          thread C:
//critical section  //critical section //critical section
```

   Tentu saja harus diatur agar pada suatu saat hanya ada dua buah *thread* yang berada pada *critical section*-nya. Hal ini dilakukan dengan menggunakan semafor *multiplex* yaitu sebuah *counting semaphore* dengan nilai awal sama dengan jumlah *resource* yang tersedia yaitu dua.

```
thread A:          thread B:          thread C:
kunci(multiplex);    kunci(multiplex);    kunci(multiplex);
//critical section  //critical section  //critical section
buka(multiplex);  buka(multiplex);  buka(multiplex);
```

   Jika dua buah thread sedang berada dalam *critical section*, *thread* berikutnya yang akan memasuki *critical section* harus menunggu kedua *thread* tersebut selesai untuk dapat memasuki *critical section*-nya.

3. **Sinkronisasi Antar-Proses.** Ada kalanya suatu *thread* memerlukan *resource* yang dihasilkan oleh *thread* lainnya. Oleh karena itu dibutuhkan suatu mekanisme untuk mengatur urutan eksekusi *thread*. Mekanisme ini dilakukan dengan memanfaatkan semafor.

   Sebagai contoh, misalnya terdapat dua buah *thread* yang sedang berjalan bersamaan:

```
thread A:              thread B:
count = count + 1;      count = count * 2;
```

Nilai awal dari variabel `count` adalah 5, nilai akhir yang diinginkan adalah 12, oleh karena itu *thread* A harus dieksekusi sebelum *thread* B. Agar hal ini dapat terjadi dibutuhkan suatu semafor *mutex* yang merupakan sebuah *binary semaphore* dengan nilai awal 0.

```
thread A:                thread B:
count = count + 1;       kunci(mutex);
buka(mutex);           count = count * 2;
```

Thread B akan menunggu sampai eksekusi *thread* A selesai sebelum melanjutkan.

Jika kita cermati fungsi `kunci`, *thread* akan terus berada dalam *waiting loop* sampai sem_value naik lebih dari 0. Padahal, di dalam *loop* tersebut *thread* tidak melakukan tugas apa-apa. Inilah yang disebut dengan *busy waiting* (semafor jenis ini disebut dengan semafor *spinlock*). Hal ini tentu saja akan berakibat buruk terhadap kinerja CPU, karena *loop* tersebut membutuhkan CPU *cycle*, sementara *loop* tersebut tidak menghasilkan apa-apa, jadi sama saja dengan menyia-nyiakan CPU *cycle*.

Untuk mengatasi hal ini, dibuatlah modifikasi dari semafor, yaitu dengan menambahkan *waiting queue* pada masing-masing semafor. Tujuannya adalah agar *thread* yang harus menunggu dipindahkan ke *waiting queue* dan kegiatannya dihentikan sementara.

Ketika *thread* keluar dari *critical section*-nya, *thread* tersebut akan memanggil fungsi `buka` yang akan mengeluarkan satu (dari beberapa) *thread* yang berada dalam *waiting queue* lalu membangunkannya untuk kemudian memasuki *critical section*-nya.

Struktur semafor ini menjadi (masih dalam bahasa C):

```
void buka(int sem_value)
{
   sem_value++;
   if(sem_value <= 0)
   {
      /*keluarkan satu thread dari waiting queue*/
      /*aktifkan thread tersebut*/
   }
}

void kunci(int sem_value)
{
   sem_value--;
   if(sem_value < 0)
   {
      /*masukkan thread ke dalam waiting queue*/
      /*blok thread tersebut*/
   }
}
```

Berbeda dengan semafor yang biasa, pada semafor yang telah dimodifikasi ini sem_value bisa menjadi negatif. Jika kita renungkan maknanya, ternyata ketika semafor bernilai negatif, nilai tersebut melambangkan banyaknya *thread* yang berada pada *waiting queue* semafor tersebut.

Keuntungan menggunakan semafor:
1. dari segi *programming*, penanganan masalah sinkronisasi dengan semafor umumnya rapi dan teratur, sehingga mudah untuk dibuktikan kebenarannya

2. semafor diimplementasikan dalam hard code sehingga penggunaannya bersifat portabel.

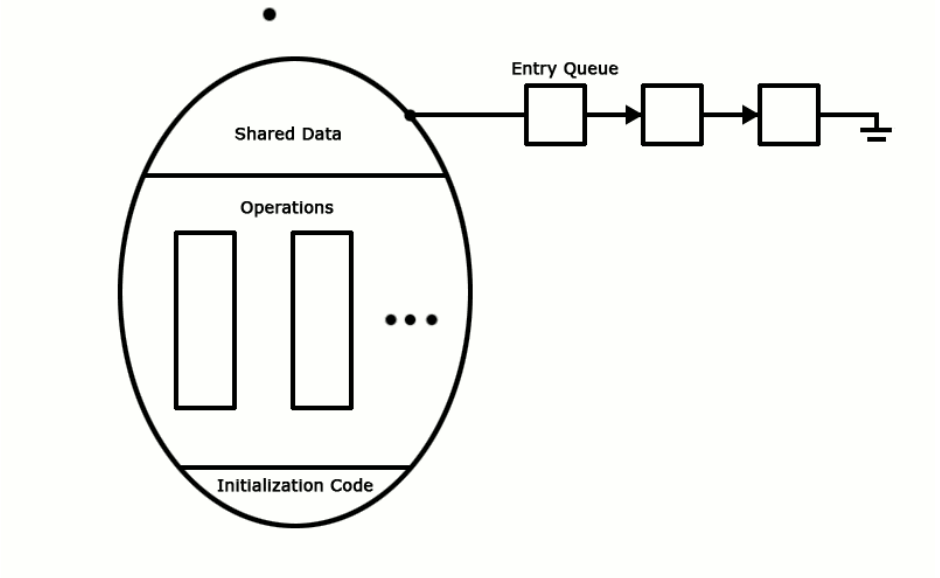
20.5. Monitor

Monitor adalah suatu tipe data abstrak yang dapat mengatur aktivitas serta penggunaan *resource* oleh beberapa *thread*. Ide monitor pertama kali diperkenalkan oleh C.A.R Hoare dan Per Brinch-Hansen pada awal 1970-an.

Monitor terdiri atas data-data *private* dengan fungsi-fungsi *public* yang dapat mengakses data-data tersebut. *Method-method* dalam suatu monitor sudah dirancang sedemikian rupa agar hanya ada satu buah *method* yang dapat bekerja pada suatu saat. Hal ini bertujuan untuk menjaga agar semua operasi dalam monitor bersifat *mutual exclusion*.

Monitor dapat dianalogikan sebagai sebuah bangunan dengan tiga buah ruangan yaitu satu buah ruangan kontrol, satu buah ruang-tunggu-masuk, satu buah ruang-tunggu-dalam. Ketika suatu *thread* memasuki monitor, ia memasuki ruang-tunggu-masuk (*enter*). Ketika gilirannya tiba, *thread* memasuki ruang kontrol (*acquire*), di sini *thread* menyelesaikan tugasnya dengan *shared resource* yang berada di ruang kontrol (*owning*). Jika tugas *thread* tersebut belum selesai tetapi alokasi waktu untuknya sudah habis atau *thread* tersebut menunggu pekerjaan *thread* lain selesai, *thread* melepaskan kendali atas monitor (*release*) dan dipindahkan ke ruang-tunggu-dalam (*waiting queue*). Ketika gilirannya tiba kembali, *thread* memasuki ruang kontrol lagi (*acquire*). Jika tugasnya selesai, ia keluar dari monitor (*release and exit*).

Gambar 20.1. Monitor

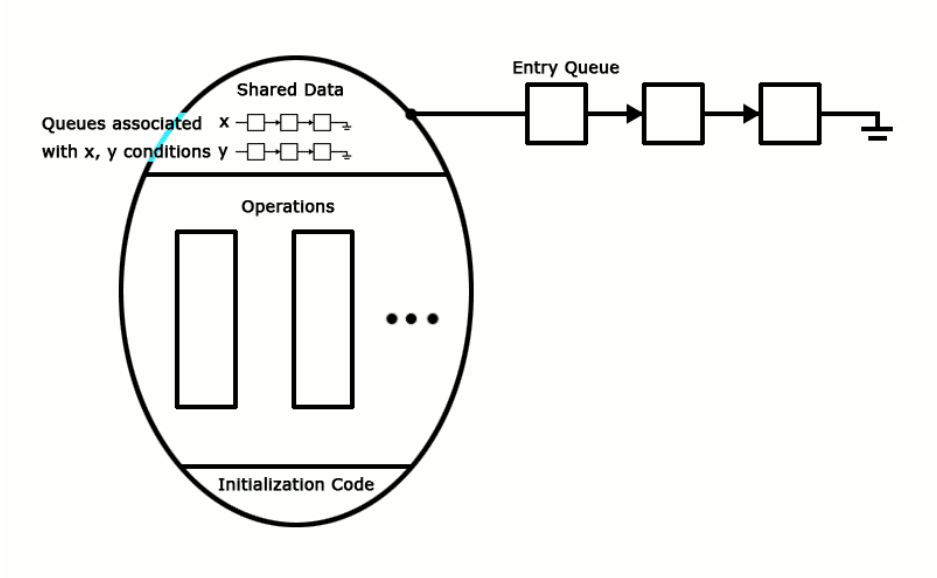


Karena masalah sinkronisasi begitu rumit dan beragam, monitor menyediakan tipe data *condition* untuk *programmer* yang ingin menerapkan sinkronisasi yang sesuai untuk masalah yang dihadapinya. *Condition* memiliki operasi-operasi:

1. `Wait`, sesuai namanya *thread* yang memanggil fungsi ini akan dihentikan kerjanya.
2. `Signal`, jika suatu *thread* memanggil fungsi ini, satu (dari beberapa) *thread* yang sedang menunggu akan dibangunkan untuk bekerja kembali. Operasi ini hanya membangunkan tepat satu buah *thread* yang sedang menunggu. Jika tidak ada *thread* yang sedang menunggu, tidak akan terjadi apa-apa (bedakan dengan operasi `buka` pada semafor).

Ilustrasi monitor dengan *condition variable*:

Gambar 20.2. Monitor dengan condition variable



Bayangkan jika pada suatu saat sebuah *thread* A memanggil fungsi `signal` pada *condition* x (`x.signal()`) dan ada sebuah *thread* B yang sedang menunggu operasi tersebut (B telah memanggil fungsi `x.wait()` sebelumnya), ada dua kemungkinan keadaan *thread* A dan B setelah A mengeksekusi `x.signal()`:

a. `Signal-and-Wait`, A menunggu sampai B keluar dari monitor atau menunggu *condition* lain yang dapat mengaktifkannya.
b. `Signal-and-Continue`, B menunggu sampai A keluar dari monitor atau menunggu *condition* lain yang dapat mengakifkannya.

Monitor dikembangkan karena penggunaan semafor yang kurang praktis. Hal itu disebabkan kesalahan pada penggunaan semafor tidak dapat dideteksi oleh *compiler*. Keuntungan memakai monitor:

1. Kompilator pada bahasa pemrograman yang telah mengimplementasikan monitor akan memastikan bahwa *resource* yang dapat diakses oleh beberapa *thread* dilindungi oleh monitor, sehingga prinsip *mutual exclusion* tetap terjaga.
2. Kompilator bisa memeriksa kemungkinan adanya *deadlock*.

20.6. Monitor Java

Java adalah bahasa pemrograman yang telah menerapkan *multithreading*. Oleh karena itu, diperlukan suatu sistem yang dapat mensinkronisasi *thread-thread* tersebut. *Java Virtual Machine* memakai monitor untuk sinkronisasi *thread*.

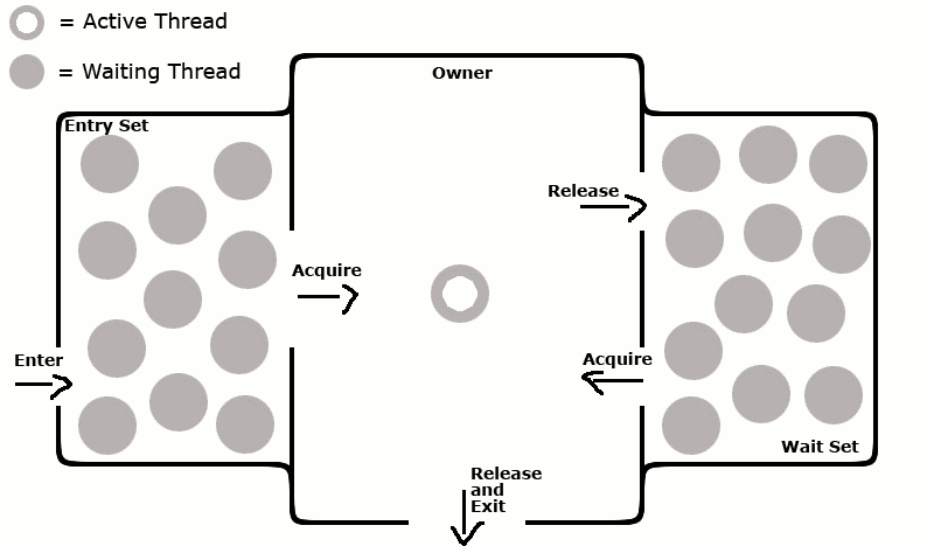
Monitor JVM mendukung dua jenis sinkronisasi yaitu:

1. ***Mutual Exclusion .*** Seperti pada monitor biasa, monitor mengatur *thread-thread* yang berbagi *resource* agar dapat bekerja sendiri-sendiri dengan baik tanpa ada yang saling menggangu.Pada umumnya pengaturan ini ditujukan untuk *thread-thread* yang saling berbagi *resource*, sehingga *thread-thread* yang tidak berbagi *resource* dapat dibiarkan bekerja sendiri. Akan tetapi, dalam JVM *thread* dengan prioritas lebih tinggi akan mendominasi *thread* dengan prioritas rendah sehingga CPU dapat dimonopoli oleh *thread* dengan prioritas tinggi dan *thread* dengan prioritas rendah tidak mendapatkann CPU *time*. Untuk mengatasi hal ini digunakanlah monitor. Secara umum monitor digunakan untuk melindungi akses terhadap *resource* yang ada di dalamnya.
2. ***Cooperation .*** *Cooperation* berarti membantu beberapa *thread* untuk bekerjasama mencapai tujuannya. *Cooperation* berperan terutama ketika suatu *thread* menginginkan *resource* dalam keadaan tertentu dan *thread* lain bertanggung jawab untuk membuat *resource* tersebut menjadi seperti yang diinginkan oleh *thread* pertama.Dalam JVM, semua object dan class diasosiasikan dengan monitor (object monitor dan class monitor). Untuk object, monitor melindungi

variabel dari instansiasi object tersebut, untuk class, monitor melindungi class variable.Untuk mengimplementasikan *mutual exclusion*, JVM mengasosiasikan `lock` untuk setiap *object* dan *class*. Jika suatu *thread* mendapatkan `lock`-nya, *thread* lain tidak akan bisa mengambil `lock` tersebut sebelum *thread* itu melepaskan `lock`-nya (suatu *thread* yang mendapatkan `lock`-nya mendapat akses untuk menggunakan monitor). Hal inilah yang disebut dengan *object locking*.

Ilustrasi monitor Java:

Gambar 20.3. Monitor JVM



Cara penggunaan monitor java

1. ***Synchronized Statements .*** Untuk membuat sebuah *synchronized statement*, gunakan *keyword* `synchronized` dengan ekspresi yang me- *refer* ke suatu *object* contoh:

```
void reverseOrder()
   {
      synchronized(this)
      {
         /*ekspresi method*/
      }
   }
```

2. ***Synchronized Methods .*** untuk membuat suatu *synchronized methods*, cukup dengan menambahkan *keyword* `synchronized` di depan nama *method* tersebut pada deklarasinya contoh:

```
synchronized void reverseOrder()
{
   /*ekspresi method*/
}
```

Seperti halnya pada monitor biasa, monitor JVM juga menyediakan *method-method* tambahan agar *programmer* dapat dengan leluasa menyelesaikan masalah sinkronisasi yang dihadapinya. *Method-method* tersebut adalah:

a. **`void wait().`** *thread* yang memanggil *method* ini akan mesuk ke *waiting queue*. *Thread* tersebut akan menunggu sampai ada *thread* lain yang memanggil *method* `notify()`.
b. **`void wait(long timeout)` dan `void wait(long timeout, int nanos).`** Mirip seperti *method* `wait`, bedanya ketika *timeout* (dalam milisekon) habis dan tidak ada *thread* lain yang memanggil *method* `notify()`, *thread* dibangunkan oleh JVM.
c. **`void notify().`** Membangunkan satu *thread* yang ada di *waiting queue* yang dipilih secara *random*. Jika tidak ada *thread* di *waiting queue*, maka tidak akan terjadi apa-apa.
d. **`void notifyAll().`** Mirip seperti `notify()`, tetapi yang dibangunkan adalah semua *thread* yang berada di *waiting queue*.

20.7. Rangkuman

- **Instruksi `TestAndSet().`** instruksi atomik yang dapat digunakan untuk menangani masalah *critical section*.
- **Semafor.** sebuah variabel yang hanya dapat diakses oleh dua buah operasi standar yaitu *increment* dan *decrement*. Dua buah jenis semafor, yaitu *Binary Semaphore* dan *Counting Semaphore*. Semafor berfungsi untuk menangani masalah *critical section*, mengatur alokasi *resource*, dan sinkronisasi antarproses.
- **Monitor.** digunakan untuk menangani masalah yang muncul karena pemakaian semafor. Monitor menjamin *mutual exclusion*. Untuk menangani masalah sinkronisasi yang lebih rumit monitor menyediakan *condition variable*.
- **JVM.** mengimplementasikan monitor. Monitor JVM bekerja dengan *object locking* dan *method-method* `wait()` serta `notify()`. Monitor JVM dapat digunakan dengan menggunakan *keyword synchronized*.

Rujukan

[Christopher2001] Thomas W. Christopher dan George K. Thiruvathukal. 2001 . *High Performance Java Platform Computing*. First Edition. Prentice Hall Ptr.

[Downey2005] Allen B. Downey. 2005 . *The Little Book of Semaphores*. Second Edition. Green Tea Press.

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

[Venners2000] Bill Venners. 2000 . *Inside the Java Virtual Machine*. Second Edition. McGraw-Hill Companies.

[WEBJAVA] Sun Microsystems. 1995 . *Synchronized methods (The Java ™; Tutorials > Essential Classes > Concurrency) http://java.sun.com/docs/books/tutorial/essential/concurrency/syncmeth.html*. Diakses 8 Maret 2007.

[WEBJAVA] Sun Microsystems. 1995 . *Thread Interference (The Java ™; Tutorials > Essential Classes > Concurrency) http://java.sun.com/docs/books/tutorial/essential/concurrency/interfere.html*. Diakses 8 Maret 2007.

[WEBWIKI] Wikipedia. 2007 . *Test-and-set - Wikipedia, the free encyclopedia http://en.wikipedia.org/wiki/Test-and-set*. Diakses 8 Maret 2007.

Bab 21. Transaksi Atomik

21.1. Pendahuluan

Transaksi merupakan sekumpulan instruksi atau operasi yang menjalankan sebuah fungsi logis. Salah satu sifat yang harus dimiliki oleh transaksi adalah keatomikan. Sifat ini menjadikan suatu transaksi sebagai suatu kesatuan sehingga pengeksekusian instruksi-instruksi di dalamnya harus dijalankan secara keseluruhan atau tidak dijalankan sama sekali. Hal ini dilakukan untuk menghindari terjadinya kesalahan hasil eksekusi bila operasi-operasi yang ada dijalankan hanya sebagian saja.

Transaksi atomik dapat diilustrasikan pada kasus transfer uang antar rekening. Pada kasus ini, setidaknya akan dilakukan dua buah operasi, yaitu debit pada rekening pengirim dan kredit pada rekening penerima. Kedua buah operasi tersebut harus dijalankan keseluruhan untuk menjaga agar data pada penerima dan pengirim uang konsisten. Bila hanya salah satu operasi saja yang dilakukan, misalnya hanya dilakukan operasi debit pada rekening pengirim, maka pada sisi pengirim akan merasa bahwa ia telah melakukan transfer uang padahal di sisi penerima merasa belum menerima uang yang ditransfer. Dengan demikian, akan terjadi kesalahpahaman di antara kedua pihak.

Untuk mempertahankan sifat keatomikan suatu transaksi, maka operasi-operasi yang sudah dijalankan hasilnya harus disimpan sementara agar bila terjadi kegagalan sistem, transaksi dapat dibatalkan (belum ada data yang berubah). Bab ini akan membahas mengenai bagaimana sistem operasi mempertahankan sifat atomik dari transaksi, yaitu mengenai proses penyimpanan hasil eksekusi instruksi dan mengenai transaksi-transaksi atomik yang dijalankan secara bersamaan agar tetap bersifat atomik.

21.2. Model Sistem

Sebuah transaksi harus memenuhi beberapa syarat. Syarat-syarat ini biasa disebut ACID *properties* dan harus terpenuhi agar pada saat terjadi *system crash*, pemulihan pada transaksi tersebut dapat dilakukan. ACID *properties* terdiri dari:

- ***Atomicity*** **.** Sebuah transaksi dijalankan secara keseluruhan atau tidak dijalankan sama sekali.
- ***Consistency*** **.** Sebuah transaksi mengubah sistem dari sebuah *state* yang konsisten menjadi *state* konsisten yang lain.
- ***Isolation*** **.** Transaksi yang belum selesai tidak dapat menunjukkan hasilnya ke transaksi yang lain sebelum transaksi tersebut *commit*.
- ***Durability*** **.** Ketika sebuah transaksi *commit*, sistem akan menjamin hasil dari operasi akan tetap, bahkan ketika terjadi kegagalan pada suatu *subsequent*.

Dengan sifat *atomicity*, sebuah transaksi dapat dipastikan dijalankan secara keseluruhan atau jika terjadi *system crash* seluruh data yang telah diubah oleh transaksi tersebut dikembalikan ke *state* awal sebelum transaksi dilakukan. *State* awal yang konsisten akan diubah menjadi *state* lain yang juga konsisten setelah sebuah transaksi sukses dijalankan dengan asumsi tidak terjadi *interleave* antar transaksi. Oleh karena itu diperlukan *consistency* pada transaksi. Dengan sifat *isolation* dapat juga dikatakan setiap *schedule* (rangkaian beberapa transaksi) bersifat *serializable*, yang akan dibahas pada bagian serialisasi. Setiap kali transaksi telah berhasil dijalankan akan dijamin bahwa hasil *update* data akan terjaga.

21.3. Pemulihan Berbasis *Log*

Log merupakan sebuah struktur data yang dibuat sistem di *stable storage*. Seperti telah dijelaskan, *atomicity* merupakan salah satu komponen penting dalam sebuah transaksi. Sebuah transaksi secara sederhana merupakan serangkaian operasi *read* dan *write* yang diakhiri dengan sebuah operasi *commit* atau *abort*. Sebuah operasi *commit* menandakan bahwa transaksi tersebut telah berakhir dengan sukses,

sedangkan operasi *abort* menandakan bahwa transaksi tersebut telah berakhir karena adanya *logical error* atau kegagalan sistem.

Ada kemungkinan sebuah transaksi yang *abort* telah memodifikasi data yang diaksesnya, sehingga *state* datanya tidak sama jika transaksi berhasil dijalankan secara atomik. Dengan sifat keatomikan, diharapkan transaksi yang *abort* tidak mengubah *state* dari data yang telah dimodifikasinya. Oleh karena itu, diperlukan *rolled-back* ke kondisi sebelum transaksi dijalankan untuk menjamin keatomikan suatu transaksi.

Untuk menentukan bagaimana sistem menjamin keatomikan, kita perlu mengetahui perangkat yang digunakan untuk menyimpan data yang diakses transaksi tersebut. Media penyimpanan berdasarkan kecepatan relatif, kapasitas dibagi menjadi:

- ***Volatile storage .*** Jika terjadi *system crashes*, informasi yang ada di *volatile storage* biasanya tidak dapat diselamatkan. Tetapi, akses ke *volatile storage* sangat cepat. Contohnya *main memory* dan *cache memory*.
- ***Non-volatile storage .*** Jika terjadi *system crashes*, informasi yang ada di *non-volatile storage* biasanya masih dapat diselamatkan. Tetapi, akses ke *non-volatile storage* tidak secepat *volatile storage*. Contohnya *disk* dan *magnetic tape*.
- ***Stable storage .*** Informasi yang ada di *stable storage* biasanya tidak pernah hilang. Untuk mengimplementasikan penyimpanan seperti itu, kita perlu mereplikasi informasi yang dibutuhkan ke beberapa *non-volatile storage* (biasanya *disk*) dengan *failure modes* yang independen.

21.4. *Checkpoint*

Salah satu cara untuk menjamin keatomikan suatu transaksi adalah adanya *rolled-back* ke kondisi sebelum transaksi. Untuk melakukan *rolled-back* tersebut, kita harus menyimpan semua informasi yang berkaitan dengan modifikasi data pada transaksi tersebut di *stable storage*. Metode yang sering digunakan untuk menyimpan hal tersebut adalah *write-ahead logging*. Dengan metode ini, setiap *log* menyimpan setiap operasi *write* dari sebuah transaksi dan terdiri dari:

- ***Transaction name .*** Nama yang unik dari transaksi yang menjalankan operasi *write*.
- ***Data item name .*** Nama yang unik dari *data item* yang ditulis.
- ***Old value .*** Nilai *data item* sebelum operasi *write* dilakukan.
- ***New value .*** Nilai yang akan dimiliki *data item* setelah dilakukan operasi *write*.

Selain untuk menyimpan operasi *write*, ada *log* lain yang menyimpan informasi-informasi penting lainnya seperti *start* dari sebuah transaksi dan *commit* atau *abort* sebuah transaksi. Sebelum transaksi mulai dilaksanakan, *log* menyimpan operasi *start* dan selama transaksi dijalankan, *log* mencatat setiap operasi *write* yang terjadi. Ketika terjadi *commit*, *log* menyimpan operasi *commit*.

Dengan menggunakan *log*, sistem dapat menangani kegagalan yang terjadi, sehingga tidak ada informasi yang hilang pada *non-volatile storage*. Algoritma pemulihan menggunakan dua buah prosedur:

- ***Undo .*** Mengembalikan nilai semua data yang telah di- *update* oleh transaksi tersebut ke nilai sebelum transaksi dijalankan. *Undo* dilakukan ketika log menyimpan operasi *start*, tetapi tidak ada catatan operasi *commit*.
- ***Redo .*** Nilai semua data yang di- *update* oleh transaksi tersebut diubah menjadi *new value* (nilai yang akan dimiliki *data item* setelah dilakukan operasi *write*). *Redo* dilakukan ketika didalam *log* tersimpan operasi *start* dan juga *commit*.

21.5. Serialisasi

Ketika kegagalan sistem terjadi, kita harus melihat ke *log* terlebih dahulu untuk memutuskan transaksi mana yang harus dilakukan *redo* atau *undo*. Oleh karena itu kita harus mencari ke seluruh *log* sebelum dapat memutuskan untuk melakukan *redo* atau *undo*. Hal ini tentunya mempunyai kekurangan:

1. Proses pencarian akan memakan waktu yang cukup lama.

2. Seandainya transaksi tersebut harus dilakukan *redo*, berarti data tersebut harus dimodifikasi dengan nilai yang sebenarnya telah di- *update*. Meskipun hal tersebut tidak berdampak buruk, tetapi proses pemulihan akan memakan waktu yang lebih lama.

Untuk mengatasi kekurangan tersebut, kita dapat menggunakan sebuah konsep *checkpoint*. Selama transaksi dijalankan, sistem membuat *write-ahead log* dan secara periodik menjalankan *checkpoint* yang dilakukan pada saat:
1. Seluruh catatan dalam *log* yang sedang berada di *volatile storage* dipindahkan ke *stable storage*.
2. Seluruh data yang dimodifikasi yang berada di *volatile storage* dipindahkan ke *stable storage*.
3. *Log* yang menyimpan operasi *checkpoint* dipindahkan ke *stable storage*.

Ketika kegagalan terjadi *routine* pemulihan memeriksa *log* untuk memutuskan transaksi mana yang terakhir kali melakukan operasi *write* dan di mana operasi *checkpoint* terjadi. Dengan menggunakan pencarian mundur dan berhasil menemukan *checkpoint* dan menemukan catatan operasi *start*, berarti kita telah menemukan bagian dari transaksi yang akan kita periksa untuk selanjutnya dilakukan *redo* atau *undo*. Dengan kata lain, kita tidak harus memeriksa keseluruhan *log* pada transaksi tersebut. Proses pemulihan dapat dilakukan dengan kondisi:
1. Jika pada bagian transaksi tersebut ditemukan operasi *commit*, maka dilakukan *redo*.
2. Jika pada bagian transaksi tersebut tidak ditemukan catatan mengenai operasi *commit*, maka dilakukan *undo*.

Sebelumnya telah dijelaskan mengenai kasus dimana hanya ada satu buah transaksi yang dapat dieksekusi pada suatu waktu. Sekarang, kita beralih pada kasus dimana ada beberapa transaksi yang harus dieksekusi secara bersamaan. Oleh karena setiap transaksi yang dilakukan bersifat atomik, maka hasil dari eksekusi akhir harus sama dengan hasil eksekusi bila transaksi-transaksi tersebut dijalankan secara berurutan. Meskipun, cara demikian akan memastikan keatomikan dari setiap transaksi, tetapi cara demikian sangat tidak efisien karena adanya pembatasan transaksi-transaksi ketika suatu transaksi dilaksanakan.

Penjadwalan *(schedule)* merupakan urutan pengeksekusian transaksi-transaksi. Misalkan ada dua buah transaksi T0 dan T1, dimana kedua transaksi ini dieksekusi secara atomik dengan urutan T0 diikuti dengan T1. Sebuah penjadwalan dimana setiap transaksi dieksekusi secara atomik sesuai dengan urutan yang ada disebut penjadwalan serial. Dengan demikian, bila ada n transaksi akan ada n! penjadwalan yang valid.

Tabel 21.1. Contoh Penjadwalan Serial: Penjadwalan T0 diikuti T1

T0	T1
Read *(A)*	
Write *(A)*	
Read *(B)*	
Write *(B)*	
	Read *(A)*
	Write *(A)*
	Read *(B)*
	Write *(B)*

Pada kasus dimana terjadi *overlapping* (ada transaksi yang dijalankan ketika transaksi lain sedang berjalan) dalam pengeksekusian transaksi-transaksi yang ada, maka penjadwalan tersebut disebut penjadwalan non-serial. Penjadwalan demikian tidak selalu menghasilkan hasil eksekusi yang salah (bisa benar bila hasilnya sama dengan hasil penjadwalan serial). Penjadwalan non-serial yang menghasilkan eksekusi yang benar disebut *conflict serializable*. Untuk memeriksa sifat *serializable* dari sebuah penjadwalan, harus diperiksa apakah terdapat konflik antara dua operasi pada transaksi yang berbeda. Konflik terjadi bila:
- Ada dua operasi Oi dan Oj pada penjadwalan S dimana keduanya mengakses data yang sama, dan
- Setidaknya ada satu operasi yang melakukan `write()`

Tabel 21.2. Contoh Penjadwalan Non-Serial (*Concurrent Serializable Schedule*)

T0	T1
Read *(A)*	
Write *(A)*	
	Read *(A)*
	Write *(A)*
Read *(B)*	
Write *(B)*	
	Read *(B)*
	Write *(B)*

Pada contoh tersebut, `write(A)` pada T0 mengalami konflik dengan `read(A)` pada T1 karena keduanya mengakses data yang sama (A) dan terdapat operasi `write(A)`. Namun, `write(A)` pada T1 tidak mengalami konflik dengan `read(B)` pada T0, karena walaupun ada operasi `write(A)` tetapi keduanya mengakses data yang berbeda. Dalam kasus dimana tidak terjadi konflik antar dua operasi, maka dapat dilakukan *swapping* sehingga terbentuk penjadwalan baru S' yang urutannya sama dengan penjadwalan serial. Pada contoh tersebut *swapping* yang dapat dilakukan adalah:
- *Swap* `write(A)` pada T1 dengan `read(B)` pada T0
- *Swap* `read(B)` pada T0 dengan `read(A)` pada T1
- *Swap* `write(B)` pada T0 dengan `write(A)` pada T1
- *Swap* `write(B)` pada T0 dengan `read(A)` pada T1

Maka, akan didapat penjadwalan baru S' yang merupakan penjadwalan serial. Penjadwalan demikian dinamakan *conflict serializable*, sedangkan proses penyusunan penjadwalan baru yang serial disebut serialisasi.

21.6. Protokol Penguncian

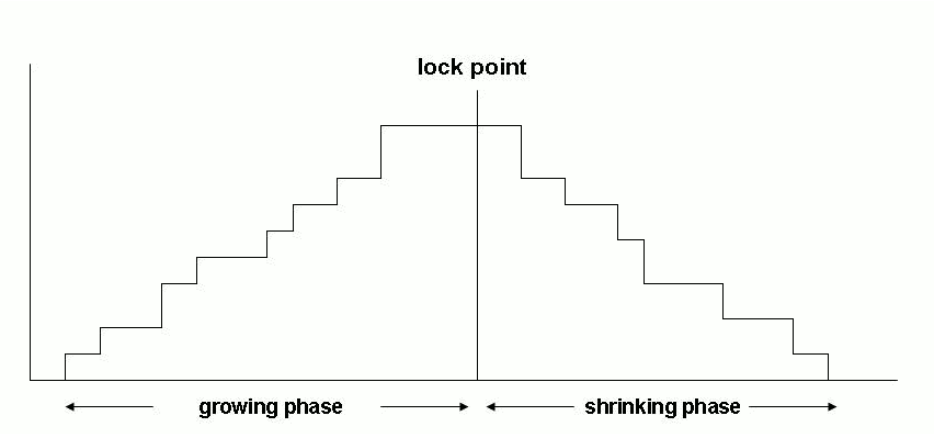
Salah satu cara untuk menjamin terjadi serialisasi adalah dengan menerapkan protokol penguncian *(locking protocol)* pada tiap data yang akan diakses. Ada dua macam cara untuk melakukan penguncian pada data:
1. ***Shared .*** Jika sebuah transaksi Ti melakukan *shared-mode lock* pada data Q, maka transaksi tersebut dapat melakukan operasi *read* pada Q tetapi tidak dapat melakukan operasi *write* pada Q.
2. ***Exclusive .*** Jika sebuah transaksi Ti melakukan *exclusive-mode lock* pada data Q, maka transaksi tersebut dapat melakukan *read* dan *write* pada Q.

Setiap transaksi harus melakukan penguncian pada data yang akan diakses, bergantung pada kebutuhan operasi yang akan dilakukan. Proses pengaksesan data Q sebuah transaksi Ti adalah sebagai berikut:
1. Menentukan mode penguncian yang akan dipergunakan
2. Memeriksa apakah data Q sedang dikunci oleh transaksi lain. Jika tidak, Ti dapat langsung mengakses Q, jika ya, Ti harus menunggu *(wait)*.
3. Bila mode penguncian yang diinginkan adalah *exclusive-lock mode*, maka Ti harus menunggu sampai data Q dibebaskan.
4. Bila mode penguncian yang diinginkan adalah *shared-lock mode*, maka Ti harus menunggu sampai data Q tidak berada dalam *exclusive-mode lock* oleh data lain. Dalam hal ini data Q bisa diakses bila sedang dalam keadaan bebas atau *shared-mode lock*.

Sebuah transaksi dapat melakukan pembebasan *(unlock)* pada suatu data yang telah dikunci sebelumnya setelah data tersebut selesai diakses. Namun, proses pembebasan data tidak langsung dilakukan sesegera mungkin karena sifat *serializable* bisa tidak terjaga. Oleh karena itu, ada pengembangan lebih lanjut dari protokol penguncian yang disebut *two-phase locking protocol*. Protokol ini terdiri dari dua fase, yaitu:

1. ***Growing phase .*** Fase dimana sebuah transaksi hanya boleh melakukan penguncian pada data. Pada fase ini, transaksi tidak boleh melakukan pembebasan pada data lain.
2. ***Shrinking phase .*** Fase dimana sebuah transaksi melakukan pembebasan pada data. Pada fase ini, transaksi tidak boleh melakukan penguncian pada data lain.

Gambar 21.1. *Two-Phase Locking Protocol*



Pada banyak kasus, *two-phase locking protocol* banyak dipergunakan untuk menjaga sifat *serializable* dari suatu penjadwalan, namun protokol ini belum dapat menjamin bahwa tidak akan terjadi *deadlock* karena masih ada transaksi yang berada dalam status menunggu *(`wait()`)*.

21.7. Protokol Berbasis Waktu

Pada protokol penguncian, transaksi-transaksi yang mengalami konflik ditangani dengan mengatur transaksi yang lebih dulu melakukan penguncian dan juga mode penguncian yang digunakan. Protokol berbasis waktu merupakan cara lain untuk melakukan pengaturan transaksi agar sifat *serializable* penjadwalan tetap terjaga.

Pada protokol ini, setiap transaksi diberikan sebuah *timestamp* yang unik yang diberi nama TS(Ti). *Timestamp* ini diberikan sebelum transaksi Ti melakukan eksekusi. Bila Ti telah diberikan *timestamp*, maka bila ada transaksi lain Tj yang kemudian datang dan diberikan *timestamp* yang unik pula, akan berlaku TS(Ti) < TS(Tj). Ada dua metode yang digunakan untuk melakukan protokol ini:
1. Gunakan waktu pada *clock system* sebagai *timestamp*. Jadi, *timestamp* sebuah transaksi sama dengan *clock system* ketika transaksi itu memasuki sistem.
2. Gunakan sebuah *counter* sebagai sebuah *timestamp*. Jadi, *timestamp* sebuah transaksi sama dengan nilai *counter* ketika transaksi mulai memasuki sistem.

Timestamp dari transaksi-transaksi akan membuat sifat *serializable* tetap terjaga. Bila TS(Ti) < TS(Tj), maka Ti akan dilakukan terlebih dahulu, baru kemudian Tj dilakukan. Setiap data item yang akan diakses akan memiliki dua nilai *timestamp*:
a. `W-timestamp(Q)`, berisi *timestamp* terbesar yang berhasil mengeksekusi perintah `write()`.
b. `R-timestamp(Q)`, berisi *timestamp* terbesar yang berhasil mengeksekusi perintah `read()`.

Timestamp yang ada akan terus diperbaharui kapan saja instruksi `read(Q)` atau `write(Q)` dieksekusi. Berdasarkan protokol berbasis waktu, semua konflik yang ada antara *read* dan *write* akan dieksekusi berdasarkan urutan *timestamp* tiap instruksi.

Protokol pembacaan transaksi:
- Jika TS(Ti) < `W-timestamp(Q)`, maka Ti perlu membaca data yang sekarang sudah ditulis dengan data lain. Maka, transaksi ini akan ditolak.
- Jika TS(Ti) >= `W-timestamp(Q)`, maka Ti akan membaca data dan `R-timestamp(Q)` akan di-set menjadi maksimum, yaitu sesuai *timestamp* Ti.

Protokol penulisan transaksi:
- Jika TS(Ti) lebih kecil dari `R-timestamp(Q)`, maka data Q yang akan ditulis oleh Ti diperlukan sebelumnya, sehingga dianggap bahwa Ti tidak perlu melakukan operasi `write()`(waktu eksekusi transaksi sudah lewat) dan transaksi ini ditolak.
- Jika TS(Ti) lebih kecil `W-timestamp(Q)`, maka transaksi Ti melakukan operasi `write()` yang hasilnya tidak diperlukan lagi (waktu Ti melakukan eksekusi sudah lewat) sehingga transaksi ini ditolak.
- Selain dua point di atas, maka transaksi akan dilakukan

Tabel 21.3. Contoh Penjadwalan dengan PROTOKOL BERBASIS WAKTU

T2	T3
Read *(B)*	
	Read *(B)*
	Write *(B)*
Read *(A)*	
	Read *(A)*
	Write *(A)*
	Read *(B)*
	Write *(B)*

Protokol berbasis waktu juga membantu dalam mengatasi *deadlock*, karena tidak ada transaksi-transaksi yang melakukan `wait()`.

21.8. Rangkuman

Transaksi merupakan sekumpulan instruksi atau operasi yang menjalankan sebuah fungsi logis dan memiliki sifat *atomicity*, *consistency*, *isolation*, dan *durability*. Sifat *atomicity* pada transaksi menyebabkan transaksi tersebut akan dijalankan secara keseluruhan atau tidak sama sekali. Operasi-operasi pada transaksi atomik disimpan dalam *log* agar dapat dilakukan *rolled-back* jika terjadi kegagalan sistem. Dengan memanfaatkan *log*, pemulihan data dapat dilakukan dengan melakukan *undo* atau *redo*. Untuk menghemat waktu pada saat *rolled-back*, kita dapat memberikan operasi *checkpoint* pada transaksi sehingga kita tidak perlu memeriksa keseluruhan transaksi untuk memutuskan melakukan *undo/redo*.

Serialisasi diperlukan ketika beberapa transaksi atomik dijalankan secara bersamaan. Hal ini dimaksudkan agar sifat konsistensi hasil eksekusi transaksi dapat terpenuhi. Ada dua cara untuk menjaga agar penjadwalan bersifat *serializable*, yaitu protokol penguncian dan protokol berbasis waktu. Pada protokol penguncian, setiap data yang akan diakses harus dikunci oleh transaksi yang akan memakainya agar transaksi lain tidak bisa mengakses data yang sama. Sedangkan, pada protokol berbasis waktu, setiap transaksi diberikan suatu *timestamp* yang unik, sehingga dapat diketahui apakah transaksi tersebut sudah dijalankan atau belum. Protokol berbasis waktu dapat mengatasi masalah *deadlock*, sedangkan protokol penguncian tidak.

Rujukan

[Bacon2003] Jean Bacon dan Tim Harris. 2003 . *Operating Systems : Concurrent And Distributed Software Design*. First Edition. Addison Wesley.

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

[Tanenbaum1992] Andrew S. Tanenbaum. 1992 . *Modern Operating Systems*. First Edition. Prentice-Hall.

[WEBWIKI2007] Wikipedia. 2007 . *Serializability – http://en.wikipedia.org/wiki/Serializability*. Diakses 06 Maret 2007.

Bab 22. Sinkronisasi Linux

22.1. Pendahuluan

Pada suatu saat dalam sebuah kernel, tidak terkecuali kernel Linux, dapat terjadi *concurrent access*. Yang dimaksud dengan *concurrent access* adalah beberapa *thread* yang sedang berjalan mengakses *resources* yang sama dalam waktu yang sama. Jika hal ini terjadi, *thread-thread* tersebut akan saling meng- *overwrite* perubahan yang dilakukan *thread* sesamanya sebelum perubahan tersebut mencapai *state* yang konsisten. Sehingga hasil dari proses tidak seperti yang diharapkan. Dalam hal ini diperlukan proteksi dalam kernel yang bersangkutan. Proteksi dapat dilakukan dengan sinkronisasi.

Proteksi *resources* dari *concurrent access* bukanlah merupakan hal yang mudah. Beberapa tahun yang lalu sebelum Linux mendukung adanya *symmetrical multiprocessing*, proteksi masih mudah dilakukan. Karena hanya ada satu processor yang didukung, satu-satunya cara bagi *resources* dapat diakses secara *concurrent* (bersama-sama) oleh *thread* adalah ketika ada *interrupt* atau memang ada jadwal dari kernel bahwa *thread* lain diperbolehkan untuk mengakses *resources* tersebut.

Namun dengan berkembangnya zaman, Linux akhirnya mendukung adanya *symmetrical multiprocessing* dalam versi 2.0 kernelnya. *Multiprocessing* artinya *kernel code* dapat dijalankan dalam dua atau lebih processor. Jika tanpa proteksi, *code* yang dijalankan dalam dua processor yang berbeda dapat mengakses *resources* yang sama dalam waktu yang sama. Dengan adanya Linux 2.6 kernel yang mendukung adanya konsep *preemptive, scheduler* dalam kernel dapat meng *interrupt kernel code* yang sedang berjalan untuk memberi kesempatan bagi *kernel code* lain untuk dijalankan. Dengan demikian pengaksesan *resources* yang sama dalam waktu yang sama dapat dihindari.

Dalam bab ini akan dijelaskan bagaimana implementasi sinkronisasi dalam Linux. Metode-metode sinkronisasi yang dibahas meliputi integer atomik yang merupakan salah satu jenis dari operasi atomik, *locking* yang terdiri dari *spin lock* dan semafor, dan dijelaskan hal-hal lain yang terkait dengan pembahasan ini.

22.2. *Critical Section*

Sebuah proses memiliki bagian dimana bagian ini akan melakukan akses dan manipulasi data. Bagian ini disebut dengan *critical section*. Ketika sebuah proses sedang dijalankan dalam *critical section* nya, tidak ada proses lain yang boleh dijalankan dalam *critical section* nya. Karena hal ini dapat memungkinkan terjadinya akses ke *resources* yang sama dalam waktu yang sama. Keadaan seperti ini disebut proses tersebut *mutually exclusive*. Oleh karena itu, diperlukan suatu mekanisme atau aturan agar proses sifat *mutually exclusive* dapat terpenuhi.

Dengan mengontrol variabel mana yang diubah baik didalam maupun diluar *critical section*, *concurrent access* dapat dicegah. *Critical section* biasanya digunakan saat program *multithreading*, dimana program tersebut terdiri dari banyak thread, akan mengubah nilai dari variabel. Dalam hal ini *critical section* diperlukan untuk melindungi variabel dari *concurrent access* yang dapat membuat nilai dari variabel tersebut menjadi tidak konsisten.

Lalu bagaimana *critical section* tersebut diimplementasikan didalam sistem operasi. Metode yang paling sederhana adalah dengan mencegah adanya *thread* lain yang mengubah variabel yang sedang digunakan dalam *critical section*. Selain itu, *system call* yang dapat menyebabkan *context switch* juga dihindari. Jika *scheduler* meng- *interrupt* proses yang sedang mengakses *critical section* nya, maka *scheduler* akan membiarkan proses tersebut menyelesaikan *critical section* nya atau menghentikannya sementara untuk memberi kesempatan bagi proses lain untuk menjalankan *critical section* nya. Proses yang sedang berada dalam *critical section* nya dijalankan secara *mutually exclusive*.

Contoh 22.1. *Critical section*

```
do{

   critical section

}while(1)
```

22.3. Penyebab Konkurensi Kernel

Perlunya sinkronisasi disebabkan karena adanya program yang dijadwalkan secara *preemptive*. Karena proses dapat di *interrupt* maka proses yang lain dapat masuk ke processor menggantikannya untuk dijalankan. Hal ini memungkinkan proses awal di *interrupt* di *critical region* nya. Jika ada proses baru yang dijadwalkan untuk dijalankan selanjutnya masuk ke *critical region* yang sama, maka *race condition* akan terjadi. Ada dua jenis *concurrency* yaitu *pseudo-concurrency* dan *true-concurrency*. *Pseudo-concurrency* terjadi ketika dua proses tidak benar-benar berjalan dalam waktu yang tepat sama namun ada waktu jeda diantara keduanya. Sedangkan *true-concurrency* terjadi ketika dua proses berjalan secara bersama-sama yang waktu yang sama. *True-concurrency* biasanya terjadi pada komputer yang berbasis *symmetrical multiprocessing*.

Ada beberapa penyebab konkurensi kernel, diantaranya:

- ***Interrupt .*** *Interrupt* dapat terjadi sewaktu-waktu, menghentikan sementara proses yang sedang berjalan.
- ***Softirqs dan Tasklets .*** Kernel dapat menjadwalkan *Softirqs dan Tasklets* sewaktu-waktu untuk menghentikan sementara proses yang sedang berjalan. *Softirqs dan Tasklets* adalah pengatur *interrupt* untuk *interrupt- interrupt* yang tidak dapat dilakukan *interrupt-handler* yang biasa. *Softirqs* dapat berjalan secara bersama-sama dalam beberapa processor bahkan dua *Softirqs* yang sama dapat berjalan bersama-sama. Sifat *tasklets* hampir sama dengan *Softirqs*, hanya saja dua *tasklets* yang sama tidak dapat berjalan secara bersama-sama.
- ***Kernel Preemption .*** Karena kernel bersifat *preemptive*, sebuah proses yang sedang berjalan dapat dihentikan sementara oleh proses yang lain.
- ***Sleeping dan Syncronization with user-space .*** Sebuah proses dalam kernel dapat *sleep* dan meminta kernel untuk menjalankan proses yang lain.
- ***Symmetrical Multiprocessing .*** Dua atau lebih processor dapat menjalankan *kernel code* dalam waktu yang sama.

22.4. Integer Atomik

Salah satu metode dalam kernel LINUX untuk sinkronisasi adalah *atomic operations*. Integer atomik adalah salah satu jenis dari *atomic operations*. *Atomic operations* menyediakan instruksi yang dijalankan secara atomik (tanpa interrupt). Contohnya sebuah *atomic increment* dapat membaca dan meng- *increment* sebuah variabel dalam sebuah step yang tidak dapat dibagi-bagi. Misalnya i diinisialisasi sama dengan 7

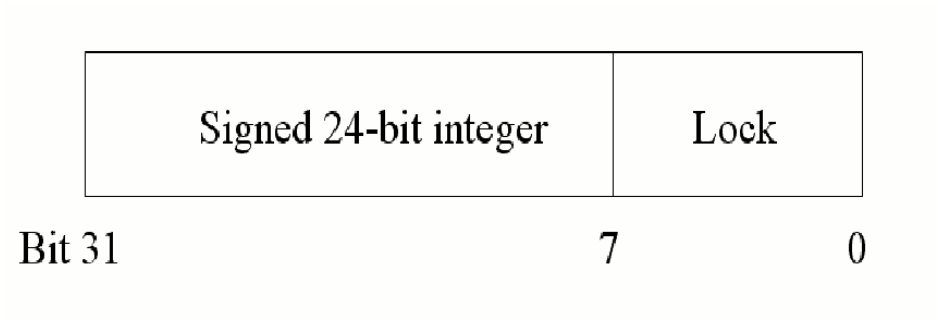
Gambar 22.1. *Atomic Operation*

```
Thread 1               Thread 2
-------------------------------------------------------------
Atomic increment i (7 -> 8)         -
    -                        Atomic increment i (8->9)
```

Atomic increment pada *thread* 1 akan dijalankan sampai selesai tanpa *interrupt* sehingga hasil dari *thread* pertama adalah 8. Setelah itu barulah *thread* 2 dijalankan. Mula-mula nilai i yang dibaca adalah 8 kemudian di i *increment* sehingga hasil akhirnya 9.

Methods integer atomik hanya berjalan untuk tipe data `atomic_t`. Fungsi atomik akan menggunakan tipe data ini dan tidak akan mengirim tipe data ini ke fungsi nonatomik. Dalam penggunaannya, *compiler* tidak mengakses langsung nilai dari data ini namun yang diakses adalah alamat memorinya. Integer atomik (atomic_t) memiliki 32 bits dalam representasinya. Namun yang dipakai untuk menyimpan suatu nilai hanya 24 bits karena 8 bits sisanya dipakai untuk lock yang berfungsi untuk memproteksi *concurrent access* ke data atomic yang bersangkutan. Hal ini diimplementasikan dalam SPARC port pada LINUX

Gambar 22.2. 32-bit atomic_t



Untuk menggunakan atomic integer operations perlu dideklarasikan terlebih dahulu `<asm/atomic.h>`. Berikut ini contoh penggunaan atomic integer :

```
atomic_t v;
atomic_t v = ATOMIC_INIT(0);

atomic_set(&v,4);
atomic_add(2,&v);
atomic_inc(&v);

printk("%d\n",atomic_read(&v));
```

Fungsi `atomic_t v` akan mendefinisikan variabel v sebagai sebuah integer atomik. Kemudian variabel v akan diinisialisasi menjadi 0 oleh fungsi `ATOMIC_INIT(0)`. Untuk mengeset nilai v misalnya menjadi 4 secara atomik dapat digunakan fungsi `atomic_set(&v,4)`. Menambah nilai v secara atomik dengan menggunakan `fungsi atomic_add(2,&v)`. Variabel v dapat juga dikenakan atomic increment dengan menggunakan fungsi `atomic_inc(&v)`. Fungsi `printk("%d\n",atomic_read(&v))` akan mencetak nilai v.

Berikut ini beberapa Atomic Integer Operations :

Tabel 22.1. Tabel Atomic Integer Operations

Atomic Integer Operation	Description
`ATOMIC_INIT(int i)`	inisialisasi atomic_t menjadi i
`atomic_read(atomic_t *v)`	membaca nilai integer dari v
`void atomic_set(atomic_t *v,int i)`	set nilai v menjadi i

Atomic Integer Operation	Description
`void atomic_add(int i,atomic_t *v)`	menambah v sebesar i
`void atomic_sub(int i,atomic_t *v)`	mengurangi c sebesar i
`void atomic_inc(atomic_t *v)`	menambah v dengan 1
`void atomic_dec(atomic_t *v)`	mengurangi v dengan 1

22.5. *Spin Locks*

Metode *Locking* adalah salah satu metode dalam sinkronisasi yang diperlukan untuk menjaga agar tidak ada proses yang berjalan bersama-sama yang memungkinkan adanya *concurrent access* ke resources yang sama. *Locking* akan menutup proses yang sedang berjalan dalam *critical region* nya dari proses yang lain sehingga hanya akan ada satu proses yang sedang berjalan dalam sebuah processor. *Locking* yang paling umum digunakan dalam LINUX adalah *spin lock*. *Spin lock* adalah *lock* yang hanya dapat dilakukan oleh satu *thread*. Ketika sebuah *thread* yang akan dijalankan meminta *spin lock* yang sedang digunakan, maka *thread* ini akan *loops* menunggu sampai *spin lock* tersebut selesai digunakan oleh *thread* yang sedang berjalan. Jika *spin lock* sedang tidak digunakan maka *thread* yang akan dijalankan akan langsung mendapatkannya. Hal ini akan mencegah adanya dua *thread* yang dijalankan bersama-sama. Dalam *spin lock* tidak diperbolehkan kernel *preemption*. Berikut ini contoh penggunaan *spin lock*:

Sebelum menggunakan spin lock, perlu dideklarasikan `<asm\spinlock.h>`.

```
spinlock-t mr_lock = SPIN_LOCK_UNLOCKED;

unsigned long flags;

spin_lock_irqsave(&mr_lock,flags);

/ * critical region * /

spin_unlock_irqrestore(&mr_lock,flags);
```

Fungsi `SPIN_LOCK_UNLOCKED` menyatakan bahwa variabel yang bersangkutan, dalam hal ini `mr_lock`, sedang tidak menjalankan *spin lock*. Fungsi `spin_lock_irqsave(&mr_lock,flags)` menyebabkan `mr_lock` mendapatkan *spin lock* dan menyimpan *state* terakhir dari *thread* yang di *interrupt*. Setelah itu, `mr_lock` akan masuk ke *critical region* nya. Setelah selesai maka `mr_lock` melepaskan *spin lock* menggunakan `spin_unlock_irqrestore(&mr_lock,flags)` dan *thread* yang di *interrupt* kembali ke *state* terakhirnya.

Berikut ini beberapa spin lock methods

Tabel 22.2. Tabel Spin Lock Methods

Method	Description
`spin_lock()`	Mendapatkan lock
`spin_lock_irq()`	Menginterrupt dan mendapatkan lock
`spin_lock_irqsave()`	Menyimpan current state dari local interrupt dan mendapatkan lock

Method	Description
`spin_unlock()`	Melepaskan lock
`spin_unlock_irq()`	Melepaskan lock dan enable local interrupt
`spin_unlock_irqrestore()`	Melepaskan lock dan memberi local interrupt previous statenya

22.6. Semafor

Semafor dalam LINUX adalah *sleeping locks*. Ketika sebuah *thread* meminta semafor yang sedang digunakan, maka semafor akan meletakkan *thread* tersebut dalam *wait queue* dan menyebabkan *thread* tersebut masuk status *sleep*. Kemudian processor menjalankan *thread* yang lain. Ketika *thread* yang memegang semafor melepaskan *lock* nya, maka satu dari *thread* yang ada di *wait queue* akan dipanggil sehingga akan mendapatkan semafor.

Beberapa konklusi yang dapat diambil berkaitan dengan *sleeping* yang terjadi dalam semafor diantaranya :

- Karena *thread* yang sedang menunggu giliran untuk dijalankan masuk dalam status *sleep*, maka semafor cocok untuk *lock* yang digunakan untuk waktu yang cukup lama.
- Sebaliknya, semafor tidak cocok untuk lock yang digunakan dalam waktu yang singkat karena waktu yang digunakan untuk *sleeping*, menjaga *wait queue*, dan membangunkan *thread* yang sedang *sleep* akan menambah waktu *lock*.
- *Thread* yang sedang memegang semafor dapat *sleep* dan tidak akan *deadlock* ketika ada *thread* yang lain yang mendapatkan semafor yang sama karena *thread* tersebut akan *sleep* dan membiarkan *thread* yang pertama untuk melanjutkan eksekusinya.
- Suatu *thread* tidak dapat memegang *spin lock* ketika menunggu untuk mendapatkan semafor karena *thread* tersebut harus *sleep* dan tidak dapat *sleep* dengan memegang *spin lock*.

Berbeda dengan *spin lock*, semafor memperbolehkan adanya kernel *preemption*. Dalam semafor juga diperbolehkan pemegang semafor lebih dari satu dalam suatu waktu yang sama. Banyaknya pemegang *lock* di sebut *usage count*. Nilai yang paling umum untuk *usage count* adalah satu yang atinya hanya ada satu *thread* yang berjalan dalam suatu waktu. Dalam hal ini jika *usage count* sama dengan 1 maka disebut *binary semaphore* atau *mutex*. Implementasi dari semafor didefinisikan dalam <asm \semaphore.h>. Semafor dapat dibuat dengan cara sebagai berikut :

```
static DECLARE_SEMAPHORE_GENERIC(name,count)
```

dimana `name` menyatakan nama variabel dan `count` adalah *usage count* dari semafor.

```
static DECLARE_MUTEX(name)
```

dimana `name` adalah nama variabel dari semafor.

Berikut ini contoh implementasi semaphore :

```
static DECLARE_MUTEX(mr_sem);

if(down_interruptible(&mr_sem)){

   /*critical region*/
```

```
up(&mr_sem);
}
```

Fungsi `down_interruptible()` untuk meminta semafor. Jika gagal, maka variabel yang bersangkutan akan *sleep* dalam state TASK_INTERRUPTIBLE. Jika kemudian variabel menerima signal maka `down_interruptible()` akan mereturn -EINTR. Sehingga fungsi `down()` akan membuat variabel masuk dalam state TASK_UNINTERRUPTIBLE dan akan dijalankan dalam *critical region* nya. Untuk melepaskan semafor, digunakan fungsi `up()`. Mengetahui apakah akan menggunakan *spin lock* atau semafor adalah cara yang baik untuk mengoptimalkan code yang dibuat Berikut ini tabel mengenai apa yang diperlukan untuk menentukan lock mana yang digunakan

Tabel 22.3. Tabel Spin Lock Versus Semaphore

Requirement	Recommended
Overhead locking yang rendah	Spin lock
Lock hold time yang singkat	Spin lock
Lock hold time yang panjang	Semaphore
Sleep ketika menunggu lock	Semaphore

22.7. SMP

Symmetrical Multiprocessing (SMP) mendukung adanya pengeksekusian secara paralel dua atau lebih *thread* oleh dua atau lebih processor. Kernel LINUX 2.0 adalah kernel LINUX pertama yang memperkenalkan konsep SMP. Untuk menjaga agar dua *thread* tidak mengakses *resources* yang sama dalam waktu yang sama, maka SMP menerapkan aturan dimana hanya ada satu processor yang dapat menjalankan *thread* dalam kernel mode dengan cara *spin lock* tunggal untuk menjalankan aturan ini. *Spin lock* ini tidak memunculkan permasalahan untuk proses yang banyak menghabiskan waktu untuk menunggu proses komputasi, tapi untuk proses yang banyak melibatkan banyak aktifitas kernel, *spin lock* 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 *spin lock* tunggal menjadi banyak *lock* 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 kernel mode secara bersamaan.

22.8. Rangkuman

Pada suatu saat dalam sebuah kernel, tidak terkecuali kernel LINUX, dapat terjadi *concurrent access*. Dalam hal ini diperlukan proteksi dalam kernel yang bersangkutan. Proteksi dapat dilakukan dengan sinkronisasi.

Sebuah proses memiliki bagian dimana bagian ini akan melakukan akses dan manipulasi data. Bagian ini disebut dengan *critical section*. Ketika sebuah proses sedang dijalankan dalam *critical section* nya, tidak ada proses lain yang boleh dijalankan dalam *critical section* nya.

Ada dua jenis *concurrency* yaitu *pseudo-concurrency* dan *true-concurrency*. Ada beberapa penyebab konkurensi kernel, diantaranya *interrupt*, *softirqs* dan *tasklets*, kernel *preemption*, *sleeping* dan *synchronization with user-space*, dan *symmetrical multiprocessing*.

Salah satu metode dalam kernel LINUX untuk sinkronisasi adalah *atomic operations*. Integer atomik adalah salah satu jenis dari *atomic operations*. Integer Atomik menyediakan instruksi yang dijalankan secara atomik (tanpa interrupt).

Locking yang paling umum digunakan dalam LINUX adalah *spin lock*. *Spin lock* adalah *lock* yang hanya dapat dilakukan oleh satu *thread*. Ketika sebuah *thread* yang akan dijalankan meminta *spin*

lock yang sedang digunakan, maka *thread* ini akan *loops* menunggu sampai *spin lock* tersebut selesai digunakan oleh *thread* yang sedang berjalan.

Semafor dalam LINUX adalah *sleeping locks*. Ketika sebuah *thread* meminta semafor yang sedang digunakan, maka semafor akan meletakkan *thread* tersebut dalam *wait queue* dan menyebabkan *thread* tersebut masuk status *sleep*.

Symmetrical multiprocessing (SMP) mendukung adanya pengeksekusian secara paralel dua atau lebih *thread* oleh dua atau lebih processor. Kernel LINUX 2.0 adalah kernel LINUX pertama yang memperkenalkan konsep SMP.

Rujukan

[RobertLove2005] Robert Love. 2005 . *Linux Kernel Development*. Novell Press.

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

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

[WEBDrake96] Donald G Drake. April 1996. *Introduction to Java threads – A quick tutorial on how to implement threads in Java – http://www.javaworld.com/ javaworld/ jw-04-1996/ jw-04-threads.html*. Diakses 29 Mei 2006.

[WEBFasilkom2003] Fakultas Ilmu Komputer Universitas Indonesia. 2003 . *Sistem Terdistribusi – http:// telaga.cs.ui.ac.id/ WebKuliah/ sisdis2003/*. Diakses 29 Mei 2006.

[WEBHarris2003] Kenneth Harris. 2003 . *Cooperation: Interprocess Communication – Concurrent Processing – http://people.cs.uchicago.edu/ ~mark/ 51081/ LabFAQ/ lab5/ IPC.html*. Diakses 2 Juni 2006.

[WEBWalton1996] Sean Walton. 1996 . *Linux Threads Frequently Asked Questions (FAQ) – http://linas.org/ linux/ threads-faq.html*. Diakses 29 Mei 2006.

[WEBWiki2006a] From Wikipedia, the free encyclopedia. 2006 . *Title – http://en.wikipedia.org/wiki/ Zombie_process*. Diakses 2 Juni 2006.

[WEBLINUX2002] Robert Love. 2002 . *Kernel Locking Techniques http://www.linuxjournal.com/article/5833/*. Diakses 3 Maret 2007.

[WEBwikipedia2007a] From Wikipedia, the free encyclopedia. 2007.. 2007 . *Symmetric Multiprocessing http:// en.wikipedia.org/wiki/Symmetric_multiprocessing/*. Diakses 3 Maret 2007.

[WEBwikipedia2007] From Wikipedia, the free encyclopedia. 2007.. 2007 . *Critical Section http:// en.wikipedia.org/wiki/Critical_section.htm/*. Diakses 10 April 2007.

Bab 23. *Deadlocks*

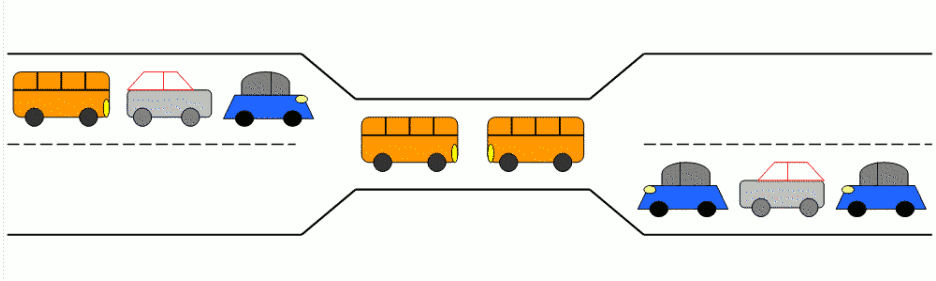
23.1. Pendahuluan

Dalam sistem komputer, terdapat banyak sumber daya yang hanya bisa dimanfaatkan oleh satu proses pada suatu waktu. Contohnya adalah penggunaan sumber daya seperti *printer*, *tape drives* dan *CD-ROM drives*. Dua buah proses yang menggunakan slot yang sama pada tabel proses dapat menyebabkan kerusakan pada sistem. Untuk itu, setiap sistem operasi memiliki mekanisme yang memberikan akses eksklusif pada sumber daya.

Pada kenyataannya, proses membutuhkan akses eksklusif untuk beberapa sumber daya sekaligus. Bayangkan apabila sebuah proses, sebut saja proses A, meminta sumber daya X dan mendapatkannya. Kemudian ada proses B yang meminta sumber daya Y dan mendapatkannya juga. Setelah itu, proses A meminta sumber daya Y dan proses B meminta sumber daya X. Pada situasi tersebut, kedua proses harus ter- *block* dan menunggu secara terus-menerus. Keadaan seperti itu dinamakan *deadlock*.

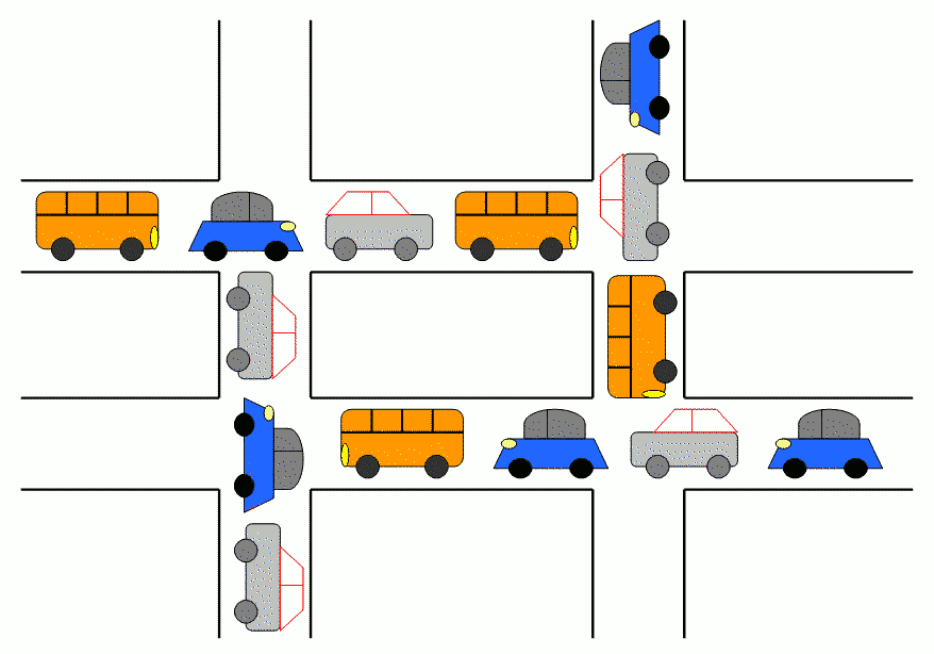
Deadlock secara bahasa berarti buntu atau kebuntuan. Dalam definisi lebih lengkap, *deadlock* berarti suatu keadaan dimana sistem seperti terhenti dikarenakan setiap proses memiliki sumber daya yang tidak bisa dibagi dan menunggu untuk mendapatkan sumber daya yang sedang dimiliki oleh proses lain. Keadaan seperti ini hanya dapat terjadi pada akses terhadap sumber daya yang tidak bisa dibagi atau *non-sharable*.

Gambar 23.1. Contoh kasus *deadlock* pada lalu lintas di jembatan



Pada contoh di atas, digambarkan ilustrasi dari kejadian *deadlock* pada dunia nyata, yaitu pada lalu lintas di jembatan. Dapat dilihat bahwa kedua mobil yang berada di tengah-tengah jembatan tidak dapat maju dan hanya menunggu. Penyelesaian dari masalah tersebut adalah salah satu dari mobil tersebut mundur, sehingga mobil yang lain dapat maju. Mobil pada kasus ini adalah proses, sedangkan jembatan adalah sumber daya. Kedua mobil berebut untuk menggunakan sumber daya, namun karena sumber daya tersebut hanya dapat digunakan oleh satu proses saja, maka terjadilah *deadlock*. Kondisi tersebut bila terjadi dalam waktu yang lama dapat menyebabkan terjadinya *starvation*.

Gambar 23.2. Contoh kasus *deadlock* pada lalu lintas di persimpangan



Gambar di atas adalah contoh lain terjadinya *deadlock* pada dunia nyata. Pada gambar jelas terlihat bahwa lalu lintas terhenti dan terjadi antrian pada empat arah datangnya mobil. Tidak ada mobil yang bisa melanjutkan perjalanan dan hanya menunggu saja. Permasalahan ini dapat dipecahkan dengan cara salah satu dari antrian tersebut mundur dan memberikan kesempatan antrian lain untuk berjalan terlebih dahulu. Kasus seperti ini sangat potensial untuk terjadinya *starvation*. Berikut ini diberikan contoh situasi *deadlock* yang dideskripsikan dengan *pseudocode*.

Contoh 23.1. TestAndSet

```
Mutex M1, M2;

/* Thread 1 */
while (1)
{
  NonCriticalSection()
  Mutex_lock(&M1);
  Mutex_lock(&M2);
  CriticalSection();
  Mutex_unlock(&M2);
  Mutex_unlock(&M1);
}

/* Thread 2 */
while (1)
{
  NonCriticalSection()
  Mutex_lock(&M2);
  Mutex_lock(&M1);
  CriticalSection();
  Mutex_unlock(&M1);
  Mutex_unlock(&M2);
}
```

Misalkan *thread* 1 berjalan dan mengunci M1. Akan tetapi sebelum ia dapat mengunci M2, ia diinterupsi. Kemudian *thread* 2 mulai berjalan dan mengunci M2. Ketika ia mencoba untuk mendapatkan dan mengunci M1, ia terblok karena M1 telah dikunci oleh *thread* 1. Selanjutnya *thread* 1 berjalan lagi dan mencoba untuk mendapatkan dan mengunci M2, namun terblok karena M2 telah dikunci oleh *thread* 2. Kedua *thread* terblok dan saling menunggu terjadinya sesuatu yang tak pernah akan terjadi. Kesimpulannya, terjadi *deadlock* yang melibatkan *thread* 1 dan *thread* 2.

Pada bahasan selanjutnya kita akan membahas tentang *deadlock* secara lebih mendalam, yaitu tentang *starvation*, bagaimana *deadlock* dapat terjadi dan cara untuk menanganinya.

23.2. *Starvation*

Pada bagian pendahuluan, telah sama-sama kita ketahui mengenai pengertian dari *deadlock*. Di contoh lalu lintas jembatan, terlihat bahwa kejadian *deadlock* yang berlangsung secara terus-menerus dan tiada akhir dapat menyebabkan terjadinya *starvation*. Akan tetapi, *deadlock* bukanlah satu-satunya penyebab terjadinya *starvation*. Lalu lintas yang didominasi oleh kendaraan-kendaraan dari satu arah pun dapat menyebabkan terjadinya *starvation*. Akibat yang terjadi adalah kendaraan dari arah lain menjadi terus menunggu giliran untuk berjalan hingga akhirnya mengalami *starvation*.

Starvation adalah keadaan dimana satu atau beberapa proses 'kelaparan' karena terus dan terus menunggu kebutuhan sumber dayanya dipenuhi. Namun, karena sumber daya tersebut tidak tersedia atau dialokasikan untuk proses lain, akhirnya proses yang membutuhkan tidak bisa memilikinya. Kondisi seperti ini merupakan akibat dari keadaan menunggu yang berkepanjangan.

23.3. Model Sistem

Keadaan dimana suatu proses yang meminta sumber daya pasti terjadi dalam suatu sistem. Untuk itu dibutuhkan cara pemodelan terhadapnya. Terdapat tipe sumber daya R_1, R_2, ..., R_m. Contohnya adalah *space* pada memori dan juga komponen-komponen M/K. Setiap tipe sumber daya R_i tersebut memiliki W_i *instances*. Misalnya sebuah sumber daya M/K memiliki dua buah *instances* yang bisa diakses oleh proses.

Sebuah proses dalam melakukan penggunaan terhadap suatu sumber daya melalui langkah-langkah sebagai berikut:

- ***Request*.** Pada langkah ini, pertama kali proses mengajukan diri untuk bisa mendapatkan sumber daya. Proses dapat meminta satu atau lebih sumber daya yang tersedia ataupun yang sedang dimiliki oleh proses yang lain.
- ***Use*.** Selanjutnya, setelah proses mendapatkan sumber daya yang dibutuhkannya, proses akan melakukan eksekusi. Sumber daya digunakan oleh proses sampai proses selesai melakukan eksekusi dan tidak membutuhkan lagi sumber daya tersebut.
- ***Release*.** Setelah memanfaatkan sumber daya untuk melakukan eksekusi, proses pun akan melepaskan sumber daya yang dimilikinya. Sumber daya tersebut dibutuhkan oleh proses lain yang mungkin sedang menunggu untuk menggunakan.

23.4. Karakteristik

Setelah pada bagian sebelumnya kita telah mengetahui mengenai pengertian dari *deadlock* dan bagaimana memodelkannya, sekarang kita akan membahas secara mendalam mengenai karakteristik dari terjadinya *deadlock*. Karakteristik-karakteristik ini harus dipenuhi keempatnya untuk terjadi *deadlock*. Namun, perlu diperhatikan bahwa hubungan kausatif antara empat karakteristik ini dengan terjadinya *deadlock* adalah implikasi. *Deadlock* mungkin terjadi apabila keempat karakteristik terpenuhi. Empat kondisi tersebut adalah:

1. ***Mutual Exclusion*.** Kondisi yang pertama adalah *mutual exclusion* yaitu proses memiliki hak milik pribadi terhadap sumber daya yang sedang digunakannya. Jadi, hanya ada satu proses yang menggunakan suatu sumber daya. Proses lain yang juga ingin menggunakannya harus menunggu

hingga sumber daya tersebut dilepaskan oleh proses yang telah selesai menggunakannya. Suatu proses hanya dapat menggunakan secara langsung sumber daya yang tersedia secara bebas.

2. ***Hold and Wait .*** Kondisi yang kedua adalah *hold and wait* yaitu beberapa proses saling menunggu sambil menahan sumber daya yang dimilikinya. Suatu proses yang memiliki minimal satu buah sumber daya melakukan *request* lagi terhadap sumber daya. Akan tetapi, sumber daya yang dimintanya sedang dimiliki oleh proses yang lain. Pada saat yang sama, kemungkinan adanya proses lain yang juga mengalami hal serupa dengan proses pertama cukup besar terjadi. Akibatnya, proses-proses tersebut hanya bisa saling menunggu sampai sumber daya yang dimintanya dilepaskan. Sambil menunggu, sumber daya yang telah dimilikinya pun tidak akan dilepas. Semua proses itu pada akhirnya saling menunggu dan menahan sumber daya miliknya.

3. ***No Preemption .*** Kondisi yang selanjutnya adalah *no preemption* yaitu sebuah sumber daya hanya dapat dilepaskan oleh proses yang memilikinya secara sukarela setelah ia selesai menggunakannya. Proses yang menginginkan sumber daya tersebut harus menunggu sampai sumber daya tersedia, tanpa bisa merebutnya dari proses yang memilikinya.

4. ***Circular Wait .*** Kondisi yang terakhir adalah *circular wait* yaitu kondisi membentuk siklus yang berisi proses-proses yang saling membutuhkan. Proses pertama membutuhkan sumber daya yang dimiliki proses kedua, proses kedua membutuhkan sumber daya milik proses ketiga, dan seterusnya sampai proses ke n-1 yang membutuhkan sumber daya milik proses ke n. Terakhir, proses ke n membutuhkan sumber daya milik proses yang pertama. Yang terjadi adalah proses-proses tersebut akan selamanya menunggu. *Circular wait* oleh penulis diistilahkan sebagai 'Lingkaran Setan' tanpa ujung.

23.5. Penanganan

Secara umum terdapat 4 cara untuk menangani keadaan *deadlock*, yaitu:

1. **Pengabaian.** Maksud dari pengabaian di sini adalah sistem mengabaikan terjadinya *deadlock* dan pura-pura tidak tahu kalau *deadlock* terjadi. Dalam penanganan dengan cara ini dikenal istilah *ostrich algorithm*. Pelaksanaan algoritma ini adalah sistem tidak mendeteksi adanya *deadlock* dan secara otomatis mematikan proses atau program yang mengalami *deadlock*. Kebanyakan sistem operasi yang ada mengadaptasi cara ini untuk menangani keadaan *deadlock*. Cara penanganan dengan mengabaikan *deadlock* banyak dipilih karena kasus *deadlock* tersebut jarang terjadi dan relatif rumit dan kompleks untuk diselesaikan. Sehingga biasanya hanya diabaikan oleh sistem untuk kemudian diselesaikan masalahnya oleh *user* dengan cara melakukan terminasi dengan *Ctrl +Alt+Del* atau melakukan *restart* terhadap komputer.

2. **Pencegahan.** Penanganan ini dengan cara mencegah terjadinya salah satu karakteristik *deadlock*. Penanganan ini dilaksanakan pada saat *deadlock* belum terjadi pada sistem. Intinya memastikan agar sistem tidak akan pernah berada pada kondisi *deadlock*. Akan dibahas secara lebih mendalam pada bagian selanjutnya.

3. **Penghindaran.** Menghindari keadaan *deadlock*. Bagian yang perlu diperhatikan oleh pembaca adalah bahwa antara pencegahan dan penghindaran adalah dua hal yang berbeda. Pencegahan lebih kepada mencegah salah satu dari empat karakteristik *deadlock* terjadi, sehingga *deadlock* pun tidak terjadi. Sedangkan penghindaran adalah memprediksi apakah tindakan yang diambil sistem, dalam kaitannya dengan permintaan proses akan sumber daya, dapat mengakibatkan terjadi *deadlock*. Akan dibahas secara lebih mendalam pada bagian selanjutnya.

4. **Pendeteksian dan Pemulihan.** Pada sistem yang sedang berada pada kondisi *deadlock*, tindakan yang harus diambil adalah tindakan yang bersifat represif. Tindakan tersebut adalah dengan mendeteksi adanya *deadlock*, kemudian memulihkan kembali sistem. Proses pendeteksian akan menghasilkan informasi apakah sistem sedang *deadlock* atau tidak serta proses mana yang mengalami *deadlock*. Akan dibahas secara lebih mendalam pada bagian selanjutnya.

23.6. Pencegahan

Pencegahan *deadlock* dapat dilakukan dengan cara mencegah salah satu dari empat karakteristik terjadinya *deadlock*. Berikut ini akan dibahas satu per satu cara pencegahan terhadap empat karakteristik tersebut.

1. ***Mutual Exclusion .*** Kondisi *mutual exclusion* pada sumber daya adalah sesuatu yang wajar terjadi, yaitu pada sumber daya yang tidak dapat dibagi (*non-sharable*). Sedangkan pada sumber daya yang bisa dibagi tidak ada istilah *mutual exclusive*. Jadi, pencegahan kondisi yang pertama ini sulit karena memang sifat dasar dari sumber daya yang tidak dapat dibagi.

2. ***Hold and Wait .*** Untuk kondisi yang kedua, sistem perlu memastikan bahwa setiap kali proses meminta sumber daya, ia tidak sedang memiliki sumber daya lain. Atau bisa dengan proses meminta dan mendapatkan sumber daya yang dimilikinya sebelum melakukan eksekusi, sehingga tidak perlu menunggu.

3. ***No Preemption .*** Pencegahan kondisi ini dengan cara membolehkan terjadinya *preemption*. Maksudnya bila ada proses yang sedang memiliki sumber daya dan ingin mendapatkan sumber daya tambahan, namun tidak bisa langsung dialokasikan, maka akan *preempted*. Sumber daya yang dimiliki proses tadi akan diberikan pada proses lain yang membutuhkan dan sedang menunggu. Proses akan mengulang kembali eksekusinya setelah mendapatkan semua sumber daya yang dibutuhkannya, termasuk sumber daya yang dimintanya terakhir.

4. ***Circular Wait .*** Kondisi 'lingkaran setan' ini dapat 'diputus' dengan jalan menentukan total kebutuhan terhadap semua tipe sumber daya yang ada. Selain itu, digunakan pula mekanisme enumerasi terhadap tipe-tipe sumber daya yang ada. Setiap proses yang akan meminta sumber daya harus meminta sumber daya dengan urutan yang menaik. Misalkan sumber daya *printer* memiliki nomor 1 sedangkan *CD-ROM* memiliki nomor 3. Proses boleh melakukan permintaan terhadap *printer* dan kemudian *CD-ROM*, namun tidak boleh sebaliknya.

23.7. Penghindaran

Penghindaran terhadap *deadlock* adalah cara penanganan yang selanjutnya. Inti dari penghindaran adalah jangan sembarangan membolehkan proses untuk memulai atau meminta lagi. Maksudnya adalah, jangan pernah memulai suatu proses apabila nantinya akan menuju ke keadaan *deadlock*. Kedua, jangan memberikan kesempatan pada proses untuk meminta sumber daya tambahan jika penambahan tersebut akan membawa sistem pada keadaan *deadlock*. Tidak mungkin akan terjadi *deadlock* apabila sebelum terjadi sudah kita hindari.

Langkah lain untuk menghindari adalah dengan cara tiap proses memberitahu jumlah kebutuhan maksimum untuk setiap tipe sumber daya yang ada. Selanjutnya terdapat *deadlock-avoidance algorithm* yang secara rutin memeriksa *state* dari sistem untuk memastikan tidak adanya kondisi *circular wait* serta sistem berada pada kondisi *safe state*. *Safe state* adalah suatu kondisi dimana semua proses mendapatkan sumber daya yang dimintanya dengan sumber daya yang tersedia. Apabila tidak bisa langsung, ia harus menunggu selama waktu tertentu, kemudian mendapatkan sumber daya yang diinginkan, melakukan eksekusi, dan terakhir melepas kembali sumber daya tersebut. Terdapat dua jenis algoritma penghindaran yaitu *resource-allocation graph* untuk *single instances resources* serta *banker's algorithm* untuk *multiple instances resources*.

Algoritma penghindaran yang pertama yaitu *resource-allocation graph* akan dijelaskan secara mendalam pada bab selanjutnya yaitu Diagram Graf. Untuk algoritma yang kedua yaitu *banker's algorithm* akan dibahas pada bab ini dan dilengkapi oleh pembahasan di bab selanjutnya.

Dalam *banker's algorithm*, terdapat beberapa struktur data yang digunakan, yaitu:

- ***Available .*** Jumlah sumber daya yang tersedia.
- ***Max .*** Jumlah sumber daya maksimum yang diminta oleh tiap proses.
- ***Allocation .*** Jumlah sumber daya yang sedang dimiliki oleh tiap proses.
- ***Need .*** Sisa sumber daya yang masih dibutuhkan oleh proses, didapat dari *max- allocation*.

Kemudian terdapat *safety algorithm* untuk menentukan apakah sistem berada pada *safe state* atau tidak.

Contoh 23.2. TestAndSet

```
01 work dan finish adalah vektor yang diinisialisasi:
   work = available
   finish[i] = FALSE untuk i= 1,2,3,..,n-1.
02 cari i yang memenuhi finish[i] == FALSE dan needi <= work
   jika tak ada, ke tahap 04
03 work = work + allocationi
   finish [i] = TRUE
   kembali ke tahap 02
04 jika finish[i]==TRUE untuk semua i, maka sistem safe state.
```

Terdapat juga algoritma lainnya yang menentukan apakah proses boleh melakukan permintaan terhadap sumber daya tambahan atau tidak. Algoritma yang bertujuan memastikan sistem tetap pada keadaan *safe state* ini dinamakan *resource-request algorithm.*

Contoh 23.3. TestAndSet

```
Request = sumber daya yang dibutuhkan proses Pi. Pada request,
Pi membutuhkan k instances dari Rj.

01 Jika Requesti <= Needi, ke tahap 02.
   Selain itu error karena melebihi maximum permintaan
02 Jika Requesti <= Available, ke tahap 03.
   Selain itu Pi harus menunggu karena tidak tersedia
03 Ubah kondisi state setelah request dikabulkan
      Available = Available - Requesti
      Allocationi = Allocationi + Requesti
      Needi = Needi - Requesti

   if safe => sumber daya dialokasikan pada Pi
   if unsafe => Pi menunggu, state kembali sebelumnya
```

Algoritma-algoritma tersebut bertujuan untuk menghindarkan sistem dari terjadinya *deadlock*. Keadaan dimana sistem bebas dari *deadlock* disebut *safe state*. Jadi, semua kebutuhan proses akan sumber daya terpenuhi. Dampaknya adalah sistem tidak mengalami *deadlock*. Selain *safe state*, terdapat pula keadaan *unsafe state*. Pada keadaan ini, sistem mempunyai kemungkinan untuk berada pada kondisi *deadlock*. Sehingga cara yang paling jitu untuk menghindari *deadlock* adalah memastikan bahwa sistem tidak akan pernah mengalami keadaan *unsafe state*.

23.8. Pendeteksian

Pada dasarnya kejadian *deadlock* sangatlah jarang terjadi. Apabila kondisi tersebut terjadi, masing-masing sistem operasi mempunyai mekanisme penanganan yang berbeda. Ada sistem operasi yang ketika terdapat kondisi *deadlock* dapat langsung mendeteksinya. Namun, ada pula sistem operasi yang bahkan tidak menyadari kalau dirinya sedang mengalami *deadlock*. Untuk sistem operasi yang dapat mendeteksi *deadlock*, digunakan algoritma pendeteksi. Secara lebih mendalam, pendeteksian kondisi *deadlock* adalah cara penanganan *deadlock* yang dilaksanakan apabila sistem telah berada pada kondisi *deadlock*. Sistem akan mendeteksi proses mana saja yang terlibat dalam kondisi *deadlock*. Setelah diketahui proses mana saja yang mengalami kondisi *deadlock*, maka diadakan mekanisme untuk memulihkan sistem dan menjadikan sistem berjalan kembali dengan normal.

Mekanisme pendeteksian adalah dengan menggunakan *detection algorithm* yang akan memberitahu sistem mengenai proses mana saja yang terkena *deadlock*. Setelah diketahui proses mana saja yang terlibat dalam *deadlock*, selanjutnya adalah dengan menjalankan mekanisme pemulihan sistem yang akan dibahas pada bagian selanjutnya. Berikut ini adalah algoritma pendeteksian *deadlock*.

23.9. Pemulihan

Pemulihan kondisi sistem terkait dengan pendeteksian terhadap *deadlock*. Apabila menurut algoritma pendeteksian *deadlock* sistem berada pada keadaan *deadlock*, maka harus segera dilakukan mekanisme pemulihan sistem. Berbahaya apabila sistem tidak segera dipulihkan dari *deadlock*, karena sistem dapat mengalami penurunan *performance* dan akhirnya terhenti.

Cara-cara yang ditempuh untuk memulihkan sistem dari *deadlock* adalah sebagai berikut:

1. **Terminasi proses.** Pemulihan sistem dapat dilakukan dengan cara melalukan terminasi terhadap semua proses yang terlibat dalam *deadlock*. Dapat pula dilakukan terminasi terhadap proses yang terlibat dalam *deadlock* secara satu per satu sampai 'lingkaran setan' atau *circular wait* hilang. Seperti diketahui bahwa *circular wait* adalah salah satu karakteristik terjadinya *deadlock* dan merupakan kesatuan dengan tiga karakteristik yang lain. Untuk itu, dengan menghilangkan kondisi *circular wait* dapat memulihkan sistem dari *deadlock*.Dalam melakukan terminasi terhadap proses yang *deadlock*, terdapat beberapa faktor yang menentukan proses mana yang akan diterminasi. Faktor pertama adalah prioritas dari proses-proses yang terlibat *deadlock*. Faktor kedua adalah berapa lama waktu yang dibutuhkan untuk eksekusi dan waktu proses menunggu sumber daya. Faktor ketiga adalah berapa banyak sumber daya yang telah dihabiskan dan yang masih dibutuhkan. Terakhir, faktor utilitas dari proses pun menjadi pertimbangan sistem untuk melakukan terminasi pada suatu proses.

2. ***Rollback and Restart .*** Dalam memulihkan keadaan sistem yang *deadlock*, dapat dilakukan dengan cara sistem melakukan *preempt* terhadap sebuah proses dan kembali ke *state* yang aman. Pada keadaan *safe state* tersebut, proses masih berjalan dengan normal, sehingga sistem dapat memulai proses dari posisi aman tersebut. Untuk menentukan pada saat apa proses akan *rollback*, tentunya ada faktor yang menentukan. Diusahakan untuk meminimalisasi kerugian yang timbul akibat memilih suatu proses menjadi korban. Harus pula dihindari keadaan dimana proses yang sama selalu menjadi korban, sehingga proses tersebut tidak akan pernah sukses menjalankan eksekusi.

23.10. Rangkuman

Deadlock adalah suatu keadaan dimana sistem seperti terhenti dikarenakan setiap proses memiliki sumber daya yang tidak bisa dibagi dan menunggu untuk mendapatkan sumber daya yang sedang dimiliki oleh proses lain.

Starvation adalah keadaan dimana satu atau beberapa proses 'kelaparan' karena terus dan terus menunggu kebutuhan sumber dayanya dipenuhi. Namun, karena sumber daya tersebut tidak tersedia atau dialokasikan untuk proses lain, akhirnya proses yang membutuhkan tidak bisa memilikinya. Kondisi seperti ini merupakan akibat dari keadaan menunggu yang berkepanjangan.

Karakteristik terjadinya *deadlock*:
- ***Mutual Exclusion .***
- ***Hold and Wait .***
- ***No Preemption .***
- ***Circular Wait .***

Mekanisme penanganan *deadlock*:
- **Pengabaian.** *Ostrich Algorithm*.
- **Pencegahan.** Mencegah terjadinya salah satu kondisi *deadlock*.
- **Penghindaran.** Memastikan sistem berada pada *safe state* dan dengan menggunakan *deadlock avoidance algorithm*.

- **Pendeteksian dan Pemulihan.** Mekanisme pendeteksian menggunakan *detection algorithm*, sedangkan pemulihan dengan cara *rollback and restart* sistem ke *safe state*.

Rujukan

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

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

[WEBRpi2004] Computer Science RPI. 2004 . *Deadlock http://www.cs.rpi.edu/academics/courses/fall04/os/c10/index.html*. Diakses 28 Maret 2007.

[WEBWiki2006a] Wikipedia. 2006 . *Deadlock http://en.wikipedia.org/wiki/Deadlock*. Diakses 05 Februari 2007.

[WEBWiki2006b] Wikipedia. 2006 . *Banker's Algorithm http://en.wikipedia.org/wiki/Banker%27s_algorithm*. Diakses 16 Februari 2007.

Bab 24. Diagram Graf

24.1. Pendahuluan

Berdasarkan penjelasan sebelumnya mengenai *deadlock*, diperlukan suatu penggambaran tentang bentuk *deadlock*. Dalam hal ini graf digunakan untuk merepresentasikan hal tersebut.

Deadlock adalah suatu kondisi dimana proses tidak berjalan lagi ataupun tidak ada komunikasi antar proses di dalam sistem operasi. Salah satu gambaran terjadinya *deadlock*, misalkan proses1 menunggu sumber daya yang sedang dipegang oleh proses2, sedangkan proses2 itu sedang menunggu sumber daya yang dipegang oleh proses1. Jadi tidak ada satu pun proses yang bisa *running*, melepaskan sumber daya, atau dibangunkan. Sumber daya, proses, dan *deadlock* tersebut dapat digambarkan dengan graf.

Sedangkan graf adalah suatu struktur diskrit yang terdiri dari simpul dan *edge*, dimana *edge* menghubungkan simpul-simpul yang ada. Berdasarkan hubungan antara *edge* dan simpulnya, graf dibagi menjadi dua, yaitu graf sederhana dan graf tak-sederhana. Berdasarkan arahnya graf dapat dibagi menjadi dua yaitu graf berarah dan graf tidak berarah. Graf berarah memperhatikan arah *edge* yang menghubungkan dua simpul, sedangkan graf tidak berarah tidak memperhatikan arah *edge* yang menghubungkan dua simpul.

Dalam Bab 24 ini akan dibahas mengenai implementasi graf dalam sistem operasi, yaitu dalam penggunaannya untuk penanganan *deadlock* pada sistem operasi. Diantaranya adalah graf alokasi sumber daya dan graf tunggu . Graf alokasi sumber daya dan graf tunggu merupakan graf sederhana dan graf berarah. Dua graf tersebut adalah bentuk visualisasi dalam mendeteksi masalah *deadlock* pada sistem operasi.

Setiap sumber daya pada sistem operasi akan digunakan oleh proses-proses yang membutuhkannya. Mekanisme hubungan dari proses-proses dan sumber daya itu dapat diwakilkan dan digambarkan dengan graf alokasi sumber daya dan graf tunggu. Dengan adanya visualisasi dari graf tersebut, maka masalah *deadlock* pada sistem operasi dapat dideteksi dan diselesaikan.

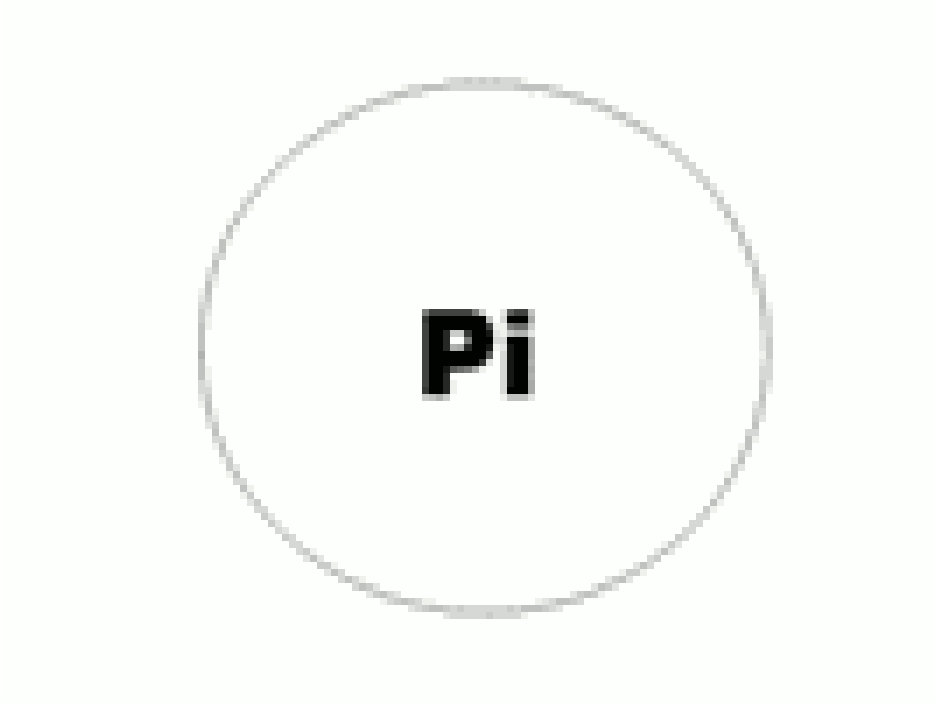
24.2. Komponen Alokasi Sumber Daya

Graf alokasi sumber daya mempunyai komponen-komponen layaknya graf biasa. Hanya saja dalam graf alokasi sumber daya ini, *vertex* dibagi menjadi 2 jenis yaitu:

1. **Proses.**

   P = {P0, P1, P2, P3,.... , Pi}. P terdiri dari semua proses yang ada di sistem. Untuk proses, vertexnya digambarkan sebagai lingkaran dengan nama prosesnya.

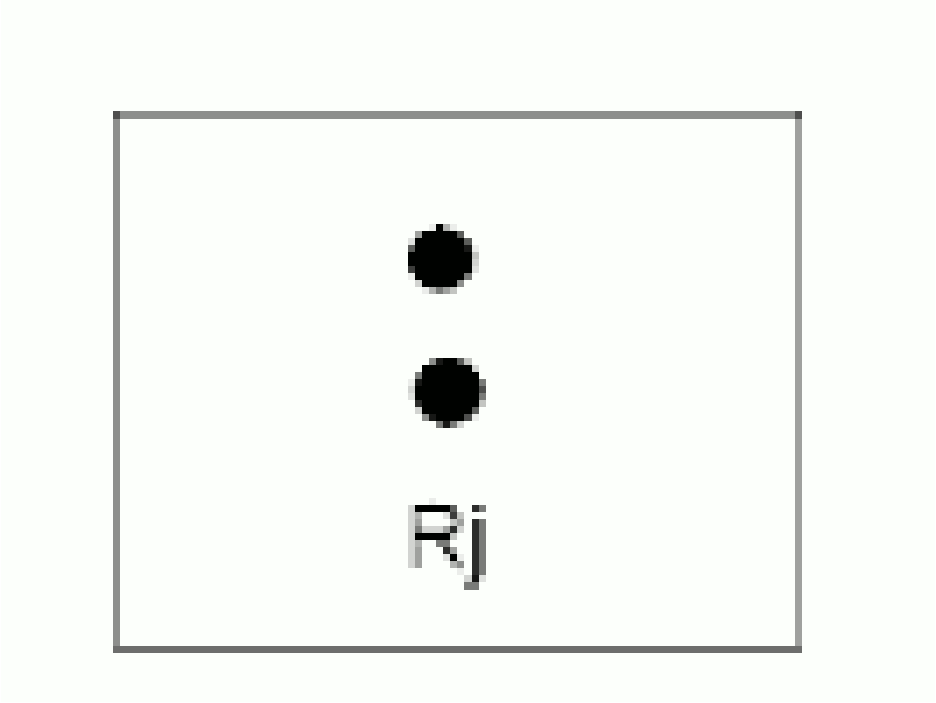
Gambar 24.1. Proses Pi



2. ***Resource .***

   Sumber daya R= {R0, R1, R2, R3, ..., Rj}. R terdiri dari semua sumber daya yang ada di sistem. Untuk sumber daya, vertexnya digambarkan sebagai segi empat dengan titik ditengahnya yang menunjukkan jumlah instans yang dapat dialokasikan serta nama sumber dayanya.

Gambar 24.2. Sumber daya Rj

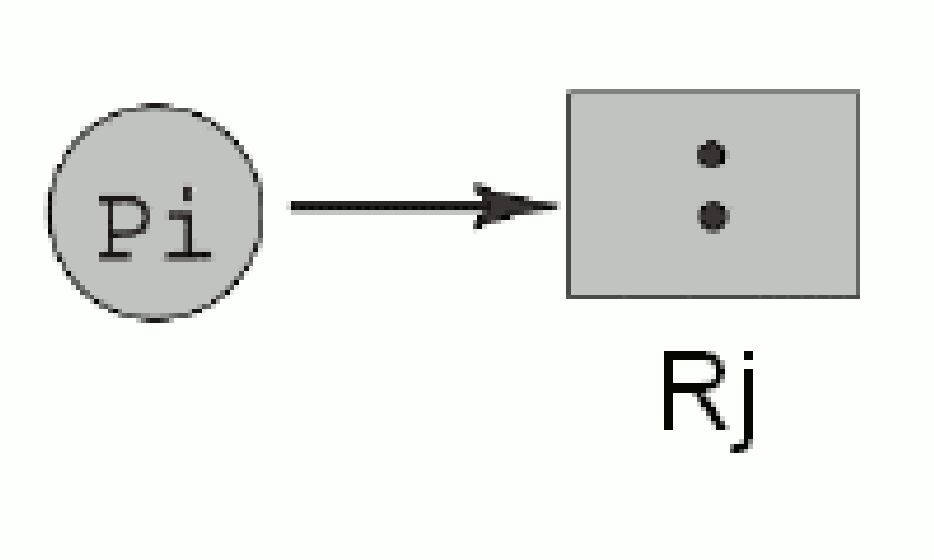


Proses dan *resource* dihubungkan oleh sebuah *edge* (sisi). Untuk *edge*, terdiri dari dua jenis yaitu:

1. ***Edge* permintaan: Pi->Rj.**

   Edge permintaan menggambarkan adanya suatu proses Pi yang meminta sumber daya Rj

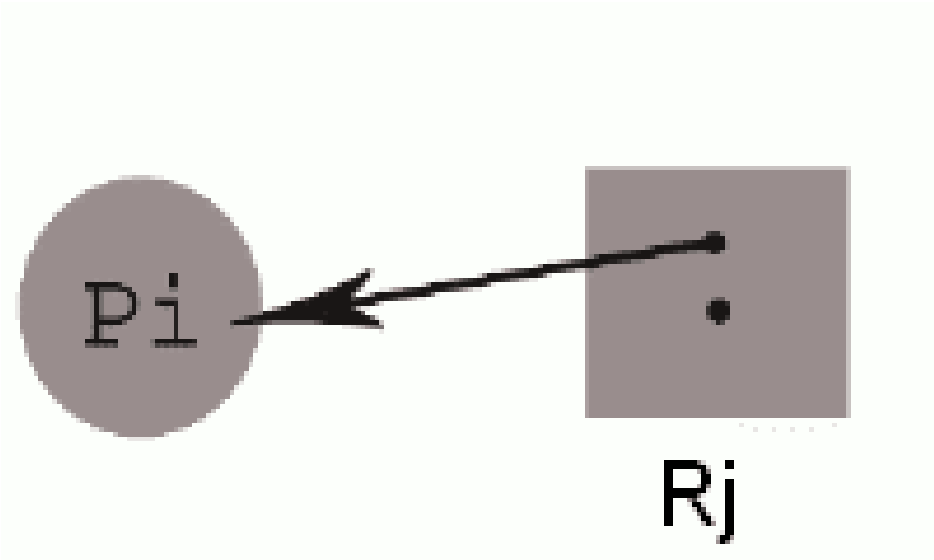
Gambar 24.3. Proses Pi meminta sumber daya Rj



2. ***Edge* Alokasi Sumber Daya: Rj->Pi.**

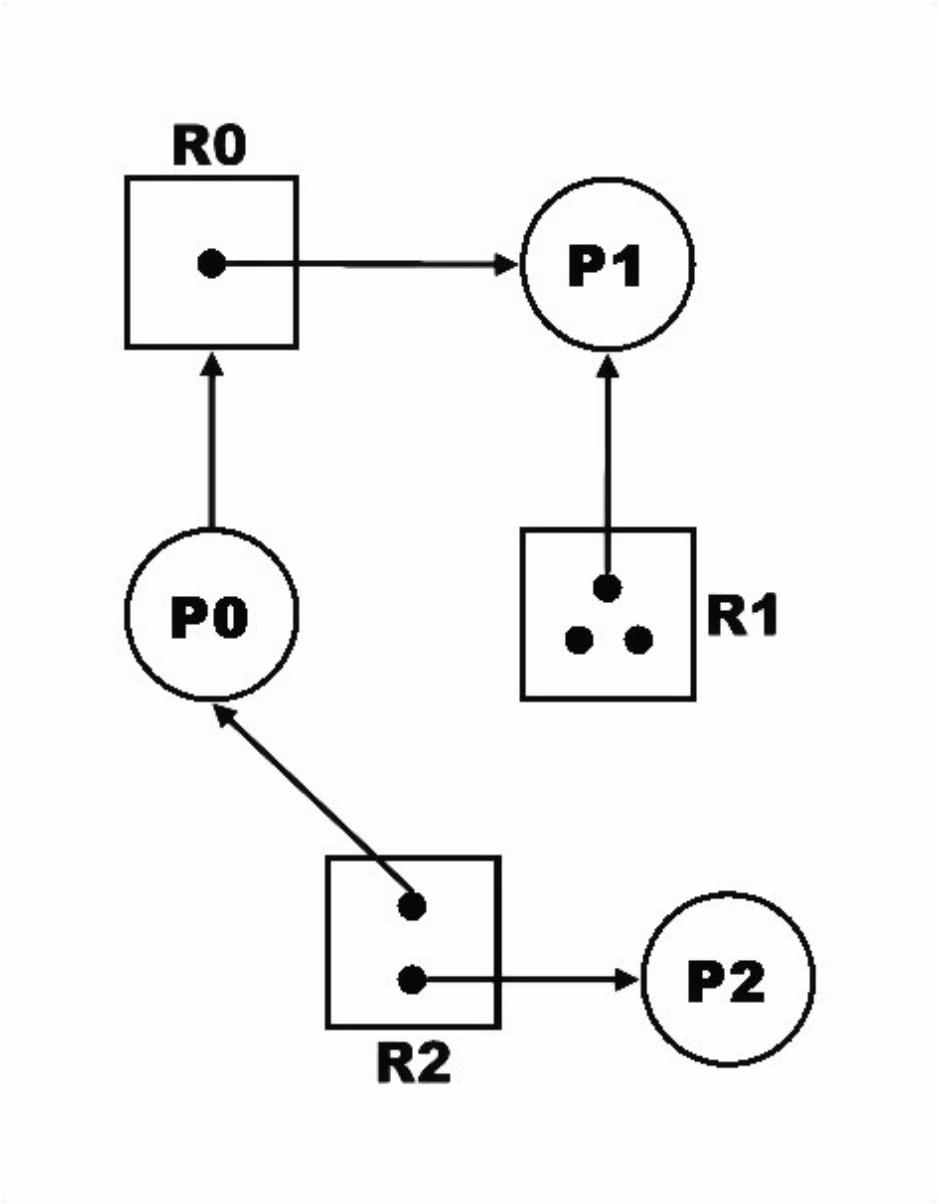
   Edge alokasi sumber daya menggambarkan adanya suatu sumber daya Rj yang mengalokasikan sumber dayanya pada Pi

Gambar 24.4. Resource Rj meminta sumber daya Pi



Setelah mengetahui bentuk *vertex* dan *edge* yang digunakan, kita akan lihat bagaimana salah satu contoh penggunaan graf alokasi sumber daya.

Gambar 24.5. Contoh graf alokasi sumber daya



Graf diatas terdiri dari 6 *vertex* dan 5 *edge*, V= {P0, P1, P2, R0, R1, R2}

E = {P0-> R0, R0-> P1, R1-> P1, R2-> P0, R2-> P2}. Keterangan Graf diatas :

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

Setelah suatu proses telah mendapatkan semua sumber daya yang diperlukan maka sumber daya tersebut dilepas dan dapat digunakan oleh proses lain.Sebuah proses menggunakan *resource* dengan urutan sebagai berikut:

1. Mengajukan permohonan *(request).* Bila Permohonan tidak dapat dikabulkan dengan segera (misal karena *resource* sedang digunakan oleh proses lain), maka proses itu harus menunggu sampai *resource* yang dimintanya tersedia.

2. Menggunakan *resource (use).* Proses dapat menggunakan *resource*, misal : printer untuk mencetak, *disk drive* untuk melakukan operasi M/K , dan sebagainya .

3. Melepaskan *resource (release).* Setelah proses menyelesaikan penggunaan *resource*, maka *resource* harus dilepaskan sehingga dapat digunakan oleh proses lain.

24.3. Metode Penghindaran

Bila metode *prevention* lebih menekankan pada cara permintaan sehingga keempat kondisi yang dapat menyebabkan *deadlock* tidak terjadi bersamaan, maka metode *avoidance* lebih mengarah pada perlunya informasi tambahan dari proses mengenai bagaimana resource akan diminta.

Pada saat sebuah proses mengajukan permintaan untuk menggunakan *resource* yang tersedia, maka algoritma *avoidance* akan bekerja dengan mendeteksi apakah alokasi yang diberikan dapat menyebabkan sistem dalam *safe state*. Bila keadaan hasilnya sistem *safe state*, maka *resource* akan dialokasikan untuk proses tersebut, tetapi bila sebaliknya maka permintaan akan ditolak.

Sebuah sistem berada dalam *safe state* bila terdapat *safe sequence* dimana proses yang memerlukan *resource* dapat ditangani. Bila P1 selesai menggunakan *resource* dan melepaskannya, maka P2 dapat menggunakan *resource* yang sedang digunakannya dan *resource* yang dilepas oleh P1 dapat digunakan P2 untuk menyelesaikan tugasnya dan kemudian melepaskan *resource* untuk digunakan oleh P3, dan seterusnya

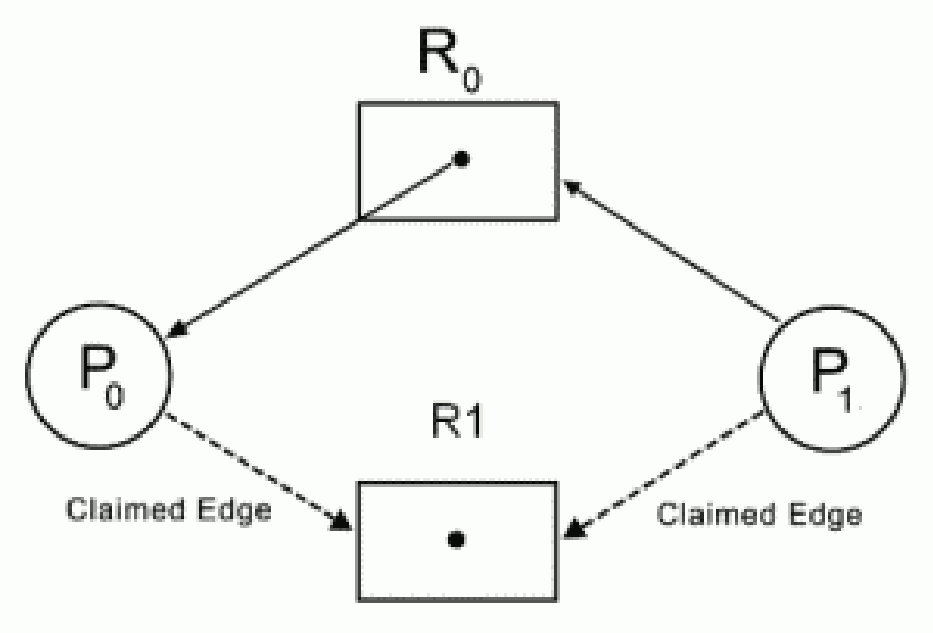
Algoritma Graf Alokasi Sumber Daya Untuk Mencegah *Deadlock*

Algoritma ini dapat dipakai untuk mencegah *deadlock* jika sumber daya hanya memiliki satu instans. Pada algoritma ini ada komponen tambahan pada *edge* yaitu *Claimed Edge*. Sama halnya dengan *edge* yang lain, *claimed edge* menghubungkan antara sumber daya dan simpul.

Claimed edge Pi ---> Rj berarti bahwa proses Pi akan meminta sumber daya Rj pada suatu waktu. *Claimed edge* sebenarnya merupakan *edge* permintaan yang digambarkan sebagai garis putus-putus. Ketika proses Pi memerlukan sumber daya Rj, *claimed edge* diubah menjadi *edge* permintaan. Dan setelah proses Pi selesai menggunakan Rj, *edge* 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.

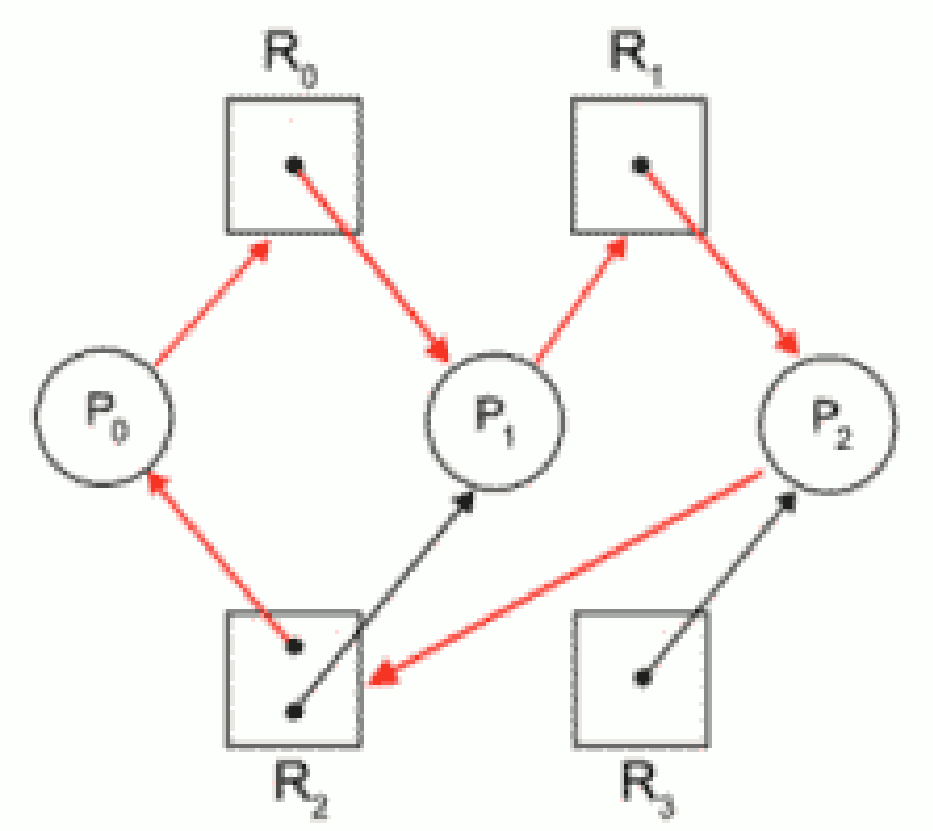
Algoritma ini memerlukan waktu n^2 dalam mendeteksi perputaran dimana n adalah jumlah proses dalam sistem. Jika tidak ada perputaran dalam graf, maka sistem berada dalam status aman. Tetapi jika perputaran ditemukan maka sistem berada dalam status tidak aman. Pada saat status tidak aman ini, proses Pi harus menunggu sampai permintaan sumber dayanya dipenuhi.

Gambar 24.6. Graf Alokasi Sumber Daya dalam status aman



Pada saat ini R1 sedang tidak mengalokasikan sumber dayanya, sehingga P1 dapat memperoleh sumber daya R1. Namun, jika *claimed edge* diubah menjadi *edge* permintaan dan kemudian diubah menjadi *edge* alokasi, hal ini dapat menyebabkan terjadinya perputaran.

Gambar 24.7. Graf dengan Deadlock



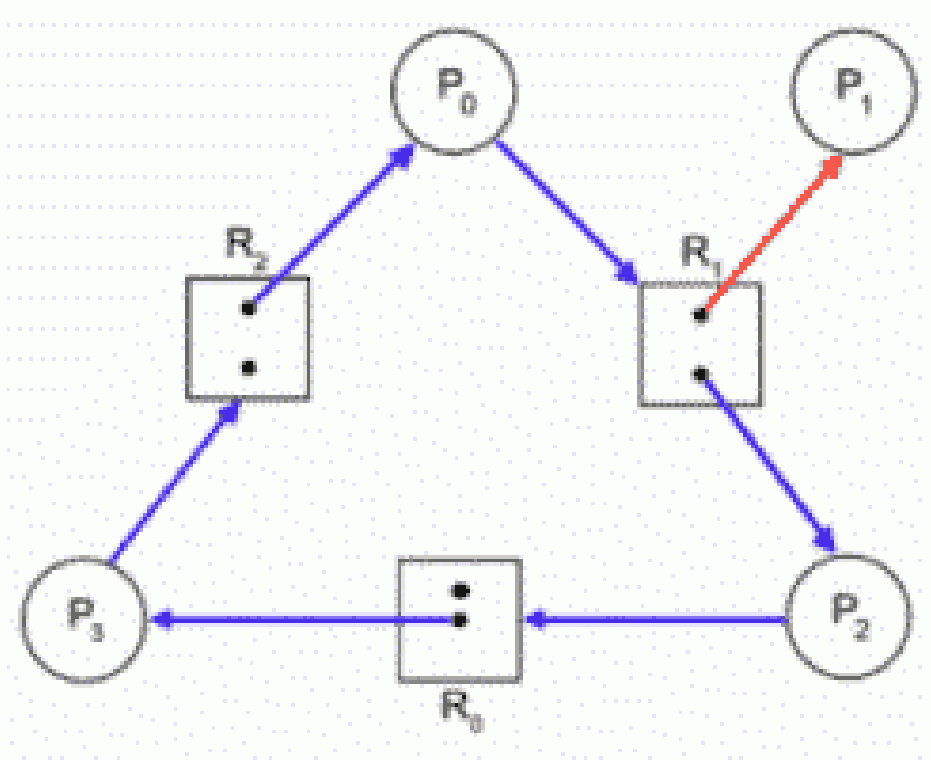
Dari gambar diatas kita dapat melihat terjadinya *deadlock* yang disebabkan oleh P0 memerlukan sumber daya R0 untuk menyelesaikan prosesnya, sedangkan R0 dialokasikan untuk P1. Di lain pihak

P1 memerlukan sumber daya R1 sedangkan R1 dialokasikan untuk P2. P2 memerlukan sumber daya R2 akan tetapi R2 mengalokasikan sumber dayanya pada P1.

1. R2 ->P0 ->R0 ->P1 -> R1 -> P2 -> R2

2. R2 -> P1-> R1 -> P2 -> R2

Gambar 24.8. Contoh Graf tanpa *Deadlock*



Gambar di atas menunjukkan beberapa hal sebagai berikut:

1. P0 meminta sumber daya dari R1

2. R1 memberikan sumber dayanya kepada P1

3. R1 memberikan satu instans sumber dayanya kepada P2

4. P2 meminta sumber daya pada P0

5. R0 memberikan sumber daya pada P3

6. P3 meminta sumber daya pada R2

7. R2 mengalokasikan sumber daya pada P0

24.4. Algoritma Bankir

Algoritma ini dapat digambarkan sebagai seorang bankir *(Resources)* di kota kecil yang berurusan dengan kelompok orang yang meminta pinjaman (Proses). 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 berada pada kondisi dimana dananya habis dan tidak dapat meminjamkan uang lagi. Jika hal tersebut terjadi, maka akan terjadi kondisi *deadlock*. Agar kondisi aman, maka asumsi setiap pinjaman harus dikembalikan waktu yang tepat.

Untuk sumber daya dengan instan banyak, ketika sebuah proses meminta sumber daya ia harus menunggu terlebih dahulu. Ketika sebuah proses telah mendapatkan semua sumber dayanya ia harus mengembalikannya dalam suatu batasan waktu. Algoritma ini dapat ditulis secara lebih jelas sebagai berikut:

```
Let P ={P1, P2, P3, ...., Pn}be set of all processes while
P is not empty do

Seek Pi, an element of P that can finish

If no Pi can be found then

End algorithm: state is unsafe

else

Remove Pi from P

Return resource of Pi to allocated pool

end if

end while

End algorithm: state is safe
```

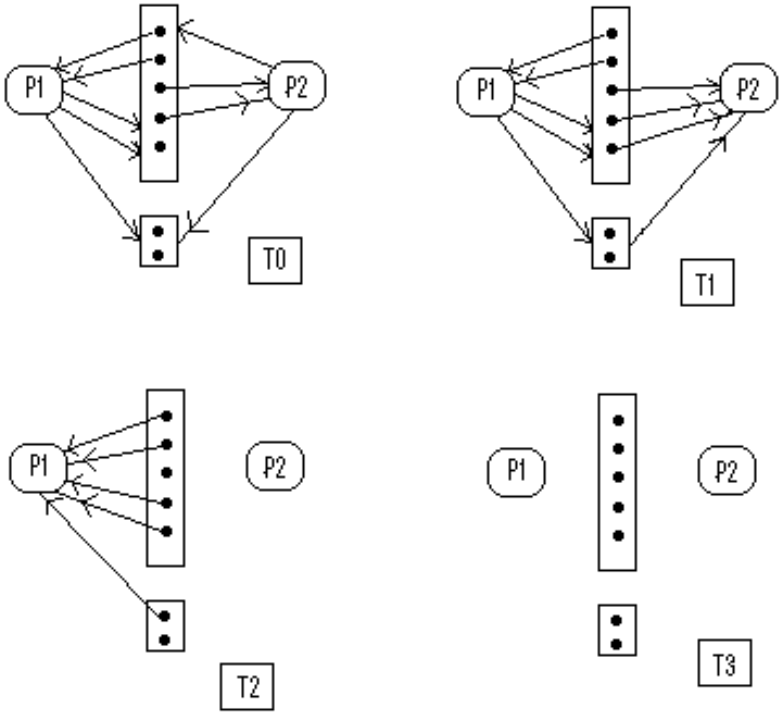
Implementasi Algoritma Bankir

Keterangan soal

Diketahui: Set P terdiri dari 2 proses (P1 &P2). Set R terdiri dari 2 sumber daya(R1 & R2). R1 = 5 instan ; R2 = 2 instan Implementasi menggunakan Algoritma Bankir. Prioritas pada proses dengan indeks kecil Setelah semua sumber daya terpenuhi, proses akan mengembalikan semua sumber daya

tsb. Gambarkan kondisi saat T0 sampai Tn saat kondisi semua sumber daya sudah dikembalikan ke R masing-masing!

Gambar 24.9. Jawaban soal



Pada saat T0, P1 mendapatkan 2 *resource* dari R1 pada saat bersamaan P1 meminta 2 *resource* ke R1 dan 1 *resource* ke R2. Sedangkan P2 mendapatkan 2 *resource* dari R1 dan pada waktu yang bersamaan P2 meminta 1 *resource* ke R1 dan 1 *resource* ke R2.

Pada saat T1, P1 belum mendapatkan *resource* yang dimintanya pada waktu T0 dari R2, karena P1 masih belum mendapatkan seluruh *resource* R1 . Alokasi dari R1 masih tetap terjadi karena pada P1 masih terjadi proses meminta *resource* R1. P1 masih belum mendapat alokasi karena *resource* R1 tidak mencukupi untuk diberikan ke P1. Sedangkan pada P2, request edge-nya ke R1 pada waktu T0 telah berubah menjadi *assignment edge*, karena *request* P2 dapat dipenuhi oleh R1. P2 juga memperoleh *resource* yang dimintanya pada waktu T0 dari R2

Pada saat T2 , P1 telah mendapatkan semua *resource* yang dimintanya dari R1, karena P2 telah melepaskan semua *resource* R1 yang dimilikinya.

Pada saat T3, P1 telah melepaskan semua *resource* yang dimilikinya sehingga R1 dan R2 sudah mendapatkan semua *resource*-nya kembali dan sudah dapat digunakan oleh proses yang lain.

24.5. Metode Pendeteksian

Deadlock akan terjadi, jika dan hanya jika grafik tunggu memiliki siklus di dalamnya.Untuk mendeteksi *deadlock*, sistem harus memiliki grafik tunggu dan menjalankan algoritma deteksi *deadlock* secara periodik. Hal yang harus diperhatikan adalah seberapa sering algoritma deteksi harus dipanggil. Hal ini tergantung dari dua faktor:

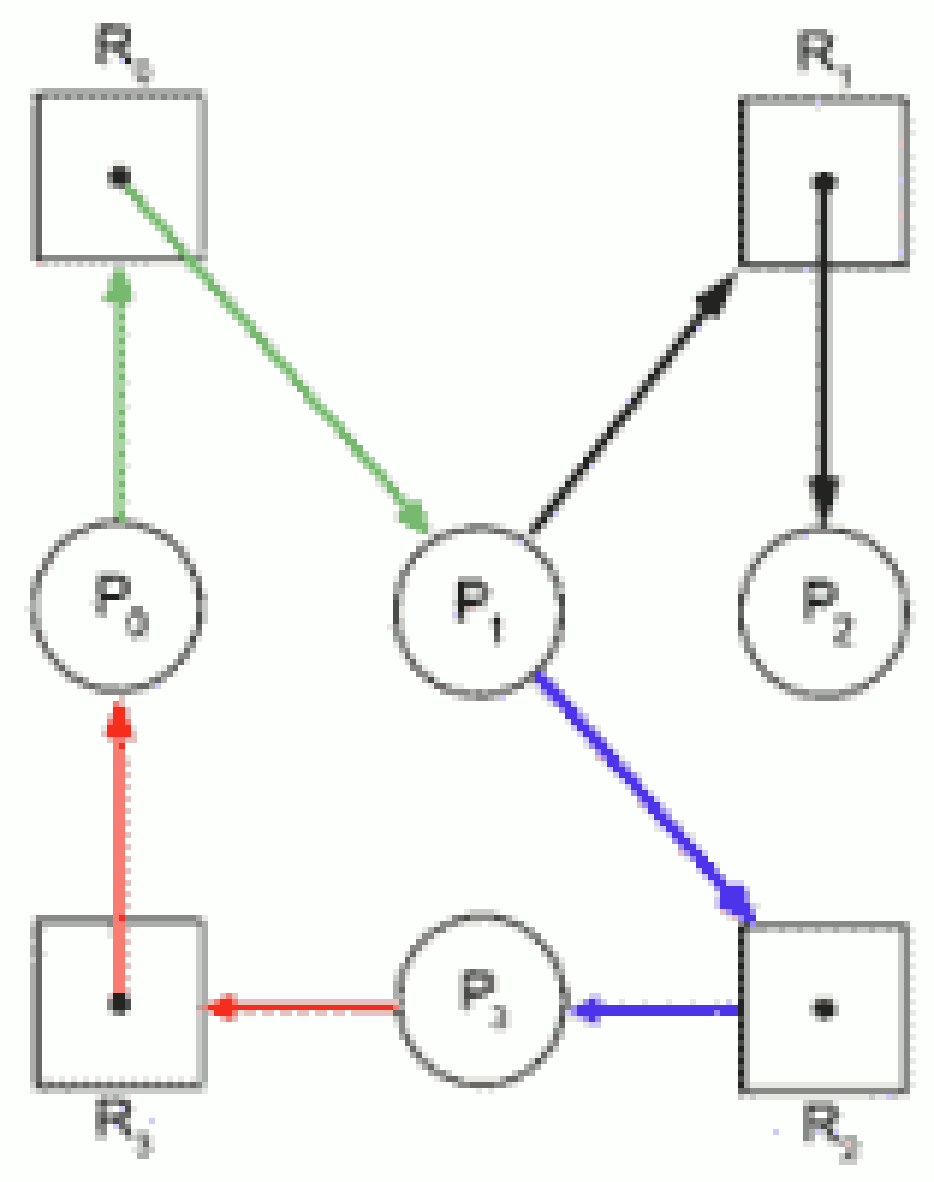
1. Frekuensi terjadinya *deadlock* pada umumnya
2. Jumlah proses yang akan terpengaruh ketika *deadlock* terjadi.

Bila *deadlock* terjadi maka algoritma deteksi harus sering dipanggil. *Resource* yang dialokasikan ke proses-proses yang mengalami *deadlock* tidak akan digunakan sampai kondisi *deadlock* diatasi. Bila *deadlock* tidak segera diatasi maka jumlah proses yang terlibat dalam *deadlock* akan semakin bertambah.

Salah satu ciri terjadinya *deadlock* adalah ketika beberapa proses mengajukan permohonan untuk *resource*, tetapi permohonan ini tidak dapat dipenuhi dengan segera. Sistem dapat saja memanggil algoritma deteksi setiap kali permohonan untuk *resource* tidak dapat diperoleh dengan segera. Namun,semakin sering algoritma deteksi dipanggil, maka waktu *overhead* yang dibutuhkan untuk komputasi menjadi semakin besar.

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:

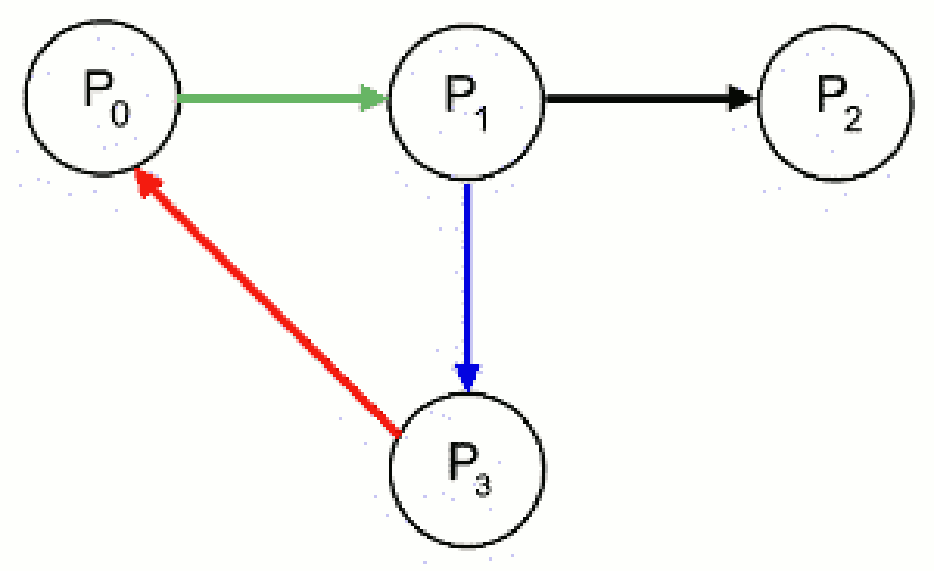
Gambar 24.10. Contoh Graf Alokasi Sumber Daya yang akan diubah menjadi graf tunggu



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 *edge* 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. Contoh Graf Tunggu



Untuk mendeteksi *deadlock*, sistem perlu membuat graf tunggu dan secara berkala memeriksa apakah ada perputaran atau tidak. Untuk mendeteksi adanya perputaran diperlukan operasi sebanyak n^2, dimana n adalah jumlah simpul dalam graf alokasi sumber daya.

24.6. Rangkuman

Deadlock adalah suatu kondisi dimana proses tidak berjalan lagi ataupun tidak ada komunikasi lagi antar proses di dalam sistem operasi. *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.

Untuk mendeteksi *deadlock* dan menyelesaikannya dapat digunakan graf sebagai visualisasinya. Jika dalam graf terlihat adanya perputaran, maka proses tersebut memiliki potensi terjadi *deadlock*. Namun, jika dalam graf tidak terlihat adanya perputaran, maka proses tersebut tidak akan terjadi *deadlock*.

implementasi graf dalam sistem operasi, yaitu penggunaannya untuk penanganan *deadlock* pada sistem operasi. Diantaranya adalah graf alokasi sumber daya dan graf tunggu.Graf alokasi sumber daya dan graf tunggu merupakan graf sederhana dan graf berarah. Dua graf tersebut adalah bentuk visualisasi dalam mendeteksi masalah *deadlock* pada sistem operasi.

Untuk mengetahui ada atau tidaknya *deadlock* dalam suatu graf alokasi sumber daya dapat dilihat dari perputaran dan sumber daya yang dimilikinya. Jika tidak ada perputaran berarti tidak *deadlock*. Jika ada perputaran, ada potensi terjadi *deadlock*. Sumber daya dengan instans tunggal dan perputaran pasti akan mengakibatkan *deadlock*.

Pada graf tunggu, *deadlock* terjadi jika dan hanya jika pada graf tersebut ada perputaran. Untuk mendeteksi adanya perputaran diperlukan operasi sebanyak n^2, dimana n adalah jumlah simpul dalam graf alokasi sumber daya.

Rujukan

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

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

[SistemOperasiModern2005] Ir.Riri Fitri Sari dan Yansen Darmaputra. 2005 . *Sistem Operasi Modern*. Edisi Pertama. Penerbit Andi.

Bab 25. *Bounded-Buffer*

25.1. Pendahuluan

Proses kooperatif adalah proses yang saling mempengaruhi satu sama lain. Proses kooperatif ini dapat berbagi sumber daya secara langsung atau berbagi data melalui pertukaran pesan. Dengan demikian perlu dilakukan sinkronisasi untuk mencegah timbulnya data yang tidak konsisten akibat akses data secara konkuren oleh lebih dari 1 proses.

Pesan-pesan yang dipertukarkan antar proses membutuhkan antrian sementara yang sering kita sebut sebagai penyangga atau *buffer*. *Buffer* dapat dibagi menjadi 3 jenis sesuai kapasitasnya, yaitu *buffer* yang kapasitasnya 0; *buffer* yang kapasitasnya tak hingga; serta *buffer* yang kapasitasnya dibatasi sebanyak n. *Buffer* dengan kapasitas terbatas inilah yang disebut sebagai *bounded-buffer*.

Salah satu ilustrasi proses yang menggunakan *bounded buffer* adalah proses produsen-konsumen, dimana produsen menaruh data ke dalam *buffer* untuk kemudian diambil oleh konsumen. Masalah yang timbul adalah *buffer* yang kemudian menjadi *critical section*. Pada satu waktu, hanya 1 proses yang boleh memasuki *critical section*, dengan demikian *buffer* hanya bisa diakses oleh produser saja atau konsumen saja pada 1 waktu. Masalah berikutnya adalah ketika produsen ingin menaruh data, namun *buffer* penuh, atau ketika konsumen ingin mengambil data, namun *buffer* masih kosong. Ini adalah salah satu masalah sinkronisasi klasik yang dikenal pula dengan nama *bounded-buffer problem* atau *producer-consumer problem*.

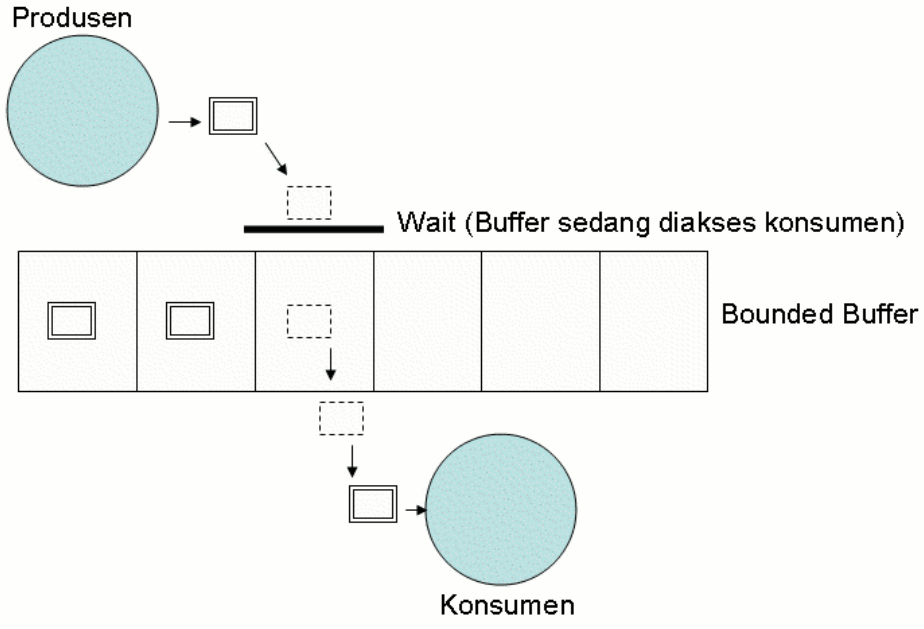
25.2. Penggunaan Semafor

Untuk mensinkronisasikan proses produsen dan proses konsumen ini, digunakan perangkat sinkronisasi semafor. Semafor yang digunakan adalah:

- **mutex.** *Binary semaphore* yang dapat bernilai 0 atau 1 untuk menjaga agar pada suatu waktu *buffer* hanya dapat diakses oleh satu proses saja. Mutex bernilai 0 jika *buffer* sedang diakses sebuah proses, dan bernilai 1 jika tidak.
- **tempat_kosong.** *Counting semaphore* berisi data tempat kosong pada *buffer*.
- **tempat_terisi.** *Counting semaphore* berisi data tempat terisi pada *buffer*.

Gambar 25.1. Produsen Menunggu Konsumen

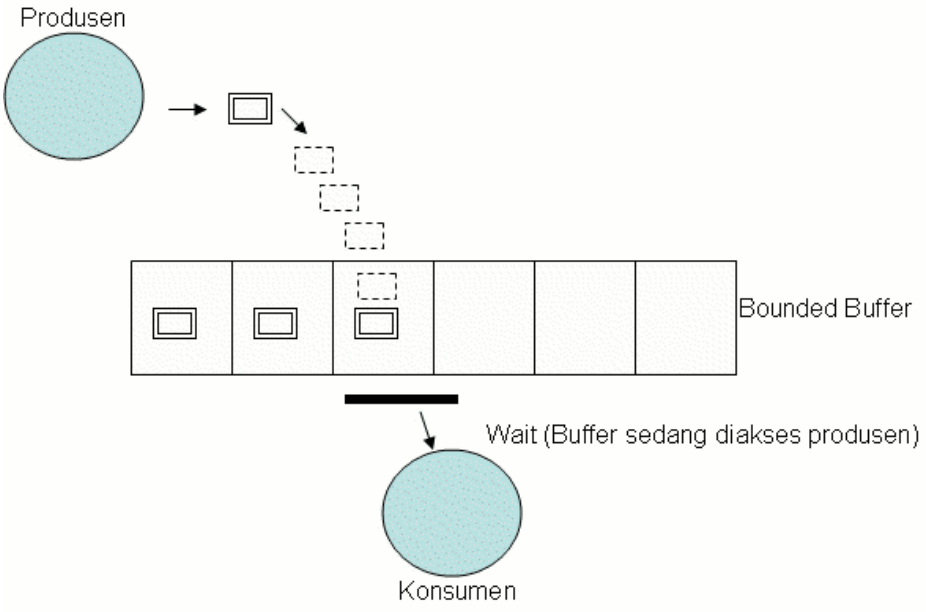
Kejadian dimana produsen ingin mengakses *buffer*, namun *buffer* sedang diakses oleh konsumen.



Dengan demikian produsen akan menunggu jika mendapati *buffer* sedang dipakai konsumen dan atau *buffer* penuh. *Buffer* yang sedang dipakai konsumen ditandai dengan semafor mutex yang bernilai 0; dan *buffer* yang penuh ditandai dengan semafor tempat_kosong yang bernilai 0, yang berarti tempat kosong di *buffer* ada 0. Konsumen akan menunggu juga jika *buffer* sedang dipakai oleh produsen atau jika *buffer* kosong. *Buffer* kosong ditandai dengan semafor tempat_terisi yang bernilai 0, yang berarti tempat terisi ada 0.

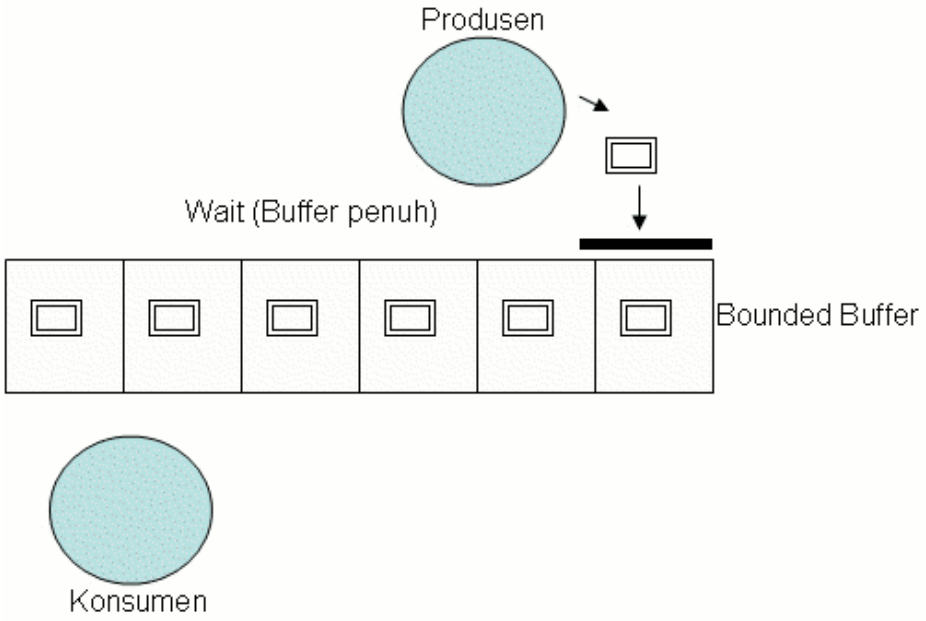
Gambar 25.2. Konsumen Menunggu Produsen

Kejadian dimana konsumen ingin mengakses *buffer*, namun *buffer* sedang diakses oleh produsen.



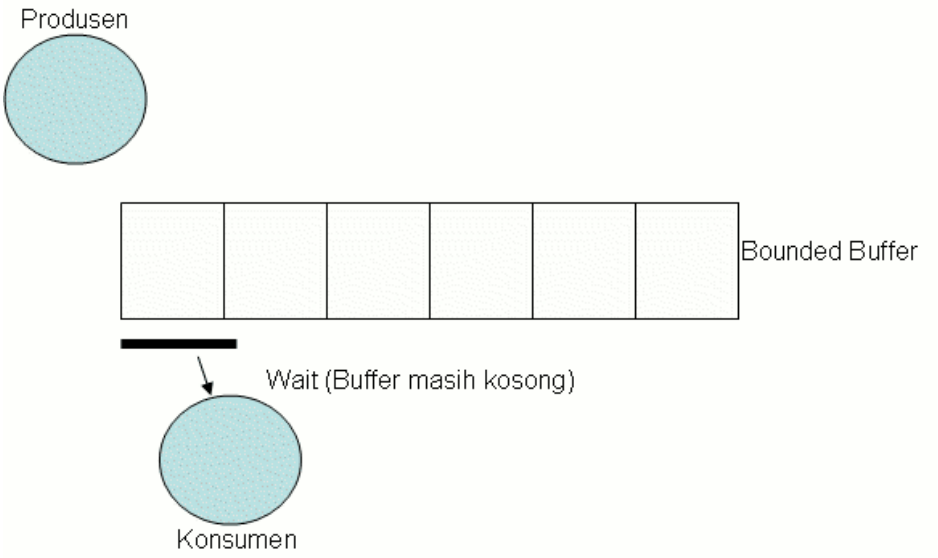
Gambar 25.3. Produsen Menunggu *Buffer* Penuh

Produsen tidak bisa menaruh apapun di *buffer* ketika *buffer* penuh (tidak bisa menimpa data yang ada).



Gambar 25.4. Konsumen Menunggu *Buffer* Kosong

Konsumen tidak bisa mengambil apapun dari *buffer* ketika *buffer* kosong.



Jika sebuah proses mendapat akses *buffer*, proses tersebut harus men -*set* mutex menjadi 0 sebagai tanda bahwa *buffer* sedang ia gunakan dan ketika selesai proses tersebut akan mengembalikan nilai mutex menjadi 1 agar *buffer* dapat diakses oleh proses yang lain.

25.3. Program

Contoh 25.1 Bounded-Buffer Problem

```
000   // Authors: Greg Gagne, Peter Galvin, Avi Silberschatz
001   // Modified by Rahmat M. Samik-Ibrahim
002   // and also so very slightly modified by Laksmita Rahadianti
003   // copyright (c) 2000 by G. Gagne, P. Galvin, A. Silberschatz
004   // Applied Operating Systems Concepts - John Wiley and Sons,
Inc.
005
006   import java.util.*;
007
008   public class MasalahProdusenKonsumen
009  {
010      public static void main(String args[])
011     {
012         BoundedBuffer server = new BoundedBuffer();
013         Produsen threadProdusen = new Produsen(server);
014         Konsumen threadKonsumen = new Konsumen(server);
015         threadProdusen.start();
016         threadKonsumen.start();
017      }
018   }
019
020   class BoundedBuffer
021   {
022      public static final int TIDUR   = 5;
023      public static final int UK_BUFFER = 3;
024      private Semaphore mutex;
```

```
025      private Semaphore tempat_kosong;
026      private Semaphore tempat_terisi;
027      private int count, tmptMsk, tmptKlr;
028      private Object[] buffer;
029      public BoundedBuffer()
030      {
031         count=tmptMsk=tmptKlr=0;
032         buffer=new Object[UK_BUFFER];
033         mutex=new Semaphore(1);
034         tempat_kosong=new Semaphore(UK_BUFFER);
035         tempat_terisi=new Semaphore(0);
036      }
037      public static void napping()
038      {
039         int tidur = (int) (TIDUR*Math.random());
040         try{ Thread.sleep(tidur*1000);}
041            catch(InterruptedException e){}
042      }
043      public void taruh(Object item)
044      {
045         tempat_kosong.acquire();   mutex.acquire();
046         ++count;
047         buffer[tmptMsk]=item;
048         tmptMsk=(tmptMsk+1)% UK_BUFFER;
049         System.out.println("Produsen memasukkan :   "+item);
050         mutex.release();   tempat_terisi.release();
051      }
052      public Object ambil()
053      {
054         Object item;
055         tempat_terisi.acquire();   mutex.acquire();
056         --count;
057         item=buffer[tmptKlr];
058         tmptKlr=(tmptKlr+1)%UK_BUFFER;
059         System.out.println("Konsumen mengambil :  "+item);
060         mutex.release(); tempat_kosong.release();
061         return item;
062      }
063   }
064
065   class Konsumen extends Thread
066   {
067      private BoundedBuffer buffer;
068      public Konsumen(BoundedBuffer b)
069      {
070          buffer = b;
071      }
072      public void run()
073      {
074         Date data;
075         while(true)
076         {
077             BoundedBuffer.napping();
078             System.out.println("Konsumen ingin mengambil.");
079             data = (Date) buffer.ambil();
080         }
081      }
082   }
```

```
083
084   class Produsen extends Thread
085   {
086      private BoundedBuffer buffer;
087      public Produsen(BoundedBuffer b)
088      {
089         buffer = b;
090      }
091      public void run()
092      {
093          Date data;
094          while(true)
095          {
096              data=new Date();
097              BoundedBuffer.napping();
098              System.out.println("Produsen menghasilkan :  "+data);
099              System.out.println("Produsen ingin menaruh.");
100              buffer.taruh(data);
101          }
102      }
103   }
104
105   final class Semaphore
106   {
107      private int value;
108      public Semaphore(int v)
109      {
110         value=v;
111      }
112      public synchronized void acquire()
113      {
114         while(value==0)
115         {
116         try{wait();}
117             catch(InterruptedException e){}
118         }
119         value--;
120      }
121      public synchronized void release()
122      {
123         ++value;
124         notify();
125      }
126   }
```

25.4. Penjelasan Program

Program ini meng- *import package* `java.util` untuk mengambil `Date` yang akan kita gunakan sebagai data yang dipertukarkan proses konsumen dan produsen melalui *buffer* (baris 006).

Contoh 25.2 MasalahProdusenKonsumen

```
008   public class MasalahProdusenKonsumen
009  {
010      public static void main(String args[])
```

```
011     {
012         BoundedBuffer server = new BoundedBuffer();
013         Produsen threadProdusen = new Produsen(server);
014         Konsumen threadKonsumen = new Konsumen(server);
015         threadProdusen.start();
016         threadKonsumen.start();
017      }
018   }
```

Kelas `MasalahProdusenKonsumen` mengandung *method* `main` yang membuat instansiasi objek `BoundedBuffer` yang diberi nama `server`. `Server` merupakan antrian sementara yang akan digunakan oleh produsen dan konsumen untuk menaruh data yang dipertukarkan.

Contoh 25.3 BoundedBuffer

```
029      public BoundedBuffer()
030      {
031         count=tmptMsk=tmptKlr=0;
032         buffer=new Object[UK_BUFFER];
033         mutex=new Semaphore(1);
034         tempat_kosong=new Semaphore(UK_BUFFER);
035         tempat_terisi=new Semaphore(0);
036      }
```

Selain itu `MasalahProdusenKonsumen` akan membuat instansiasi dari kelas `Produsen` dan `Konsumen`, yang masing-masing berjalan sebagai *thread* yang paralel dengan `main`. Ketika *thread* produsen dan konsumen dimulai di `main` dengan dipanggilnya fungsi `start()`(baris 015 dan 016), masing-masing proses itu akan menjalankan *method* `run()`-nya seperti di bawah ini.

Contoh 25.4 Method Run Konsumen

```
072      public void run()
073      {
074         Date data;
075         while(true)
076         {
077             BoundedBuffer.napping();
078             System.out.println("Konsumen ingin mengambil.");
079             data = (Date) buffer.ambil();
080         }
081      }
```

Contoh 25.5 Method Run Produsen

```
091      public void run()
092      {
093          Date data;
094          while(true)
095          {
096              data=new Date();
097              BoundedBuffer.napping();
```

```
098             System.out.println("Produsen menghasilkan :  "+data);
099             System.out.println("Produsen ingin menaruh.");
100             buffer.taruh(data);
101          }
102      }
```

Dapat dilihat bahwa masing-masing *thread* akan tidur sejenak dengan fungsi `napping()`(baris 077 dan baris 097), untuk memberi jeda antar proses. Waktu untuk tidur ini adalah angka *random* yang dihasilkan dari `Math.random()`(baris 039).

Contoh 25.6 Napping

```
037      public static void napping()
038      {
039         int tidur = (int) (TIDUR*Math.random());
040         try{ Thread.sleep(tidur*1000);}
041            catch(InterruptedException e){}
042      }
```

Dalam fungsi `napping()`, program akan menangkap jika ada `InterruptedException`(baris 040-041). Perlu diingat bahwa program ini tidak akan terminasi sendiri (akan terjadi *loop* yang terus-menerus). Dengan demikan, kita harus menterminasi program ini dengan paksa (dengan perintah CTRL+C), akan ditangkap di bagian `catch` ini, sehingga program akan diterminasi.

Fungsi lain yang dipanggil pada saat *thread* produsen dan konsumen menjalankan `run()` adalah `taruh` yang dipanggil oleh produsen dan `ambil` yang dipanggil oleh konsumen. `taruh` berfungsi untuk memasukkan data ke dalam *buffer* sedangkan `ambil` untuk mengambil data.

Contoh 25.7 Taruh dan Ambil

```
043      public void taruh(Object item)
044      {
045         tempat_kosong.acquire();   mutex.acquire();
046         ++count;
047         buffer[tmptMsk]=item;
048         tmptMsk=(tmptMsk+1)% UK_BUFFER;
049         System.out.println("Produsen memasukkan :   "+item);
050         mutex.release();   tempat_terisi.release();
051      }
052      public Object ambil()
053      {
054         Object item;
055         tempat_terisi.acquire();   mutex.acquire();
056         --count;
057         item=buffer[tmptKlr];
058         tmptKlr=(tmptKlr+1)%UK_BUFFER;
059         System.out.println("Konsumen mengambil :  "+item);
060         mutex.release(); tempat_kosong.release();
061         return item;
062      }
```

Pada kelas `BoundedBuffer` tadi, telah diinstansiasikan 3 buah semafor yang diinisialisasi dengan nilai 1 untuk *mutex*, ukuran buffer untuk *tempat_kosong*, dan 0 untuk *tempat_terisi*. Semafor ini

digunakan pada saat percobaan memasukkan atau mengambil data dari *buffer* oleh *thread* produsen dan konsumen.

Contoh 25.8 Instansiasi dan Inisialisasi Semafor

```
024      private Semaphore mutex;
025      private Semaphore tempat_kosong;
026      private Semaphore tempat_terisi;
.
.
.
033      mutex=new Semaphore(1);
034      tempat_kosong=new Semaphore(UK_BUFFER);
035      tempat_terisi=new Semaphore(0);
```

Dengan demikian, pada saat produsen memanggil `taruh`, hal pertama yang dilakukan adalah percobaan mengurangi jumlah tempat kosong melalui *acquire* semafor tempat_kosong (baris 045). Hal ini diikuti oleh percobaan pengambilalihan *mutex*. (baris 045). Sedangkan konsumen mencoba mengurangi jumlah tempat terisi melalui pengambilalihan semafor tempat_terisi (baris 055). Setelah itu konsumen berusaha mengambil alih juga semafor *mutex*(baris 055). Setelah produsen berhasil memperoleh *mutex* dan mengurangi tempat_kosong, maka ia akan mengeksekusikan baris 046-049 dari program `taruh` yang merupakan *critical section*. Ketika selesai, ia akan melepas kembali *mutex* dan mengupdate jumlah tempat terisi dengan cara melepas semafor tempat_terisi. Demikian pula dengan konsumen, setelah memperoleh semafor mutex dan tempat_terisi, akan mengeksekusi baris 056-059 dari program `ambil`(yang juga merupakan *critical section*), lalu melepas kembali mutex dan mengupdate jumlah tempat kosong dengan melepas semafor tempat_kosong.

Contoh 25.9 Acquire Semafor

```
111   public synchronized void acquire()
112   {
113      while(value==0)
114      {
115         try{wait();}
116         catch(InterruptedException e){}
117      }
118      value--;
119   }
```

Contoh 25.10 Release Semafor

```
120   public synchronized void release()
121   {
122      ++value;
123      notify();
124   }
```

Semafor *mutex* bernilai 1 untuk menandai bahwa *buffer* sedang dipakai oleh sebuah proses. Ketika diinisialisasi pun ia bernilai 1, yang artinya di keadaan awal *buffer* tidak dipakai oleh proses manapun. Ketika sebuah proses berusaha meng- *acquire* semafor ini, dan menemui bahwa keadaannya sedang

bernilai 0 (baris 113) yang berarti *buffer* sedang dipakai proses lain, maka ia akan menunggu (baris 115) dengan fungsi `wait()` sampai nilai *mutex* bernilai 1 (*buffer* bebas). Ketika sebuah proses selesai, ia akan me- *release mutex* dengan menjadikan nilainya kembali ke 1. (baris 122) dan memberitahu proses lain yang sedang menunggu dengan memanggil `notify()`(baris 123).

Semafor tempat_kosong adalah jumlah tempat kosong. Dengan demikian ketika di- *acquire* oleh produsen, jika bernilai 0 (*buffer* penuh), maka produsen akan menunggu (baris 115), karena jika penuh maka produsen tidak bisa menimpa data yang tersimpan. Jika tidak 0, maka produsen akan mengurangi nilainya (mengurangi jumlah tempat yang kosong). Setelah produsen selesai maka ia akan mengubah nilai tempat_terisi (menambahkan 1 tempat yang terisi) dengan me- *release* semafor tempat_terisi. Untuk konsumen sama saja, namun yang di- *acquire* adalah tempat_terisi (jika *tempat_terisi*=0 berarti *buffer* kosong, belum ada yang terisi) dan yang di- *release* ketika proses selesai adalah tempat_kosong.

Dengan demikian, produsen dan konsumen akan secara bergantian menaruh atau mengambil data dari *buffer*, dan produsen tidak akan menaruh data jika *buffer* penuh (tidak akan menimpa data yang sudah ada), dan konsumen tidak bisa pula mengambil data jika *buffer* kosong.

Salah satu contoh keluaran program ini adalah:

Contoh 25.11 Contoh keluaran

```
Konsumen ingin mengambil.
Produsen menghasilkan :  Tue Apr 10 16:12:16 ICT 2007
Produsen ingin menaruh.
Produsen memasukkan :   Tue Apr 10 16:12:16 ICT 2007
Konsumen mengambil :  Tue Apr 10 16:12:16 ICT 2007
Produsen menghasilkan :  Tue Apr 10 16:12:19 ICT 2007
Produsen ingin menaruh.
Produsen memasukkan :   Tue Apr 10 16:12:19 ICT 2007
Konsumen ingin mengambil.
Konsumen mengambil :  Tue Apr 10 16:12:19 ICT 2007
Produsen menghasilkan :  Tue Apr 10 16:12:20 ICT 2007
Konsumen ingin mengambil.
Produsen ingin menaruh.
Produsen memasukkan :   Tue Apr 10 16:12:20 ICT 2007
Konsumen mengambil :  Tue Apr 10 16:12:20 ICT 2007
Konsumen ingin mengambil.
Produsen menghasilkan :  Tue Apr 10 16:12:22 ICT 2007
Produsen ingin menaruh.
Produsen memasukkan :   Tue Apr 10 16:12:22 ICT 2007
Konsumen mengambil :  Tue Apr 10 16:12:22 ICT 2007
```

25.5. Rangkuman

Proses yang kooperatif bisa berbagi data melalui penukaran pesan. Pesan-pesan yang dikirim antar proses akan disimpan dalam sebuah antrian sementara, yaitu *buffer*. Jika kapasitas *buffer* tersebut terbatas, maka dia disebut *bounded-buffer*. Untuk mencegah inkonsistensi data yang terjadi akibat akses data oleh proses kooperatif yang berjalan secara konkuren, maka diperlukan sinkronisasi antar proses-proses tersebut.

Permasalahan *bounded-buffer* ini diilustrasikan dalam proses produsen-konsumen. Masalah-masalah yang timbul adalah

- *buffer* yang merupakan *critical section*, sehingga hanya boleh diakses satu proses pada satu waktu;
- keadaan dimana produsen ingin menaruh data di antrian, namun antrian penuh
- keadaan dimana konsumen ingin mengambil data dari antrian namun antrian kosong.

Untuk menyelesaikan masalah, digunakanlah perangkat sinkronisasi semafor. Semafor yang digunakan adalah

- **mutex.** yang menjaga buffer hanya diakses satu proses pada satu waktu;
- **tempat_kosong.** jumlah tempat kosong.
- **tempat_terisi.** jumlah tempat terisi

Dengan demikian produsen yang ingin menaruh data atau konsumen yang ingin mengakses data harus memeriksa apakah proses lain sedang memakai *buffer* (menggunakan *mutex*) dan memeriksa apakah *buffer* penuh atau kosong (menggunakan *tempat_kosong* dan *tempat_terisi*).

Rujukan

[Deitel2005] Harvey M Deitel dan Paul J Deitel. 2005 . *Java How To Program*. Sixth Edition. Prentice Hall.

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

Bab 26. *Readers/Writers*

26.1. Pendahuluan

Masalah *Readers-Writers* merupakan masalah klasik dalam sinkronisasi. Solusi dari masalah ini diperlukan untuk dapat menjaga konsistensi data.

Banyak *thread* bisa berbagi sumber daya penyimpanan yang sama. Ada *thread* yang membaca, ada juga yang menulis. *Thread* yang membaca disebut *readers* (pembaca), sedangkan yang menulis disebut *writers* (penulis).

Jika lebih dari satu *thread* mengakses data yang sama pada satu waktu, bisa terjadi korupsi data. Kondisi yang harus dipenuhi agar tidak terjadi korupsi data adalah :

1. **Sebuah objek data bisa dibaca oleh beberapa *thread* secara simultan.**
2. **Sebuah objek data yang sedang ditulis oleh sebuah *thread* tidak dapat dibagi aksesnya kepada *thread* yang lain baik pembaca maupun penulis.**

Writer harus memiliki akses yang eksklusif terhadap suatu objek data, sehingga tidak boleh ada proses lain yang mengakses sebuah objek yang sedang diakses oleh *writer*.

26.2. Penggunaan Semafor

Semaphore adalah sebuah variabel bertipe integer yang selain saat inisialisasi, hanya dapat diakses melalui dua *method*, yaitu :
1. **`P()`(atau `wait()`, `tunggu()`, dan lain-lain).** Berfungsi sebagai *increment method*
2. **`V()`(atau `signal()`, `sinyal()`, dan lain-lain).** Berfungsi sebagai *decrement method*.

Jika *method* `P()` dipanggil, *thread* yang memanggilnya akan melakukan `wait()` (yang ada di dalam *method* `P()`) sampai di- `notify()`. `notify()` itu sendiri berada di dalam *method* `V()`, sehingga *thread* akan bisa mulai melakukan sesuatu pada objek data jika *method* `V()` telah dipanggil.

Pada program *Reader/Writer*, *Semaphore* digunakan untuk sinkronisasi antar *readers* atau antar *writers* atau antar *readers* dengan *writers*.

Implementasinya dalam kode bahasa Java

```
public synchronized void P()
{
   while (value <= 0)
   {
      try { wait(); }
      catch (InterruptedException e) { }
   }
   value--;
}
public synchronized void V()
{
   ++value;
   notify();
}
```

26.3. Program

```
001   /*_____________________________________________________________*/
002   /*class ReaderWriterServer_____________________________________*/
003   /*_____________________________________________________________*/
004
005   public class ReaderWriterServer
006   {
007      public static void main(String args[])
008      {
009         Database server = new Database();
010         Reader[] readerArray = new Reader[NUM_OF_READERS];
011         Writer[] writerArray = new Writer[NUM_OF_WRITERS];
012         for (int i = 0; i < NUM_OF_READERS; i++)
013         {
014            readerArray[i] = new Reader(i, server);
015            readerArray[i].start();
016         }
017         for (int i = 0; i < NUM_OF_WRITERS; i++)
018         {
019            writerArray[i] = new Writer(i, server);
020            writerArray[i].start();
021         }
022      }
023      private static final int NUM_OF_READERS = 3;
024      private static final int NUM_OF_WRITERS = 2;
025   }
026
027   /*_____________________________________________________________*/
028   /*class Reader_________________________________________________*/
029   /*_____________________________________________________________*/
030
031   class Reader extends Thread
032   {
033      public Reader(int r, Database db)
034      {
035         readerNum = r;
036         server = db;
037      }
038      public void run()
039      {
040         int c;
041         while (true)
042         {
043            Database.napping();
044            System.out.println("reader " + readerNum +
045            " wants to read.");
046            c = server.startRead();
047            System.out.println("reader " + readerNum +
048            " is reading. Reader Count = " + c);
049            Database.napping();
```

```
050            System.out.print("reader " + readerNum +
051            " is done reading. ");
052            c = server.endRead();
053         }
054      }
055      private Database server;
056      private int readerNum;
057   }
058
059   /*_____________________________________________________*/
060   /*class Writer_________________________________________*/
061   /*_____________________________________________________*/
062
063   class Writer extends Thread
064   {
065      public Writer(int w, Database db)
066      {
067         writerNum = w;
068         server = db;
069      }
070      public void run()
071      {
072         while (true)
073         {
074            System.out.println("writer " + writerNum +
075            " is sleeping.");
076            Database.napping();
077            System.out.println("writer " + writerNum +
078            " wants to write.");
079            server.startWrite();
080            System.out.println("writer " + writerNum +
081            " is writing.");
082            Database.napping();
083            System.out.println("writer " + writerNum +
084            " is done writing.");
085            server.endWrite();
086         }
087      }
088      private Database server;
089      private int writerNum;
090   }
091
092   /*_____________________________________________________*/
093   /*class Semaphore______________________________________*/
094   /*_____________________________________________________*/
095
096   final class Semaphore
097   {
098      public Semaphore()
099      {
100         value = 0;
101      }
102      public Semaphore(int v)
103      {
104         value = v;
105      }
106      public synchronized void P()
107      {
```

```
108         while (value <= 0)
109         {
110            try { wait(); }
111            catch (InterruptedException e) { }
112         }
113         value--;
114      }
115      public synchronized void V()
116      {
117         ++value;
118         notify();
119      }
120      private int value;
121   }
122
123   /*_____________________________________________________________*/
124   /*class Database_____________________________________________*/
125   /*_____________________________________________________________*/
126
127   class Database
128   {
129      public Database()
130      {
131         readerCount = 0;
132         mutex = new Semaphore(1);
133         db = new Semaphore(1);
134      }
135      public static void napping()
136      {
137         int sleepTime = (int) (NAP_TIME * Math.random() );
138         try { Thread.sleep(sleepTime*1000); }
139         catch(InterruptedException e) {}
140      }
141      public int startRead()
142      {
143         mutex.P();
144         ++readerCount;
145         if (readerCount == 1)
146         {
147            db.P();
148         }
149         mutex.V();
150         return readerCount;
151      }
152      public int endRead()
153      {
154         mutex.P();
155         --readerCount;
156         if (readerCount == 0)
157         {
158            db.V();;
159         }
160         mutex.V();
161         System.out.println("Reader count = " + readerCount);
162         return readerCount;
163      }
164      public void startWrite()
165      {
```

```
166         db.P();
167      }
168      public void endWrite()
169      {
170         db.V();
171      }
172      private int readerCount;
173      Semaphore mutex;
174      Semaphore db;
175      private static final int NAP_TIME = 15; }
```

Contoh keluaran. Contoh keluaran tidak mutlak seperti ini, bisa bervariasi, penyebab dari hal ini akan dijelaskan pada bagian Penjelasan Program.

```
writer 1 is sleeping.
writer 0 is sleeping.
writer 0 wants to write.
writer 0 is writing.
reader 2 wants to read.
writer 0 is done writing.
reader 2 is reading. Reader Count = 1
writer 0 is sleeping.
reader 2 is done reading. Reader count = 0
reader 0 wants to read.
reader 0 is reading. Reader Count = 1
reader 1 wants to read.
reader 1 is reading. Reader Count = 2
reader 1 is done reading. Reader count = 1
reader 0 is done reading. Reader count = 0
reader 2 wants to read.
reader 2 is reading. Reader Count = 1
writer 0 wants to write.
writer 1 wants to write.
reader 1 wants to read.
reader 1 is reading. Reader Count = 2
reader 2 is done reading. Reader count = 1
reader 1 is done reading. Reader count = 0
writer 0 is writing.
writer 0 is done writing.
writer 0 is sleeping.
writer 1 is writing.
writer 1 is done writing.
writer 1 is sleeping.
reader 0 wants to read.
reader 0 is reading. Reader Count = 1
reader 1 wants to read.
reader 1 is reading. Reader Count = 2
writer 0 wants to write.
reader 2 wants to read.
reader 2 is reading. Reader Count = 3
reader 2 is done reading. Reader count = 2
reader 0 is done reading. Reader count = 1
writer 1 wants to write.
reader 1 is done reading.Reader count = 0
writer 0 is writing.
```

```
reader 2 wants to read.
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 2 is reading. Reader Count = 1
reader 0 is reading. Reader Count = 2
reader 1 is reading. Reader Count = 3
reader 2 is done reading. Reader count = 2
writer 0 wants to write.
reader 0 is done reading. Reader count = 1
```

26.4. Penjelasan Program

Pada program *readers/writers* diatas, ada 5 class:

`ReaderWriterServer` (baris 5-25). Kelas ini merupakan kelas yang memuat *method* `main()`. Kelas ini membuat 5 objek, 3 objek *reader* dan 2 objek *writer* serta menjalankan semua objek-objek tadi.

`Reader`(baris 31-57). Kelas ini berfungsi sebagai *thread* yang membaca data. Kelas ini sebelum mulai membaca akan mencetak *"reader ... wants to read."* terlebih dahulu kemudian memanggil *method* `startRead()` untuk mulai membaca, jika tidak ditahan pada salah satu *semaphore* (di kelas `Database`) akan ada output *"reader ... is reading..."*; kemudian setelah waktu yang acak (oleh `Database.napping()`) akan tercetak *"reader ... is done reading..."* dan memanggil *method* `endRead()` pada kelas `Database` untuk berhenti membaca.

`Writer`(baris 63-90). Kelas ini berfungsi sebagai *thread* yang menulis data. Kelas ini sebelum mulai akan tidur terlebih dahulu, lalu akan mencetak *"writer ... wants to write."* terlebih dahulu kemudian memanggil *method* `startWrite()` untuk mulai menulis, jika tidak ditahan pada *semaphore* `db` (di kelas `Database`) akan ada output *"writer ... is writing..."*; kemudian setelah waktu yang acak (oleh `Database.napping()`) akan tercetak *"writer ... is done writing."* dan memanggil *method* `endWrite()` pada kelas `Database` untuk berhenti menulis.

`Semaphore`(baris 96-121). Seperti yang sudah dijelaskan pada bagian sebelumnya, pada program ini, *Semaphore* digunakan untuk sinkronisasi antar *readers* atau antar *writers* atau antar *readers* dengan *readers*. *Method* yang ada pada kelas ini adalah *method* `P()` yang akan mencoba mengurangi nilai variabel `value`, variabel tersebut akan bisa dikurangi jika variabel tersebut bernilai lebih dari nol. Jika variabel `value` sama dengan atau kurang dari nol maka *thread* yang memanggil method ini akan memanggil method `wait()` yang membuatnya menunggu. *Method* ini berfungsi untuk menghalangi *thread* untuk tidak masuk ke dalam *critical section*. *Method* `P()` memanggil *method* `wait()` yang akan menyebabkan *thread- thread* yang akan memasuki *critical section* menunggu. Selain itu pada kelas ini terdapat *method* `V()` yang di dalamnya menambahkan nilai `value` sehingga nilainya tidak nol lagi, serta memanggil *method* `notify()`. *Method* `notify()` akan membangunkan salah satu *thread* yang sedang menunggu secara acak.

`Database`(baris 127-176). Kelas ini mengimplementasi semua pekerjaan yang dilaksanakan oleh kelas `Writer` dan kelas `Reader`. Kelas Database ini mempunyai 5 *method* yaitu `napping()`, `startRead()`, `endRead()`, `startWrite()`, dan `endWrite()`. *Method* `napping()` berfungsi membuat *thread* yang mengaksesnya akan memanggil *method* `sleep()` sehingga *thread* tersebut 'tertidur'. Waktu tidur tersebut acak dan tidak mutlak, hal ini menyebabkan output dari program akan bervariasi, tapi tetap memenuhi ketentuan. *Method* kedua adalah `startRead()`. *Method* ini me *return* sebuah *int* merepresentasikan jumlah reader yang sedang mengakses *database* pada saat itu. Dia hanya mengijinkan hanya satu *reader* pada satu waktu yang dapat mengakses

`readerCount` dengan mengunci *Semaphore* `mutex`(dengan `mutex.P()`). Bila jumlah reader == 1 maka *reader* tersebut akan menutup *Semaphore* db (dengan `db.P()`) sehingga *writer* tidak diijinkan masuk untuk mengakses data. *Method* selanjutnya adalah `endRead()`. *Method* ini diakses oleh *reader* yang telah selesai mengakses data. Untuk mengurangi `readerCount` maka *reader* tersebut harus mengakses *Semaphore* `mutex` kembali. Jika nilai `readerCount`== 0, yang berarti sudah tidak ada lagi *reader* yang sedang mengakses data, maka reader memanggil *method* `V()` untuk membuka *Semaphore* `db`, sehingga *writer* yang sedang mengantri dapat masuk untuk mengakses data. *Method* selanjutnya adalah `startWrite()`. *Method* ini hanya memanggil satu *method* yaitu mengunci *Semaphore* `db`(dengan `db.P()`). Hal ini dimaksudkan bila ada *writer* yang sedang mengakses data, maka tidak ada *writer* lain atau *reader* yang diperbolehkan masuk untuk mengakses data. *Method* `endWrite()` dipanggil oleh kelas writer yang telah selesai mengakses database. *Method* ini memanggil *method* `V()` untuk membuka *Semaphore* db sehingga reader atau writer lain dapat masuk untuk mengakses data.

26.5. Rangkuman

Readers/Writers merupakan sebuah masalah klasik dalam contoh sinkronisasi untuk menjaga validitas data. Jika reader sedang mengakses data, reader-reader yang lain boleh ikut mengakses data, tapi writer harus menunggu sampai data tidak diakses siapapun. Jika writer sedang mengakses data, tidak boleh ada *thread* lain yang mengakses data. *Semaphore* digunakan untuk sinkronisasi antar *thread* (baik readers maupun writers).

Rujukan

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

[WEBWiki2007] From Wikipedia, the free encyclopedia. 2007 . *Readers-writers_problem – http://en.wikipedia.org/wiki/Readers-writers_problem*. Diakses 21 Maret 2007.

Bab 27. Sinkronisasi Dengan Semafor

27.1. Pendahuluan

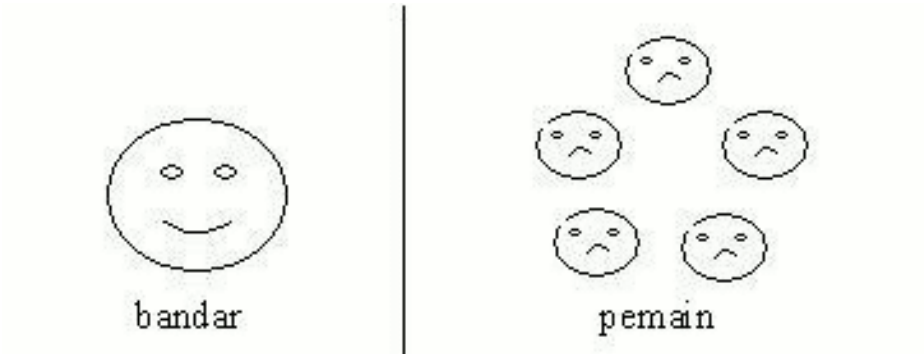
Sinkronisasi dua arah adalah suatu mekanisme dimana suatu *thread* dapat mengendalikan sinkronisasi *thread* lain, begitu pula sebaliknya. Berikut ini adalah sebuah program yang menggunakan proses sinkronisasi dua arah. Program yang dibuat dalam bahasa Java ini, bernama Hompimpah. Contoh program sinkronisasi dua arah adalah semafor berjenis *non-spinlock*. Semafor jenis *non-spinlock* berarti implementasi dari kelas Semafor, dimana semafor yang digunakan adalah jika ada proses yang sedang berjalan, maka proses lain akan menunggu, sampai ada *thread* lain yang memanggilnya. Hal ini sesuai dengan prinsip *critical section* yaitu *mutualy exclusive*, hanya satu thread yang diizinkan mengakses *critical section*; *progress*, *critical section* pasti ada yang mengakses; *bounded waiting*, setiap thread dijamin dapat meng-akses *critical section*. Hal ini dapat dilihat dalam program yaitu ketika sang bandar sedang berada pada *critical section*, pemain akan menunggu hingga bandar selesai mengaksesnya. Sedangkan kebalikannya adalah semafor jenis *spinlock*, yang akan melakukan *infinite loop*, sehingga *thread* tersebut tidak ada jaminan akan masuk *critical section*. Semafor jenis *non-spinlock* ini terdiri dari 2 operasi untuk proses sinkronisasi, yaitu operasi `buka()`,dan operasi `kunci()`.

27.2. Penggunaan Semafor

Program ini berjalan layaknya permainan Hompimpah yang telah kita kenal. Adapun yang dilakukan dalam permainan hompimpah adalah sebagai berikut:

1. Peranan

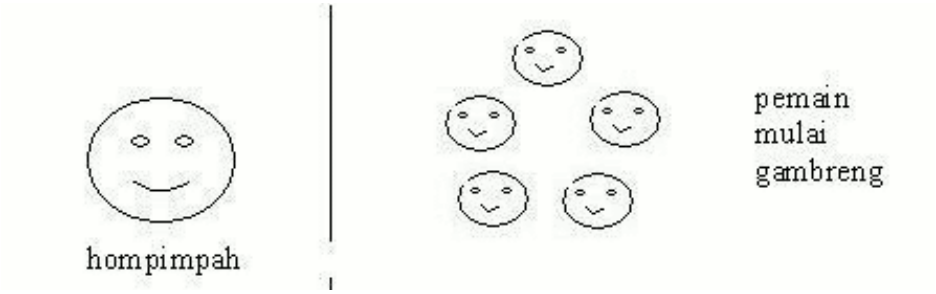
Gambar 27.1. Peranan yang terdapat dalam permainan



Dalam permainan Hompimpah kita mengenal dua peranan, yaitu pemain-pemain yang melakukan hompimpah (gambar sebelah kanan yang berjumlah 5) dan seorang bandar (gambar sebelah kiri), yang berfungsi menetukan, kapan permainan dimulai dan kapan permainan berakhir. Dalam hal ini bandar bertindak sebagai pengatur jalannya permainan.

2. Memulai permainan

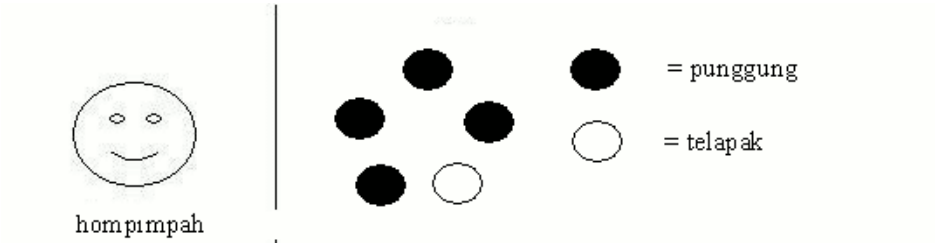
Gambar 27.2. Bandar memulai permainan



Permainan mulai ketika bandar menyerukan Hompimpah. Ketika itu secara serentak pemain-pemain akan melakukan hompimpah dengan tangannya. Kemudian bandar akan menyerukan gambreng sebagai penanda hompimpah berhenti, sehingga pemain pun menghentikan hompimpah dan masing-masing akan menunjukan tangannya, apakah telapak atau punggung.

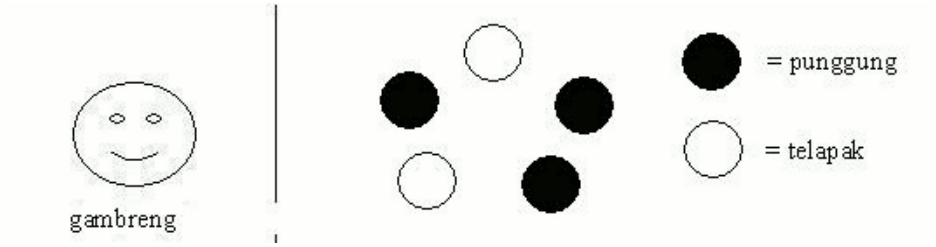
3. Pengecekan pemenang

Gambar 27.3. Bandar memeriksa pemenang



Terlihat pada gambar, setelah bandar menyerukan gambreng dan hompimpah berhenti, bandar akan melakukan perhitungan banyaknya telapak dan punggung. Apabila bandar menemukan ada diantara telapak atau punggung yang berjumlah satu, maka satu-satunya pemain itu akan keluar sebagai pemenang. Namun apabila tidak ada diantara telapak atau punggung yang berjumlah satu, seperti pada gambar maka bandar akan mengulang permainan dengan menyerukan gambreng. Hal tersebut akan berulang terus-menerus hingga terdapat 1 tangan yang berbeda. Dengan kata lain, bandar akan terus menyerukan gambreng dan mengulang permainan hingga menemukan pemenang.

Gambar 27.4. Bandar mengulang gambreng



Seperti halnya permainan Hompimpah yang diilustrasikan di atas, program Hompimpah juga berjalan demikian layaknya permainan Hompimpah sebenarnya. Tetapi, ada perbedaan terhadap jumlah bandamya. Program Hompimpah memiliki jumlah bandar yang sama dengan jumlah pemainnya, sehingga setiap pemain memiliki satu bandar. Adapun tujuan sinkronisasi dua arah adalah mengendalikan *thread-thread* sesuai dengan keinginan kita melalui kendali `bandarGambreng()` yang diimplementasikan pada program Hompimpah.

27.3. Program

```
134         for(int ii=0; ii<jumlahPemain;ii++)
135         {
136           pemain[ii].buka();
137           bandar[ii].kunci();
138         }
139       }
140
141       private void syncBandar()
142       {
143        for(int ii=0;ii<jumlahPemain;ii++)
144        {
145          bandar[ii].kunci();
146        }
147       }
148       private void syncPemain()
149       {
150         for(int ii=0;ii<jumlahPemain;ii++)
151         {
152           pemain[ii].buka();
153         }
154       }
155       private void syncBandarPemain(int ii)
156       {
157         bandar[ii].buka();
158         pemain[ii].kunci();
159       }
160
161
162 }
163
164 class Semafor
165 {
166       public Semafor()
167       {
168         value=0;
169       }
170       public Semafor(int val)
171       {
172         value=val;
173       }
174       public synchronized void kunci()
175       {
176         while(value==0)
177         {
178           try
179           {
180              wait();
181
182           }
183           catch(InterruptedException e)
184           {
185           }
186           value--;
187         }
188
189       }
190       public synchronized void buka()
191       {
192         value++;
193         notify();
194       }
195       private int value;
196 }
```

Contoh 27.1. Program yang menggunakan proses sinkronisasi dua arah

27.4. Penjelasan Program

Program Hompimpah terdiri dari 4 *class* yaitu *class* Hompimpah, *class* Pemain, *class* Gambreng, *class* Semafor. Masing-masing *class* berfungsi sebagai berikut:

1. *Class* Hompimpah: dalam kelas ini program Hompimpah dapat dijalankan
2. *Class* Gambreng: dalam kelas ini seluruh proses permainan Hompimpah dijalankan.
3. *Class* Semafor: kelas ini menjalankan proses sinkronisasi.
4. *Class* Hompimpah: dalam kelas terdapat fungsi `run()` dengan fungsi ini tiap-tiap pemain akan dihidupkan.

Penjelasan selengkapnya sebagai berikut:

Contoh 27.2. *Class* Hompimpah

```
001 public class Hompimpah
002 {
003
004      private static final int JUMLAH_PEMAIN=6;
005      public static void main(String args[])throws Exception
006      {
007       Gambreng gamserver=new Gambreng(JUMLAH_PEMAIN);
008         Thread[] pemain=new Thread[JUMLAH_PEMAIN];
009         for (int ii=0; ii<JUMLAH_PEMAIN; ii++)
010         {
011            pemain[ii]= new Thread (new Pemain(gamserver, ii));
012           pemain[ii].start();
013
014         }
015         gamserver.bandarGambreng();
017      }
018
019 }
020
```

Pada *class* Hompimpah, dibuat objek baru dari *class* Gambreng (baris 7), dimana dalam kelas tersebut dibuat 2 jenis *thread* yaitu thread bandar dan *thread* pemain. Masing-masing *thread* adalah *array* dari objek Semafor, yang berkapasitas JUMLAH_PEMAIN, yaitu 6. Kemudian kedua jenis *thread* di set *value*-nya sebagai 0 (terdapat pada *class* Gambreng). Hal ini mengindikasikan bahwa *thread* bandar dan pemain sedang tidur. Dalam kelas Gambreng, mutex sebagai objek dari *class* Semafor, di set 1,yang berarti bahwa *critical section* sedang dalam keadaan *available* atau tidak ada *thread* yang sedang memakai. Kemudian iterasiGambreng di set 0 dan pemanggilan *method* `resetGambreng()` pada *class* Gambreng sebagai penanda awal permainan

Pada baris ke 8 *class* Hompimpah dibuat *array* object baru dari *class* Thread yaitu pemain sebanyak JUMLAH_PEMAIN yang di set 6. Tiap-tiap *thread* memiliki `no_pemain` dan memiliki objek dari *class* Gambreng. Selanjutnya *thread- thread* tersebut menjalankan *method* `run()`-nya, dimana dalam *method* tersebut tiap-tiap *thread* pemain mengirimkan nomornya untuk diakses dalam *method* `pemainGambreng(int)` yang berada pada *class* Gambreng.

Contoh 27.3. method pemainGambreng

```
078      public void pemainGambreng (int nomor)
079      {
080        syncBandarPemain(nomor);
081        while(!menangGambreng())
082        {
083          mutex.kunci();
084
085          if((int)(Math.random()*2)==1)
086          {
087             truePemain=nomor;
088            trueCount++;
089
090          }
091          else
092          {
093            falsePemain=nomor;
094            falseCount++;
095          }
096          mutex.buka();
097          syncBandarPemain(nomor);
098        }
099      }
100      public void resetGambreng()
101      {
102        mutex.kunci();
103        adaPemenang=false;
104        truePemain=0;
105        trueCount=0;
106        falsePemain=0;
107        falseCount=0;
108
109        mutex.buka();
110      }
111
```

Pada fungsi tersebut tiap-tiap *thread* akan mengakses *critical section* secara bergantian, sesuai dengan prinsip *mutual exclusion*. Di dalam *critical section*, *thread-thread* akan dicek apakah truePemain(telapak) atau falsePemain(punggung) kemudian dihitung berapa jumlah telapak dan punggungnya. Proses ini akan terus me- *loop* sampai ada pemenang. Sebelum memasuki *critical section* dan melakukan pengecekan, sistem melakukan proses sinkronisasi bandar dan pemain, dengan memangil fungsi `syncBandarPemain()`

Contoh 27.4. syncBandarPemain

```
155      private void syncBandarPemain(int ii)
156      {
157        bandar[ii].buka();
158        pemain[ii].kunci();
159      }
160
```

dalam fungsi tersebut bandar melepaskan kunci, dan memanggil *thread* pemain dengan `notify()` yang merupakan bagian dari fungsi `buka()` dari *class* Semafor, untuk kemudian *thread* tersebut dapat mengakses *critical section*.

Setelah seluruh *thread* pemain mengakses *critical section* dan melakukan pengecekan, program memanggil fungsi `bandarGambreng()` pada *class* Gambreng,untuk melakukan proses sinkronisasi bandar yaitu memanggil fungsi `syncBandar()`

Contoh 27.5. syncBandar

```
141      private void syncBandar()
142      {
143       for(int ii=0;ii<jumlahPemain;ii++)
144       {
145         bandar[ii].kunci();
146       }
147      }
```

, sehingga bandar terkunci dalam *critical section* dan memegang kekuasaan dalam Gambreng untuk melakukan perulangan fungsi-fungsi berikut, yang masing-masing akan dipanggil dalam fungsi `bandarGambreng()`:

1. Me-reset permainan gambreng dengan memanggil fungsi `resetGambreng()`.

Contoh 27.6. resetGambreng

```
100      public void resetGambreng()
101      {
102        mutex.kunci();
103        adaPemenang=false;
104        truePemain=0;
105        trueCount=0;
106        falsePemain=0;
107        falseCount=0;
108
109        mutex.buka();
110      }
111
```

Dalam fungsi `resetGambreng()`, `truePemain`, `falsePemain`, `trueCount` dan `falseCount`= 0, kemudian ada pemenang diset false, sebagai tanda awal dari permainan.

2. Melakukan sinkronisasi pemain dan bandar dengan memanggil fungsi `syncPemainBandar()`.

Contoh 27.7. syncPemainBandar

```
132      private void syncPemainBandar()
133      {
134        for(int ii=0; ii<jumlahPemain;ii++)
135        {
136          pemain[ii].buka();
137          bandar[ii].kunci();
138        }
139      }
```

Ketika melakukan pemanggilan fungsi ini, seluruh pemain dibangunkan kemudian tiap-tiap bandar akan tidur. Dengan fungsi ini, bandar akan tetap menunggu sampai semua pemain membangunkannya. Dan ketika bandar bangun, pemain akan ditidurkan kembali.

3. Melakukan penghitungan dengan fungsi `hitungGambreng()`

Contoh 27.8. hitungGambreng

```
116      private void hitungGambreng()
117      {
118        mutex.kunci();
119
120        if(trueCount==1)
121        {
122          adaPemenang=true;
123          nomorPemenang=truePemain;
124        }
125        else(falseCount==1)
126        {
127          adaPemenang=true;
128          nomorPemenang=falsePemain;
129        }
130        mutex.buka();
131      }
```

Dengan fungsi ini, kembali melakukan pengaksesan variable. Dengan melakukan pengecekan, apabila salah satu diantara trueCount atau falseCount bernilai 1, maka adaPemenang. Nilai ini akan diproses oleh fungsi `menangGambreng()`, yang akan mengembalikan nilai boolean(true atau false). Jika fungsi tersebut bernilai *true* maka proses loop berhenti.

4. Menghitung iterasi perulangan, sampai fungsi `menangGambreng()` mengembalikan nilai *true*, yang menandakan adanya pemenang.

Saat proses perulangan berhenti, pemain melepaskan kunci kemudian sistem menampilkan output saat *execute*, berupa `no_pemain` yang keluar sebagai pemenang. Setiap proses execute akan menampilkan nomor pemain yang berbeda-beda , karena proses random dan urutan pengaksesan *critical section* oleh thread pada tiap peng- *execute*-an berbeda-beda.

Contoh 27.9. Keluaran Program

```
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[3] Jumlah Iterasi[12]
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[2] Jumlah Iterasi[1]
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[0] Jumlah Iterasi[6]
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[4] Jumlah Iterasi[2]
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[1] Jumlah Iterasi[14]
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[3] Jumlah Iterasi[9]
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[5] Jumlah Iterasi[3]
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[3] Jumlah Iterasi[2]
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[4] Jumlah Iterasi[5]
Nomor Peserta Pemain [0] - [5] Pemenang Pemain nomor[0] Jumlah Iterasi[11]
```

Contoh di atas menunjukkan hasil Hompimpah tidak selalu sama. Jumlah iterasi menunjukkan berapa kali bandar menyerukan gambreng dan mengulang permainan.

27.5. Rangkuman

Program Hompimpah merupakan ilustrasi dimana sebuah *thread* memegang kendali sinkronisasi thread lainnya. Seperti yang dijelaskan dalam program masing-masing dari pemain saling mengendalikan satu sama lain, dengan menggunakan alat sinkronisasi yang bernama semafor. Semafor dalam program adalah semafor buatan berupa *class* Semafor yang dibuat dalam bahasa Java. Adapun di dalamnya terdapat 2 fungsi yaitu fungsi `buka()` dan fungsi `kunci()` dengan fungsi-fungsi inilah masing-masing *thread* dapat mengendalikan satu sama lain

Rujukan

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

Daftar Rujukan Utama

[CC2001] 2001. *Computing Curricula 2001*. Computer Science Volume. ACM Council. IEEE-CS Board of Governors.

[Deitel2005] Harvey M Deitel dan Paul J Deitel. 2005. *Java How To Program*. Sixth Edition. Prentice Hall.

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

[HenPat2002] John L Hennessy dan David A Patterson. 2002. *Computer Architecture* . A Quantitative Approach . Third Edition. Morgan Kaufman. San Francisco.

[Hyde2003] Randall Hyde. 2003. *The Art of Assembly Language*. First Edition. No Strach Press.

[KennethRosen1999] Kenneth H Rosen. 1999. *Discrete Mathematics and Its Application*. McGraw Hill.

[KrutzVines2001] Ronald L Krutz dan Russell D Vines. 2001. *The CISSP Prep Guide Mastering the Ten Domains of Computer Security*. John Wiley & Sons.

[Kusuma2000] Sri Kusumadewi. 2000. *Sistem Operasi* . Edisi Dua. Graha Ilmu. Yogyakarta.

[Love2005] Robert Love. 2005. *Linux Kernel Development* . Second Edition. Novell Press.

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

[PeterDavie2000] Larry L Peterson dan Bruce S Davie. 2000. *Computer Networks A Systems Approach*. Second Edition. Morgan Kaufmann.

[SariYansen2005] Riri Fitri Sari dan Yansen. 2005. *Sistem Operasi Modern* . Edisi Pertama. Andi. Yogyakarta.

[Sidik2004] Betha Sidik. 2004. *Unix dan Linux*. Informatika. Bandung.

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

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

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

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

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

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

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

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

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

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

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

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

[WEBAmirSch2000] Yair Amir dan Theo Schlossnagle. 2000. *Operating Systems 00.418: Memory Management – http://www.cs.jhu.edu/ ~yairamir/ cs418/ os5/* . Diakses 29 Mei 2006.

[WEBArpaciD2005] Andrea C Arpaci-Dusseau dan Remzi H Arpaci-Dusseau. 2005. *CS 537: Introduction to Operating Systems – File System: User Perspective – http://www.cs.wisc.edu/ ~remzi/ Classes/537/ Fall2005/ Lectures/ lecture18.ppt* . Diakses 8 Juli 2006.

[WEBBabicLauria2005] G Babic dan Mario Lauria. 2005. *CSE 660: Introduction to Operating Systems – Files and Directories – http://www.cse.ohio-state.edu/ ~lauria/ cse660/ Cse660.Files.04-08-2005.pdf* . Diakses 8 Juli 2006.

[WEBBraam1998] Peter J Braam. 1998. *Linux Virtual File System – http://www.coda.cs.cmu.edu/ doc/ talks/ linuxvfs/* . Diakses 25 Juli 2006.

[WEBCACMF1961] John Fotheringham. " Dynamic Storage Allocation in the Atlas Computer Including an Automatic Use of a Backing Store – http://www.eecs.harvard.edu/ cs261/ papers/ frother61.pdf ". Diakses 29 Juni 2006. *Communications of the ACM* . 4. 10. October 1961.

[WEBCarter2004] John Carter. 2004. *CS 5460 Operating Systems – Lecture 19: File System Operations and Optimizations – http://www.cs.utah.edu/ classes/ cs5460/ lectures/ lecture19.pdf* . Diakses 29 Mei 2006.

[WEBChung2005] Jae Chung. 2005. *CS4513 Distributed Computer Systems – File Systems – http:// web.cs.wpi.edu/ ~goos/ Teach/ cs4513-d05/ slides/ fs1.ppt* . Diakses 7 Juli 2006.

[WEBCook2006] Tony Cook. 2006. *G53OPS Operating Systems – Directories – http://www.cs.nott.ac.uk/ ~acc/ g53ops/ lecture14.pdf* . Diakses 7 Juli 2006.

[WEBCornel2005] Cornel Computer Science Department. 2005. *Classic Sync Problems Monitors – http:// www.cs.cornell.edu/ Courses/ cs414/ 2005fa/ docs/ cs414-fa05-06-semaphores.pdf* . Diakses 13 Juni 2006.

[WEBDrake96] Donald G Drake. April 1996. *Introduction to Java threads – A quick tutorial on how to implement threads in Java – http://www.javaworld.com/ javaworld/ jw-04-1996/ jw-04-threads.html* . Diakses 29 Mei 2006.

[WEBEgui2006] Equi4 Software. 2006. *Memory Mapped Files – http://www.equi4.com/ mkmmf.html* . Diakses 3 Juli 2006.

[WEBFasilkom2003] Fakultas Ilmu Komputer Universitas Indonesia. 2003. *Sistem Terdistribusi – http:// telaga.cs.ui.ac.id/ WebKuliah/ sisdis2003/* . Diakses 29 Mei 2006.

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

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

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

[WEBFunkhouser2002] Thomas Funkhouser. 2002. *Computer Science 217 Introduction to Programming Systems: Memory Paging – http://www.cs.princeton.edu/ courses/ archive / spring02/ cs217/ lectures/ paging.pdf* . Diakses 28 Juni 2006.

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

[WEBGooch1999] Richard Gooch. 1999. *Overview of the Virtual File System – http://www.atnf.csiro.au/ people/ rgooch/ linux/ docs/ vfs.txt* . Diakses 29 Mei 2006.

[WEBGottlieb2000] Allan Gottlieb. 2000. *Operating Systems: Page tables – http://allan.ultra.nyu.edu/ ~gottlieb/ courses/ 1999-00-spring/ os/ lecture-11.html* . Diakses 28 Juni 2006.

[WEBHarris2003] Kenneth Harris. 2003. *Cooperation: Interprocess Communication – Concurrent Processing – http://people.cs.uchicago.edu/ ~mark/ 51081/ LabFAQ/ lab5/ IPC.html* . Diakses 2 Juni 2006.

[WEBHP1997] Hewlett-Packard Company. 1997. *HP-UX Memory Management – Overview of Demand Paging – http://docs.hp.com/ en/ 5965-4641/ ch01s10.html* . Diakses 29 Juni 2006.

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

[WEBIBMNY] IBM Corporation. NY. *General Programming Concepts – Writing and Debugging Programs – http://publib16.boulder.ibm.com/ pseries/ en_US/ aixprggd/ genprogc/ ls_sched_subr.htm* . Diakses 1 Juni 2006.

[WEBIBM1997] IBM Coorporation. 1997. *General Programming Concepts: Writing and Debugging Programs – Threads Scheduling http://www.unet.univie.ac.at/ aix/ aixprggd/ genprogc/ threads_sched.htm* . Diakses 1 Juni 2006.

[WEBIBM2003] IBM Corporation. 2003. *System Management Concepts: Operating System and Devices – http:// www.ncsa.uiuc.edu/ UserInfo/ Resources/ Hardware/ IBMp690 / IBM/ usr/ share/ man/ info/ en_US/ a_doc_lib/ aixbman/ admnconc/ mount_overview.htm* . Diakses 29 Mei 2006.

[WEBInfoHQ2002] InfoHQ. 2002. *Computer Maintenance Tips – http://www.infohq.com/ Computer/ computer_maintenance_tip.htm* . Diakses 11 Agustus 2006.

[WEBITCUV2006] IT& University of Virginia. 2006. *Mounting File Systems (Linux) – http:// www.itc.virginia.edu/ desktop/ linux/ mount.html* . Diakses 20 Juli 2006.

[WEBJeffay2005] Kevin Jeffay. 2005. *Secondary Storage Management – http://www.cs.unc.edu/ ~jeffay/ courses/ comp142/ notes/ 15-SecondaryStorage.pdf* . Diakses 7 Juli 2006.

[WEBJonesSmith2000] David Jones dan Stephen Smith. 2000. *85349 – Operating Systems – Study Guide – http:// www.infocom.cqu.edu.au/ Courses/ aut2001/ 85349/ Resources/ Study_Guide/ 85349.pdf* . Diakses 20 Juli 2006.

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

[WEBJupiter2004] Jupitermedia Corporation. 2004. *Virtual Memory – http://www.webopedia.com/ TERM/ v/ virtual_memory.html* . Diakses 29 Juni 2006.

[WEBKaram1999] Vijay Karamcheti. 1999. *Honors Operating Systems – Lecture 15: File Systems – http:// cs.nyu.edu/ courses/ spring99/ G22.3250-001/ lectures/ lect15.pdf* . Diakses 5 Juli 2006.

[WEBKessler2005] Christhope Kessler. 2005. *File System Interface – http://www.ida.liu.se/ ~TDDB72/ slides/ 2005/ c10.pdf* . Diakses 7 Juli 2006.

[WEBKozierok2005] Charles M Kozierok. 2005. *Reference Guide – Hard Disk Drives http:// www.storagereview.com/ guide/* . Diakses 9 Agustus 2006.

[WEBLee2000] Insup Lee. 2000. *CSE 380: Operating Systems – File Systems – http://www.cis.upenn.edu/ ~lee/ 00cse380/ lectures/ ln11b-fil.ppt* . Diakses 7 Juli 2006.

[WEBLindsey2003] Clark S Lindsey. 2003. *Physics Simulation with Java – Thread Scheduling and Priority – http://www.particle.kth.se/ ~fmi/ kurs/ PhysicsSimulation/ Lectures/ 10A/ schedulePriority.html* . Diakses 1 Juni 2006.

[WEBMassey2000] Massey University. May 2000. *Monitors & Critical Regions – http://www-ist.massey.ac.nz/ csnotes/ 355/ lectures/ monitors.pdf* . Diakses 29 Mei 2006.

[WEBMooreDrakos1999] Ross Moore dan Nikos Drakos. 1999. *Converse Programming Manual – Thread Scheduling Hooks – http://charm.cs.uiuc.edu/ manuals/ html/ converse/ 3_3Thread_ Scheduling_Hooks.html* . Diakses 1 Juni 2006.

[WEBOCWEmer2005] Joel Emer dan Massachusetts Institute of Technology. 2005. *OCW – Computer System Architecture – Fall 2005 – Virtual Memory Basics – http://ocw.mit.edu/ NR/ rdonlyres/ Electrical -Engineering -and -Computer -Science/ 6 -823Computer -System -ArchitectureSpring2002/ C63EC0D0 -0499 -474F -BCDA -A6868A6827C4/ 0/ lecture09.pdf* . Diakses 29 Juni 2006.

[WEBOSRLampson1983] Butler W Lampson. "Hints for Computer System Design – http:// research.microsoft.com/ copyright/ accept.asp? path=/~lampson/ 33-Hints/ Acrobat.pdf&pub=acm ". Diakses 10 Agustus 2006. *Operating Systems Review*. 15. 5. Oct 1983.

[WEBQuirke2004] Chris Quirke. 2004. *What is a Maintenance OS? – http://cquirke.mvps.org/ whatmos.htm* . Diakses 11 Agustus 2006.

[WEBRamam2005] B Ramamurthy. 2005. *File Management – http://www.cse.buffalo.edu/ faculty/ bina/ cse421/ spring2005/ FileSystemMar30.ppt* . Diakses 5 Juli 2006.

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

[WEBRegehr2002] John Regehr dan University of Utah. 2002. *CS 5460 Operating Systems – Demand Halamand Virtual Memory – http://www.cs.utah.edu/ classes/ cs5460-regehr/ lecs/ demand_paging.pdf* . Diakses 29 Juni 2006.

[WEBRobbins2003] Steven Robbins. 2003. *Starving Philosophers: Experimentation with Monitor Synchronization – http://vip.cs.utsa.edu/ nsf/ pubs/ starving/ starving.pdf* . Diakses 29 Mei 2006.

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

[WEBRustling1997] David A Rusling. 1997. *The Linux Kernel – The EXT2 Inode – http://www.science.unitn.it/ ~fiorella/ guidelinux/ tlk/ node96.html* . Diakses 1 Agustus 2006.

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

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

[WEBSamik2005a] Rahmat M Samik-Ibrahim. 2005. *IKI-20230 Sistem Operasi - Kumpulan Soal Ujian 2002-2005 – http://rms46.vlsm.org/ 1/ 94.pdf* . vLSM.org. Pamulang. Diakses 29 Mei 2006.

[WEBSchaklette2004] Mark Shacklette. 2004. *CSPP 51081 Unix Systems Programming: IPC – http:// people.cs.uchicago.edu/ ~mark/ 51081/ LabFAQ/ lab5/ IPC.html* . Diakses 29 Mei 2006.

[WEBSolomon2004] Marvin Solomon. 2004. *CS 537 Introduction to Operating Systems: Lecture Notes Part 7 – http://www.cs.wisc.edu/ ~solomon/ cs537/ paging.html* . Diakses 28 Juni 2006.

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

[WEBVolz2003] Richard A Volz. 2003. *Real Time Computing – Thread and Scheduling Basics – http:// linserver.cs.tamu.edu/ ~ravolz/ 456/ Chapter-3.pdf* . Diakses 1 Juni 2006.

[WEBWalton1996] Sean Walton. 1996. *Linux Threads Frequently Asked Questions (FAQ) – http://linas.org/ linux/ threads-faq.html* . Diakses 29 Mei 2006.

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

[WEBWirzOjaStafWe2004] Lars Wirzenius, Joanna Oja, dan StephenAlex StaffordWeeks. 2004. *The Linux System Administrator's Guide – The boot process in closer look http://www.tldp.org/ LDP/ sag/ html/ boot-process.html* . Diakses 7 Agustus 2006.

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

[WEBWiki2006a] From Wikipedia, the free encyclopedia. 2006. *Title – http://en.wikipedia.org/ wiki/ Zombie_process* . Diakses 2 Juni 2006.

[WEBWiki2006b] From Wikipedia, the free encyclopedia. 2006. *Atomicity– http://en.wikipedia.org/ wiki/ Atomicity* . Diakses 6 Juni 2006.

[WEBWiki2006c] From Wikipedia, the free encyclopedia. 2006. *Memory Management Unit – http:// en.wikipedia.org/ wiki/ Memory_management_unit* . Diakses 30 Juni 2006.

[WEBWiki2006d] From Wikipedia, the free encyclopedia. 2006. *Page Fault – http://en.wikipedia.org/ wiki/ Page_fault* . Diakses 30 Juni 2006.

[WEBWiki2006e] From Wikipedia, the free encyclopedia. 2006. *Copy on Write – http://en.wikipedia.org/ wiki/ Copy_on_Write* . Diakses 03 Juli 2006.

[WEBWiki2006f] From Wikipedia, the free encyclopedia. 2006. *Page replacement algorithms – http:// en.wikipedia.org/ wiki/ Page_replacement_algorithms* . Diakses 04 Juli 2006.

[WEBWiki2006g] From Wikipedia, the free encyclopedia. 2006. *File system – http://en.wikipedia.org/ wiki/ File_system* . Diakses 04 Juli 2006.

[WEBWiki2006h] From Wikipedia, the free encyclopedia. 2006. *Keydrive – http://en.wikipedia.org/ wiki/ Keydrive* . Diakses 09 Agustus 2006.

[WEBWiki2006i] From Wikipedia, the free encyclopedia. 2006. *Tape drive – http://en.wikipedia.org/ wiki/ Tape_drive* . Diakses 09 Agustus 2006.

[WEBWiki2006j] From Wikipedia, the free encyclopedia. 2006. *CD-ROM – http://en.wikipedia.org/ wiki/ CD-ROM* . Diakses 09 Agustus 2006.

[WEBWiki2006k] From Wikipedia, the free encyclopedia. 2006. *DVD – http://en.wikipedia.org/ wiki/ DVD* . Diakses 09 Agustus 2006.

[WEBWiki2006l] From Wikipedia, the free encyclopedia. 2006. *CD – http://en.wikipedia.org/ wiki/ CD* . Diakses 09 Agustus 2006.

[WEBWiki2006m] From Wikipedia, the free encyclopedia. 2006. *DVD-RW – http://en.wikipedia.org/ wiki/ DVD-RW* . Diakses 09 Agustus 2006.

[WEBWiki2006n] From Wikipedia, the free encyclopedia. 2006. *Magneto-optical drive – http://en.wikipedia.org/ wiki/ Magneto-optical_drive* . Diakses 09 Agustus 2006.

[WEBWiki2006o] From Wikipedia, the free encyclopedia. 2006. *Floppy disk – http://en.wikipedia.org/ wiki/ Floppy_disk* . Diakses 09 Agustus 2006.

Lampiran A. *GNU Free Documentation License*

Version 1.2, November 2002

A.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties ## for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

A.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. Aggregation with Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

A.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/ copyleft/ .

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. ADDENDUM

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME.
>
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Lampiran B. Kumpulan Soal Ujian Bagian Pertama

Berikut merupakan kumpulan soal Ujian Tengah Semester (UTS) dan Ujian Akhir Semester (UAS) antara 2003 dan 2008 untuk Mata Ajar IKI-20230/80230 Sistem Operasi, Fakultas Ilmu Komputer Universitas Indonesia. Waktu pengerjaan setiap soal [kecuali "Pasangan Konsep"] ialah 30 menit.

(2003-2007) Pasangan Konsep

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

a. *OS View: "Resource Allocator" vs. "Control Program"*.
b. *"Graceful Degradation" vs. "Fault Tolerant"*.
c. *Dual Mode Operation: "User Mode" vs. "Monitor Mode"*.
d. *Operating System Goal: "Convenient" vs. "Efficient"*.
e. *"System Components" vs. "System Calls"*.
f. *"Operating System Components" vs. "Operating System Services"*.
g. *"Symetric Multiprocessing" vs. "Asymetric Multiprocessing"*.
h. *"Distributed Systems' vs. "Clustered Systems"*.
i. *"Client Server System" vs. "Peer-to-peer system"*.
j. *"Microkernels" vs. "Virtual Machines"*.
k. *"Random Access Memory" vs. "Magnetic Disk"*.
l. *"Hard Real-time" vs "Soft Real-time"*.
m. *Job: "Batch system" vs. "Time-Sharing System"*.
n. *System Design: "Mechanism" vs. "Policy"*.
o. *Burst Cycle: "I/O Burst" vs. "CPU Burst"*.
p. *Process Bound: "I/O Bound" vs. "CPU Bound"*.
q. *"Process State" vs. "Process Control Block"*.
r. *"Waiting Time" vs. "Response Time"*.
s. *Process Type: "Lightweight" vs. "Heavyweight"*. .
t. *Multithread Model: "One to One" vs. "Many to Many"*.
u. *Scheduling Process: "Short Term" vs. "Long Term"*.
v. *Scheduling Algorithm: "FCFS (First Come First Serve)" vs. "SJF (Shortest Job First)"*.
w. *"Preemptive Shortest Job First" vs. "Non-preemptive Shortest Job First"*.
x. *Inter Process Communication: "Direct Communication" vs. "Indirect Communication"*.
y. *Process Synchronization: "Monitor" vs. "Semaphore"*.
z. *"Deadlock Avoidance" vs. "Deadlock Detection"*.
aa. *"Deadlock" vs. "Starvation"*.
ab. *Address Space: "Logical" vs. "Physical"*.
ac. *Dynamic Storage Allocation Strategy: "Best Fit" vs. "Worse Fit"*.
ad. *Virtual Memory Allocation Strategy: "Global" vs. "Local Replacement"*.
ae. *File Operations: "Deleting" vs. "Truncating"*.
af. *Storage System: "Volatile" vs. "Non-volatile"*.
ag. *File Allocation Methods: "Contiguous" vs. "Linked"*.
ah. *Disk Management: "Boot Block" vs. "Bad Block"*.
ai. *I/O Data-Transfer Mode: "Character" vs. "Block"*.
aj. *I/O Access Mode: "Sequential" vs. "Random"*.
ak. *I/O Transfer Schedulle: "Synchronous" vs. "Asynchronous"*.
al. *I/O Sharing: "Dedicated" vs. "Sharable"*. .
am *I/O direction: "Read only" vs. "Write only"*.
an. *"I/O Structure" vs. "Storage Structure"*.
ao. *Software License: "Free Software" vs. "Copyleft"*.

B.1. Konsep Dasar Perangkat Komputer

(B01-2005-01) Perangkat Lunak Bebas

a. Terangkan ke-empat (3+1) definisi Perangkat Lunak Bebas (PLB) menurut *Free Software Foundation* (*FSF*).
b. Terangkan perbedaan dan persamaan antara PLB dan *Open Source Software*.
c. Terangkan perbedaan dan persamaan antara PLB dan Perangkat Lunak "*Copyleft*".
d. Berikan contoh/ilustrasi Perangkat Lunak Bebas yang bukan "*Copyleft*".
e. Berikan contoh/ilustrasi Perangkat Lunak Bebas "*Copyleft*" yang bukan *GNU Public License*.

B.2. Konsep Dasar Sistem Operasi

(B02-2003-01) 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!

(B02-2004-01) Kernel Linux 2.6.X (=KL26) (2004)

a. Terangkan, apa yang dimaksud dengan Perangkat Lunak Bebas (PLB) yang berbasis lisensi GNU GPL (*General Public Licence*)!
b. KL26 diluncurkan Desember 2003. Terangkan mengapa hingga kini (Januari 2005), belum juga dibuka cabang pengembangan Kernel Linux versi 2.7.X!
c. KL26 lebih mendukung sistem berskala kecil seperti Mesin Cuci, Kamera, Ponsel, mau pun PDA. Terangkan, bagaimana kemampuan (*feature*) opsi tanpa MMU (*Memory Management Unit*) dapat mendukung sistem berskala kecil.
d. KL26 lebih mendukung sistem berskala sangat besar seperti "*Enterprise System*". Terangkan sekurangnya dua kemampuan (*feature*) agar dapat mendukung sistem berskala sangat besar.
e. KL26 lebih mendukung sistem interaktif seperti "*Work Station*". Terangkan sekurangnya satu kemampuan (*feature*) agar dapat mendukung sistem interaktif.

(B02-2005-01) Konsep Sistem Operasi

a. Terangkan/jabarkan sekurangnya empat komponen utama dari sebuah Sistem Operasi.
b. Terangkan/jabarkan peranan/pengaruh dari keempat komponen di atas terhadap sebuah Sistem Operasi Waktu Nyata (*Real Time System*).
c. Terangkan/jabarkan peranan/pengaruh dari keempat komponen di atas terhadap sebuah Sistem Prosesor Jamak (*Multi Processors System*).
d. Terangkan/jabarkan peranan/pengaruh dari keempat komponen di atas terhadap sebuah Sistem Operasi Terdistribusi (*Distributed System*).
e. Terangkan/jabarkan peranan/pengaruh dari keempat komponen di atas terhadap sebuah Sistem Operasi Telepon Seluler (*Cellular Phone*).

(B02-2007-01) *System Calls*

Antar-muka layanan Sistem Operasi tersedia melalui "*System Calls*". Sistem Operasi itu sendiri terdiri dari komponen manajer-manajer seperti "proses", "memori", "M/K", "sistem berkas", "jaringan", "keamanan", dan lain sebagainya. Berikan ilustrasi sebanyak 10 *system calls*, sekurangnya satu ilustrasi per manajer tersebut di atas.

B.3. Proses dan Penjadwalan

(B03-2002-01) Rancangan Sistem

Rancang sebuah sistem yang secara rata-rata:

- sanggup melayani secara bersamaan (*concurrent*) hingga 1000 pengguna (*users*).
- hanya 1% dari pengguna yang aktif mengetik pada suatu saat, sedangkan sisanya (99%) tidak mengerjakan apa-apa (*idle*).
- kecepatan mengetik 10 karakter per detik.
- setiap ketukan (ketik) menghasilkan *response CPU burst* dengan ukuran 10000 instruksi mesin.
- setiap instruksi mesin dijalankan dalam 2 (dua) buah siklus mesin (*machine cycle*).
- utilisasi CPU 100%.

a. Gambarkan GANTT chart dari proses-proses tersebut di atas. Lengkapi gambar dengan yang dimaksud dengan *burst time* dan *response time*!
b. Berapa lama, durasi sebuah *CPU burst* tersebut?
c. Berapa lama, kasus terbaik (*best case*) *response time* dari ketikan tersebut?
d. Berapa lama, kasus terburuk (*worse case*) *response time* dari ketikan tersebut?
e. Berapa MHz. *clock-rate CPU* pada kasus butir tersebut di atas?

(B03-2003-01) Tabel Proses I

Berikut merupakan sebagian dari keluaran menjalankan perintah "**top b n 1**" pada sebuah sistem GNU/Linux yaitu "**bunga.mhs.cs.ui.ac.id**" pada tanggal 10 Juni 2003 yang lalu.

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

  PID USER   PRI NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
    1 root     0  0   112   72    56 S     0.0  0.0   0:11 init
    2 root     0  0     0    0     0 SW    0.0  0.0   0:03 kflushd
    4 root     0  0     0    0     0 SW    0.0  0.0 156:14 kswapd
    ...
14953 root     0  0   596  308   236 S     0.0  0.2  19:12 sshd
31563 daemon   0  0   272  256   220 S     0.0  0.2   0:02 portmap
 1133 user1   18  0  2176 2176  1752 R     8.1  1.7   0:00 top
 1112 user1    0  0  2540 2492  2144 S     0.0  1.9   0:00 sshd
 1113 user1    7  0  2480 2480  2028 S     0.0  1.9   0:00 bash
30740 user2    0  0  2500 2440  2048 S     0.0  1.9   0:00 sshd
30741 user2    0  0  2456 2456  2024 S     0.0  1.9   0:00 bash
30953 user3    0  0  2500 2440  2072 S     0.0  1.9   0:00 sshd
30954 user3    0  0  2492 2492  2032 S     0.0  1.9   0:00 bash
 1109 user3    0  0  3840 3840  3132 S     0.0  3.0   0:01 pine
...
 1103 user8    0  0  2684 2684  1944 S     0.0  2.1   0:00 tin
```

a. Jam berapakah program tersebut di atas dijalankan?
b. Berapa waktu sebelumnya (dari tanggal 10 Juni tersebut), server "bunga.mhs.cs.ui.ac.id" terakhir kali *(re)boot*?
c. Apakah yang dimaksud dengan "*load average*"?
d. Sebutkan nama dari sebuah proses di atas yang statusnya "*running*"!
e. Sebutkan nama dari sebuah proses di atas yang statusnya "*waiting*"!

(B03-2003-02) Tabel Proses II

```
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
.....
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
```

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

a. Berapakah nomer *Process Identification* dari program "*top*" tersebut?
b. Siapakah yang mengeksekusi program "*top*" tersebut?
c. Sekitar jam berapakah, program tersebut dieksekusi?
d. Sudah berapa lama sistem GNU/Linux tersebut hidup/menyala?
e. Berapa pengguna yang sedang berada pada sistem tersebut?
f. Apakah yang dimaksud dengan "*load average*"?
g. Apakah yang dimaksud dengan proses "*zombie*" ?

(B03-2004-01) Tabel Proses III

Berikut merupakan sebagian dari keluaran hasil eksekusi perintah ``**top b n 1**" pada sebuah sistem GNU/Linux yaitu "**rmsbase.vlsm.org**" beberapa saat yang lalu.

a. Berapakah nomor *Process Identification* dari program "*top*" tersebut?
b. Sekitar jam berapakah, program tersebut dieksekusi?
c. Apakah yang dimaksud dengan proses "*nice*" ?
d. Dalam sistem Linux, "*process*" dan "*thread*" berbagi "*process table*" yang sama. Identifikasi/ tunjukkan (nomor *Process Identification*) dari salah satu *thread*. Terangkan alasannya!
e. Terangkan, mengapa sistem yang 46.6% idle dapat memiliki "*load average*" yang tinggi!

```
top - 17:31:56 up 10:14 min,  1 user,  load average: 8.64, 5.37, 2.57
Tasks:     95 total,     2 running, 93 sleeping,  0 stopped,   0 zombie
Cpu(s):  14.1% user,  35.7% system,   3.6% nice,  46.6% idle
Mem:    256712k total,   252540k used,     4172k free,    13772k buffers
Swap:   257032k total,     7024k used,   250008k free,   133132k cached
```

```
PID USER  PR NI  VIRT  RES  SHR S %CPU %MEM  TIME+  COMMAND
809 root  19 19  6780 6776 6400 S 42.2  2.6 1:02.47 rsync
709 root  20 19  6952 6952  660 R 29.3  2.7 1:46.72 rsync
710 root  19 19  6492 6484 6392 S  0.0  2.5 0:02.12 rsync
818 rms46 13  0   880  880  668 R  7.3  0.3 0:00.10 top
...
660 rms46  9  0  1220 1220  996 S  0.0  0.5 0:00.00 bash
661 rms46  9  0  1220 1220  996 S  0.0  0.5 0:00.01 bash
...
712 rms46  9  0  9256 9256 6068 S  0.0  3.6 0:06.82 evolution
781 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:02.59 evolution-mail
803 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:00.41 evolution-mail
804 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:00.00 evolution-mail
805 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:07.76 evolution-mail
806 rms46  9  0 16172  15m 7128 S  0.0  6.3 0:00.02 evolution-mail
766 rms46  9  0  5624 5624 4572 S  0.0  2.2 0:01.01 evolution-calen
771 rms46  9  0  4848 4848 3932 S  0.0  1.9 0:00.24 evolution-alarm
788 rms46  9  0  5544 5544 4516 S  0.0  2.2 0:00.55 evolution-addre
792 rms46  9  0  4608 4608 3740 S  0.0  1.8 0:01.08 evolution-execu
...
713 rms46  9  0 23580  23m  13m S  0.0  9.2 0:04.33 firefox-bin
763 rms46  9  0 23580  23m  13m S  0.0  9.2 0:00.57 firefox-bin
764 rms46  9  0 23580  23m  13m S  0.0  9.2 0:00.00 firefox-bin
796 rms46  9  0 23580  23m  13m S  0.0  9.2 0:00.18 firefox-bin
```

(B03-2006-02) Tabel Proses IV

Berikut merupakan keluaran dari menjalankan ``**top b n 1**" pada sebuah sistem GNU/Linux yaitu "**telaga.cs.ui.ac.id**" (beberapa baris dihapus):

a. Ada berapa CPU pada sistem tersebut di atas?
b. Siapakah *user name* yang menjalankan program top tersebut?
c. Berapakah nomor *user ID* dari yang menjalankan program *top* tersebut?
d. Berapakah nomor *process ID* dari program *top* tersebut?
e. Siapakah parent dari proses *top* tersebut? Sebutkan nama program dan PID-nya!
f. Siapakah *grand parent* dari proses *top* tersebut? Sebutkan nama program dan PID-nya!
g. Gambarkan bagan "*Process Tree*" dari semua program tersebut di atas. Asumsikan, init (PID=1) merupakan root, serta *parent* dari semua proses yang tidak tercantum *parent*-nya ialah PID=1.

```
top - 11:31:54 up 40 days,  2:04,  9 users,  load average: 0.25, 0.43, 0.35
Tasks: 198 total,   1 running, 197 sleeping,   0 stopped,   0 zombie
 Cpu0 :   1.2% user,   1.4% system,  61.6% nice,  35.8% idle
 Cpu1 :   3.2% user,   0.9% system,  61.8% nice,  34.1% idle
 Cpu2 :   4.4% user,   1.1% system,  62.0% nice,  32.5% idle
 Cpu3 :   2.2% user,   0.6% system,  62.2% nice,  35.0% idle
Mem:   1032692k total,  1005108k used,    27584k free,     9776k buffers
Swap:   506008k total,   180172k used,   325836k free,   675336k cached

  PID  PPID   UID  USER   GROUP  PR NI  S  %CPU TIME COMMAND
    1     0     0  root   root    0  0  S  0.0  0:43 init
 5147  5141  1411  user2  staff   9  0  S  0.0  0:00 sshd
 5148  5147  1411  user2  staff   9  0  S  0.0  0:00 bash
```

```
 5290  5286 51018  user1  staff   9  0  S  0.0  0:00 sshd
 5291  5290 51018  user1  staff   9  0  S  0.0  0:00 bash
 5822  5148  1411  user2  staff   9  0  S  0.0  0:00 pine
 5979 24964  1030  user3  staff   9  0  S  0.0  0:00 ssh
 6207  5291 51018  user1  staff   9  0  S  0.0  0:00 mutt
 6439 24580  1248  user5  staff  13  0  R  2.7  0:00 top
23142 26022  1762  user4  staff   9  0  S  0.0  0:01 pine
24577 24575  1248  user5  staff   9  0  S  0.0  0:01 sshd
24580 24577  1248  user5  staff   9  0  S  0.0  0:00 bash
24963 24959  1030  user3  staff  10  0  S  1.3  0:01 sshd
24964 24963  1030  user3  staff   9  0  S  0.0  0:00 bash
26021 26015  1762  user4  staff   9  0  S  0.0  0:01 sshd
26022 26021  1762  user4  staff   9  0  S  0.0  0:00 bash
```

(B03-2003-03) Status Proses I

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.

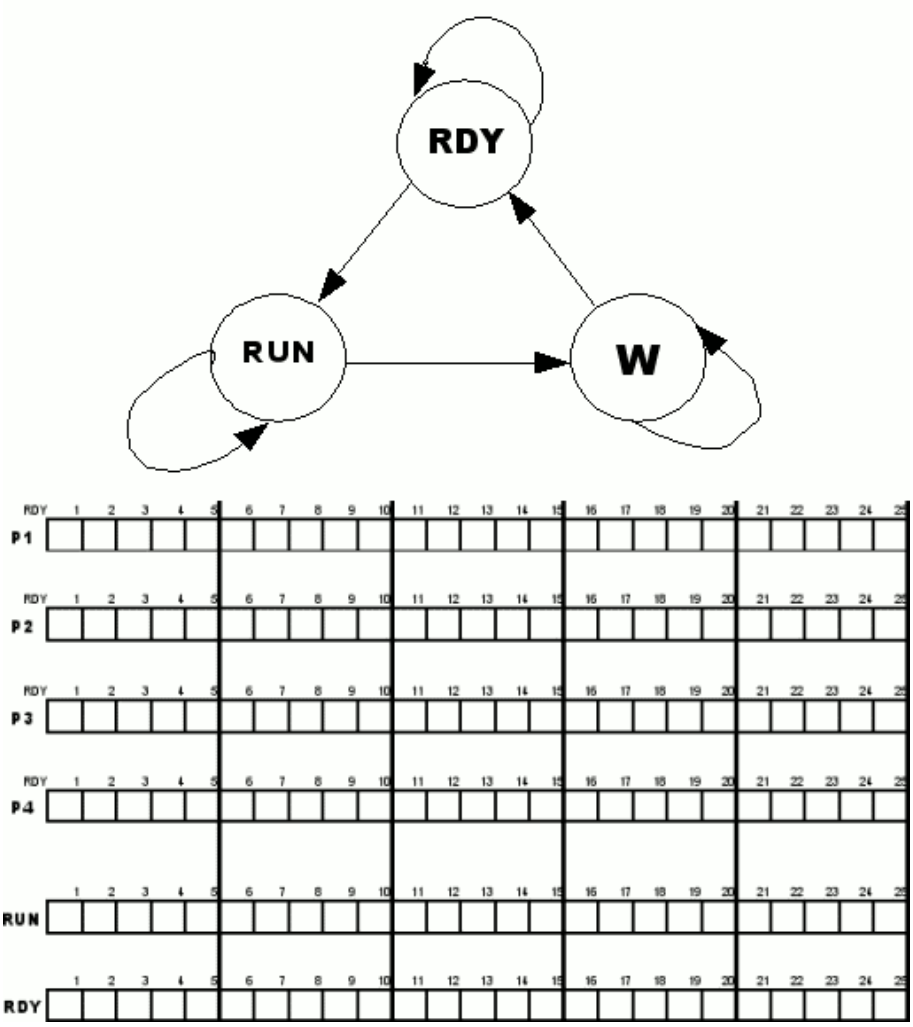
(B03-2005-01) Status Proses II

Diketahui empat proses (P_1, P_2, P_3, P_4) yang pada t_0 berada pada status "*RDY*" (*READY*). Pada satu saat, hanya satu proses yang boleh memiliki status "*RUN*". Status "*W*" (*Wait*) dan "*RDY*" dapat dimiliki beberapa proses setiap saat. Peralihan status proses dari "*RDY*" ke "*RUN*" diatur sebagai berikut:

- Prioritas diberikan kepada proses yang paling lama berada di "RDY" (bukan kumulatif).
- Proses yang tiba di "RDY" dapat langsung transit ke "RUN".
- Utamakan ID yang lebih kecil, jika proses-proses memiliki prioritas yang sama.

Pola *RUN*/*Wait* dari P_1 bergantian sebagai berikut: (3, 9, 3, 9, 3, 9, ...). Sedangkan pola *RUN*/*Wait*, berturut-turut: P_2 (2, 6, 2, 6, 2, 6, ...), P_3 (1, 6, 1, 6, 1, 6, ...), P_4 (1, 8, 1, 8, 1, 8, ...).

a. Gambarkan *Gantt Chart* selama 25 satuan waktu selanjut, untuk setiap Proses, serta status *RUN*/*CPU* dan *RDY*.
b. Berapa % utilitasi dari *CPU*?
c. Berapakah rata-rata *load RDY*?



(B03-2007-01) Status Proses III

Serupa dengan B03-2005-01 di atas, dengan $P_1 = P_2 = (1, 9, 1, 9, 1, 9, ...)$ sedangkan $P_3 = P_4 = (2, 8, 2, 8, 2, 8, ...)$.

(B03-2006-03) Status Proses IV

Berikut merupakan tabel utilitasi CPU terhadap derajat multi-program.

	I/O Wait	Derajat Multiprogram			
		1	2	3	4
Utilisasi CPU Total	80%	20%	36%	49%	59%
	20%	80%	96%	99.2%	99.8%
Utilisasi CPU per proses	80%	20%	18%	16%	15%
	20%	80%	48%	33.1%	25%

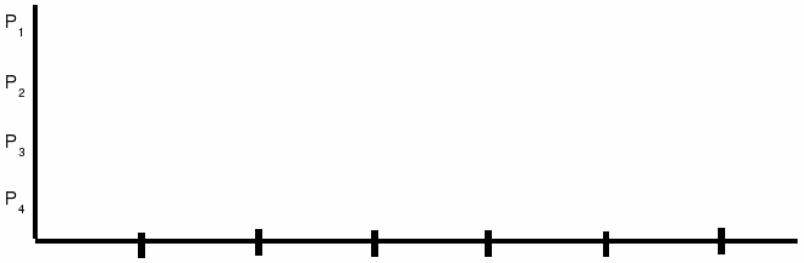
a. Gambarkan bagan "utilisasi CPU total" sebagai fungsi dari "derajat multiprogram". Gabungkan *I/O wait* 80% dan 20% dalam satu bagan!
b. Diketahui empat (4) proses – P1, P2, P3, P4 – start pada saat bersamaan dengan ``I/O Wait" 80%. ``Waktu CPU" keempat proses tersebut, berturut-turut 69, 49, 31, dan 15 detik. Gambarkan bagan waktu CPU masing-masing proses sebagai terhadap waktu.
c. Bandingkan "waktu total" dari P1 di atas dibandingkan dengan jika P1 dijalankan sendirian tanpa P2, P3, dan P4.

(B03-2007-02) Status Proses V

Diketahui proses jenis A dengan komponen *I/O Wait* sebesar 90%, serta proses jenis B dengan komponen *I/O Wait* sebesar 80%. Berikut merupakan tabel utilisasi CPU terhadap kombinasi multi-program A/B.

	A	**2A**	**B**	**2B**	**2A+2B**	**2A+B**	**A+2B**
CPU *idle*	90%	81%	80%	64%	52%	65%	58%
CPU *busy*	10%	19%	20%	36%	48%		
CPU/*procA*	10%	10%	-	-	8%		
CPU/*procB*	-	-	20%	18%	16%		

a. Lengkapi bagian tabel yang masih kosong.
b. Diketahui sebuah proses jenis A dengan jumlah waktu CPU = 12 detik. Berapa waktu total hingga proses tersebut selesai dieksekusi?
c. Diketahui sebuah proses jenis B dengan jumlah waktu CPU = 12 detik. Berapa waktu total hingga proses tersebut selesai dieksekusi?
d. Diketahui dua proses jenis A dan dua proses jenis B secara serentak mulai dieksekusi. Jumlah waktu CPU masing-masing proses ialah 12 detik. Berapa waktu total hingga proses terakhir selesai dieksekusi?
e. Buatkan diagram waktu dari butir d di atas.



(B03-2007-03) Status Proses VI

Diketahui empat proses, P1(0:6), P2(5:4), P3(10:4), P4(15:2.7); [Pn(A:B); n=nomor proses; A=waktu start; B=waktu CPU] dengan tabel utilitasi CPU - derajat multi-program berikut:

I/O Wait 60%	**Derajat Multiprogram**			
	1	**2**	**3**	**4**
Utilisasi CPU Total	40%	64%	78%	88%
Utilisasi CPU per proses	40%	32%	26%	22%

Gambarkan bagan masing-masing proses terhadap waktu.

(B03-2005-02) `Fork()` I

Silakan menelusuri program C berikut ini. Diasumsikan bahwa PID dari program tersebut (baris 17) ialah 5000, serta tidak ada proses lain yang terbentuk kecuali dari `fork()` program ini.

a. Tuliskan keluaran dari program tersebut.
b. Ubahlah MAXLEVEL (baris 04) menjadi "5"; lalu kompail ulang dan jalankan kembali! Tuliskan bagian keluaran dari modifikasi program tersebut.
c. Jelaskan asumsi pemilihan PID pada butir "b" di atas!

```
01 #include <sys/types.h>
02 #include <stdio.h>
03 #include <unistd.h>
04 #define  MAXLEVEL 4
```

```
06 char*  turunan[]=
07        {"", "pertama","kedua","ketiga","keempat","kelima"};
08 main()
09 {
10    int   idx     = 1;
11    int   putaran = 0;
12    int   deret0  = 0;
13    int   deret1  = 1;
14    int   tmp;
15    pid_t pid;
16
17    printf("PID INDUK %d\n", (int) getpid());
18    printf("START deret Fibonacci... ");
      printf(" %d... %d...\n", deret0, deret1);
20    while (putaran < MAXLEVEL)
21    {
22       tmp=deret0+deret1;
23       deret0=deret1;
24       deret1=tmp;
25
26       pid = fork();             /* FORK     */
28       if (pid > 0)              /* Induk?   */
29       {
30          wait(NULL);
31          printf("INDUK %s selesai menunggu ", turunan[idx]);
32          printf("PID %d...\n", (int) pid);
33          putaran++;
34       } else if (pid==0) {      /* Turunan? */
35          printf("Deret Fibonacci selanjutnya...");
     printf(" %d...\n", deret1);
36          idx++;
37          exit (0);
38       } else {                  /* Error?   */
39          printf("Error...\n");
40          exit (1);
41       }
42    };
43    exit (0);
44 }
```

(B03-2005-03) `Fork()` II

Silakan menelusuri program "*multifork*" berbahasa C berikut ini. Diasumsikan bahwa *PID* dari program tersebut (baris 14) ialah 5000, serta tidak ada proses lain yang terbentuk kecuali dari `fork()` program ini. Tuliskan keluaran dari program tersebut!

```
002 /* multifork (c) 2005 Rahmat M. Samik-Ibrahim, GPL-like */
003 /*****************************************************/
005 #include <sys/types.h>
006 #include <stdio.h>
007 #include <unistd.h>
009 /********************************************* main ***/
010 main()
011 {
012    pid_t pid1, pid2, pid3;
013
```

```
014    printf("PID_INDUK   ******* %5.5d  ******* ******\n",
015                                       (int) getpid());
017    pid1 = fork();
018    wait(NULL);
019    pid2 = fork();
020    wait(NULL);
021    pid3 = fork();
022    wait(NULL);
023    printf("PID1(%5.5d) -- PID2(%5.5d) -- PID3(%5.5d)\n",
024                      (int) pid1, (int) pid2, (int) pid3);
025 }
027 /*********************************************************/
```

(B03-2006-01) Fork() III

Diketahui, yang PID berikut ini ialah 5000. Tuliskan keluaran dari program tersebut!

```
001  /*************************************************************/
002  /* qforkng (c) 2006 Rahmat M. Samik-Ibrahim, GPL-like        */
003  /*                                                           */
004  /* Suppose its process ID (PID) is 5000                      */
005  /* Write down the output of this process!                    */
006  /*************************************************************/
007
008  #include <stdio.h>
009  #include <stdlib.h>
010
011  /****************************************************** main ***/
012  main()
013  {
014     pid_t pid1, pid2, pid3, pid4;
015
016     pid1 = getpid();        /* getpid: get my current PID       */
017     pid2 = fork();          /* fork:   clone parent -> child    */
018     wait(NULL);             /* wait:   wait until child is done */
019     pid3 = fork();
020     wait(NULL);
021     pid4 = getpid();
022     printf("PID1[%5.5d] PID2[%5.5d] PID3[%5.5d] PID4[%5.5d]\n",
023        (int) pid1, (int) pid2, (int) pid3, (int) pid4 );
024  }
025
026  /*************************************************************/
```

(B03-2007-04) Fork() IV

Bagaimana keluaran dari program "isengfork" pada halaman berikut?

```
01 /*****************************************************/
02 /* isengfork (c) 2007 Rahmat M. Samik-Ibrahim, GPL-like */
03 /*****************************************************/
04 #include <sys/types.h>
05 #include <stdio.h>
```

```
06 #include <unistd.h>
08 main()
09 {
10    int ii=0;
11    if (fork() == 0) ii++;
12    wait(NULL);
13    if (fork() == 0) ii++;
14    wait(NULL);
15    if (fork() == 0) ii++;
16    wait(NULL);
17    printf ("Result = %3.3d \n",ii);
18 }
19 / ******************************************************/
```

(B03-2001-01) Penjadwalan Proses I

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

- P_1(0,9)
- P_2(2,7)
- P_3(4,1)
- P_4(6,3)
- P_5(8,2)

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

a. Berapakah rata-rata *turn-around time* dan *waiting time* dari kelima proses tersebut, jika diimplementasikan dengan algoritma penjadwalan FCFS (*First Come, First Served*)?
b. Bandingkan *turnaround time* dan *waiting time* tersebut, dengan sebuah algoritma penjadwalan dengan ketentuan sebagai berikut:
   - *Pre-emptive*: pergantian proses dapat dilakukan kapan saja, jika ada proses lain yang memenuhi syarat. Namun durasi setiap proses dijamin minimum dua (2) satuan waktu, sebelum boleh diganti.
   - Waktu alih (*switch-time*) sama dengan di atas, yaitu sebesar satu (1) satuan waktu (*unit time*).
   - Jika proses telah menunggu >= 15 satuan waktu:
     - dahulukan proses yang telah menunggu paling lama
     - lainnya: dahulukan proses yang menunggu paling sebentar.
   - Jika kriteria yang terjadi seri: dahulukan proses dengan nomor urut yang lebih kecil (umpama: P_1 akan didahulukan dari P_2).

(B03-2002-02) Penjadwalan Proses II

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 penjadwalan 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.

(B03-2004-02) Penjadwalan Proses III

Diketahui tiga (3) proses *preemptive* dengan nama berturut-turut P_1(0), P_2(2), dan P_3(4). Angka dalam kurung menunjukkan waktu tiba ("*arrival time*"). Ketiga proses tersebut memiliki *burst time* yang sama yaitu 4 satuan waktu (*unit time*). Setiap memulai/peralihan proses, selalu diperlukan waktu-alih (*switch time*) sebesar satu (1) satuan waktu.

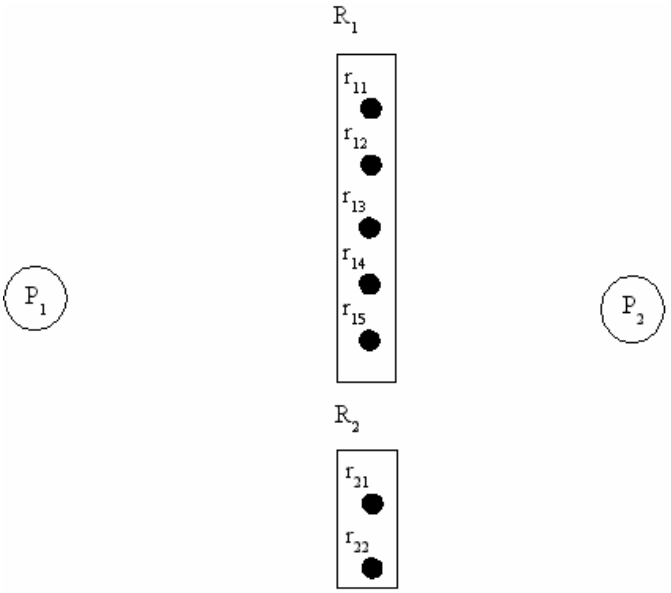
Berapakah rata-rata *turn-around time* dan *waiting time* dari ketiga proses tersebut, jika diimplementasikan dengan algoritma penjadwalan:
- *Shortest Waiting First*: mendahulukan proses dengan waiting time terendah.
- *Longest Waiting First*: mendahulukan proses dengan waiting time tertinggi.

Jika kriteria penjadwalan seri, dahulukan proses dengan nomor urut yang lebih kecil (umpama: P_1 akan didahulukan dari P_2). Jangan lupa membuat *Gantt Chart*-nya!

B.4. Proses dan Sinkronisasi

(B04-2003-01) *Deadlock* I

Gambarkan graf pada urutan T_0, T_1,... dan seterusnya, hingga semua permintaan sumber-daya terpenuhi dan dikembalikan. Sebutkan, jika terjadi kondisi "*unsafe*"!



Diketahui:
- set P yang terdiri dari dua (2) proses; P = { P_1, P_2 }.
- set R yang terdiri dari dua (2) sumber-daya (*resources*); dengan berturut-turut lima (5) dan dua (2) *instances*; R = { R_1, R_2 } = { {r_{11}, r_{12}, r_{13}, r_{14}, r_{15} }, {r_{21}, r_{22} } }.
- Plafon (jatah maksimum) sumber-daya untuk masing-masing proses ialah:

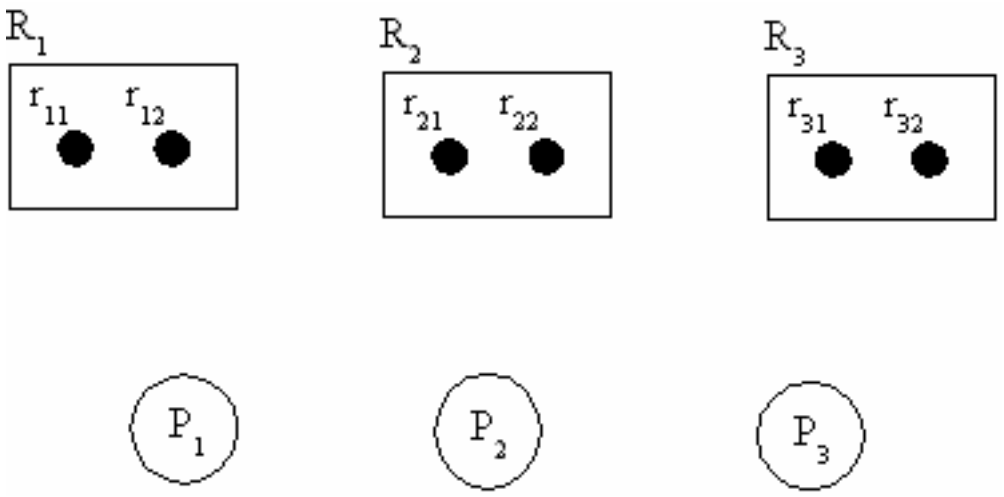
	r_1	**r_2**
p_1	5	1
p_2	3	1

- Pencegahan *deadlock* dilakukan dengan *Banker's Algorithm*.
- Alokasi sumber-daya yang memenuhi kriteria *Banker's Algorithm* di atas, akan diprioritaskan pada proses dengan indeks yang lebih kecil.
- Setelah mendapatkan semua sumber-daya yang diminta, proses akan mengembalikan SELURUH sumber-daya tersebut.
- Pada saat T_0, "Teralokasi" serta "Permintaan" sumber-daya proses ditentukan sebagai berikut:

	TERALOKASI		**PERMINTAAN**	
	R_1	**R_2**	**R_1**	**R_2**
p_1	2	0	2	1
p_2	2	0	1	1

(B04-2003-02) *Deadlock* II

Diketahui:
- set P yang terdiri dari tiga (3) proses; P = { P_1, P_2, P_3 }.
- set R yang terdiri dari tiga (3) sumber-daya (*resources*); masing-masing terdiri dari dua (2) instan (*instances*); R = { R_1, R_2, R_3 } = { {r_{11}, r_{12} }, {r_{21}, r_{22} }, {r_{31}, r_{32} } }.
- Prioritas alokasi sumber-daya akan diberikan pada proses dengan indeks yang lebih kecil.
- Jika tersedia: permintaan alokasi sumber-daya pada T_N akan dipenuhi pada urutan berikutnya T_{N+1}.
- Proses yang telah dipenuhi semua permintaan sumber-daya pada T_M ; akan melepaskan semua sumber-daya tersebut pada urutan berikutnya T_{M+1}.
- Pencegahan *deadlock* dilakukan dengan menghindari *circular wait*.
- Pada saat T_0, set E_0 = { } (atau kosong), sehingga gambar graf-nya sebagai berikut:



Jika set E pada saat T_1 menjadi: E_1 = { P_1 –> R_1, P_1 –> R_2, P_2 –> R_1, P_2 –> R_2, P_3 –> R_1, P_3 –> R_2, P_3 –> R_3 }, gambarkan graf pada urutan T_1, T_2,... serta (E_2, E_3, ...) berikutnya hingga semua permintaan sumber-daya terpenuhi dan dikembalikan.

(B04-2005-01) *Deadlock* III

a. Terangkan/jabarkan secara singkat, keempat kondisi yang harus dipenuhi agar terjadi *Deadlock*! Gunakan graf untuk menggambarkan keempat kondisi tersebut!
b. Terangkan/jabarkan secara singkat, apakah akan selalu terjadi *Deadlock* jika keempat kondisi tersebut dipenuhi?!

(B04-2007-01) *Deadlock* IV

Perhatikan bagian Bagian B.4, “(B04-2007-02) Jembatan ” di bawah.
a. Berikan sebuah ilustrasi, bagaimana *deadlock* dapat terjadi.
b. Berikan sebuah ilustrasi, bagaimana stravation tanpa *deadlock* dapat terjadi.
c. Terangkan bagaimana di atas *deadlock* dapat dicegah.

(B04-2007-03) *Deadlock* V

a. Sebutkan cara yang lazim dilakukan sistem operasi dewasa ini (GNU/Linux dan MS Windows) untuk menanggulangi *deadlocks*?
b. Jelaskan mengapa cara butir a di atas merupakan cara yang paling lazim!
c. Sebutkan apa perbedaan antara *deadlock avoidance* dan *deadlock prevention*!
d. Berikut ilustrasi/contoh butir c di atas!

(B04-2001-01) 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.

```
001 // Gabungan ReaderWriterServer.java Reader.java Writer.java
002 //          Semaphore.java Database.java
003 // (c) 2000 Gagne, Galvin, Silberschatz
004
005 public class ReaderWriterServer {
006    public static void main(String args[]) {
007       Database server = new Database();
008       Reader[] readerArray = new Reader[NUM_OF_READERS];
009       Writer[] writerArray = new Writer[NUM_OF_WRITERS];
010       for (int i = 0; i < NUM_OF_READERS; i++) {
011          readerArray[i] = new Reader(i, server);
012          readerArray[i].start();
013       }
014       for (int i = 0; i < NUM_OF_WRITERS; i++) {
015          writerArray[i] = new Writer(i, server);
016          writerArray[i].start();
017       }
018    }
019    private static final int NUM_OF_READERS = 3;
020    private static final int NUM_OF_WRITERS = 2;
021 }
022
023 class Reader extends Thread {
024    public Reader(int r, Database db) {
025       readerNum = r;
026       server = db;
027    }
028    public void run() {
029      int c;
030      while (true) {
031        Database.napping();
032        System.out.println("reader " + readerNum
     + " wants to read.");
033        c = server.startRead();
034        System.out.println("reader " + readerNum +
035             " is reading. Reader Count = " + c);
036        Database.napping();
037        System.out.print("reader " + readerNum +
  " is done reading. ");
038        c = server.endRead();
039       }
040    }
041    private Database  server;
042    private int       readerNum;
043 }
045 class Writer extends Thread {
```

```
046    public Writer(int w, Database db) {
047       writerNum = w;
048       server = db;
049    }
050    public void run() {
051      while (true) {
052        System.out.println("writer " + writerNum + " is sleeping.");
053        Database.napping();
054        System.out.println("writer " + writerNum + " wants to write.");
055        server.startWrite();
056        System.out.println("writer " + writerNum + " is writing.");
057        Database.napping();
058        System.out.println("writer " + writerNum + " is done writing.");
059        server.endWrite();
060       }
061    }
062    private Database  server;
063    private int       writerNum;
064 }
065
066 final class Semaphore {
067    public Semaphore() {
068       value = 0;
069    }
070    public Semaphore(int v) {
071       value = v;
072    }
073    public synchronized void P() {
074       while (value <= 0) {
075          try { wait(); }
076          catch (InterruptedException e) { }
077       }
078       value--;
079    }
080    public synchronized void V() {
081       ++value;
082       notify();
083    }
084    private int value;
085 }
086
087 class Database {
088    public Database() {
089       readerCount = 0;
090       mutex = new Semaphore(1);
091       db = new Semaphore(1);
092    }
093    public static void napping() {
094      int sleepTime = (int) (NAP_TIME * Math.random() );
095      try { Thread.sleep(sleepTime*1000); }
096      catch(InterruptedException e) {}
097    }
098    public int startRead() {
099       mutex.P();
100       ++readerCount;
101       if (readerCount == 1) {
102          db.P();
103       }
```

```
104       mutex.V();
105       return readerCount;
106    }
107    public int endRead() {
108       mutex.P();
109       --readerCount;
110       if (readerCount == 0) {
111          db.V();;
112       }
113       mutex.V();
114       System.out.println("Reader count = " + readerCount);
115       return readerCount;
116    }
117    public void startWrite() {
118       db.P();
119    }
120    public void endWrite() {
121       db.V();
122    }
123    private int readerCount;
124    Semaphore mutex;
125    Semaphore db;
126    private static final int NAP_TIME = 15;
127 }
128
129 // The Class java.lang.Thread
130 // When a thread is created, it is not yet active; it begins
131 // to run when method start is called. Invoking the start
132 // method causes this thread to begin execution; by calling
    // the run method.
133 // public class Thread implements Runnable {
134 //     ...
135 //     public void run();
136 //     public void start()
137 //        throws IllegalThreadStateException;
138 //     ...
139 // }
```

(B04-2002-01) Problem *Reader/Writer* II

Perhatikan berkas "`ReaderWriterServer.java`" pada soal yang lalu, yang merupakan gabungan berbagai berkas seperti "`ReaderWriterServer.java`", "`Reader.java`", "`Writer.java`", "`Semaphore.java`", "`Database.java`", oleh Gagne, Galvin, dan Silberschatz. Terangkan berdasarkan berkas tersebut:

a. akan terbentuk berapa *thread*, jika menjalankan program *class* "`ReaderWriterServer`" ini? Apa yang membedakan antara sebuah *thread*, dengan *thread*, lainnya?
b. mengapa: jika ada "`Reader`" yang sedang membaca, tidak ada "`Writer`" yang dapat menulis; dan mengapa: jika ada "`Writer`" yang sedang menulis, tidak ada "`Reader`" yang dapat membaca?
c. mengapa: jika ada "`Reader`" yang sedang membaca, boleh ada "`Reader`" lainnya yang turut membaca?
d. modifikasi kode program tersebut (cukup mengubah baris terkait), sehingga akan terdapat 5 (lima) "`Reader`" dan 4 (empat) "`Writer`"!

Modifikasi kode program tersebut (cukup mengubah `method` terkait), sehingga pada saat **RAJA** (Reader 0) ingin membaca, tidak boleh ada **RAKYAT** (`Reader` lainnya) yang sedang/akan membaca. **JANGAN MEMPERSULIT DIRI SENDIRI**: jika **RAJA** sedang membaca, **RAKYAT** boleh turut membaca.

(B04-2004-01) Problem *Reader/Writer* III

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`.

```
001 /*************************************************************
002  * Gabungan/Modif: Factory.java Database.java RWLock.java
003  * Reader.java Semaphore.java SleepUtilities.java Writer.java
004  * Operating System Concepts with Java - Sixth Edition
005  * Gagne, Galvin, Silberschatz Copyright John Wiley &amp;amp; Sons-2003.
006  */
007
008 public class Factory
009 {
010     public static void main(String args[])
011     {
012         System.out.println("INIT Thread...");
013         Database server   = new Database();
014         Thread   readerX  = new Thread(new Reader(server));
015         Thread   writerX  = new Thread(new Writer(server));
016         readerX.start();
017         writerX.start();
018         System.out.println("Wait...");
019     }
020 }
022 // Reader // ************************************************
023 class Reader implements Runnable
024 {
025     public Reader(Database db) { server = db; }
026
027     public void run() {
028         while (--readercounter > 0)
029          {
030             SleepUtilities.nap();
031             System.out.println("readerX: wants to read.");
032             server.acquireReadLock();
033             System.out.println("readerX: is reading.");
034             SleepUtilities.nap();
035             server.releaseReadLock();
036             System.out.println("readerX: done...");
037         }
038     }
039
040     private Database server;
041     private int      readercounter = 3;
042 }
043
044 // Writer // ************************************************
045 class Writer implements Runnable
046 {
047     public Writer(Database db) { server = db; }
049     public void run() {
```

```
050         while (writercounter-- > 0)
051         {
052            SleepUtilities.nap();
053             System.out.println("writerX: wants to write.");
054             server.acquireWriteLock();
055             System.out.println("writerX: is writing.");
056              SleepUtilities.nap();
057             server.releaseWriteLock();
058             System.out.println("writerX: done...");
059         }
060     }
062     private Database server;
063     private int      writercounter = 3;
064 }
065 // Semaphore // ************************************************
066 class Semaphore
067 {
068     public Semaphore()        { value = 0; }
069     public Semaphore(int val) { value = val; }
070     public synchronized void acquire() {
071         while (value == 0) {
072             try { wait(); }
073             catch (InterruptedException e) { }
074         }
075         value--;
076     }
077
078     public synchronized void release() {
079         ++value;
080         notifyAll();
081     }
082     private int value;
083 }
084
085 // SleepUtilities // ********************************************
086 class SleepUtilities
087 {
088     public static void nap() { nap(NAP_TIME); }
089
090     public static void nap(int duration) {
091         int sleeptime = (int) (duration * Math.random() );
092           try { Thread.sleep(sleeptime*1000); }
093         catch (InterruptedException e) {}
094     }
095     private static final int NAP_TIME = 3;
096 }
098 // Database // **************************************************
099 class Database implements RWLock
100 {
101     public Database()              { db = new Semaphore(1); }
102     public void acquireReadLock()  { db.acquire(); }
103     public void releaseReadLock()  { db.release(); }
104     public void acquireWriteLock() { db.acquire(); }
105     public void releaseWriteLock() { db.release(); }
106     Semaphore                      db;
107 }
108 // An interface for reader-writer locks. // *********************
109 interface RWLock
```

```
110 {
111     public abstract void acquireReadLock();
112     public abstract void releaseReadLock();
113     public abstract void acquireWriteLock();
114     public abstract void releaseWriteLock();
115 }
```

(B04-2003-03) *Bounded Buffer*

Perhatikan berkas "`BoundedBufferServer.java`" pada halaman berikut.

a. Berapakah ukuran penyangga (*buffer*)?
b. Modifikasi program (sebutkan nomor barisnya) agar ukuran penyangga menjadi 6 (enam).
c. Tuliskan/perkirakan keluaran (*output*) 10 baris pertama, jika menjalankan program ini.
d. Jelaskan fungsi dari ketiga *semaphore* (*mutex*, *full*, *empty*) pada program tersebut.
e. Tambahkan (sebutkan nomor barisnya) sebuah *thread* dari *class* Supervisor yang berfungsi:
   i. pada awal dijalankan, melaporkan ukuran penyangga.
   ii. secara berkala (acak), melaporkan jumlah pesan (*message*) yang berada dalam penyangga.
f. *Semaphore* mana yang paling relevan untuk modifikasi butir "e" di atas?

```
001 // Authors: Greg Gagne, Peter Galvin, Avi Silberschatz
002 // Slightly Modified by: Rahmat M. Samik-Ibrahim
003 // Copyright 2000 by Greg Gagne, Peter Galvin, Avi Silberschatz
004 // Applied Operating Systems Concepts-John Wiley &amp;amp; Sons, Inc.
005 //
006 // Class "Date":
007 //     Allocates a Date object and initializes it so that
008 //     it represents the time at which it was allocated,
009 //     (E.g.): "Wed Apr 09 11:12:34 JAVT 2003"
010 // Class "Object"/ method "notify":
011 //     Wakes up a single thread that is waiting on this
    //     object's monitor.
012 // Class "Thread"/ method "start":
013 //     Begins the thread execution and calls the run method
    //     of the thread.
014 // Class "Thread"/ method "run":
015 //     The Runnable object's run method is called.
016
017 import java.util.*;
018 // main ****************************************************
019 public class BoundedBufferServer
020 {
021     public static void main(String args[])
022     {
023         BoundedBuffer server         = new BoundedBuffer();
024         Producer      producerThread = new Producer(server);
025         Consumer      consumerThread = new Consumer(server);
026         producerThread.start();
027         consumerThread.start();
028     }
029 }
030
031 // Producer ************************************************
032 class Producer extends Thread
033 {
034     public Producer(BoundedBuffer b)
```

```
035     {
036          buffer = b;
037     }
038
039     public void run()
040     {
041         Date message;
042         while (true)
043         {
044             BoundedBuffer.napping();
045
046             message = new Date();
047             System.out.println("P: PRODUCE  " + message);
048             buffer.enter(message);
049         }
050     }
051     private  BoundedBuffer buffer;
052 }
053
054 // Consumer ******************************************************
055 class Consumer extends Thread
056 {
057    public Consumer(BoundedBuffer b)
058    {
059       buffer = b;
060    }
061    public void run()
062    {
063          Date message;
064          while (true)
065          {
066              BoundedBuffer.napping();
067              System.out.println("C: CONSUME  START");
068              message = (Date)buffer.remove();
069          }
070     }
071     private  BoundedBuffer buffer;
072 }
074 // BoundedBuffer.java ********************************************
075 class BoundedBuffer
076 {
077    public BoundedBuffer()
078    {
079         count  = 0;
080         in     = 0;
081         out    = 0;
082         buffer = new Object[BUFFER_SIZE];
083         mutex  = new Semaphore(1);
084         empty  = new Semaphore(BUFFER_SIZE);
085         full   = new Semaphore(0);
086    }
087    public static void napping()
088    {
089         int sleepTime = (int) (NAP_TIME * Math.random() );
090         try { Thread.sleep(sleepTime*1000); }
091         catch(InterruptedException e) { }
092    }
093    public void enter(Object item)
```

```
094    {
095         empty.P();
096         mutex.P();
097         ++count;
098         buffer[in] = item;
099         in = (in + 1) % BUFFER_SIZE;
100         System.out.println("P: ENTER    " + item);
101         mutex.V();
102         full.V();
103    }
104    public Object remove()
105    {
106         Object item;
107         full.P();
108         mutex.P();
109         --count;
110         item = buffer[out];
111         out = (out + 1) % BUFFER_SIZE;
112         System.out.println("C: CONSUMED " + item);
113         mutex.V();
114         empty.V();
115         return item;
116     }
117     public  static final int NAP_TIME    = 5;
118     private static final int BUFFER_SIZE = 3;
119     private Semaphore        mutex;
120     private Semaphore        empty;
121     private Semaphore        full;
122     private int              count, in, out;
123     private Object[]         buffer;
124 }
125
126 // Semaphore.java ************************************************
128 final class Semaphore
129 {
130     public Semaphore()
131     {
132         value = 0;
133     }
134     public Semaphore(int v)
135     {
136        value = v;
137     }
138     public synchronized void P()
139     {
140         while (value <= 0)
141         {
142             try { wait(); }
143             catch (InterruptedException e) { }
144         }
145         value --;
146     }
147     public synchronized void V()
148     {
149         ++value;
150         notify();
151     }
152     private int value;
```

```
153 }
```

(B04-2005-02) Sinkronisasi I

a. Terangkan peranan/fungsi dari semafor-semafor pada program Java berikut ini!
b. Tuliskan keluaran dari program tersebut!
c. Modifikasi program (baris mana?), agar object proses dengan index tinggi mendapat prioritas didahulukan dibandingkan proses dengan index rendah.
d. Terangkan kelemahan dari program ini! Kondisi bagaimana yang mengakibatkan semafor tidak berperan seperti yang diinginkan!

```
 0 /*********************************************************
 1  * SuperProses (c) 2005 Rahmat M. Samik-Ibrahim, GPL-like   */
 2
 3 // ********* SuperProses *
 4 public class SuperProses {
 5    public static void main(String args[]) {
 6       Semafor[] semafor1 = new Semafor[JUMLAH_PROSES];
 7       Semafor[] semafor2 = new Semafor[JUMLAH_PROSES];
 8       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
 9          semafor1[ii] = new Semafor();
10          semafor2[ii] = new Semafor();
11       }
12
13       Thread superp=new Thread(new SuperP(semafor1,semafor2,JUMLAH_PROSES));
14       superp.start();
15
16       Thread[] proses= new Thread[JUMLAH_PROSES];
17       for (int ii = 0; ii < JUMLAH_PROSES; ii++) {
18          proses[ii]=new Thread(new Proses(semafor1,semafor2,ii));
19          proses[ii].start();
20       }
21    }
22
23    private static final int JUMLAH_PROSES = 5;
24 }
25
26 // ** SuperP ********************
27 class SuperP implements Runnable {
28    SuperP(Semafor[] sem1, Semafor[] sem2, int jmlh) {
29       semafor1      = sem1;
30       semafor2      = sem2;
31       jumlah_proses = jmlh;
32    }
33
34    public void run() {
35       for (int ii = 0; ii < jumlah_proses; ii++) {
36          semafor1[ii].kunci();
37       }
38       System.out.println("SUPER PROSES siap...");
39       for (int ii = 0; ii < jumlah_proses; ii++) {
40          semafor2[ii].buka();
41          semafor1[ii].kunci();
42       }
43    }
```

```
44
45    private Semafor[] semafor1, semafor2;
46    private int       jumlah_proses;
47
48
49 }// ** Proses ********************
50 class Proses implements Runnable {
51    Proses(Semafor[] sem1, Semafor[] sem2, int num) {
52       num_proses = num;
53       semafor1   = sem1;
54       semafor2   = sem2;
55    }
57    public void run() {
58       semafor1[num_proses].buka();
59       semafor2[num_proses].kunci();
60       System.out.println("Proses " + num_proses + " siap...");
61       semafor1[num_proses].buka();
62    }
64    private Semafor[] semafor1, semafor2;
65    private int       num_proses;
66 }
68 // ** Semafor *
69 class Semafor {
70    public Semafor()        { value = 0;   }
71    public Semafor(int val) { value = val; }
72
73    public synchronized void kunci() {
74       while (value == 0) {
75          try   { wait(); }
76          catch (InterruptedException e) { }
77       }
78       value--;
79    }
80
81    public synchronized void buka() {
82       value++;
83       notify();
84    }
85
86    private int value;
87 }
```

(B04-2005-03) Sinkronisasi II

Silakan menelusuri program Java "Hompimpah" pada lampiran berikut ini.

a. Berapa jumlah pemain "Hompimpah" tersebut?
b. Sebuatkan nama *object* `semafor` yang digunakan untuk melindungi *Critical Section*? Sebutkan baris berapa saja yang termasuk *Critical Section* tersebut.
c. Tuliskan salah satu kemungkinan keluaran dari program ini.
d. Terangkan fungsi/peranan dari metoda-metoda berikut ini: `syncPemainBandar()`; `syncBandar()`; `syncPemain()`; `syncBandarPemain()`.

```
001 /*******************************************************/
002 /* Hompimah (c) 2005 Rahmat M. Samik-Ibrahim, GPL-like  */
003 /* Nilai Tangan:TRUE="telapak" FALSE="punggung tangan"  */
```

```
004 /*****************************************************/

006 // ************************************* Hompimpah ***
007 public class Hompimpah {
008    public static void main(String args[]) {
009       Gambreng gamserver = new Gambreng(JUMLAH_PEMAIN);
010       Thread[] pemain    = new Thread[JUMLAH_PEMAIN];
011       for (int ii = 0; ii < JUMLAH_PEMAIN; ii++) {
012          pemain[ii]=new Thread(new Pemain(gamserver,ii));
013          pemain[ii].start();
014       }
015       gamserver.bandarGambreng();
016    }
017    // ************************************************
018    private static final int JUMLAH_PEMAIN = 6;
019 }
021 // ************************************** Pemain ***
022 class Pemain implements Runnable {
023    Pemain(Gambreng gserver, int nomer) {
024       gamserver  = gserver;
025       no_pemain  = nomer;
026    }
027    // ********************************** Pemain.run ***
028    public void run() {
029       gamserver.pemainGambreng(no_pemain);
030    }
031    // ************************************************
032    private Gambreng gamserver;
033    private int      no_pemain;
034 }
036 // ************************************** Gambreng ***
037 class Gambreng {
038    public Gambreng(int jumlah) {
039       bandar          = new Semafor[jumlah];
040       pemain          = new Semafor[jumlah];
041       for (int ii=0; ii<jumlah; ii++) {
042          bandar[ii]   = new Semafor();
043          pemain[ii]   = new Semafor();
044       }
045       mutex           = new Semafor(1);
046       jumlahPemain    = jumlah;
047       iterasiGambreng = 0;
048       resetGambreng();
049    }
050    // ************************ Gambreng.bandarGambreng ***
051    public void bandarGambreng() {
052       syncBandar();
053       while(! menangGambreng()) {
054          resetGambreng();
055          syncPemainBandar();
056          hitungGambreng();
057          iterasiGambreng++;
058       }
059       syncPemain();
060       System.out.println("Nomor Peserta Pemain [0] - [" +
061          (jumlahPemain-1) + "]  Pemenang Pemain Nomor[" +
062          nomorPemenang + "]  Jumlah Iterasi[" +
063          iterasiGambreng + "]");
```

```
064    }
065    // ********************** Gambreng.pemainGambreng ***
066    public void pemainGambreng(int nomor) {
067       syncBandarPemain(nomor);
068       while(! menangGambreng()) {
069          mutex.kunci();
070          // TRUE="telapak" FALSE="punggung tangan" *******
071          if ((int)(Math.random()*2)==1) {
072             truePemain=nomor;
073             trueCount++;
074          } else {
075             falsePemain=nomor;
076             falseCount++;
077          }
078          mutex.buka();
079          syncBandarPemain(nomor);
080       }
081    }
082    // ************************ Gambreng.resetGambreng ***
083    private void resetGambreng() {
084       mutex.kunci();
085       adaPemenang   = false;
086       truePemain    = 0;
087       trueCount     = 0;
088       falsePemain   = 0;
089       falseCount    = 0;
090       mutex.buka();
091    }
092    // ************************ Gambreng.menangGambreng ***
093    private boolean menangGambreng() {
094       return adaPemenang;
095    }
096    // ********************** Gambreng.hitungGambreng ***
097    private void hitungGambreng() {
098       mutex.kunci();
099       if (trueCount == 1) {
100          adaPemenang=true;
101          nomorPemenang=truePemain;
102       } else if (falseCount == 1) {
103          adaPemenang=true;
104          nomorPemenang=falsePemain; }
106       mutex.buka();
107    }
108    // ******************** Gambreng.syncPemainGambreng ***
109    private void syncPemainBandar() {
110       for (int ii=0; ii<jumlahPemain; ii++) {
111          pemain[ii].buka();
112          bandar[ii].kunci();
113       }
114    }
115    // *************************** Gambreng.syncBandar ***
116    private void syncBandar() {
117       for (int ii=0; ii<jumlahPemain; ii++)
118          bandar[ii].kunci();
119    }
120    // **************************** Gambreng.syncPemain ***
121    private void syncPemain() {
122       for (int ii=0; ii<jumlahPemain; ii++)
```

```
123          pemain[ii].buka();
124    }
125    // ********************* Gambreng.syncBandarPemain ***
126    private void syncBandarPemain(int ii) {
127       bandar[ii].buka();
128       pemain[ii].kunci();
129    }
130    // ***************************************************
131    private boolean   adaPemenang;
132    private int       truePemain, trueCount, iterasiGambreng;
133    private int       falsePemain, falseCount;
134    private int       nomorPemenang, jumlahPemain;
135    private Semafor[] bandar, pemain;
136    private Semafor   mutex;
137 }
139 // ************************************* Semafor ***
140 class Semafor {
141    public Semafor()        { value = 0;   }
142    public Semafor(int val) { value = val; }
143    // ********************************* Semafor.kunci ***
144    public synchronized void kunci() {
145       while (value == 0) {
146          try   { wait(); }
147          catch (InterruptedException e) { }
148       }
149       value--;
150    }
151    // ********************************** Semafor.buka ***
152    public synchronized void buka() {
153       value++;
154       notify();
155    }
156    // ***************************************************
157    private int value;
158 }
160 // ***************************************************
```

(B04-2006-01) Sinkronisasi III

a. Tuliskan keluaran dari program *java* berikut ini!
b. Terangkan peranan/kegunaan dari masing-masing *semaphore*: *control1*, *control2*, dan *control3*.
c. Silakan memodifikasi program, agar >*thread* "p2" yang pertama menulis "*Player 2 is up...*"

```
001  /*********************************************************/
002  /* TrioThreads (c) 2006 Rahmat M. Samik-Ibrahim, GPL-like   */
003  /* (a) Please write down the output of this java program!   */
004  /* (b) Slighty modify the program so that                   */
005  /*       "Player2 is up..." appears first!                  */
006  /*********************************************************/
007
008  // ************************************* SGUProcess ***
009  public class TrioTreads {
010     public static void main(String args[]) {
011        Engine  engine = new Engine();
012        Thread player1 = new Thread(new Player1(engine));
```

```
013         Thread player2 = new Thread(new Player2(engine));
014         Thread player3 = new Thread(new Player3(engine));
015         player1.start();
016         player2.start();
017         player3.start();
018      }
019   }
020
021   // ********************************************* Player1 ***
022   class Player1 implements Runnable {
023      Player1(Engine eng) { engine = eng; }
024      public void run()   { engine.p1();  }
025      private  Engine engine;
026   }
027
028   // ********************************************* Player2 ***
029   class Player2 implements Runnable {
030      Player2(Engine eng) { engine = eng; }
031      public void run()   { engine.p2();  }
032      private Engine engine;
033   }
034
035   // ********************************************* Player3 ***
036   class Player3 implements Runnable {
037      Player3(Engine eng) { engine = eng; }
038      public void run()   { engine.p3();  }
039      private Engine engine;
040   }
041
042   // ********************************************** Engine ***
043   class Engine {
044      public Engine() {
045         control1 = new Semaphore();
046         control2 = new Semaphore();
047         control3 = new Semaphore();
048      }
049      // ***************************************** Engine.p1 ***
050      public void p1() {
051         control3.release();
052         control2.release();
053         control1.acquire();
054         control1.acquire();
055         System.out.println("Player1 is up...");
056      }
057
058      // ***************************************** Engine.p2 ***
059      public void p2() {
060         control3.release();
061         control2.acquire();
062         control2.acquire();
063         System.out.println("Player2 is up...");
064         control1.release();
065      }
067      // ***************************************** Engine.p3 ***
068      public void p3() {
069         control3.acquire();
070         control3.acquire();
071         System.out.println("Player3 is up...");
```

```
072        control2.release();
073        control1.release();
074     }
077     private Semaphore  control1, control2, control3;
078  }
080  // ********************************************* Semaphore ***
081  class Semaphore {
082     public Semaphore()      { value = 0; }
083     public Semaphore(int v) { value = v; }
084     // ******************************** Semaphore.acquire ***
085     public synchronized void acquire() {
086        while (value == 0) {
087           try   { wait(); }
088           catch (InterruptedException e) { }
089        }
090        value--;
091     }
092     // ********************************** Semafor.release ***
093     public synchronized void release() {
094        value++;
095        notify();
096     }
097     // ********************************************************
098     private int value;
099  }
101  // ********************************************************
```

(B04-2003-04) IPC

Perhatikan berkas program java berikut ini:

a. Berapakah jumlah *object* dari "`Worker Class`" yang akan terbentuk?
b. Sebutkan nama-nama *object* dari "`Worker Class`" tersebut!
c. Tuliskan/perkirakan keluaran (*output*) 10 baris pertama, jika menjalankan program ini!
d. Apakah keluaran pada butir "c" di atas akan berubah, jika parameter CS_TIME diubah menjadi dua kali NON_CS_TIME? Terangkan!
e. Apakah keluaran pada butir "c" di atas akan berubah, jika selain parameter CS_TIME diubah menjadi dua kali NON_CS_TIME, dilakukan modifikasi NN menjadi 10? Terangkan!

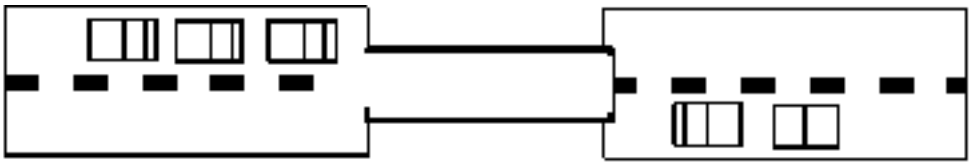
```
001 /* Gabungan Berkas:
002  * FirstSemaphore.java, Runner,java, Semaphore.java, Worker.java.
003  * Copyright (c) 2000 oleh Gagne, Galvin, Silberschatz.
004  * Applied Operating Systems Concepts-John Wiley &amp;amp; Sons, Inc.
005  * Slightly modified by Rahmat M. Samik-Ibrahim.
006  *
007  * Informasi Singkat (RMS46):
008  * Threat.start()   --> memulai thread yang memanggil Threat.run().
009  * Threat.sleep(xx) --> thread akan tidur selama xx milidetik.
010  * try {...} catch(InterruptedException e){} --> terminasi program.
011  */
012
013 public class FirstSemaphore
014 {
015     public static void main(String args[]) {
016         Semaphore  sem = new Semaphore(1);
017         Worker[]  bees = new Worker[NN];
```

```
018         for (int ii = 0; ii < NN; ii++)
019             bees[ii] = new Worker(sem, ii);
020         for (int ii = 0; ii < NN; ii++)
021             bees[ii].start();
022     }
023     private final static int NN=4;
024 }
025
026 // Worker ========================================================
027 class Worker extends Thread
028 {
029     public Worker(Semaphore sss, int nnn) {
030         sem     = sss;
031         wnumber = nnn;
032          wstring = WORKER + (new Integer(nnn)).toString();
033     }
034
035     public void run() {
036         while (true) {
037             System.out.println(wstring + PESAN1);
038             sem.P();
039             System.out.println(wstring + PESAN2);
040             Runner.criticalSection();
041             System.out.println(wstring + PESAN3);
042             sem.V();
043             Runner.nonCriticalSection();
044        }
045     }
046     private Semaphore sem;
047     private String    wstring;
048     private int       wnumber;
049     private final static String PESAN1=
         " akan  masuk   ke  Critical Section.";
050     private final static String PESAN2=
         " berada  di  dalam Critical Section.";
051     private final static String PESAN3=
         " telah keluar dari Critical Section.";
052     private final static String WORKER=
         "PEKERJA ";
053 }
054
055 // Runner ========================================================
056 class Runner
057 {
058     public static void criticalSection() {
059         try {
060             Thread.sleep( (int) (Math.random()*CS_TIME * 1000));
061         }
062         catch (InterruptedException e) { }
063     }
064
065     public static void nonCriticalSection() {
066         try {
067             Thread.sleep( (int) (Math.random()*NON_CS_TIME*1000));
068         }
069         catch (InterruptedException e) { }
070     }
071     private final static int     CS_TIME = 2;
```

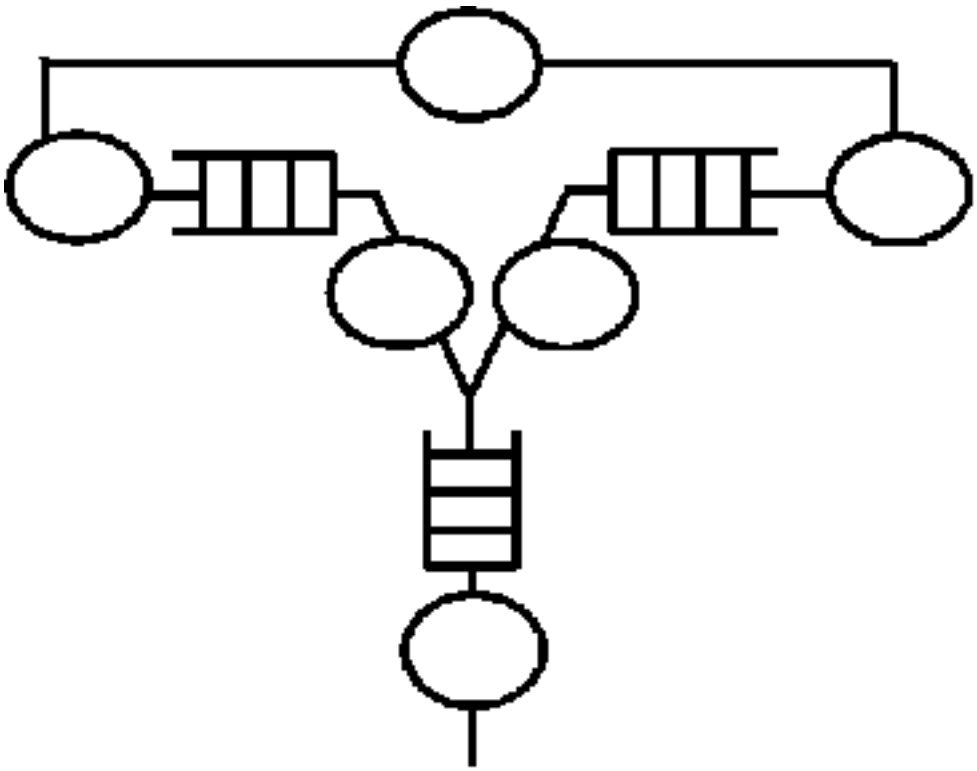
```
072     private final static int NON_CS_TIME = 2;
073 }
074
075 // Semaphore ==========================================================
076 final class Semaphore
077 {
078     public Semaphore() {
079         value = 0;
080     }
081
082     public Semaphore(int v) {
083         value = v;
084     }
085
086     public synchronized void P() {
087         while (value <= 0) {
088             try {
089                 wait();
090             }
091             catch (InterruptedException e) { }
092         }
093         value --;
094     }
095
096     public synchronized void V() {
097         ++value;
098         notify();
099     }
100
101     private int value;
102 }
103
104 // END ================================================================
```

(B04-2007-02) Jembatan

Perhatikan program Java terlampir. Diketahui sebuah jembatan, yang pada satu saat hanya dapat dilalui dari satu arah.



a. Lengkapi model berikut ini dengan obyek-obyek yang ada.



b. Sebutkan nama semua obyek class "Semafor" yang digunakan, berikut jenis semafor yang digunakan.
c. Sebutkan nama semua obyek class "Thread" yang digunakan, berikut peranan dari masing-masing obyek.
d. Bagaimana kira-kira, keluaran 10 baris pertama dari program ini?

```
001 import java.util.*;
002
003 public class Jembatan {
004    public static void main(String args[]) {
005       Semafor sync_ki  = new Semafor(0);
006       Semafor sync_ka  = new Semafor(0);
007
008       Antrian antri_ki = new Antrian( "kiri", MAX_KI);
009       Antrian antri_ka = new Antrian("kanan", MAX_KA);
010       Antrian jembatan = new Antrian("",      MAXJMB);
011
012       Thread tiba_ki   = new Thread(new Tiba(antri_ki, INTERVAL_KI));
013       Thread tiba_ka   = new Thread(new Tiba(antri_ka, INTERVAL_KA));
014       Thread masuk_ki  = new Thread(new Masuk (antri_ki, jembatan,
                                 sync_ki));
015       Thread masuk_ka  = new Thread(new Masuk (antri_ka, jembatan,
                                 sync_ka));
016       Thread pak_ogah  = new Thread(new
017                            Pak_Ogah(antri_ki, antri_ka,
                                    INTERVAL_OGAH));
018       Thread keluar    = new Thread(new
019                            Keluar(jembatan, INTERVAL_JMB, sync_ki,
                                    sync_ka));
020
021       tiba_ki.start();
```

```
022       masuk_ki.start();
023       tiba_ka.start();
024       masuk_ka.start();
025       keluar.start();
026       pak_ogah.start();
027    }
028    private static final int MAX_KI        = 24;
029    private static final int MAX_KA        = 24;
030    private static final int MAXJMB        =  3;
031    private static final int INTERVAL_KI   =  3;
032    private static final int INTERVAL_KA   =  3;
033    private static final int INTERVAL_JMB  =  1;
034    private static final int INTERVAL_OGAH = 12;
035 }
036
037 class Tiba implements Runnable {
038    public Tiba(Antrian ant, int in) {
039       antrian  = ant;
040       interval = in;
041       urutan   = 0;
042    }
043
044    public void run() {
045       while (true) {
046          Mobil mobil = new Mobil(antrian.arah(), ++urutan);
047          antrian.tiba(mobil);
048          System.out.println("MOBIL["+mobil.arah()+","
049              +mobil.urutan()+"] tiba.");
050          antrian.jeda(interval);
051       }
052    }
053
054    private Antrian antrian;
055    private int     interval;
056    private int     urutan;
057 }
058 class Pak_Ogah implements Runnable {
059    public Pak_Ogah(Antrian aki, Antrian aka, int in) {
060       antrian_ki = aki;
061       antrian_ka = aka;
062       interval   = in;
063       giliran    = false;
064    }
065
066    public void run() {
067       while (true) {
068          giliran    ^= true;          // giliran = (giliran xor true)
069          if (giliran) { antrian = antrian_ki; }
070          else         { antrian = antrian_ka; }
071          antrian.jeda(interval);
072          Mobil mobil = new Mobil("pogah",-1);    // minta ganti arah
073          antrian.tiba(mobil);
074          System.out.println("Pak OGAH menghentikan arus "+
                                antrian.arah()+".");
075       }
076    }
077
078    private Antrian antrian, antrian_ki, antrian_ka;
```

```
079    private Semafor sync_ki;
080    private Semafor sync_ka;
081    private int     interval;
082    private boolean giliran;
083 }
084
085 class Masuk implements Runnable {
086    public Masuk(Antrian dr, Antrian tj, Semafor gl) {
087       dari    = dr;
088       tujuan  = tj;
089       giliran = gl;
090    }
091
092    public void run() {
093       giliran.kunci();
094       while (true) {
095          Mobil mobil = dari.berangkat();
096          tujuan.tiba(mobil);
097          if ((mobil.arah()).equals("pogah")) {
098             giliran.kunci();
099          } else {
100             System.out.println("MOBIL[" + mobil.arah() + ","
101             + mobil.urutan() + "] memasuki jembatan." );
102          }
103       }
104    }
105
106    private Antrian dari, tujuan;
107    private Semafor giliran;
108 }
109 class Keluar implements Runnable {
110    public Keluar(Antrian jm, int in, Semafor gki, Semafor gka) {
111       jembatan = jm;
112       gilir_ki = gki;
113       gilir_ka = gka;
114       interval = in;
115       giliran  = false;
116    }
117
118    public void run() {
119       gilir_ki.buka();
120       while (true) {
121          jembatan.jeda(interval);
122          Mobil mobil = jembatan.berangkat();
123          if ((mobil.arah()).equals("pogah")) {
124             giliran ^= true;
125             if (giliran) { gilir_ka.buka(); }
126             else         { gilir_ki.buka(); }
127             System.out.println("Pergantian arah!");
128          } else {
129             System.out.println("MOBIL[" + mobil.arah() + ","
130             + mobil.urutan() + "] melanjutkan perjalanan.");
131          }
132       }
133    }
134
135    private  Antrian jembatan;
136    private  Semafor gilir_ki;
```

```
137    private  Semafor gilir_ka;
138    private  int     interval;
139    private  boolean giliran;
140 }
141
142 class Antrian {
143    public Antrian(String ar, int uk) {
144       masuk       = 0;
145       keluar      = 0;
146       arah        = ar;
147       ukuran      = uk;
148       antrian     = new Mobil[ukuran];
149       mutex       = new Semafor(1);
150       boleh_isi   = new Semafor(ukuran);
151       boleh_ambil = new Semafor(0);
152    }
153
154    public void tiba(Mobil mobil) {
155       boleh_isi.kunci();
156       mutex.kunci();
157          antrian[masuk] = mobil;
158          masuk          = (masuk + 1) % ukuran;
159       mutex.buka();
160       boleh_ambil.buka();
161    }
162    public Mobil berangkat() {
163       boleh_ambil.kunci();
164       mutex.kunci();
165          Mobil mobil = antrian[keluar];
166          keluar      = (keluar + 1) % ukuran;
167       mutex.buka();
168       boleh_isi.buka();
169       return mobil;
170    }
171
172    public String arah() {
173       return  arah;
174    }
175
176    public static void jeda() { jeda(WAKTU_JEDA); }
177
178    public static void jeda(int waktu) {
179       try   { Thread.sleep( (int) (waktu * Math.random() * 1000)); }
180       catch (InterruptedException e) {}
181    }
183    private int     urutan;
184    private String  arah;
185    private Mobil[] antrian;
186    private int     ukuran, masuk, keluar;
187    private static  final int WAKTU_JEDA = 5;
188    private Semafor mutex, boleh_isi, boleh_ambil;
189 }
190
191 class Semafor
192 {
193    public Semafor()       { nilai = 0;  }
194    public Semafor(int nl) { nilai = nl; }
195
```

```
196    public synchronized void kunci() {
197       while (nilai == 0) {
198          try   { wait(); }
199          catch (InterruptedException e) { }
200       }
201       nilai--;
202    }
203
204    public synchronized void buka() { ++nilai; notify(); }
205
206    private int nilai;
207 }
208
209 class Mobil {
210    public Mobil(String str, int ur) {
211       arah   = str;
212       urutan = ur;
213    }
214
215    public  String arah()   { return arah;   }
216    public  int    urutan() { return urutan; }
217
218    private String arah;
219    private int    urutan;
220 }
```

(B04-2007-04) Harry Potter 7

Perhatikan program HP7.java berikut ini.

a. Ada berapa obyek semafor yang digunakan? Sebutkan satu per satu obyek semafor tersebut!
b. Bagaimana keluaran dari program tersebut? Tuliskan!
c. Lakukan sedikit modifikasi, agar keluaran program menjadi “Harry Potter And The Deadly Hallow”. Kerjakan langsung pada halaman program tersebut!

```
001 /*********************************************************/
002 /* HarryPotter7 (c)2007 Rahmat M. Samik-Ibrahim, GPL-like */
003 /*********************************************************/
004
005 public class HP7 {
006    public static void main(String args[]) {
007       Engine    engine = new Engine(strings, strseq);
008       Thread[] printer = new Thread[strings.length];
009       for (int ii = 0; ii < strings.length; ii++) {
010          printer[ii]=new Thread(new Printer(ii, engine));
011          printer[ii].start();
012       }
013    }
014    private final static String strings[]=
015       {"And","Deathly","Hallows","Harry","Potter","The"};
016    private final static int strseq[]=      {5,4,3,2,1,0};
017 }
018
019 /*********************************************************/
020 class Engine {
021    Engine(String str[],int strseq[]) {
```

```
022       this.str    = str;
023       this.strseq = strseq;
024       semaphore   = new Semaphore[str.length];
025       for (int ii=0; ii<str.length; ii++) {
026          semaphore[ii] = new Semaphore();
027       }
028       sequence    = 0;
029       semaphore[strseq[sequence++]].release();
030    }
031    public void go(int ii) {
032       semaphore[ii].acquire();
033       System.out.print(str[ii] + " ");
034       if (sequence < strseq.length)
035          semaphore[strseq[sequence++]].release();
036       else
037          System.out.println();
038    }
039    private Semaphore[] semaphore;
040    private String      str[];
041    private int         strseq[];
042    private int         sequence;
043 }
044
046 class Printer implements Runnable {
047    Printer(int ii, Engine ee) {
048       number = ii;
049       engine = ee;
050    }
051    public void run() {
052       engine.go(number);
053    }
054    private int    number;
055    private Engine engine;
056 }
058 /*********************************************************/
059 class Semaphore {
060    public Semaphore()      { value = 0; }
061    public Semaphore(int v) { value = v; }
062    public synchronized void acquire() {
063       while (value == 0) {
064          try   { wait(); }
065          catch (InterruptedException e) { }
066       }
067       value--;
068    }
069    public synchronized void release() {
070       value++;
071       notify();
072    }
073    private int value;
074 }
075 /*****************************************************/
```

Indeks

L

M

O

P

R

S

T

U

V

W