Sistem Operasi

Buku Referensi
Informatika dan Sistem Informasi

Sistem operasi merupakan sebuah penghubung antara pengguna dari komputer dengan perangkat keras komputer. Sebelum ada sistem operasi, orang hanya mengunakan komputer dengan menggunakan sinyal analog dan sinyal digital. Seiring dengan berkembangnya pengetahuan dan teknologi, pada saat ini terdapat berbagai sistem operasi dengan keunggulan masing-masing. Untuk lebih memahami sistem operasi maka sebaiknya perlu diketahui terlebih dahulu beberapa konsep dasar mengenai sistem operasi itu sendiri.

Sistem Operasi juga memiliki proses, Suatu proses adalah lebih dari kode program, dimana kadang kala dikenal sebagai bagian tulisan. Proses juga termasuk aktivitas yang sedang terjadi, sebagaimana digambarkan oleh nilai pada program counter dan isi dari daftar prosesor/ processor's register. Suatu proses umumnya juga termasuk process stack, yang berisikan data temporer (seperti parameter metoda, address yang kembali, dan variabel lokal) dan sebuah data section, yang berisikan variabel global.

Didalam buku ini juga membahas tentang Sinkronisasi dan Deadlock Sistem Operasi, serta Memori Sistem Operasi, dimana Memori merupakan inti dari sistem komputer modern. CPU mengambil instruksi dari memori sesuai yang ada pada program counter. Instruksi dapat berupa menempatkan/ menyimpan dari/ ke alamat di memori, penambahan, dan sebagainya. Dalam managemen memori ini, kita akan membahas bagaimana urutan alamat memori yang dibuat oleh program yang berjalan.

Sistem berkas merupakan mekanisme penyimpanan on-line serta untuk akses, baik data mau pun program yang berada dalam Sistem Operasi. Terdapat dua bagian penting dalam sistem berkas.

Secara umum, terdapat beberapa jenis seperti device penyimpanan (disk, tape), transmission device (network card, modem), dan human-interface device (screen, keyboard, mouse). Device tersebut dikendalikan oleh instruksi I/O. Alamat-alamat yang dimiliki oleh device akan digunakan oleh direct I/O instruction dan memory-mapped I/O.

Beberapa konsep yang umum digunakan ialah port, bus (daisy chain/ shared direct access), dan controller (host adapter). Port adalah koneksi yang digunakan oleh device untuk berkomunikasi dengan mesin. Bus adalah koneksi yang menghubungkan beberapa device menggunakan kabel-kabel. Controller adalah alat-alat elektronik yang berfungsi untuk mengoperasikan port, bus, dan device.

Semoga Buku Sistem Operasi ini dapat digunakan sebagai suatu Referensi untuk Bidang Informatika dan Sistem Informasi.

SEFA BUMI PERSADA
Jl. Malikussaleh No. 3 Bayu - Aceh Utara
email: sefabumipersada@gmail.com
Telp. 085260363550

ISBN 978-602-6960-79-5

Dr. Hartono, S.Kom, M.Kom, IPM
Dr. Dahlan Abdullah, ST, M.KOM, IPM
Fadlisyah, S.Si, MT
Cut Ita Erliana, ST, MT, IPM

Sistem Operasi

Sistem Operasi

Buku Referensi
Informatika dan Sistem Informasi

Dr. Hartono, S.Kom, M.Kom, IPM
Dr. Dahlan Abdullah, ST, M.KOM, IPM
Fadlisyah, S.Si, MT
Cut Ita Erliana, ST, MT, IPM

Sistem Operasi

Buku Referensi
Informatika dan Sistem Informasi

Dr. Hartono, S.Kom, M.Kom, IPM
Dr. Dahlan Abdullah, ST, M.KOM, IPM
Fadlisyah, S.Si, MT
Cut Ita Erliana, ST, MT, IPM

Diterbitkan Oleh:

SISTEM OPERASI
(Buku Referensi dan Sistem Informasi)
Oleh : Dr. Hartono, S.Kom, M.Kom, IPM
Dr. Dahlan Abdullah, ST, M.KOM, IPM
Fadlisyah, S.Si, MT
Cut Ita Erliana, ST, MT, IPM
Hak Cipta © 2018 pada Penulis
Editor : -

Cover Design : *M. Rizki, S. Kom. I (SEFA)*

Layout : *M. Rizki, S. Kom. I (SEFA)*

Pracetak dan Produksi : *CV. Sefa Bumi Persada*

Hak Cipta dilindungi undang-undang.
Dilarang memperbanyak atau memindahkan sebagian atau seluruh isi buku ini dalam bentuk apapun, baik secara elektronis maupun mekanis, termasuk memfotokopi, merekam atau dengan sistem penyimpanan lainnya, tanpa izin tertulis dari Penulis
Penerbit:

SEFA BUMI PERSADA

Jl. B. Aceh – Medan, Alue Awe - Lhokseumawe

email: sefabumipersada@gmail.com

Telp. 085260363550

Cetakan I : 2018

ISBN – 978-602-6960-79-5
1. Hal. xiv + 238: 15,8 x 23 cm I. Judul

Kata Pengantar

Dengan mengucapkan puji dan syukur kehadirat Allah SWT, di mana atas rahmat dan karunia-Nya Penulis telah dapat menyelesaikan buku yang berjudul "***SISTEM OPERASI***".

Rasa terima kasih penulis ucapkan kepada seluruh rekan-rekan yang telah membantu untuk menyelesaikan buku ini dan pihak-pihak yang telah memberi bantuan dan bimbingan sehingga penulis dapat menyelesaikan naskah Sistem Operasi ini, yang tidak dapat disebutkan satu per satu.

Adapun materi-materi yang dibahas di dalam buku ini mencakup: Pengantar Sistem Operasi, Proses dan Thread, Sinkronisasi dan Deadlock, Memori, Sistem Berkas dan I/O dan Disk.

Penulis hanya mengucapkan selamat membaca.

Penulis

Daftar Isi

Halaman Cover i

Kata Pengantar iii

Daftar Isi iv

BAB I PENDAHULUAN

1.1 Sistem Operasi 1

1.1.1 Fungsi Dasar 1

1.1.2 Tujuan Mempelajari Sistem Operasi 2

1.1.3. Sasaran Sistem Operasi 2

1.1.4. Sejarah Sistem Operasi 2

1.1.5. Layanan Sistem Operasi 3

1.2. Struktur Komputer 4

1.2.1. Sistem Operasi Komputer 4

1.2.2. Struktur I/O 4

1.2.3. Struktur Penyimpanan 5

1.2.4. Storage Hierarchy 6

1.2.5. Proteksi Perangkat Keras 6

1.3. Struktur Sistem Operasi 8

1.3.1. Komponen-komponen Sistem 8

1.3.2. Managemen Proses 9

1.3.3. Managemen Memori Utama 9

1.3.4. Managemen Secondary-Storage 9

1.3.5. Managemen Sistem I/O 10

1.3.6. Managemen Berkas 10

1.3.7. Sistem Proteksi 10

1.3.8. Jaringan 10

1.3.9. Command-Interpreter System 11
1.3.10. Layanan Sistem Operasi 11
1.3.11. System Calls 11
1.3.12. Mesin Virtual 12
1.3.13. Perancangan Sistem dan Implementasi 12
1.3.14. System Generation (SYSGEN) 13
1.4. Rangkuman 13
1.5. Pertanyaan 16
1.6. Rujukan 17

BAB II Proses dan Thread

2.1. Proses 19
2.1.1. Konsep Dasar dan Definisi Proses 19
2.1.2. Keadaan Proses 20
2.1.3. Process Control Block 21
2.1.4. Threads 23
2.2. Penjadualan Proses 23
2.2.1. Penjadualan Antrian 23
2.2.2. Penjadual 25
2.2.3. Alih Konteks 27
2.3. Operasi-Operasi Pada Proses 29
2.3.1. Pembuatan Proses 30
2.3.2. Terminasi Proses 31
2.4. Hubungan Antara Proses 32
2.4.1. Proses yang Kooperatif 33
2.4.2. Komunikasi Proses Dalam Sistem 35
2.5. Thread 42
2.5.1. Konsep Dasar 42

2.5.2. Keuntungan 43
2.5.3. User Threads 43
2.5.4. Kernel Threads 44
2.6. Model Multithreading 44
2.6.1. Model Many to One 45
2.6.2. Model One to One 45
2.6.3. Model Many to Many 46
2.6.4. Thread Dalam Solaris 2 47
2.6.5. Thread Java 49
2.6.6. Managemen Thread 51
2.6.7. Keadaan Thread 52
2.6.8. Thread dan JVM 52
2.6.9. JVM dan Sistem Operasi 53
2.6.10. Contoh Solusi Multithreaded 53
2.7. Penjadual CPU 53
2.7.1. Konsep Dasar 53
2.7.3. Penjadual Shortest Job First 58
2.7.4. Penjadual Prioritas 59
2.7.5. Penjadual Round Robin 60
2.8. Penjadualan Multiprocessor 62
2.8.1. Penjadualan Multiple Processor 62
2.8.2. Penjadualan Real Time 62
2.8.3. Penjadualan Thread 64
2.9. Java Thread dan Algoritmanya 64
2.9.1. Penjadualan Java Thread 64
2.9.2. Evaluasi Algoritma 66
2.10. Kesimpulan 68
2.10.1. Proses 68

2.10.2. Thread 69
2.10.3. Penjadualan CPU 70
2.11. Soal-soal Latihan 71
2.11.1. Proses 71
2.11.2. Thread 71
2.11.3. Penjadualan CPU 72
2.12. Rujukan 73
2.13. Daftar Istilah 74

BAB III Sinkronisasi dan Deadlock

3.1. Sinkronisasi 77
3.1.1. Latar Belakang 77
3.1.2. Critical Section 80
3.1.3. Solusi Hardware pada Sinkronisasi 84
3.1.4. Semaphore 85
3.1.5. Problem Klasik pada Sinkronisasi 87
3.1.6. Monitors 90
3.2. Deadlock 91
3.2.1. Latar Belakang 91
3.2.2. Resources-Allocation Graph 92
3.2.3. Model Sistem 94
3.2.4. Strategi menghadapi Deadlock 94
3.2.5. Mencegah Deadlock 95
3.2.6. Menghindari Deadlock 97
3.2.7. Algoritma Bankir 98
3.2.8. Mendeteksi Deadlock dan Memulihkan Deadlock 99
3.3. Kesimpulan 99
3.4. Latihan 100

3.5. Rujukan 101
3.5.1. Rujukan Sinkronisasi 101
3.5.2. Rujukan Deadlock 102

BAB IV Memori

4.1. Latar Belakang 103
4.1.1. Pengikatan Alamat 103
4.1.2. Ruang Alamat Fisik dan Logik 104
4.1.3. Penempatan Dinamis 104
4.1.4. Perhubungan Dinamis dan Berbagi Library 105
4.1.5. Lapisan Atas 106
4.2. Penukaran (Swap) 107
4.3. Alokasi Memori Yang Berdampingan 109
4.4. Pemberian Halaman 112
4.4.1. Metode Dasar 113
4.4.2. Struktur Tabel Halaman 116
4.5. Segmentasi 121
4.5.1. Metode Dasar 121
4.5.2. Perangkat Keras 122
4.5.3. Pemeliharaan dan Pembagian 123
4.5.4. Fragmentasi 123
4.6. Segmentasi Dengan Pemberian Halaman 124
4.6.1. Pengertian 124
4.6.2. Kelebihan Segmentasi dengan Pemberian Halaman 124
4.6.3. Perbedaan Segmentasi dan Paging 125
4.6.4. Pengimplementasian Segmentasi dengan Pemberian Halaman Intel i386 126
4.7. Memori Virtual 127

4.7.1. Pengertian 128
4.7.2. Keuntungan 129
4.7.3. Implementasi 129
4.8. Permintaan Pemberian Halaman (Demand Paging) 129
4.8.1. Permasalahan pada Page Fault 130
4.8.2. Skema Bit Valid - Tidak Valid 130
4.9. Pemindahan Halaman 132
4.9.1. Skema Dasar 133
4.9.2. Pemindahan Halaman Secara FIFO 134
4.9.3. Pemindahan Halaman Secara Optimal 135
4.9.4. Pemindahan Halaman Secara LRU 135
4.9.5. Pemindahan Halaman Secara Perkiraan LRU 137
4.9.6. Dasar Perhitungan Pemindahan Halaman 138
4.9.7. Algoritma Page-Buffering 139
4.10. Alokasi Frame 139
4.10.1. Jumlah Frame Minimum 140
4.10.2. Algoritma Alokasi 141
4.10.3. Alokasi Global lawan Local 142
4.11. Thrashing 143
4.11.1. Penyebab Thrashing 143
4.11.2. Model Working Set 144
4.11.3. Frekuensi Kesalahan Halaman 146
4.12. Contoh Pada Sistem Operasi 147
4.12.1. Windows NT 147
4.12.2. Solaris 2 147
4.12.3. Linux 148
4.13. Pertimbangan Lain 149

4.13.1. Sebelum Pemberian Halaman 149
4.13.2. Ukuran Halaman 149
4.13.3. Tabel Halaman yang Dibalik 150
4.13.4. Struktur Program 151
4.13.5. Penyambungan Masukan dan Keluaran 151
4.13.6. Pemrosesan Waktu Nyata 152

BAB V Sistem Berkas

5.1. Pengertian 153
5.2. Berkas 153
5.2.1. Konsep Dasar 153
5.2.2. Atribut Pada Berkas 154
5.2.3. Operasi Pada Berkas 155
5.2.4. Jenis Berkas 158
5.2.5. Struktur Berkas 159
5.2.6. Struktur Berkas Pada Disk 160
5.2.7. Penggunaan Berkas Secara Bersama-sama 161
5.3. Metode Akses 161
5.3.1. Akses Secara Berurutan 161
5.3.2. Akses Langsung 162
5.3.3. Akses Dengan Menggunakan Indeks 162
5.4. Struktur Direktori 162
5.4.1. Operasi Pada Direktori 162
5.4.2. Direktori Satu Tingkat 163
5.4.3. Direktori Dua Tingkat 163
5.4.4. Direktori Dengan Struktur Tree 163
5.4.5. Direktori Dengan Struktur Acyclic-Graph 164
5.4.6. Direktori Dengan Struktur Graph 164

5.5. Proteksi Berkas 164
5.5.1. Tipe Akses Pada Berkas 165
5.5.2. Akses List dan Group 165
5.5.3. Pendekatan Sistem Proteksi yang Lain 167
5.6. Struktur Sistem Berkas 167
5.6.1. Organisasi Sistem Berkas 167
5.6.2. Mounting Sistem Berkas 168
5.7. Metode Alokasi Berkas 169
5.7.1. Alokasi Secara Berdampingan (Contiguous Allocation) 169
5.7.2. Alokasi Secara Berangkai (Linked Allocation) 170
5.7.3. Alokasi Dengan Indeks (Indexed Allocation) 172
5.7.4. Kinerja Sistem Berkas 173
5.8. Managemen Ruang Kosong (Free Space) 174
5.8.1. Menggunakan Bit Vektor 175
5.8.2. Linked List 175
5.8.3. Grouping 176
5.8.4. Counting 176
5.9. Implementasi Direktori 176
5.9.1. Linear List 176
5.9.2. Hash Table 177
5.10. Efisiensi dan Unjuk Kerja 178
5.10.1. Efisiensi 178
5.10.2. Kinerja 178
5.11. Recovery 179
5.11.1. Pemeriksaan Rutin 179
5.11.2. Back Up and Restore 180
5.12. Macam-macam Sistem Berkas 181

5.12.1. Sistem Berkas Pada Windows 181
5.12.2. Sistem Berkas pada UNIX (dan turunannya) 182
5.12.3. Perbandingan antara Windows dan UNIX 184
5.12.4. Macam-macam Sistem Berkas di UNIX 185
5.13. Kesimpulan 185
5.14. Soal-Soal Sistem Berkas 188

BAB VI I/O dan Disk

6.1. Perangkat Keras I/O 189
6.1.2. Interupsi 190
6.1.3. DMA.. 192
6.2. Interface Aplikasi I/O 194
6.2.1. Peralatan Block dan Karakter 195
6.2.2. Peralatan Jaringan 195
6.2.3. Jam dan Timer 195
6.2.4. Blocking dan Nonblocking I/O 196
6.3. Kernel I/O Subsystem 196
6.3. 1. I/O Scheduling 196
6.3.2. Buffering 197
6.3.3. Caching 198
6.3.4. Spooling dan Reservasi Device 199
6.3.5. Error Handling 200
6.3.6. Kernel Data Structure 200
6.4. Penanganan Permintaan I/O 201
6.5. Kinerja I/O 203
6.5.1. Pengaruh I/O pada Kinerja 203
6.5.2. Cara Meningkatkan Efisiensi I/O 203
6.5.3. Implementasi Fungsi I/O 204

6.6. Struktur Disk 204
6.7. Penjadualan Disk 205
6.7.1. Penjadualan FCFS 206
6.7.2. Penjadualan SSTF 207
6.7.3. Penjadualan SCAN 208
6.7.4. Penjadualan C-SCAN 208
6.7.5. Penjadualan LOOK 209
6.7.6. Pemilihan Algoritma Penjadualan Disk 209
6.8. Managemen Disk 210
6.8.1. Memformat Disk 210
6.8.2. Boot Block 211
6.8.3. Bad Blocks 211
6.9. Penanganan Swap-Space 211
6.9.1. Penggunaan Swap-Space 212
6.9.2. Lokasi Swap-Space 212
6.9.3. Pengelolaan Swap-Space 213
6.10. Kehandalan Disk 215
6.11. Implementasi Stable-Storage 216
6.12. Tertiary-Storage Structure 217
6.12.1. Macam-macam Tertiary-Strorage Structure 218
6.12.2. Masalah-Masalah yang Berkaitan Dengan Sistem Operasi 221
6.12.3. interface Aplikasi 221
6.12.4. Penamaan Berkas 223
6.12.5. Managemen Penyimpanan Hirarkis 223
6.13. Rangkuman 224
6.13.1. I/O 224
6.13.2. Disk 224

6.14. Soal Latihan 225
6.15. Rujukan 227
6.16. Daftar Istilah 227

Bab 1

Pendahuluan

Bab ini berisi tiga pokok pembahasan. Pertama, membahas hal-hal umum seputar sistem operasi.Selanjutnya, menerangkan konsep perangkat keras sebuah komputer. Sebagai penutup akan diungkapkan, pokok konsep dari sebuah sistem operasi.

1.1. Sistem Operasi

Sistem operasi merupakan sebuah penghubung antara pengguna dari komputer dengan perangkat keras komputer. Sebelum ada sistem operasi, orang hanya menggunakan komputer dengan menggunakan sinyal analog dan sinyal digital. Seiring dengan berkembangnya pengetahuan dan teknologi, pada saat ini terdapat berbagai sistem operasi dengan keunggulan masing-masing. Untuk lebih memahami sistem operasi maka sebaiknya perlu diketahui terlebih dahulu beberapa konsep dasar mengenai sistem operasi itu sendiri.

Pengertian sistem operasi secara umum ialah pengelola seluruh sumber-daya yang terdapat pada sistem komputer dan menyediakan sekumpulan layanan (*system calls*) ke pemakai sehingga memudahkan dan menyamankan penggunaan serta pemanfaatan sumber-daya sistem komputer.

1.1.1. Fungsi Dasar

Sistem komputer pada dasarnya terdiri dari empat komponen utama, yaitu perangkat-keras, program aplikasi, sistem-operasi, dan para pengguna. Sistem operasi berfungsi untuk mengatur dan mengawasi penggunaan perangkat keras oleh berbagai program aplikasi serta para pengguna.

Sistem operasi berfungsi ibarat pemerintah dalam suatu negara, dalam arti membuat kondisi komputer agar dapat menjalankan

program secara benar. Untuk menghindari konflik yang terjadi pada saat pengguna menggunakan sumber-daya yang sama, sistem operasi mengatur pengguna mana yang dapat mengakses suatu sumber-daya. Sistem operasi juga sering disebut *resource allocator*. Satu lagi fungsi penting sistem operasi ialah sebagai program pengendali yang bertujuan untuk menghindari kekeliruan (*error*) dan penggunaan komputer yang tidak perlu.

1.1.2.Tujuan Mempelajari Sistem Operasi

Tujuan mempelajari sistem operasi agar dapat merancang sendiri serta dapat memodifikasi sistem yang telah ada sesuai dengan kebutuhan kita, agar dapat memilih alternatif sistem operasi, memaksimalkan penggunaan sistem operasi dan agar konsep dan teknik sistem operasi dapat diterapkan pada aplikasi-aplikasi lain.

1.1.3. Sasaran Sistem Operasi

Sistem operasi mempunyai tiga sasaran utama yaitu *kenyamanan* -- membuat penggunaan komputer menjadi lebih nyaman, *efisien* -- penggunaan sumber-daya sistem komputer secara efisien, serta mampu *berevolusi* -- sistem operasi harus dibangun sehingga memungkinkan dan memudahkan pengembangan, pengujian serta pengajuan sistem-sistem yang baru.

1.1.4. Sejarah Sistem Operasi

Menurut Tanenbaum, sistem operasi mengalami perkembangan yang sangat pesat, yang dapat dibagi kedalam empat generasi:

- Generasi Pertama (1945-1955)
  Generasi pertama merupakan awal perkembangan sistem komputasi elektronik sebagai pengganti sistem komputasi mekanik, hal itu disebabkan kecepatan manusia untuk menghitung terbatas dan manusia sangat mudah untuk membuat kecerobohan, kekeliruan bahkan kesalahan. Pada generasi ini belum ada sistem operasi, maka sistem komputer diberi instruksi yang harus dikerjakan secara langsung.

- Generasi Kedua (1955-1965)
  Generasi kedua memperkenalkan *Batch Processing System*, yaitu Job yang dikerjakan dalam satu rangkaian, lalu dieksekusi secara berurutan.Pada generasi ini sistem komputer belum dilengkapi sistem operasi, tetapi beberapa fungsi sistem operasi telah ada, contohnya fungsi sistem operasi ialah FMS dan IBSYS.

- Generasi Ketiga (1965-1980)
  Pada generasi ini perkembangan sistem operasi dikembangkan untuk melayani banyak pemakai sekaligus, dimana para pemakai interaktif berkomunikasi lewat terminal secara on-line ke komputer, maka sistem operasi menjadi *multi-user* (di gunakan banyak pengguna sekaligus) dan *multi-programming* (melayani banyak program sekali gus).
- Generasi Keempat (Pasca 1980an)
  Dewasa ini, sistem operasi dipergunakan untuk jaringan komputer dimana pemakai menyadari keberadaan komputer-komputer yang saling terhubung satu sama lainnya. Pada masa ini para pengguna juga telah dinyamankan dengan *Graphical User Interface* yaitu antar-muka komputer yang berbasis grafis yang sangat nyaman, pada masa ini juga dimulai era komputasi tersebar dimana komputasi-komputasi tidak lagi berpusat di satu titik, tetapi dipecah dibanyak komputer sehingga tercapai kinerja yang lebih baik.

1.1.5. Layanan Sistem Operasi

Sebuah sistem operasi yang baik menurut Tanenbaum harus memiliki layanan sebagai berikut: pembuatan program, eksekusi program, pengaksesan *I/O Device*, pengaksesan terkendali terhadap berkas pengaksesan sistem, deteksi dan pemberian tanggapan pada kesalahan, serta akunting.

Pembuatan program yaitu sistem operasi menyediakan fasilitas dan layanan untuk membantu para pemrogram untuk menulis program; Eksekusi Program yang berarti Instruksi-instruksi dan data-data harus dimuat ke memori utama, perangkat-parangkat masukan/ keluaran dan berkas harus di-inisialisasi, serta sumber-daya yang ada harus disiapkan, semua itu harus di tangani oleh sistem operasi; Pengaksesan *I/O Device*, artinya Sistem Operasi harus mengambil alih sejumlah instruksi yang rumit dan sinyal kendali menjengkelkan agar pemrogram dapat berfikir sederhana dan perangkat pun dapat beroperasi; Pengaksesan terkendali terhadap berkas yang artinya disediakannya mekanisme proteksi terhadap berkas untuk mengendalikan pengaksesan terhadap berkas; Pengaksesan sistem artinya pada pengaksesan digunakan bersama (*shared system*); Fungsi pengaksesan harus menyediakan proteksi terhadap sejumlah sumber-daya dan data dari pemakai tak terdistorsi serta menyelesaikan konflik-konflik dalam perebutan sumber-daya; Deteksi dan Pemberian tanggapan pada kesalahan, yaitu jika muncul permasalahan muncul pada sistem komputer maka sistem operasi harus memberikan tanggapan yang menjelaskan kesalahan yang terjadi serta dampaknya terhadap aplikasi yang sedang berjalan; dan Akunting yang artinya Sistem Operasi yang bagus mengumpulkan data statistik

penggunaan beragam sumber-daya dan memonitor parameter kinerja.

1.2. Struktur Komputer

Struktur sebuah sistem komputer dapat dibagi menjadi:

- Sistem Operasi Komputer
- Struktur I/O.
- Struktur Penyimpanan.
- *Storage Hierarchy*.
- Proteksi Perangkat Keras.

1.2.1. Sistem Operasi Komputer

Dewasa ini sistem komputer multiguna terdiri dari CPU (*Central Processing Unit*); serta sejumlah *device controller* yang dihubungkan melalui *bus* yang menyediakan akses ke memori. Setiap device *controller* bertugas mengatur perangkat yang tertentu (contohnya *disk drive*, *audio device*, dan *video display*). CPU dan *device controller* dapat dijalankan secara bersamaan, namun demikian diperlukan mekanisme sinkronisasi untuk mengatur akses ke memori.

Pada saat pertama kali dijalankan atau pada saat *boot*, terdapat sebuah program awal yang mesti dijalankan. Program awal ini disebut program *bootstrap*. Program ini berisi semua aspek dari sistem komputer, mulai dari register CPU, *device controller*, sampai isi memori.

Interupsi merupakan bagian penting dari sistem arsitektur komputer. Setiap sistem komputer memiliki mekanisme yang berbeda. Interupsi bisa terjadi apabila perangkat keras (*hardware*) atau perangkat lunak (*software*) minta "dilayani" oleh prosesor. Apabila terjadi interupsi maka prosesor menghentikan proses yang sedang dikerjakannya, kemudian beralih mengerjakan *service routine* untuk melayani interupsi tersebut. Setelah selesai mengerjakan *service routine* maka prosesor kembali melanjutkan proses yang

tertunda.

1.2.2. Struktur I/O

Bagian ini akan membahas struktur I/O, interupsi I/O, dan DMA, serta perbedaan dalam penanganan interupsi.

1.2.2.1. Interupsi I/O

Untuk memulai operasi I/O, CPU me-*load* register yang bersesuaian ke *device controller*. Sebaliknya *device controller*

memeriksa isi register untuk kemudian menentukan operasi apa yang harus dilakukan.

Pada saat operasi I/O dijalankan ada dua kemungkinan, yaitu synchronous I/O dan asynchronous I/O. Pada synchronous I/O, kendali dikembalikan ke proses pengguna setelah proses I/O selesai dikerjakan. Sedangkan pada *asynchronous I/O*, kendali dikembalikan ke proses pengguna tanpa menunggu proses I/O selesai. Sehingga proses I/O dan proses pengguna dapat dijalankan secara bersamaan.

1.2.2.2. Struktur DMA

Direct Memory Access (DMA) suatu metoda penanganan I/O dimana *device controller* langsung berhubungan dengan memori tanpa campur tangan CPU. Setelah men-set *buffers*, *pointers*, dan *counters* untuk perangkat I/O, *device controller* mentransfer blok data langsung ke penyimpanan tanpa campur tangan CPU. DMA digunakan untuk perangkat I/O dengan kecepatan tinggi. Hanya terdapat satu interupsi setiap blok, berbeda dengan perangkat yang mempunyai kecepatan rendah dimana interupsi terjadi untuk setiap *byte* (*word*).

1.2.3. Struktur Penyimpanan

Program komputer harus berada di memori utama (biasanya RAM) untuk dapat dijalankan. Memori utama adalah satu-satunya tempat penyimpanan yang dapat diakses secara langsung oleh prosesor. Idealnya program dan data secara keseluruhan dapat disimpan dalam memori utama secara permanen. Namun demikian hal ini tidak mungkin karena:

- Ukuran memori utama relatif kecil untuk dapat menyimpan data dan program secara keseluruhan.
- Memori utama bersifat *volatile*, tidak bisa menyimpan secara permanen, apabila komputer dimatikan maka data yang tersimpan di memori utama akan hilang.

1.2.3.1. Memori Utama

Hanya memori utama dan register merupakan tempat penyimpanan yang dapat diakses secara langsung oleh prosesor. Oleh karena itu instruksi dan data yang akan dieksekusi harus disimpan di memori utama atau register.

Untuk mempermudah akses perangkat I/O ke memori, pada arsitektur komputer menyediakan fasilitas pemetaan memori ke I/O. Dalam hal ini sejumlah alamat di memori dipetakan dengan *device register*. Membaca dan menulis pada alamat memori ini menyebabkan data ditransfer dari dan ke device register. Metode

ini cocok untuk perangkat dengan waktu respon yang cepat seperti video controller.

Register yang terdapat dalam prosesor dapat diakses dalam waktu 1 clock cycle. Hal ini menyebabkan register merupakan media penyimpanan dengan akses paling cepat bandingkan dengan memori utama yang membutuhkan waktu relatif lama. Untuk mengatasi perbedaan kecepatan, dibuatlah suatu penyangga (*buffer*) penyimpanan yang disebut *cache*.

1.2.3.2. *Magnetic Disk*

Magnetic Disk berperan sebagai *secondary storage* pada sistem komputer modern. *Magnetic Disk* disusun dari piringan-piringan seperti CD. Kedua permukaan piringan diselimuti oleh bahan-bahan magnetik. Permukaan dari piringan dibagi-bagi menjadi *track* yang memutar, yang kemudian dibagi lagi menjadi beberapa sektor.

1.2.4. *Storage Hierarchy*

Dalam *storage hierarchy structure*, data yang sama bisa tampil dalam level berbeda dari sistem penyimpanan. Sebagai contoh integer A berlokasi pada bekas B yang ditambahkan 1, dengan asumsi bekas B terletak pada *magnetic disk*. Operasi penambahan diproses dengan pertama kali mengeluarkan operasi I/O untuk menduplikat disk block pada A yang terletak pada memori utama Operasi ini diikutidengan kemungkinan penduplikatan A ke dalam *cache* dan penduplikatan A ke dalam internal register. Sehingga penduplikatan A terjadi di beberapa tempat. Pertama terjadi di internal register dimana nilai A berbeda dengan yang di sistem penyimpanan. Dan nilai di A akan kembali sama ketika nilai baru ditulis ulang ke magnetic disk.

Pada kondisi multi prosesor, situasi akan menjadi lebih rumit. Hal ini disebabkan masing-masing prosesor mempunyai local cache. Dalam kondisi seperti ini hasil duplikat dari A mungkin hanya ada di beberapa *cache*. Karena CPU (register-register) dapat dijalankan secara bersamaan maka kita harus memastikan perubahan nilai A pada satu *cache* akan mengubah nilai A pada semua *cache* yang ada. Hal ini disebut sebagai *Cache Coherency*.

1.2.5. Proteksi Perangkat Keras

Sistem komputer terdahulu berjenis *programmer-operated systems*. Ketika komputer dioperasikan dalam konsul mereka (pengguna) harus melengkapi sistem terlebih dahulu. Akan tetapi setelah sistem operasi lahir maka hal tersebut diambil alih oleh

sistem operasi. Sebagai contoh pada monitor yang proses I/O sudah diambil alih oleh sistem operasi, padahal dahulu hal ini dilakukan oleh pengguna.

Untuk meningkatkan utilisasi sistem, sistem operasi akan membagi sistem sumber daya sepanjang program secara simultan. Pengertian *spooling* adalah suatu program dapat dikerjakan walau pun I/O masih mengerjakan proses lainnya dan disk secara bersamaan menggunakan data untuk banyak proses. Pengertian *multi programming* adalah kegiatan menjalankan beberapa program pada memori pada satu waktu.

Pembagian ini memang menguntungkan sebab banyak proses dapat berjalan pada satu waktu akan tetapi mengakibatkan masalah-masalah baru. Ketika tidak di *sharing* maka jika terjadi kesalahan hanyalah akan membuat kesalahan program. Tapi jika di-*sharing* jika terjadi kesalahan pada satu proses/ program akan berpengaruh pada proses lainnya.

Sehingga diperlukan pelindung (proteksi). Tanpa proteksi jika terjadi kesalahan maka hanya satu saja program yang dapat dijalankan atau seluruh output pasti diragukan.

Banyak kesalahan pemprograman dideteksi oleh perangkat keras. Kesalahan ini biasanya ditangani oleh sistem operasi. Jika terjadi kesalahan program, perangkat keras akan meneruskan kepada sistem operasi dan sistem operasi akan menginterupsi dan mengakhirinya. Pesan kesalahan disampaikan, dan memori dari program akan dibuang. Tapi memori yang terbuang biasanya tersimpan pada disk agar *programmer* bisa membetulkan kesalahan dan menjalankan program ulang.

1.2.5.1. Operasi Dual Mode

Untuk memastikan operasi berjalan baik kita harus melindungi sistem operasi, program, dan data dari program-program yang salah. Proteksi ini memerlukan share resources. Hal ini bisa dilakukan sistem operasi dengan cara menyediakan pendukung perangkat keras yang mengizinkan kita membedakan mode pengeksekusian program.

Mode yang kita butuhkan ada dua mode operasi yaitu:

- Mode Monitor.
- Mode Pengguna.

Pada perangkat keras akan ada bit atau Bit Mode yang berguna untuk membedakan mode apa yang sedang digunakan dan apa yang sedang dikerjakan. Jika Mode Monitor maka akan benilai 0, dan jika Mode Pengguna maka akan bernilai 1.

Pada saat *boot time*, perangkat keras bekerja pada mode monitor dan setelah sistem operasi di-*load* maka akan mulai masuk ke

mode pengguna. Ketika terjadi *trap* atau interupsi, perangkat keras akan men-*switch* lagi keadaan dari mode pengguna menjadi mode monitor (terjadi perubahan *state* menjadi bit 0). Dan akan kembali menjadi mode pengguna jikalau sistem operasi mengambil alih proses dan kontrol komputer (*state* akan berubah menjadi bit 1).

1.2.5.2. Proteksi I/O

Pengguna bisa mengacaukan sistem operasi dengan melakukan instruksi I/O ilegal dengan mengakses lokasi memori untuk sistem operasi atau dengan cara hendak melepaskan diri dari prosesor. Untuk mencegahnya kita menganggap semua instruksi I/O sebagai *privilidge* instruction sehingga mereka tidak bisa mengerjakan instruksi I/O secara langsung ke memori tapi harus lewat sistem operasi terlebih dahulu. Proteksi I/O dikatakan selesai jika pengguna dapat dipastikan tidak akan menyentuh mode monitor. Jika hal ini terjadi proteksi I/O dapat dikompromikan.

1.2.5.3. Proteksi Memori

Salah satu proteksi perangkat keras ialah dengan proteksi memori yaitu dengan pembatasan penggunaan memori. Disini diperlukan beberapa istilah yaitu:

- Base Register yaitu alamat memori fisik awal yang dialokasikan/ boleh digunakan oleh pengguna.
- Limit Register yaitu nilai batas dari alamat memori fisik awal yang dialokasikan/boleh digunakan oleh pengguna
- Proteksi Perangkat Keras.

Sebagai contoh sebuah pengguna dibatasi mempunyai base register 300040 dan mempunyai limit register 120900 maka pengguna hanya diperbolehkan menggunakan alamat memori fisik antara 300040 hingga 420940 saja.

1.3. Struktur Sistem Operasi

1.3.1. Komponen-komponen Sistem

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

- Managemen Proses.
- Managemen Memori Utama.
- Managemen *Secondary-Storage*.
- Managemen Sistem I/O.
- Managemen Berkas.

- Sistem Proteksi.
- Jaringan.
- *Command-Interpreter system*.

1.3.2. Managemen Proses

Proses adalah keadaan ketika sebuah program sedang di eksekusi. Sebuah proses membutuhkan beberapa sumber daya untuk menyelesaikan tugasnya. sumber daya tersebut dapat berupa *CPU time*, memori, berkas-berkas, dan perangkat-perangkat I/O.

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

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

1.3.3. Managemen Memori Utama

Memori utama atau lebih dikenal sebagai memori adalah sebuah *array* yang besar dari *word* atau *byte*, yang ukurannya mencapai ratusan, ribuan, atau bahkan jutaan. Setiap *word* atau *byte* mempunyai alamat tersendiri. Memori Utama berfungsi sebagai tempat penyimpanan yang akses datanya digunakan oleh CPU atau perangkat I/O. Memori utama termasuk tempat penyimpanan data yang sementara (*volatile*), artinya data dapat hilang begitu sistem dimatikan. Sistem operasi bertanggung jawab atas aktivitas-aktivitas yang berkaitan dengan managemen memori seperti:

- Menjaga *track* dari memori yang sedang digunakan dan siapa yang menggunakannya.
- Memilih program yang akan di-*load* ke memori.
- Mengalokasikan dan meng-dealokasikan ruang memori sesuai kebutuhan.

1.3.4. Managemen *Secondary-Storage*

Data yang disimpan dalam memori utama bersifat sementara dan jumlahnya sangat kecil. Oleh karena itu, untuk meyimpan keseluruhan data dan program komputer dibutuhkan *secondary-storage* yang bersifat permanen dan mampu menampung banyak data. Contoh dari *secondary-storage* adalah *harddisk*, disket, dll.

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

1.3.5. Managemen Sistem I/O

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

Komponen Sistem Operasi untuk sistem I/O:

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

1.3.6. Managemen Berkas

Berkas adalah kumpulan informasi yang berhubungan sesuai dengan tujuan pembuat berkas tersebut. Berkas dapat mempunyai struktur yang bersifat hirarkis (direktori, volume, dll.).

Sistem operasi bertanggung-jawab:

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

1.3.7. Sistem Proteksi

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

- membedakan antara penggunaan yang sudah diberi izin dan yang belum.
- *specify the controls to be imposed*.
- *provide a means of enforcement*.

1.3.8. Jaringan

Sistem terdistribusi adalah sekumpulan prosesor yang tidak berbagi memori atau *clock*. Tiap prosesor mempunyai memori sendiri. Prosesor-prosesor tersebut terhubung melalui jaringan komunikasi Sistem terdistribusi menyediakan akses pengguna ke bermacam sumber-daya sistem. Akses tersebut menyebabkan:

- *Computation speed-up*.
- *Increased data availability*.
- *Enhanced reliability*.

1.3.9. *Command-Interpreter System*

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

1.3.10. Layanan Sistem Operasi

Eksekusi program adalah kemampuan sistem untuk "*load*" program ke memori dan menjalankan program. Operasi I/O: pengguna tidak dapat secara langsung mengakses sumber daya perangkat keras, sistem operasi harus menyediakan mekanisme untuk melakukan operasi I/O atas nama pengguna. Sistem manipulasi berkas dalah kemampuan program untuk operasi pada berkas (membaca, menulis, membuat, dan menghapus berkas). Komunikasi adalah pertukaran data/ informasi antar dua atau lebih proses yang berada pada satu komputer (atau lebih). Deteksi *error* adalah menjaga kestabilan sistem dengan mendeteksi "*error*", perangkat keras mau pun operasi.

Efesisensi penggunaan sistem:

- *Resource allocator* adalah mengalokasikan sumber-daya ke beberapa pengguna atau *job* yang jalan pada saat yang bersamaan.
- Proteksi menjamin akses ke sistem sumber daya dikendalikan (pengguna dikontrol aksesnya ke sistem).
- *Accounting* adalah merekam kegiatan pengguna, jatah pemakaian sumber daya (keadilan atau kebijaksanaan).

1.3.11. *System Calls*

System call menyediakan interface antara program (program pengguna yang berjalan) dan bagian OS.a*System call* menjadi jembatan antara proses dan sistem operasi. *System call* ditulis dalam bahasaa*assembly* atau bahasa tingkat tinggi yang dapat mengendalikan mesin (C). Contoh: UNIX menyediakan *system call*: *read, write* => operasi I/O untuk berkas.

Sering pengguna program harus memberikan data (parameter) ke OS yang akan dipanggil. Contoh pada UNIX: read(buffer, max_size, file_id);

Tiga cara memberikan parameter dari program ke sistem operasi:

- Melalui registers (sumber daya di CPU).

- Menyimpan parameter pada data struktur (table) di memori, dan alamat table tsb ditunjuk oleh *pointer* yang disimpan di register.
- *Push* (*store*) melalui "*stack*" pada memori dan OS mengambilnya melalui *pop* pada *stack* tsb.

1.3.12. Mesin Virtual

Sebuah mesin virtual (*Virtual Machine*) menggunakan misalkan terdapat sistem program => control program yang mengatur pemakaian sumber daya perangkat keras. Control program = trap *System call* + akses ke perangkat keras. Control program memberikan fasilitas ke proses pengguna. Mendapatkan jatah CPU dan memori. Menyediakan *interface* "identik" dengan apa yang disediakan oleh perangkat keras => *sharing devices* untuk berbagai proses.

Mesin Virtual (MV) (MV) => control program yang minimal MV memberikan ilusi *multitasking*: seolah-olah terdapat prosesor dan memori ekslusif digunakan MV. MV memilah fungsi *multitasking* dan implementasi *extended machine* (tergantung proses pengguna) => flexible dan lebih mudah untuk pengaturan. Jika setiap pengguna diberikan satu MV => bebas untuk menjalankan OS (kernel) yang diinginkan pada MV tersebut. Potensi lebih dari satu OS dalam satu komputer. Contoh: IBM VM370: menyediakan MV untuk berbagai OS: CMS (interaktif), MVS, CICS, dll. Masalah: Sharing disk => OS mempunyai sistem berkas yang mungkin berbeda. IBM: virtual disk (minidisk) yang dialokasikan untuk pengguna melalui MV.

Konsep MV menyediakan proteksi yang lengkap untuk sumberdaya sistem, dikarenakan tiap MV terpisah dari MV yang lain. Namun, hal tersebut menyebabkan tidak adanya *sharing* sumberdaya secara langsung. MV merupakan alat yang tepat untuk penelitian dan pengembangan sistem operasi. Konsep MV susah untuk diimplementasi sehubungan dengan usaha yang diperlukan untuk menyediakan duplikasi dari mesin utama.

1.3.13. Perancangan Sistem dan Implementasi

Target untuk pengguna: sistem operasi harus nyaman digunakan, mudah dipelajari, dapat diandalkan, aman dan cepat. Target untuk sistem: sistem operasi harus gampang dirancang, diimplementasi, dan dipelihara, sebagaimana fleksibel, *error*, dan efisien.

Mekanisme dan Kebijaksanaan:

- Mekanisme menjelaskan bagaimana melakukan sesuatu kebijaksanaan memutuskan apa yang akan
- dilakukan. Pemisahan kebijaksanaan dari mekanisme merupakan hal yang sangat penting; ini mengizinkan

fleksibilitas yang tinggi bila kebijaksanaan akan diubah nanti.

- Kebijaksanaan memutuskan apa yang akan dilakukan.

Pemisahan kebijaksanaan dari mekanisme merupakan hal yang sangat penting; ini mengizinkanfleksibilitas yang tinggi bila kebijaksanaan akan diubah nanti.

Implementasi Sistem biasanya menggunakan bahas assembly, sistem operasi sekarang dapat ditulis dengan menggunakan bahasa tingkat tinggi. Kode yang ditulis dalam bahasa tingkat tinggi: dapat dibuat dengan cepat, lebih ringkas, lebih mudah dimengerti dan didebug. Sistem operasi lebih mudah dipindahkan ke perangkat keras yang lain bila ditulis dengan bahasa tingkat tinggi.

1.3.14. *System Generation* (SYSGEN)

Sistem operasi dirancang untuk dapat dijalankan di berbagai jenis mesin; sistemnya harus di konfigurasi untuk tiap komputer. Program SYSGEN mendapatkan informasi mengenai konfigurasi khusus dari sistem perangkat keras.

- *Booting*: memulai komputer dengan me-*load* kernel.
- *Bootstrap program*: kode yang disimpan di code ROM yang dapat menempatkan kernel, memasukkannya kedalam memori, dan memulai eksekusinya.

1.4. Rangkuman

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

Pada awalnya, sistem komputer digunakan dari depan konsul. Perangkat lunak seperti assembler, loader, linkerdan compiler meningkatkan kenyamanan dari sistem pemrograman, tapi juga memerlukan waktu set-up yang banyak. Untuk mengurangi waktu set-up tersebut, digunakan jasa operator dan menggabungkan tugas-tugas yang sama (sistem batch). Sistem batch mengizinkan pengurutan tugas secara otomatis dengan menggunakan sistem operasi yang resident dan memberikan peningkatan yang cukup besar dalam utilisasi komputer. Komputer tidak perlu lagi menunggu operasi oleh pengguna. Tapi utilisasi CPU tetap saja rendah. Hal ini dikarenakan lambatnya kecepatan alat-alat untuk I/O 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 multi-programming. Dengan multiprogramming, beberapa tugas disimpan dalam memori dalam satu waktu; CPU digunakan secara bergantian sehingga menambah utilisasi CPU dan mengurangi total waktu yang dibutuhkan untuk menyelesaikan tugas-tugas tersebut. Multiprogramming, yang dibuat untuk meningkatkan kemampuan, juga mengizinkan time sharing. Sistem operasi yang bersifat time-shared memperbolehkan banyak pengguna untuk menggunakan komputer secara interaktif pada saat yang bersamaan. Komputer Personal adalah mikrokomputer yang dianggap lebih kecil dan lebih murah dibandingkan komputer mainframe. Sistem operasi untuk komputer-komputer seperti ini diuntungkan oleh pengembangan sistem operasi untuk komputer mainframe dalam beberapa hal. Namun, semenjak penggunaan komputer untuk keperluan pribadi, maka utilisasi CPU tidak lagi menjadi perhatian utama. Karena itu, beberapa desain untuk komputer mainframe tidak cocok untuk sistem yang lebih kecil.

Sistem parallel mempunyai lebih dari satu CPU yang mempunyai hubungan yang erat; CPU-CPU tersebut berbagi bus komputer, dan kadang-kadang berbagi memori dan perangkat yang lainnya. Sistem seperti itu dapat meningkatkan *throughput* dan reliabilititas. **Sistem *hard real-time*** sering kali digunakan sebagai alat pengontrol untuk applikasi yang dedicated. Sistem operasi yang ***hard real-time*** mempunyai batasan waktu yang tetap yang sudah didefinisikan dengan baik.Pemrosesan harus selesai dalam batasan-batasan yang sudah didefinisikan, atau sistem akan gagal. **Sistem *soft real-time*** mempunyai lebih sedikit batasan waktu yang keras, dan tidak mendukung penjadwalan dengan menggunakan batas akhir. Pengaruh dari internet dan *World Wide Web*baru-baru ini telah mendorong pengembangan sistem operasi modern yang menyertakan web browser serta perangkat lunak jaringan dan komunikasi sebagai satu kesatuan.

Multiprogramming dan sistem time-sharing meningkatkan kemampuan komputer dengan melampaui batas operasi (overlap) CPU dan I/O dalam satu mesin. Hal seperti itu memerlukan perpindahan dataantara CPU dan alat I/O, ditangani baik dengan polling atau interrupt-driven akses ke I/O port, atau dengan perpindahan DMA. Agar komputer dapat menjalankan suatu program, maka program tersebut harus berada di memori utama (memori utama). Memori utama adalah satu-satunya tempat penyimpanan yang besar yang dapat diakses secara langsung oleh prosessor, merupakan suatu array dari word atau byte, yang mempunyai ukuran ratusan sampai jutaan ribu. Setiap word memiliki alamatnya

sendiri. Memori utama adalah tempat penyimpanan yang volatile, dimana isinya hilang bila sumber energinya (energi listrik) dimatikan. Kebanyakan sistem komputer menyediakan secondary storage sebagai perluasan dari memori utama. Syarat utama dari secondary storage adalah dapat menyimpan data dalam jumlah besar secara permanen. Secondary storage yang paling umum adalah disk magnetik, yang meyediakan penyimpanan untuk program mau pun data. Disk magnetik adalah alat penyimpanan data yang nonvolatile 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).

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 I/O 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. Sistem operasi menyediakan banyak pelayanan. Di tingkat terrendah, sistem calls mengizinkan program yang sedang berjalan untuk membuat permintaan secara langsung dari sistem operasi. Di tingkat tertinggi, command interpreter atau shell menyediakan mekanisme agar pengguna dapat membuat permintaan tanpa menulis program. Command dapat muncul dari bekas sewaktu jalankan batch-mode, atau secara langsung dari terminal ketika dalam mode interaktive atau time-shared. Program sistem disediakan untuk memenuhi kebanyakan dari permintaan pengguna. Tipe dari permintaan beragam sesuai dengan levelnya. Level sistem call harus menyediakan fungsi dasar, seperti kontrol proses serta manipulasi alat dan bekas. Permintaan dengan level yang lebih tinggi (command interpreter atau program sistem) diterjemahkan kedalam urutan sistem call.

Pelayanan sistem dapat dikelompokkan kedalam beberapa kategori: kontrol program, status permintaan dan permintaan I/O. Program error dapat dipertimbangkan sebagai permintaan yang implisit untuk pelayanan. Bila sistem pelayanan sudah terdefinisi, maka struktur dari sistem operasi dapat dikembangkan. Berbagai

macam tabel diperlukan untuk menyimpan informasi yang mendefinisikan status dari sistem komputer dan status dari sistem tugas. Perancangan dari suatu sistem operasi yang baru merupakan tugas yang utama. Sangat penting bahwa tujuan dari sistem sudah terdefinisi dengan baik sebelum memulai perancangan. Tipe dari sistem yang diinginkan adalah landasan dalam memilih beragam algoritma dan strategi yang akan digunakan. Karena besarnya sistem operasi, maka modularitas adalah hal yang penting. Merancang sistem sebagai suatu urutan dari layer atau dengan menggunakan mikrokernel merupakan salah satu teknik yang baik. Konsep virtual machine mengambil pendekatan layer dan memperlakukan baik itu kernel dari sistem operasi dan perangkat kerasnya sebagai suatu perangkat keras. Bahkan sistem operasi yang lain dapat dimasukkan diatas virtual machine tersebut. Setiap sistem operasi yang mengimplemen JVM dapat menjalankan semua program java, karena JVM mendasari dari sistem ke program java, menyediakan arsitektur tampilan yang netral.

Didalam daur perancangan sistem operasi, kita harus berhati-hati untuk memisahkan pembagian kebijakan (policy decision) dengan detail dari implementasi (mechanism). Pemisahan ini membuat fleksibilitas yang maksimal apabila policy decision akan diubah kemudian. Sistem operasi sekarang ini hampir selalu ditulis dengan menggunakan bahasa tingkat tinggi. Hal ini meningkatkan implementasi, perawatan portabilitas. Untuk membuat sistem operasi untuk suatu konfigurasi mesin tertentu, kita harus melakukan *system generation*.

1.5. Pertanyaan

1. Sebutkan tiga tujuan utama dari sistem operasi!
2. Sebutkan keuntungan dari *multiprogramming*!
3. Sebutkan perbedaan utama dari sistem operasi antara komputer *mainframe* dan PC?
4. Sebutkan kendala-kendala yang harus diatasi oleh *programmer* dalam menulis sistem operasi untuk lingkungan waktu nyata?
5. Jelaskan perbedaan antara *symmetric* dan *asymmetric multiprocessing*. Sebutkan keuntungan dan kerugian dari sistem *multiprosessor*!
6. Apakah perbedaan antara *trap* dan *interrupt*? Sebutkan penggunaan dari setiap fungsi tersebut!
7. Untuk jenis operasi apakah **DMA** itu berguna? Jelaskan jawabannya!

8. Sebutkan dua kegunaan dari *memory cache*! Problem apakah yang dapat dipecahkan dan juga muncul dengan adanya *cache* tersebut?
9. Beberapa **CPU** menyediakan lebih dari dua mode operasi. Sebutkan dua kemungkinan penggunaan dari mode tersebut?
10. Sebutkan lima kegiatan utama dari sistem operasi yang berhubungan dengan managemen proses!
11. Sebutkan tiga kegiatan utama dari sistem operasi yang berhubungan dengan managemen memori!
12. Sebutkan tiga kegiatan utama dari sistem operasi yang berhubungan dengan managemen *secondary-storage*!
13. Sebutkan lima kegiatan utama dari sistem operasi yang berhubungan dengan managemen berkas!
14. Apakah tujuan dari *command interpreter*? Mengapa biasanya hal tersebut terpisah dengan *kernel*?

1.6. Rujukan

1. *http://www.csc.uvic.ca/~mcheng/360/notes/NOTES2.html*
2. *http://www.chipcenter.com/circuitcellar/march02/c0302dc4.htm*
3. *http://www.osdata.com/kind/history.htm*
4. *http://www.imm.dtu.dk/courses/02220/OS/OH/week7.pdf*
5. *http://www.mcsr.olemiss.edu/unixhelp/concepts/history.html*
6. *http://www.cs.panam.edu/fox/CSCI4334/ch3.ppt*
7. *http://www.cis.umassd.edu/~rbalasubrama/*
8. *http://legion.virginia.edu/presentations/sc2000/sld001.htm*
9. *http://www.cs.wpi.edu/~cs502/s99/*
10. *http://cs-www.cs.yale.edu/homes/avi/os-book/osc/slide-dir/*
11. *http://www.hardware.fr/articles/338/page1.html*
12. *http://www.cs.technion.ac.il/~hagit/OSS98* (
13. *http://www.ignou.ac.in/virtualcampus/adit/course/index-tr1.htm*
14. *http://www.techrescue.net/guides/insthware.asp*

15. *http://agt.buka.org/concept.html*

16. *http://kos.enix.org/pub/greenwald96synergy.pdf* (

Bab 2

Proses dan Thread

2.1. Proses

Satu selingan pada diskusi kita mengenai sistem operasi yaitu bahwa ada sebuah pertanyaan mengenai apa untuk menyebut semua aktivitas CPU. Sistem batch mengeksekusi jobs, sebagaimana suatu sistem *time-shared* telah menggunakan program pengguna, atau tugas-tugas/ pekerjaan-pekerjaan. Bahkan pada sistem tunggal, seperti *Microsoft Windows* dan *Macintosh OS*, seorang pengguna mampu untuk menjalankan beberapa program pada saat yang sama: sebuah *Word Processor*, *Web Browser*, dan paket *e-mail*. Bahkan jika pengguna dapat melakukan hanya satu program pada satu waktu, sistem operasi perlu untuk mendukung aktivitas program internalnya sendiri, seperti managemen memori. Dalam banyak hal, seluruh aktivitas ini adalah serupa, maka kita menyebut seluruh program itu proses-proses (processes).

Istilah job dan proses digunakan hampir dapat dipertukarkan pada tulisan ini. Walau kami pribadi lebih menyukai istilah proses, banyak teori dan terminologi sistem-operasi dikembangkan selama suatu waktu ketika aktivitas utama sistem operasi adalah job processing. Akan menyesatkan untuk menghindari penggunaan istilah umum yang telah diterima bahwa memasukkn kata job (seperti penjadualan job) hanya karena proses memiliki job pengganti/ pendahulu.

2.1.1. Konsep Dasar dan Definisi Proses

Secara informal; proses adalah program dalam eksekusi. Suatu proses adalah lebih dari kode program, dimana kadang kala dikenal sebagai bagian tulisan. Proses juga termasuk aktivitas yang sedang terjadi, sebagaimana digambarkan oleh nilai pada program counter dan isi dari daftar prosesor/ processor’s register. Suatu proses umumnya juga termasuk process stack, yang

berisikan data temporer (seperti parameter metoda, address yang kembali, dan variabel lokal) dan sebuah data section, yang berisikan variabel global.

Kami tekankan bahwa program itu sendiri bukanlah sebuah proses; suatu program adalah satu entitas pasif; seperti isi dari sebuah berkas yang disimpan didalam disket, sebagaimana sebuah proses dalam suatu entitas aktif, dengan sebuah program counter yang mengkhususkan pada instruksi selanjutnya untuk dijalankan dan seperangkat sumber daya/ resource yang berkenaan dengannya.

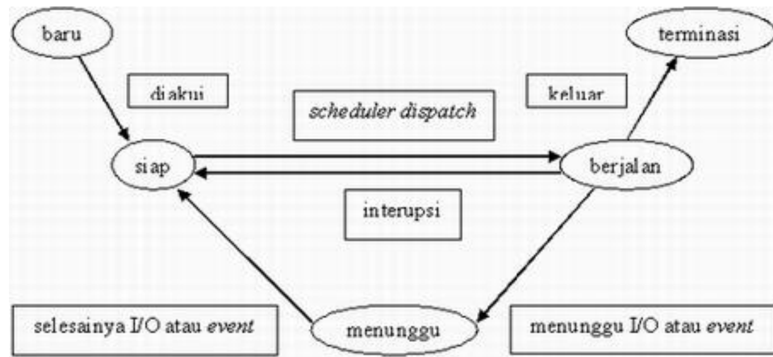
Walau dua proses dapat dihubungkan dengan program yang sama, program tersebut dianggap dua urutan eksekusi yang berbeda. Sebagai contoh, beberapa pengguna dapat menjalankan copy yang berbeda pada mail program, atau pengguna yang sama dapat meminta banyak copy dari program editor. Tiap-tiap proses ini adakah proses yang berbeda dan walau bagian tulisan-text adalah sama, data section bervariasi. Juga adalah umum untuk memiliki proses yang menghasilkan banyak proses begitu ia bekerja. Kami mendiskusikan masalah tersebut pada Bagian 2.4.

2.1.2. Keadaan Proses

Sebagaimana proses bekerja, maka proses tersebut merubah state (keadaan statis/ asal). Status dari sebuah proses didefinisikan dalam bagian oleh aktivitas yang ada dari proses tersebut. Tiap proses mungkin adalah satu dari keadaan berikut ini:

- *New*: Proses sedang dikerjakan/ dibuat.
- *Running*: Instruksi sedang dikerjakan.
- *Waiting*: Proses sedang menunggu sejumlah kejadian untuk terjadi (seperti sebuah penyelesaian I/Oatau penerimaan sebuah tanda/ signal).
- *Ready*: Proses sedang menunggu untuk ditugaskan pada sebuah prosesor.
- *Terminated*: Proses telah selsesai melaksanakan tugasnya/ mengeksekusi.

Nama-nama tersebut adalah arbitrer/ berdasar opini, istilah tersebut bervariasi disepanjang sistem operasi. Keadaan yang mereka gambarkan ditemukan pada seluruh sistem. Namun, sistem operasi tertentu juga lebih baik menggambarkan keadaan/ status proses. Adalah penting untuk menyadari bahwa hanya satu proses dapat berjalan pada prosesor mana pun pada waktu kapan pun. Namun, banyak proses yang dapat ready atau waiting. Keadaan diagram yang berkaitan dangan keadaan tersebut dijelaskan pada Gambar 2-1.



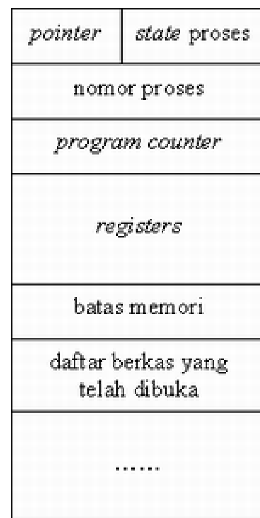
Gambar 2-1. Keadaan Proses

2.1.3. *Process Control Block*

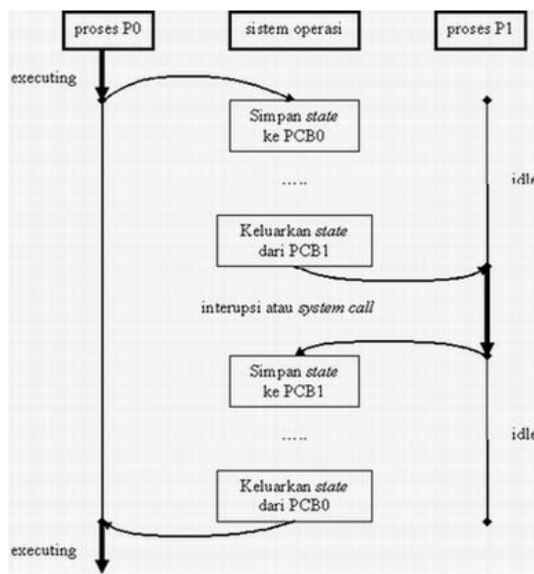
Tiap proses digambarkan dalam sistem operasi oleh sebuah *process control block* (PCB) - juga disebut sebuah *control block*. Sebuah PCB ditunjukkan dalam Gambar 2-2. PCB berisikan banyak bagian dari informasi yang berhubungan dengan sebuah proses yang spesifik, termasuk ini:

- Keadaan proses: Keadaan mungkin, *new*, *ready*, *running*, *waiting*, *halted*, dan juga banyak lagi.
- *Program counter*: *Counter* mengindikasikan address dari perintah selanjutnya untuk dijalankan untuk proses ini.
- CPU register: Register bervariasi dalam jumlah dan jenis, tergantung pada rancangan komputer.
- Register tersebut termasuk accumulator, index register, stack pointer, general-puposes register, ditambah code information pada kondisi apa pun. Besertaan dengan program counter, keadaan/ status informasi harus disimpan ketika gangguan terjadi, untuk memungkinkan proses tersebut berjalan/bekerja dengan benar setelahnya (lihat Gambar 2-3).
- Informasi managemen memori: Informasi ini dapat termasuk suatu informasi sebagai nilai dari dasar dan batas register, tabel page/ halaman, atau tabel segmen tergantung pada sistem memori yangdigunakan oleh sistem operasi (lihat Bab 4).
- Informasi pencatatan: Informasi ini termasuk jumlah dari CPU dan waktu riil yang digunakan, batas waktu, jumlah akun, jumlah job atau proses, dan banyak lagi.
- Informasi status I/O: Informasi termasuk daftar dari perangkat I/O yang di gunakan pada proses ini, suatu daftar open berkas dan banyak lagi.

- PCB hanya berfungsi sebagai tempat menyimpan/ gudang untuk informasi apa pun yang dapat bervariasi dari prose ke proses.



Gambar 2-2. ***Process Control Block*.**



Gambar 2-3. ***CPU Register*.**

2.1.4. *Threads*

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

Kontrol thread tunggal ini hanya memungkinkan proses untuk menjalankan satu tugas pada satu waktu. Banyak sistem operasi modern telah memiliki konsep yang dikembangkan agar memungkinkan sebuah proses untuk memiliki eksekusi *multithreads*, agar dapat dapat secara terus menerus mengetik dalam karakter dan menjalankan pengecek ejaan didalam proses yang sama. Maka sistem operasi tersebut memungkinkan proses untuk menjalankan lebih dari satu tugas pada satu waktu. Pada Bagian 2.5 akan dibahas proses *multithreaded*.

2.2. Penjadualan Proses

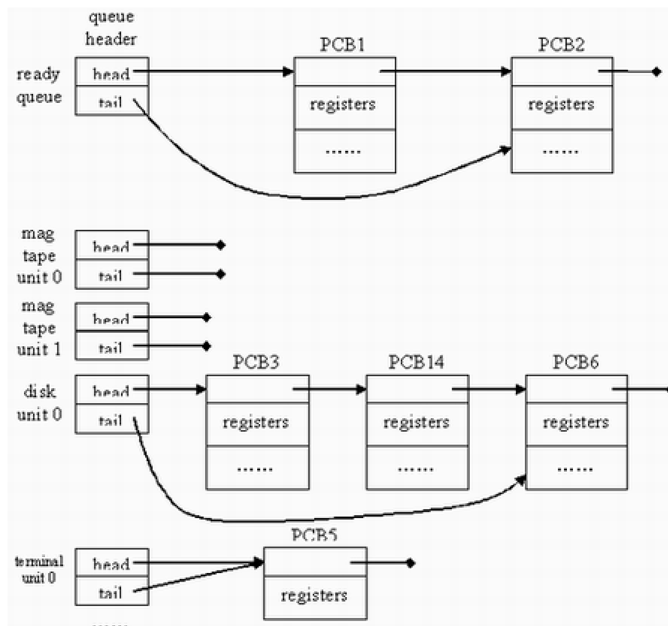
Tujuan dari multiprogramming adalah untuk memiliki sejumlah proses yang berjalan pada sepanjang waktu, untuk memaksimalkan penggunaan CPU. Tujuan dari pembagian waktu adalah untuk mengganti CPU diantara proses-proses yang begitu sering sehingga pengguna dapat berinteraksi dengan setiap program sambil CPU bekerja. Untuk sistem uniprosesor, tidak akan ada lebih dari satu proses berjalan. Jika ada proses yang lebih dari itu, yang lainnya akan harus menunggu sampai CPU bebas dan dapat dijadualkan kembali.

2.2.1. Penjadualan Antrian

Ketika proses memasuki sistem, mereka diletakkan dalam antrian job. Antrian ini terdiri dari seluruh proses dalam sistem. Proses yang hidup pada memori utama dan siap dan menunggu/ wait untuk mengeksekusi disimpan pada sebuah daftar bernama ready queue. Antrian ini biasanya disimpan sebagai daftar penghubung. Sebuah header ready queue berisikan penunjuk kepada PCB-PCB awal dan akhir. Setiap PCB memiliki pointer field yang menunjukkan proses selanjutnya dalam ready queue.

Juga ada antrian lain dalam sistem. Ketika sebuah proses mengalokasikan CPU, proses tersebut berjalan/bekerja sebentar lalu berhenti, di interupsi, atau menunggu suatu kejadian tertentu, seperti penyelesaian suatu permintaan I/O. Pada kasus ini sebuah permintaan I/O, permintaan seperti itu mungkin untuk sebuah tape drive yang telah diperuntukkan, atau alat yang berbagi, seperti disket. Karena ada banyak proses dalam sistem, disket bisa jadi sibuk dengan permintaan I/O untuk proses lainnya. Maka

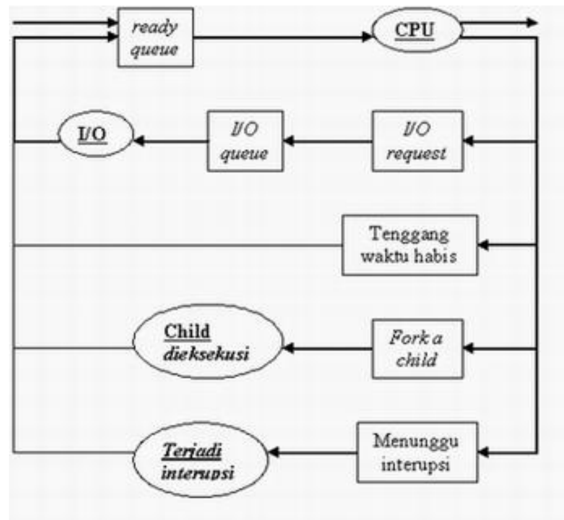
proses tersebut mungkin harus menunggu untuk disket tersebut. Daftar dari proses yang menunggu untuk peralatan I/O tertentu disebut sebuah device queue. Tiap peralatan memiliki device queuenya sendiri (Lihat Gambar 2-4).



Gambar 2-4. Device Queue.

Reprensentasi umum untuk suatu diskusi mengenai penjadualan proses adalah diagram antrian, seperti pada Gambar 2-5. Setiap kotak segi empat menunjukkan sebuah antrian. Dua tipe antrian menunjukan antrian yang siap dan suatu perangkat device queues. Lingkaran menunjukkan sumber-sumber yang melayani sistem. Sebuah proses baru pertama-tama ditaruh dalam ready queue. Lalu menunggu dalam ready queue sampai proses tersebut dipilih untuk dikerjakan/lakukan atau di dispatched. Begitu proses tersebut mengalokasikan CPU dan menjalankan/ mengeksekusi, satu dari beberapa kejadian dapat terjadi.

- Proses tersebut dapat mengeluarkan sebuah permintaan I/O, lalu di tempatkan dalam sebuah antrian I/O.
- Proses tersebut dapat membuat subproses yang baru dan menunggu terminasinya sendiri.
- Proses tersebut dapat digantikan secara paksa dari CPU, sebagai hasil dari suatu interupsi, dan diletakkan kembali dalam ready queue.



Gambar 2-5. Diagram Anrian.

Dalam dua kasus pertama, proses akhirnya berganti dari waiting state menjadi ready state, lalu diletakkan kembali dalam ready queue. Sebuah proses meneruskan siklus ini sampai berakhir, disaat dimana proses tersebut diganti dari seluruh queue dan memiliki PCB nya dan sumber-sumber/ resources dialokasikan kembali.

2.2.2. Penjadual

Sebuah proses berpindah antara berbagai penjadualan antrian selama umur hidupnya. Sistem operasi harus memilih, untuk keperluan penjadualan, memproses antrian-antrian ini dalam cara tertentu. Pemilihan proses dilaksanakan oleh penjadual yang tepat/ cocok. Dalam sistem batch, sering ada lebih banyak proses yang diserahkan daripada yang dapat dilaksanakan segera. Proses ini dipitakan/ disimpan pada suatu alat penyimpan masal (biasanya disket), dimana proses tersebut disimpan untuk eksekusi dilain waktu. Penjadualan long term, atau penjadual job, memilih proses dari pool ini dan mengisinya kedalam memori eksekusi.

Sebuah proses dapat mengeksekusi untuk hanya beberapa milidetik sebelum menunggu permintaan I/O. Seringkali, penjadualan shorterm mengeksekusi paling sedikit sekali setiap 100 milidetik. Karena durasi waktu yang pendek antara eksekusi, penjadualan shorterm haruslah cepat. Jika memerlukan 10 mili detik untuk menentukan suatu proses eksekusi selama 100 mili

detik, maka 10/(100 + 10) = 9 persen CPU sedang digunakan (terbuang) hanya untuk pekerjaan penjadualan.

Penjadualan longterm pada sisi lain, mengeksekusi jauh lebih sedikit. Mungkin ada beberapa menit antara pembuatan proses baru dalam sistem. Penjadualan longterm mengkontrol derajat multiprogramming (jumlah proses dalam memori). Jika derajat multiprogramming stabil, lalu tingkat rata-rata dari penciptaan proses harus sama dengan tingkat kepergian rata rata dari proses yang meninggalkan sistem. Maka penjadualan longterm mungkin diperlukan untuk dipanggil hanya ketika suatu proses meninggalkan sistem. Karena interval yang lebih panjang antara eksekusi, penjadualan longterm dapat memakai waktu yang lebih lama untuk menentukan proses mana yang harus dipilih untuk dieksekusi.

Adalah penting bagi penjadualan longterm membuat seleksi yang hati-hati. Secara umum, kebanyakan proses dapat dijelaskan sebagai I/O bound atau CPU bound. Sebuah proses I/O bound adalah salah satu yang membuang waktunya untuk mengerjakan I/O dari pada melakukan perhitungan. Suatu proses CPU-bound, pada sisi lain, adalah salah satu yang jarang menghasilkan permintaan I/O, menggunakan lebih banyak waktunya melakukan banyak komputasi daripada yang digunakan oleh proses I/O bound. Penting untuk penjadualan longterm memilih campuran proses yang baik antara proses I/O bound dan CPU bound. Jika seluruh proses adalah I/O bound, ready queue akan hampir selalu kosong, dan penjadualan short term akan memiliki sedikit tugas. Jika seluruh proses adalah CPU bound, I/O waiting queue akan hampir selalu kosong, peralatan akan tidak terpakai, dan sistem akan menjadi tidak imbang. Sistem dengan kinerja yang terbaik akan memiliki kombinasi proses CPU bound dan I/O bound.

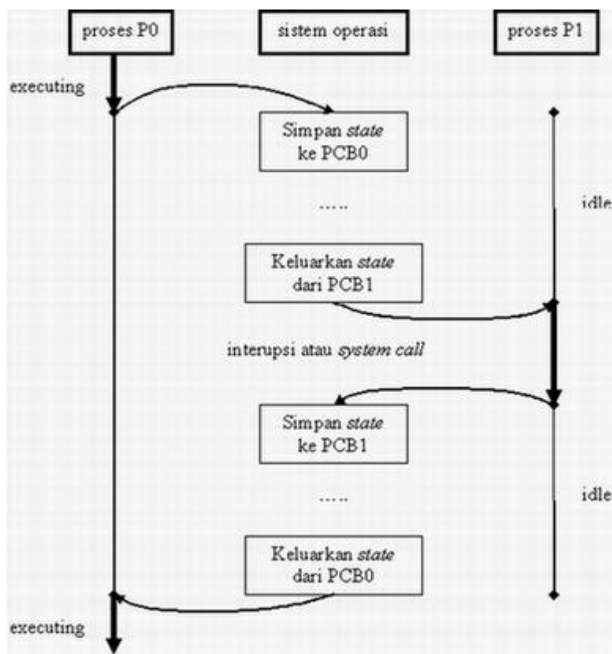
Gambar 2-6. Penjadual Medium-term.

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

Sebagian sistem operasi, seperti sistem time sharing, dapat memperkenalkan sebuah tambahan, penjadualan tingkat menengah. Penjadual medium-term ini digambarkan pada Gambar 2-5. Ide utama/kunci dibelakang sebuah penjadual medium term adalah kadang kala akan menguntungkan untuk memindahkan proses dari memori (dan dari pengisian aktif dari CPU), dan maka untuk mengurangi derajat dari multiprogramming. Dikemudian waktu, proses dapat diperkenalkan kedalam memori dan eksekusinya dapat dilanjutkan dimana proses itu di tinggalkan/ diangkat. Skema ini disebut *swapping*. Proses di *swapped out*, dan lalu di *swapped in*, oleh penjadual jangka menengah. *Swapping* mungkin perlu untuk meningkatkan pencampuran proses, atau karena suatu perubahan dalam persyaratan memori untuk dibebaskan. *Swapping* dibahas dalam Bagian 4.2.

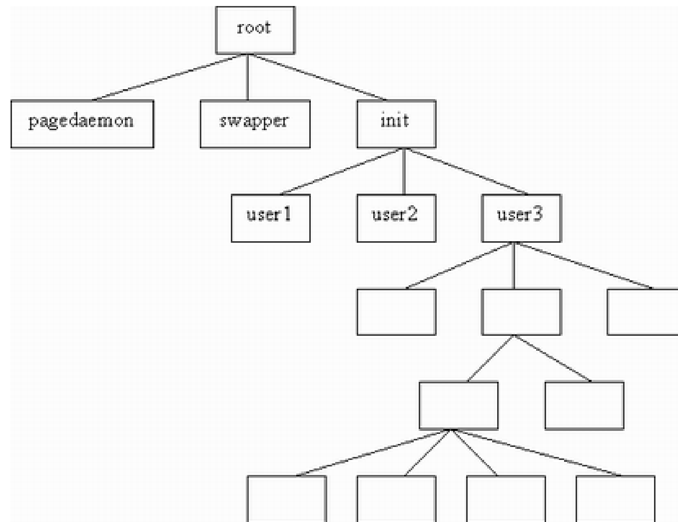
2.2.3. Alih Konteks

Mengganti CPU ke proses lain memerlukan penyimpanan suatu keadaan proses lama (*state of old process*) dan kemudian beralih ke proses yang baru. Tugas tersebut diketahui sebagai alih konteks (*context switch*). Alih konteks sebuah proses digambarkan dalam PCB suatu proses; termasuk nilai dari CPU register, status proses (lihat Gambar 2-7). dan informasi managemen memori. Ketika alih konteks terjadi, kernel menyimpan konteks dari proses lama kedalam PCB nya dan mengisi konteks yang telah disimpan dari process baru yang telah terjadual untuk berjalan. Pergantian waktu konteks adalah murni overhead, karena sistem melakukan pekerjaan yang tidak perlu. Kecepatannya bervariasi dari mesin ke mesin, bergantung pada kecepatan memori, jumlah register yang harus di copy, dan keberadaan instruksi khusus (seperti instruksi tunggal untuk mengisi atau menyimpan seluruh register). Tingkat kecepatan umumnya berkisar antara 1 sampai 1000 mikro detik



Gambar 2-7. Alih Konteks.

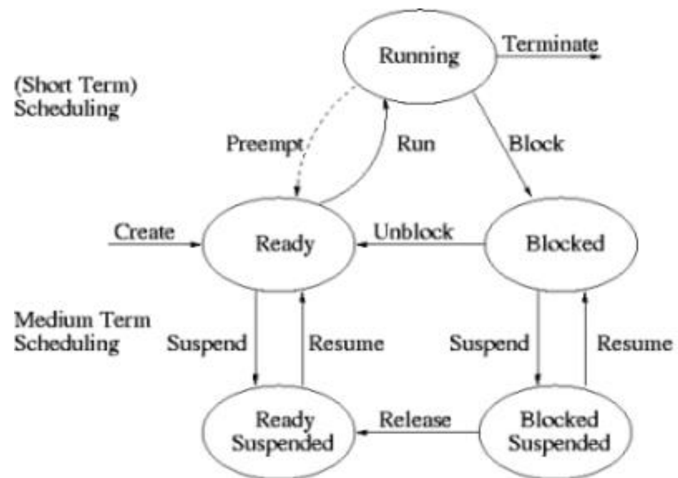
Waktu alih konteks sangat begantung pada dukungan perangkat keras. Sebagai contoh, prosesor seperti UltraSPARC menyediakan dua rangkap register. Sebuah alih konteks hanya memasukkan perubahan pointer ke perangkat register yang ada. Tentu saja, jika ada lebih proses-proses aktif yang ada dari pada yang ada di perangkat register, sistem menggunakan bantuan untuk meng-copy data register pada dan dari memori, sebagaimana sebelumnya. Semakin sistem operasi kompleks, makin banyak pekerjaan yang harus dilakukan selama alih konteks. Sebagaimana dilihat pada Bab 4, teknik managemen memori tingkat lanjut dapat mensyaratkan data tambahan untuk diganti dengan tiap konteks. Sebagai contoh, ruang alamat dari proses yang ada harus dijaga sebagai ruang pada pekerjaan berikutnya untuk digunakan. Bagaimana ruang alamat di jaga, berapa banyak pekerjaan dibutuhkan untuk menjaganya, tergantung pada metoda managemen memori dari sistem operasi. Sebagaimana akan kita lihat pada Bab 4, alih konteks telah menjadi suatu keharusan, bahwa programmer menggunakan struktur (*threads*) untuk menghindarinya kapan pun memungkinkan.



Gambar 2-8. Pohon Proses.

2.3. Operasi-Operasi Pada Proses

Proses dalam sistem dapat dieksekusi secara bersama-sama, proses tersebut harus dibuat dan dihapus secara dinamis. Maka, sistem operasi harus menyediakan suatu mekanisme umtuk pembuatan proses dan terminasi proses.



Gambar 2-9. Operasi pada Proses.

2.3.1. Pembuatan Proses

Suatu proses dapat membuat beberapa proses baru, melalui sistem pemanggilan pembuatan proses, selama jalur eksekusi. Pembuatan proses dinamakan induk proses, sebagaimana proses baru disebut anak dari proses tersbut. Tiap proses baru tersebut dapat membuat proses lainnya, membentuk suatu pohon proses (lihat Gambar 2-7).

Secara umum, suatu proses akan memerlukan sumber tertentu (waktu CPU, memori, berkas, perangkat I/O) untuk menyelesaikan tugasnya. Ketika suatu proses membuat sebuah subproses, sehingga subproses dapat mampu untuk memperoleh sumbernya secara langsung dari sistem operasi. Induk mungkin harus membatasi sumber diantara anaknya, atau induk dapat berbagi sebagian sumber (seperti memori berkas) diantara beberapa dari anaknya. Membatasi suatu anak proses menjadi subset sumber daya induknya mencegah proses apa pun dari pengisian sistem yang telalu banyak dengan menciptakan terlalu banyak subproses.

Sebagai tambahan pada berbagai sumber fisik dan logis bahwa suatu proses diperoleh ketika telah dibuat, data pemula (masukan) dapat turut lewat oleh induk proses sampai anak proses. Sebagai contoh, anggap suatu proses yang fungsinya untuk menunjukkan status sebuah berkas, katakan F1, pada layar terminal.

Ketika dibuat, akan menjadi sebagai sebuah masukan dari proses induknya, nama dari berkas F1, dan akan mengeksekusi menggunakan kumpulan data tersebut untuk memperoleh informasi yang diinginkan. Proses tersebut juga mendapat nama dari perangkat luar. Sebagian sistem operasi melewati sumber-sumber ke anak proses. Pada sistem tersebut, proses baru bisa mendapat dua berkas terbuka yang baru, F1 dan perangkat terminal dan hanya perlu untuk mentransfer data antara kedua berkas tersebut.

Ketika suatu proses membuat proses baru, dua kemungkinan ada dalam term eksekusi:

1. Induk terus menerus untuk mengeksekusi secara bersama-sama dengan anaknya.
2. Induk menunggu sampai sebagian dari anaknya telah diakhiri/terminasi.

Juga ada dua kemungkinan dalam term dari *address space* pada proses baru:

1. Anak proses adalah duplikat dari induk proses.
2. Anak proses memiliki program yang terisikan didalamnya.

Untuk mengilustrasikan implementasi yang berbeda ini, mari kita mempelajari sistem operasi UNIX. Dalam UNIX, tiap proses

diidentifikasi oleh pengidentifikasi proses, yang merupakan integer yang unik. Proses baru dibuat oleh sistem pemanggilan fork system call. Proses baru tersebut terdiri dari sebuah copy ruang alamat dari proses aslinya (original). Mekanisme tersebut memungkinkan induk proses untuk berkomunikasi dengan mudah dengan anak proses. Kedua proses (induk dan anak) meneruskan eksekusi pada instruksi setelah fork dengan satu perbedaan: Kode kembali untuk fork adalah nol untuk proses baru (anak), sebagaimana proses pengidentifikasi non nol (non zero) dari anak dikembalikan kepada induk.

Umumnya, sistem pemanggilan execlp digunakan setelah sistem pemanggilan fork. Oleh satu dari dua proses untuk menggantikan proses ruang memori dengan program baru. Sistem pemanggilan execlp mengisi suatu berkas binary kedalam memori (menghancurkan gambar memori pada program yang berisikan sistem pemanggilan execlp) dan memulai eksekusinya. Dengan cara ini, kedua proses mampu untuk berkomunikasi, dan lalu untuk pergi ke arah yang berbeda. Induk lalu dapat membuat anak yang lebh banyak atau jika induk tidak punya hal lain untuk dilakukan ketika anak bekerja, induk dapat mengeluarkan sistem pemanggilan wait untuk tidak menggerakkan dirinya sendiri pada suatu antrian yang siap sampai anak berhenti. Program C. Induk membuat anak proses menggunakan sistem pemanggilan fork(). Kini kita mempunyai dua proses yang berbeda yang menjalankan sebuah copy pada program yang sama. Nilai dari pid untuk anak proses adalah nol (zero): maka untuk induk adalah nilai integer yang lebih besar dari nol. Anak proses meletakkan ruang alamatnya dengan UNIX command/bin/ls (digunakan untuk mendapatkan pendaftaran directory) menggunakan sistem pemanggilan execlp(). Ketika anak proses selesai, induk proses menyimpulkan dari pemanggilan untuk wait() dimana induk proses menyelesaikannya dengan menggunakan sistem pemanggilan exit().

Secara kontras, sistem operasi DEC VMS membuat sebuah proses baru dengan mengisi program tertentu kedalam proses tersebut, dan memulai pekerjaannya. Sistem operasi Microsoft Windows NT mendukung kedua model: Ruang alamat induk proses dapat di duplikasi, atau induk dapat menspesifikasi nama dari sebuah program untuk sistem operasi untuk diisikan kedalam ruang alamat pada proses baru.

2.3.2. Terminasi Proses

Sebuah proses berakhir ketika proses tersebut selesai mengeksekusi pernyataan akhirnya dan meminta sistem operasi untuk menghapusnya dengan menggunakan sistem pemanggilan exit. Pada titik itu, proses tersebut dapat mengembalikan data

(keluaran) pada induk prosesnya (melalui sistem pemanggilan wait). Seluruh sumber-sumber dari proses-termasuk memori fisik dan virtual, membuka berkas, dan penyimpanan I/O di tempatkan kembali oleh sistem operasi.

Ada situasi tambahan tertentu ketika terminasi terjadi. Sebuah proses dapat menyebabkan terminasi dari proses lain melalui sistem pemanggilan yang tepat (contoh abort). Biasanya, sistem seperti itu dapat dipanggil hanya oleh induk proses tersebut yang akan diterminasi. Bila tidak, pengguna dapat secara sewenang-wenang membunuh job antara satu sama lain. Catat bahwa induk perlu tahu identitas dari anaknya. Maka, ketika satu proses membuat proses baru, identitas dari proses yang baru diberikan kepada induknya.

Induk dapat menterminasi/ mengakhiri satu dari anaknya untuk beberapa alasan, seperti:

- Anak telah melampaui kegunaannya atas sebagaian sumber yang telah diperuntukkan untuknya.
- Pekerjaan yang ditugaskan kepada anak telah keluar, dan sistem operasi tidak memeperbolehkan sebuah anak untuk meneruskan jika induknya berakhir.

Untuk menentukan kasus pertama, induk harus memiliki mekanisme untuk memeriksa status anaknya. Banyak sistem, termasuk VMS, tidak memperbolehkan sebuah anak untuk ada jika induknya telah berakhir. Dalam sistem seperti ini, jika suatu proses berakhir (walau secara normal atau tidak normal), maka seluruh anaknya juga harus diterminasi. Fenomena ini, mengacu pada terminasi secara cascading, yang normalnya dimulai oleh sistem operasi.

Untuk mengilustrasikan proses eksekusi dan proses terminasi, kita menganggap bahwa, dalam UNIX, kami dapat mengakhiri suatu proses dengan sistem pemanggilan exit; proses induknya dapat menunggu untuk terminasi anak proses dengan menggunakan sistem pemanggilan wait. Sistem pemanggilan wait kembali ke pengidentifikasi proses dari anak yang telah diterminasi, maka induk dapat memberitahu kemungkinanan anak mana yang telah diterminasi. Jika induk menterminasi. Maka, anaknya masih punya sebuah induk untuk mengumpulkan status mereka dan mengumpulkan statistik eksekusinya.

2.4. Hubungan Antara Proses

Sebelumnya kita telah ketahui seluk beluk dari suatu proses mulai dari pengertiannya, cara kerjanya, sampai operasi-operasinya seperti proses pembentukannya dan proses pemberhentiannya setelah selesai melakukan eksekusi. Kali ini kita akan mengulas

bagaimana hubungan antar proses dapat berlangsung, misal bagaimana beberapa proses dapat saling berkomunikasi dan bekerja-sama.

2.4.1. Proses yang Kooperatif

Proses yang bersifat simultan (*concurrent*) dijalankan pada sistem operasi dapat dibedakaan menjadi yaitu proses independent dan proses kooperatif. Suatu proses dikatakan independen apabila proses tersebut tidak dapat terpengaruh atau dipengaruhi oleh proses lain yang sedang dijalankan pada sistem.

Berarti, semua proses yang tidak membagi data apa pun (baik sementara/ tetap) dengan proses lain adalah independent. Sedangkan proses kooperatif adalah proses yang dapat dipengaruhi atau pun terpengaruhi oleh proses lain yang sedang dijalankan dalam sistem. Dengan kata lain, proses dikatakan kooperatif bila proses dapat membagi datanya dengan proses lain. Ada empat alasan untuk penyediaan sebuah lingkungan yang memperbolehkan terjadinya proses kooperatif:

1. Pembagian informasi: apabila beberapa pengguna dapat tertarik pada bagian informasi yang sama (sebagai contoh, sebuah berkas bersama), kita harus menyediakan sebuah lingkungan yang mengizinkan akses secara terus menerus ke tipe dari sumber-sumber tersebut.
2. Kecepatan penghitungan/ komputasi: jika kita menginginkan sebuah tugas khusus untuk menjalankan lebih cepat, kita harus membagi hal tersebut ke dalam subtask, setiap bagian dari subtask akan dijalankan secara parallel dengan yang lainnya. Peningkatan kecepatan dapat dilakukan hanya jika komputer tersebut memiliki elemen-elemen pemrosesan ganda (seperti CPU atau jalur I/O).
3. Modularitas: kita mungkin ingin untuk membangun sebuah sistem pada sebuah model modular-modular, membagi fungsi sistem menjadi beberapa proses atau threads.
4. Kenyamanan: bahkan seorang pengguna individu mungkin memiliki banyak tugas untuk dikerjakan secara bersamaan pada satu waktu. Sebagai contoh, seorang pengguna dapat mengedit, memcetak, dan meng-*compile* secara paralel.

```
import java.util.*;

public class BoundedBuffer {
    public BoundedBuffer() {
        // buffer diinisialisasikan kosong
        count = 0;
        in = 0;
```

```
out = 0;
        buffer = new Object[BUFFER_SIZE];
}

// produser memanggil method ini
   public void enter( Object item ) {
     while ( count == BUFFER_SIZE )
        ; // do nothing

     // menambahkan suatu item ke dalam buffer
    ++count;
    buffer[in] = item;
   in = ( in + 1 ) % BUFFER_SIZE;

   if ( count == BUFFER_SIZE )
        System.out.println( "Producer Entered " +
             item + " Buffer FULL" );
   else
        System.out.println( "Producer Entered " +
             item + " Buffer Size = " + count );
    }

// consumer memanggil method ini
   public Object remove() {
      Object item ;

     while ( count == 0 )
         ; // do nothing

     // menyingkirkan suatu item dari buffer
    --count;
    item = buffer[out];
    out = ( out + 1 ) % BUFFER_SIZE;

   if ( count == 0 )
       System.out.println( "Consumer consumed " +
             item + " Buffer EMPTY" );
   else
        System.out.println( "Consumer consumed " +
              item + " Buffer Size = " +count );
  return item;
 }

 public static final int NAP_TIME = 5;
 private static final int BUFFER_SIZE = 5;

 private volatile int count;
 private int in; // arahkan ke posisi kosong selanjutnya
 private int out; // arahkan ke posisi penuh selanjutnya
 private Object[] buffer;
}
```

Gambar 2-10. Program Produser Konsumer.

Sebuah proses produser membentuk informasi yang dapat digunakan oleh konsumen proses. Sebagai contoh sebuah cetakan program yang membuat banyak karakter yang diterima oleh driver pencetak. Untuk memperbolehkan produser dan konsumer proses agar dapat berjalan secara terus menerus, kita harus menyediakan sebuah item *buffer* yang dapat diisi dengan proses produser dan dikosongkan oleh proses konsumer. Proses produser dapat memproduksi sebuah item ketika konsumer sedang mengkonsumsi item yang lain. Produser dan konsumer harus dapat selaras. Konsumer harus menunggu hingga sebuah item diproduksi.

2.4.2. Komunikasi Proses Dalam Sistem

Cara lain untuk meningkatkan efek yang sama adalah untuk sistem operasi yaitu untuk menyediakan alat-alat proses kooperatif untuk berkomunikasi dengan yang lain lewat sebuah komunikasi dalam proses (IPC = Inter-Process Communication). IPC menyediakan sebuah mekanisme untuk mengizinkan proses-proses untuk berkomunikasi dan menyelaraskan aksi-aksi mereka tanpa berbagi ruang alamat yang sama. IPC adalah khusus digunakan dalam sebuah lingkungan yang terdistribusi dimana proses komunikasi tersebut mungkin saja tetap ada dalam komputer-komputer yang berbeda yang tersambung dalam sebuah jaringan. IPC adalah penyedia layanan terbaik dengan menggnakan sebuah sistem penyampaian pesan, dan sistem-sistem pesan dapat diberikan dalam banyak cara.

2.4.2.1. Sistem Penyampaian Pesan

Fungsi dari sebuah sistem pesan adalah untuk memperbolehkan komunikasi satu dengan yang lain tanpa perlu menggunakan pembagian data. Sebuah fasilitas IPC menyediakan paling sedikit dua operasi yaitu kirim (pesan) dan terima (pesan). Pesan dikirim dengan sebuah proses yang dapat dilakukan pada ukuran pasti atau variabel. Jika hanya pesan dengan ukuran pasti dapat dikirimkan, level sistem implementasi adalah sistem yang sederhana. Pesan berukuran variabel menyediakan sistem implementasi level yang lebih kompleks.

Berikut ini ada beberapa metode untuk mengimplementasikan sebuah jaringan dan operasi pengiriman/penerimaan secara logika:

- Komunikasi langsung atau tidak langsung.
- Komunikasi secara simetris/ asimetris.
- *Buffer* otomatis atau eksplisit.
- engiriman berdasarkan salinan atau referensi.
- Pesan berukuran pasti dan variabel.

2.4.2.2. Komunikasi Langsung

Proses-proses yang ingin dikomunikasikan harus memiliki sebuah cara untuk memilih satu dengan yang lain. Mereka dapat menggunakan komunikasi langsung/ tidak langsung.

Setiap proses yang ingin berkomunikasi harus memiliki nama yang bersifat eksplisit baik penerimaan atau pengirim dari komunikasi tersebut. Dalam konteks ini, pengiriman dan penerimaan pesan secara primitive dapat dijabarkan sebagai:

- Send (P, message) - mengirim sebuah pesan ke proses P.
- Receive (Q, message) - menerima sebuah pesan dari proses Q.

Sebuah jaringan komunikasi pada bahasan ini memiliki beberapa sifat, yaitu:

- Sebuah jaringan yang didirikan secara otomatis diantara setiap pasang dari proses yang ingin dikomunikasikan. Proses tersebut harus mengetahui identitas dari semua yang ingin dikomunikasikan.
- Sebuah jaringan adalah terdiri dari penggabungan dua proses.
- Diantara setiap pesan dari proses terdapat tepat sebuah jaringan.

Pembahasan ini memperlihatkan sebuah cara simetris dalam pemberian alamat. Oleh karena itu, baik keduanya yaitu pengirim dan penerima proses harus memberi nama bagi yang lain untuk berkomunikasi, hanya pengirim yang memberikan nama bagi penerima sedangkan penerima tidak menyediakan nama bagi pengirim. Dalam konteks ini, pengirim dan penerima secara sederhana dapat dijabarkan sebagai:

- Send (P, message) - mengirim sebuah pesan kepada proses P.
- Receive (id, message) - menerima sebuah pesan dari semua proses. Variabel id diatur sebagai nama dari proses dengan komunikasi.

2.4.2.3. Komunikasi Tidak Langsung

Dengan komunikasi tidak langsung, pesan akan dikirimkan pada dan diterima dari/ melalui *mailbox*(kotak surat) atau terminal-terminal, sebuah *mailbox* dapat dilihat secara abstrak sebagai sebuah objek didalam setiap pesan yang dapat ditempatkan dari proses dan dari setiap pesan yang bias dipindahkan. Setiap kotak surat memiliki sebuah identifikasi (identitas) yang unik, sebuah proses dapat berkomunikasi dengan beberapa proses lain melalui sebuah nomor dari *mailbox* yang berbeda. Dua proses dapat saling berkomunikasi apabila kedua proses tersebut sharing *mailbox*. Pengirim dan penerima dapat dijabarkan

sebagai:

- Send (A, message) - mengirim pesan ke *mailbox* A.
- Receive (A, message) - menerima pesan dari *mailbox* A.

Dalam masalah ini, link komunikasi mempunyai sifat sebagai berikut:

• Sebuah link dibangun diantara sepasang proses dimana kedua proses tersebut membagi *mailbox*.

- Sebuah link mungkin dapat berasosiasi dengan lebih dari dua proses.
- Diantara setiap pasang proses komunikasi, mungkin terdapat link yang berbeda-beda, dimana setiap link berhubungan pada satu *mailbox*.

Misalkan terdapat proses P1, P2 dan P3 yang semuanya share *mailbox*. Proses P1 mengirim pesan ke A, ketika P2 dan P3 masing-masing mengeksekusi sebuah kiriman dari A. Proses mana yang akan menerima pesan yang dikirim P1? Jawabannya tergantung dari jalur yang kita pilih:

- Mengizinkan sebuah link berasosiasi dengan paling banyak 2 proses.
- Mengizinkan paling banyak satu proses pada suatu waktu untuk mengeksekusi hasil kiriman (*receive operation*).
- Mengizinkan sistem untuk memilih secara mutlak proses mana yang akan menerima pesan (apakah itu P2 atau P3 tetapi tidak keduanya, tidak akan menerima pesan). Sistem mungkin mengidentifikasi penerima kepada pengirim.

Mailbox mungkin dapat dimiliki oleh sebuah proses atau sistem operasi. Jika *mailbox* dimiliki oleh proses, maka kita mendefinisikan antara pemilik (yang hanya dapat menerima pesan melalui *mailbox*) dan pengguna dari *mailbox* (yang hanya dapat mengirim pesan ke *mailbox*). Selama setiap *mailbox* mempunyai kepemilikan yang unik, maka tidak akan ada kebingungan tentang siapa yang harus menerima pesan dari mailbox. Ketika proses yang memiliki mailbox tersebut diterminasi, mailbox akan hilang. Semua proses yang mengirim pesan ke mailbox ini diberi pesan bahwa mailbox tersebut tidak lagi ada.

Dengan kata lain, mempunyai mailbox sendiri yang independent, dan tidak melibatkan proses yang lain. Maka sistem operasi harus memiliki mekanisme yang mengizinkan proses untuk melakukan hal-hal dibawah ini:

- Membuat *mailbox* baru.
- Mengirim dan menerima pesan melalui *mailbox*.
- Menghapus *mailbox*.

Proses yang membuat *mailbox* pertama kali secara default akan memiliki *mailbox* tersebut. Untuk pertama kali, pemilik adalah satu-satunya proses yang dapat menerima pesan melalui *mailbox* ini. Bagaimana pun, kepemilikan dan hak menerima pesan mungkin dapat dialihkan ke proses lain melalui sistem pemanggilan.

2.4.2.4. Sinkronisasi

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

- Pengiriman yang diblok: Proses pengiriman di blok sampai pesan diterima oleh proses penerima (*receiving process*) atau oleh *mailbox*.
- Pengiriman yang tidak diblok: Proses pengiriman pesan dan mengkalkulasi operasi.
- Penerimaan yang diblok: Penerima mem blok samapai pesan tersedia.
- Penerimaan yang tidak diblok: Penerima mengembalikan pesan valid atau null.

2.4.2.5. *Buffering*

Baik komunikasi itu langsung atau tak langsung, penukaran pesan oleh proses memerlukan antrian sementara. Pada dasarnya, terdapat tiga jalan dimana antrian tersebut diimplementasikan:

- Kapasitas nol: antrian mempunyai panjang maksimum 0, maka link tidak dapat mempunyai penungguan pesan (*message waiting*). Dalam kasus ini, pengirim harus memblok sampai penerima menerima pesan.
- Kapasitas terbatas: antrian mempunyai panjang yang telah ditentukan, paling banyak n pesan dapat dimasukkan. Jika antrian tidak penuh ketika pesan dikirimkan, pesan yang baru akan menimpa, dan pengirim pengirim dapat melanjutkan eksekusi tanpa menunggu. Link mempunyai kapasitas terbatas.
- Jika link penuh, pengirim harus memblok sampai terdapat ruang pada antrian.
- Kapasitas tak terbatas: antrian mempunyai panjang yang tak terhingga, maka, semua pesan dapat menunggu disini. Pengirim tidak akan pernah di blok.

2.4.2.6. Contoh Produser-Konsumer

Sekarang kita mempunyai solusi problem produser-konsumer yang menggunakan penyampaian pesan. Produser dan konsumer akan berkomunikasi secara tidak langsung menggunakan *mailbox* yang dibagi. Buffer menggunakan java.util.Vector class sehingga *buffer* mempunyai kapasitas tak terhingga. Dan send() dan read() method adalah nonblocking. Ketika produser memproduksi suatu item, item tersebut diletakkan ke *mailbox* melalui send() method. Konsumer menerima item dari *mailbox* menggunakan receive() method. Karena receive() nonblocking, consumer harus mengevaluasi nilai dari Object yang di-*return* dari receive(). Jika null, *mailbox* kosong.

```
import java.util.*;

public class Producer extends Thread {
   private MessageQueueueue mbox;

   public Producer( MessageQueueueue m ) {
      mbox = m;
   }

   public void run() {
      Date message;

      while ( true ) {
         int sleeptime = ( int ) ( Server.NAP_TIME *
Math.random() );
            System.out.println( "Producer sleeping for " +
               sleeptime + " seconds" );
            try {
               Thread.sleep(sleeptime*1000);
            } catch( InterruptedException e ) {}

            message = new Date();
            System.out.println( "Producer produced " + message );
            mbox.send( message );
    }
   }
}
import java.util.*;

public class Consumer extends Thread {
      private MessageQueueueue mbox;

      public Consumer( MessageQueueueue m ) {
            mbox = m;
      }

public void run() {
      Date message;
```

```
        while ( true ) {
               int     sleeptime     =     (int)     (Server.NAP_TIME     *
    Math.random());
               System.out.println("Consumer sleeping for " +
                   sleeptime + " seconds" );
               try {
                  Thread.sleep( sleeptime * 1000 );
               } catch( InterruptedException e ) {}

               message = ( Date ) mbox.receive();
               if ( message != null )
                   System.out.println("Consumer consume " + message
    );
         }
       }
    }
```

Gambar 2-11. Program Produser Konsumer Alternatif.

Kita memiliki dua aktor di sini, yaitu Produser dan Konsumer. Produser adalah thread yang

menghasilkan waktu (Date) kemudian menyimpannya ke dalam antrian pesan. Produser juga mencetak waktu tersebut di layer (sebagai umpan balik bagi kita). Konsumer adalah thread yang akan mengakses antrian pesan untuk mendapatkan waktu (date) itu dan tak lupa mencetaknya di layer. Kita menginginkan supaya konsumer itu mendapatkan waktu sesuatu dengan urutan sebagaimana produser menyimpan waktu tersebut. Kita akan menghadapi salah satu dari dua kemungkinan situasi di bawah ini:

• Bila p1 lebih cepat dari c1, kita akan memperoleh output sebagai berikut:

```
………..
Consumer consume Wed May 07 14:11:12 ICT 2003
Consumer sleeping for 3 seconds
Producer produced Wed May 07 14:11:16 ICT 2003
Producer sleeping for 4 seconds
// p1 sudah mengupdate isi mailbox waktu dari Wed May 07
// 14:11:16 ICT 2003 ke Wed May 07 14:11:17 ICT 2003,
// padahal c1 belum lagi mengambil waktu Wed May 07 14:11:16
Producer produced Wed May 07 14:11:17 ICT 2003
Producer sleeping for 4 seconds
Consumer consume Wed May 07 14:11:17 ICT 2003
Consumer sleeping for 4 seconds
// Konsumer melewatkan waktu Wed May 07 14:11:16
. . .
```

Gambar 2-12. Keluaran Program Produser Konsumer.

• Bila p1 lebih lambat dari c1, kita akan memperoleh keluaran seperti berikut:

```
. . .
Producer produced Wed May 07 14:11:11 ICT 2003
Producer sleeping for 1 seconds
Consumer consume Wed May 07 14:11:11 ICT 2003
Consumer sleeping for 0 seconds
// c1 sudah mengambil isi dari mailbox, padahal p1 belum
// lagi megupdate isi dari mailbox dari May 07 14:11:11
// ICT 2003 ke May 07 14:11:12 ICT 2003, c1 mendapatkan
// waktu Wed May 07 14:11:11 ICT 2003 dua kali.
Consumer consume Wed May 07 14:11:11 ICT 2003
Consumer sleeping for 0 seconds
Producer sleeping for 0 seconds
Producer produced Wed May 07 14:11:12 ICT 2003
. . .
```

Gambar 2-13. Keluaran Program Produser Konsumer.

Situasi di atas dikenal dengan race conditions. Kita dapat menghindari situasi itu dengan

mensinkronisasikan aktivitas p1 dan c1 (sehubungan dengan akses mereka ke mailbox). Proses tersebut akan didiskusikan pada Bagian 3.2.

2.4.2.7. *Mailbox*

```
import java.util.*;

public class MessageQueue {
    private Vector q;

    public MessageQueue() {
        q = new Vector();
     }

// Mengimplementasikan pengiriman nonblocking
public void send( Object item ) {
     q.addElement( item );
}

// Mengimplementasikan penerimaan nonblocking
public Object receive() {
    Object item;
    if ( q.size() == 0 )
       return null;
    else {
        item = q.firstElement();
        q.removeElementAt(0);

    return item;
   }
  }
}
```

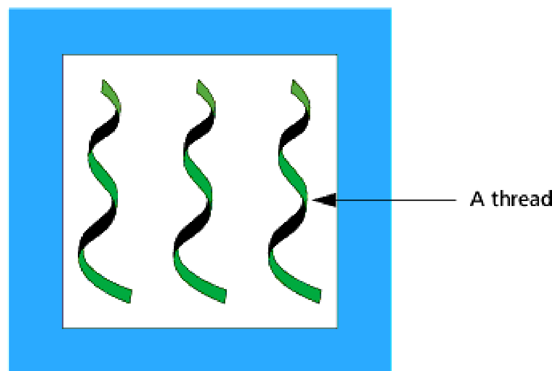
Gambar 2-14. Program Send/ Receive.

1. Menunggu sampai batas waktu yang tidak dapat ditentukan sampai terdapat ruang kosong pada *mailbox*.
2. Menunggu paling banyak n milidetik.
3. Tidak menunggu, tetapi kembali (*return*) secepatnya.

Satu pesan dapat diberikan kepada sistem operasi untuk disimpan, walau pun *mailbox* yang dituju penuh. Ketika pesan dapat disimpan pada *mailbox*, pesan akan dikembalikan kepada pengirim (*sender*). Hanya satu pesan kepada *mailbox* yang penuh yang dapat diundur (*pending*) pada suatu waktu untuk diberikan kepada thread pengirim.

2.5. *Thread*

Thread, atau kadang-kadang disebut proses ringan (*lightweight*), adalah unit dasar dari utilisasi CPU. Di dalamnya terdapat ID thread, program counter, register, dan stack. Dan saling berbagi dengan thread lain dalam proses yang sama.



Gambar 2-15. Thread.

2.5.1. Konsep Dasar

Secara informal, proses adalah program yang sedang dieksekusi. Ada dua jenis proses, proses berat (*heavyweight*) atau biasa dikenal dengan proses tradisional, dan proses ringan atau kadang disebut thread.

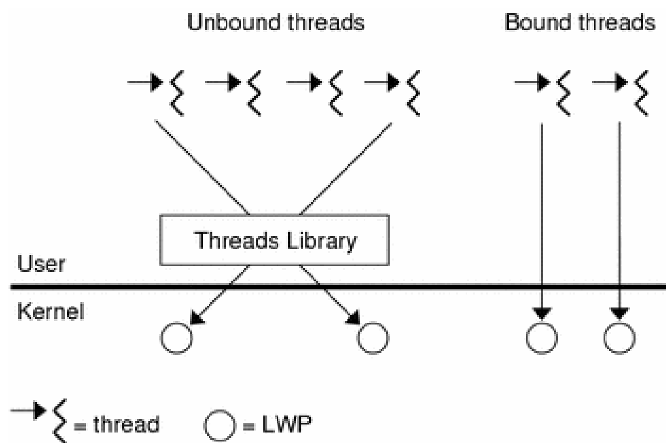
Thread saling berbagi bagian program, bagian data dan sumber daya sistem operasi dengan thread lain yang mengacu pada proses yang sama. Thread terdiri atas ID thread, program counter, himpunan register, dan stack. Dengan banyak kontrol thread

proses dapat melakukan lebih dari satu pekerjaan pada waktu yang sama.

2.5.2. Keuntungan

1. Tanggap: *Multithreading* mengizinkan program untuk berjalan terus walau pun pada bagian program tersebut di block atau sedang dalam keadaan menjalankan operasi yang lama/ panjang. Sebagai contoh, multithread web browser dapat mengizinkan pengguna berinteraksi dengan suatu thread ketika suatu gambar sedang diload oleh thread yang lain.
2. Pembagian sumber daya: Secara default, *thread* membagi memori dan sumber daya dari proses. Keuntungan dari pembagian kode adalah aplikasi mempunyai perbedaan aktifitas thread dengan alokasi memori yang sama.
3. Ekonomis: Mengalokasikan memori dan sumber daya untuk membuat proses adalah sangat mahal. Alternatifnya, karena thread membagi sumber daya dari proses, ini lebih ekonomis untuk membuat threads.
4. Pemberdayaan arsitektur multiprosesor: Keuntungann dari multithreading dapat ditingkatkan dengan arsitektur multiprosesor, dimana setiap thread dapat jalan secara parallel pada prosesor yang berbeda. Pada arsitektur prosesor tunggal, CPU biasanya berpindah-pindah antara setiap thread dengan cepat, sehingga terdapat ilusi paralelisme, tetapi pada kenyataannya hanya satu thread yang berjalan di setiap waktu.

2.5.3. *User Threads*



Gambar 2-16. User dan Kernel Thread.

User thread didukung oleh kernel dan diimplementasikan oleh thread library ditingkat pengguna. Library mendukung untuk pembentukan thread, penjadualan, dan managemen yang tidak didukung oleh kernel.

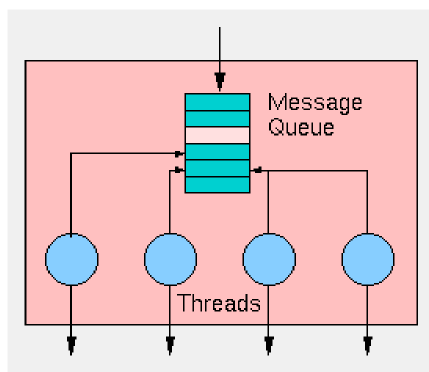
2.5.4. *Kernel Threads*

Kernel thread didukung secara langsung oleh sistem operasi: pembentukan thread, penjadualan, dan managemen dilakukan oleh kernel dalam ruang kernel. Karena managemen thread telah dilakukan oleh sistem operasi, kernel thread biasanya lebih lambat untuk membuat dan mengelola daripada pengguna thread. Bagaimana pun, selama kernel mengelola thread, jika suatu thread di block tehadap sistem pemanggilan, kernel dapat menjadualkan thread yang lain dalam aplikasi untuk dieksekusi. Juga, di dalam lingkungan multiprosesor, kernel dapat menjadualkan thread dalam prosesor yang berbeda. Windows NT, Solaris, dan Digital UNIX adalah sistem operasi yang mendukung kernel thread.

2.6. Model *Multithreading*

Dalam sub bab sebelumnya telah dibahas pengertian dari thread, keuntungannya, tingkatan atau levelnya seperti pengguna dan kernel. Maka dalam sub-bab ini pembahasan akan dilanjutkan dengan jenis-jenis thread tersebut dan contohnya baik pada Solaris mau pun Java.

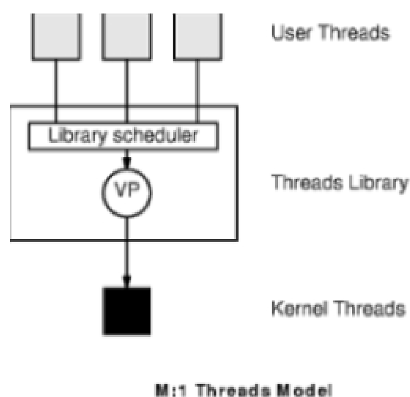
Sistem-sistem yang ada sekarang sudah banyak yang bisa mendukung untuk kedua pengguna dan kernel thread, sehingga model-model multithreading-nya pun menjadi beragam. Implementasi multithreading yang umum akan kita bahas ada tiga, yaitu model many-to-one, one-to-one, dan many-to-many.



Gambar 2-17. Model Multithreading.

2.6.1. Model *Many to One*

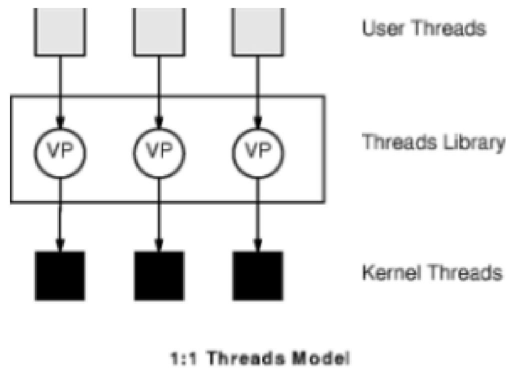
Model many-to-one ini memetakan beberapa tingkatan pengguna thread hanya ke satu buah kernel thread. Managemen proses thread dilakukan oleh (di ruang) pengguna, sehingga menjadi efisien, tetapi apabila sebuah thread melakukan sebuah pemblokingan terhadap sistem pemanggilan, maka seluruh proses akan berhenti (*blocked*). Kelemahan dari model ini adalah multihreads tidak dapat berjalan atau bekerja secara paralel di dalam multiprosesor dikarenakan hanya satu thread saja yang bisa mengakses kernel dalam suatu waktu.



Gambar 2-18. Model Many to One.

2.6.2. Model *One to One*

Model one-to-one memetakan setiap thread pengguna ke dalam satu kernel thread. Hal ini membuat model one-to-one lebih sinkron daripada model many-to-one dengan mengizinkan thread lain untuk berjalan ketika suatu thread membuat pemblokingan terhadap sistem pemanggilan; hal ini juga mengizinkan multiple thread untuk berjalan secara parallel dalam multiprosesor. Kelemahan model ini adalah dalam pembuatan thread pengguna dibutuhkan pembuatan korespondensi thread pengguna. Karena dalam proses pembuatan kernel thread dapat mempengaruhi kinerja dari aplikasi maka kebanyakan dari implementasi model ini membatasi jumlah thread yang didukung oleh sistem. Model one-to-one diimplementasikan oleh Windows NT dan OS/2.

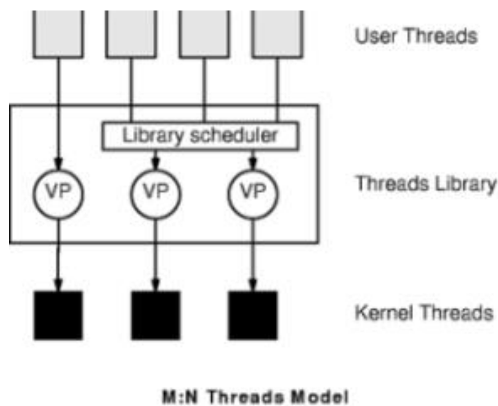


Gambar 2-19. Model One to One.

2.6.3. Model *Many to Many*

Beberapa tingkatan *thread* pengguna dapat menggunakan jumlah kernel thread yang lebih kecil atau sama dengan jumlah *thread* pengguna. Jumlah dari kernel *thread* dapat dispesifikasikan untuk beberapa aplikasi dan beberapa mesin (suatu aplikasi dapat dialokasikan lebih dari beberapa kernel thread dalam multiprosesor daripada dalam uniprosesor) dimana model many-to-one mengizinkan pengembang untuk membuat thread pengguna sebanyak mungkin, konkurensi tidak dapat tercapai karena hanya satu thread

yang dapat dijadualkan oleh kernel dalam satu waktu. Model one-to-one mempunyai konkurensi yang lebih tinggi, tetapi pengembang harus hati-hati untuk tidak membuat terlalu banyak thread tanpa aplikasi dan dalam kasus tertentu mungkin jumlah thread yang dapat dibuat dibatasi.



Gambar 2-20. Model Many to Many.

2.6.4. *Thread* Dalam Solaris 2

Solaris 2 merupakan salah satu versi dari UNIX yang sampai dengan tahun 1992 hanya masih mendukung proses berat (*heavyweight*) dengan kontrol oleh satu buah *thread*. Tetapi sekarang Solaris 2 sudah berubah menjadi sistem operasi yang modern yang mendukung threads di dalam level kernel dan pengguna, multiprosesor simetrik (SMP), dan penjadualan real-time.

Threads di dalam Solaris 2 sudah dilengkapi dengan library mengenai API-API untuk pembuatan dan managemen thread. Di dalam Solaris 2 terdapat juga level tengah thread. Di antara level pengguna dan level kernel thread terdapat proses ringan/ lightweight (LWP). Setiap proses yang ada setidaknya mengandung minimal satu buah LWP. Library thread memasangkan beberapa thread level pengguna ke ruang LWP-LWP untuk diproses, dan hanya satu user-level thread yang sedang terpasang ke suatu LWP yang bisa berjalan. Sisanya bisa diblok mau pun menunggu untuk LWP yang bisa dijalankan.

Operasi-operasi di kernel seluruhnya dieksekusi oleh kernel-level threads yang standar. Terdapat satu kernel-level thread untuk tiap LWP, tetapi ada juga beberapa kernel-level threads yang berjalan di bagian kernel tanpa diasosiasikan dengan suatu LWP (misalnya thread untuk pengalokasian disk). Thread kernel-level merupakan satu-satunya objek yang dijadualkan ke dalam sistem (lihat Bagian 2.7 mengenai scheduling). Solaris menggunakan model many-to-many.

Thread level pengguna dalam Solaris bisa berjenis bound mau pun unbound. Suatu bound thread level pengguna secara permanen terpasang ke suatu LWP. Jadi hanya thread tersebut yang bekerja di LWP, dan dengan suatu permintaan, LWP tersebut bisa diteruskan ke suatu prosesor. Dalam beberapa situasi yang membutuhkan waktu respon yang cepat (seperti aplikasi real-time), mengikat suatu thread sangatlah berguna. Suatu thread yang unbound tidak secara permanen terpasang ke suatu LWP. Semua threads unbound dipasangkan (secara multiplex) ke dalam suatu ruang yang berisi LWP-LWP yang tersedia

untuk aplikasi. Secara default thread-thread yang ada adalah unbound. Misalnya sistem sedang beroperasi, setiap proses bisa mempunyai threads level pengguna yang banyak.

User-user level thread ini bisa dijadual dan diganti di antara LWP-LWP-nya oleh thread library tanpa intervensi dari kernel. User-level threads sangatlah efisien karena tidak dibutuhkan bantuan kerja kernel oleh thread library untuk menukar dari satu user-level thread ke yang lain.

Setiap LWP terpasang dengan tepat satu kernel-level thread, dimana setiap user-level thread tidak tergantung dari kernel. Suatu proses mungkin mempunyai banyak LWP, tetapi mereka hanya dibutuhkan ketika thread harus berkomunikasi dengan kernel. Misalnya, suatu LWP akan dibutuhkan untuk setiap thread yang bloknya konkuren di sistem pemanggilan. Anggap ada lima buah pembacaan berkas yang muncul. Jadi dibutuhkan lima LWP, karena semuanya mungkin mengunggu untuk penyelesaian proses I/O di kernel. Jika suatu proses hanya mempunyai empat LWP, maka permintaan yang kelima harus menunggu unuk salah satu LWP kembali dari kernel. Menambah LWP yang keenam akan sia-sia jika hanya terdapat tempat untuk lima proses.

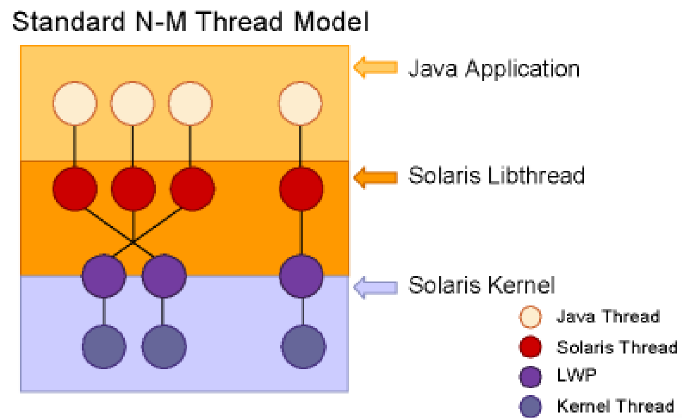
Kernel-kernel threads dijadual oleh penjadual kernel dan dieksekusi di CPU atau CPU-CPU dalam sistemnya. Jika suatu kernel thread memblok (misalnya karena menunggu penyelesaian suatu proses I/O), prosesor akan bebas untuk menjalankan kernel thread yang lain. Jika thread yang sedang terblok sedang menjalankan suatu bagian dari LWP, maka LWP tersebut akan ikut terblok. Di tingkat yang lebih atas lagi, user-level thread yang sedang terpasang ke LWP tersebut akan terblok juga. Jika suatu proses mempunyai lebih dari satu LWP, maka LWP lain bisa dijadual oleh kernel.

Para pengembang menggunakan struktur-struktur data sebagai berikut untuk mengimplementasikan thread-thread dalam Solaris 2:

- Suatu user-level thread mempunyai thread ID, himpunan register (mencakup suatu PC dan stack pointer), stack dan prioritas (digunakan oleh library untuk penjadualan). Semua struktur data tersebut berasal dari ruang user.

- Suatu LWP mempunyai suatu himpunan register untuk user-level thread yang ia jalankan, juga memori dan informasi pencatatan. LWP merupakan suatu struktur data dari kernel, dan bertempat pada ruang kernel.
- Suatu kernel thread hanya mempunyai struktur data yang kecil dan sebuah stack. Struktur datanya melingkupi copy dari kernel-kernel registers, suatu pointer yang menunjuk ke LWP yang terpasang dengannya, dan informasi tentang prioritas dan penjadualan.

Setiap proses dalam Solaris 2 mempunyai banyak informasi yang terdapat di process control block (PCB). Secara umum, suatu proses di Solaris mempunyai suatu proses id (PID), peta memori, daftar dari berkas yang terbuka, prioritas, dan pointer yang menunjuk ke daftar LWP yang terasosiasi kedalam proses.



Gambar 2-21. Thread Solaris dan Java.

2.6.5. *Thread* Java

Seperti yang telah kita lihat, thread didukung selain oleh sistem operasi juga oleh paket library thread. Sebagai contoh, Win32 library mempunyai API untuk multithreading aplikasi Windows, dan Pthreads mempunyai fungsi manajmen thread untuk sistem POSIX-compliant. Java adalah unik dalam mendukung tingkatan bahasa untuk membuat dan managemen thread.

Semua program java mempunyai paling sedikit satu kontrol thread. Bahkan program java yang sederhana mempunyai hanya satu main() method yang berjalan dalam thread tunggal dalam JVM. Java menyediakan perintah-perintah yang mendukung pengembang untuk membuat dan memanipulasi kontrol thread pada program.

Satu cara untuk membuat thread secara eksplisit adalah dengan membuat kelas baru yang diturunkan dari kelas thread, dan menimpa run() method dari kelas Thread tersebut.

Object yang diturunkan dari kelas tersebut akan menjalankan sebagian thread control dalam JVM. Bagaimana pun, membuat suatu objek yang diturunkan dari kelas Thread tidak secara spesifik membuat thread baru, tetapi start() method lah yang sebenarnya membuat thread baru.

Memanggil start() method untuk objek baru mengalokasikan memori dan menginisialisasikan thread baru dalam JVM dan memanggil run() method membuat thread pantas untuk dijalankan oleh JVM.

(Catatan: jangan pernah memanggil run() method secara langsung. Panggil start() method dan ini secara langsung akan memanggil run() method).

Ketika program ini dijalankan, dua thread akan dibuat oleh JVM. Yang pertama dibuat adalah thread yang berasosiasi dengan aplikasi-thread tersebut mulai dieksekusi pada main() method. Thread kedua adalah runner thread secara ekspilisit dibuat dengan start() method. Runner thread memulai eksekusinya dengan run() method.

Pilihan lain untuk membuat sebuah thread yang terpisah adalah dengan mendefinisikan suatu kelas yang mengimplementasikan runnable interface. Runnable interface tersebut didefinisikan sebagai berikut:

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

Gambar 2-22. Runnable.

Sehingga, ketika sebuah kelas diimplementasikan dengan runnable, kelas tersebut harus mendefinisikan run() method. Kelas thread yang berfungsi untuk mendefinisikan static dan instance method, juga mengimplementasikan runnable interface. Itu menerangkan bahwa mengapa sebuah kelas diturunkan dari thread harus mendefinisikan run() method.

Implementasi dari runnable interface sama dengan mengekstend kelas thread, satu-satunya kemungkinan untuk mengganti "extends thread" dengan "implements runnable".

```
Class worker2 implements Runnable
{
    Public void run() {
        System. Out. Println ("I am a worker
thread. ");
    }
}
```

Gambar 2-23. ClassWorker2.

Membuat sebuah thread dari kelas yang diimplementasikan oleh runnable berbeda dengan membuat thread dari kelas yang mengekstend thread. Selama kelas baru tersebut tidak mengekstend thread, dia tidak mempunyai akses ke objek static atau instance method —seperti start() method— dari kelas thread. Bagaimana pun, sebuah objek dari kelas thread adalah

tetap dibutuhkan, karena yang membuat sebuah thread baru dari kontrol adalah start() method.

Di kelas kedua, sebuah objek thread baru dibuat melalui runnable objek dalam konstruktornya. Ketika thread dibuat oleh start() method, thread baru mulai dieksekusi pada run() method dari runnable objek. Kedua method dari pembuatan thread tersebut adalah cara yang paling sering digunakan.

2.6.6. Managemen *Thread*

Java menyediakan beberapa fasilitas API untuk mengatur thread — thread, diantaranya adalah:

- Suspend(): berfungsi untuk menunda eksekusi dari thread yang sedang berjalan.
- Sleep(): berfungsi untuk menempatkan thread yang sedang berjalan untuk tidur dalam beberapa waktu.
- Resume(): hasil eksekusi dari thread yang sedang ditunda.
- Stop(): menghentikan eksekusi dari sebuah thread; sekali thread telah dihentikan dia tidak akan memulainya lagi.

Setiap method yang berbeda untuk mengontrol keadaan dari thread mungkin akan berguna dalam situasi tertentu. Sebagai contoh: Applets adalah contoh alami untuk *multithreading* karena mereka biasanya memiliki grafik, animasi, dan audio—semuanya sangat baik untuk mengatur berbagai thread yang terpisah. Bagaimana pun, itu tidak akan mungkin bagi sebuah applet untuk berjalan ketika dia sedang tidak ditampilkan, jika applet sedang menjalankan CPU secara intensif. Sebuah cara untuk menangani situasi ini adalah dengan menjalankan applet sebagai thread terpisah dari kontrol, menunda thread ketika applet sedang tidak ditampilkan dan melaporkannya ketika applet ditampilkan kembali.

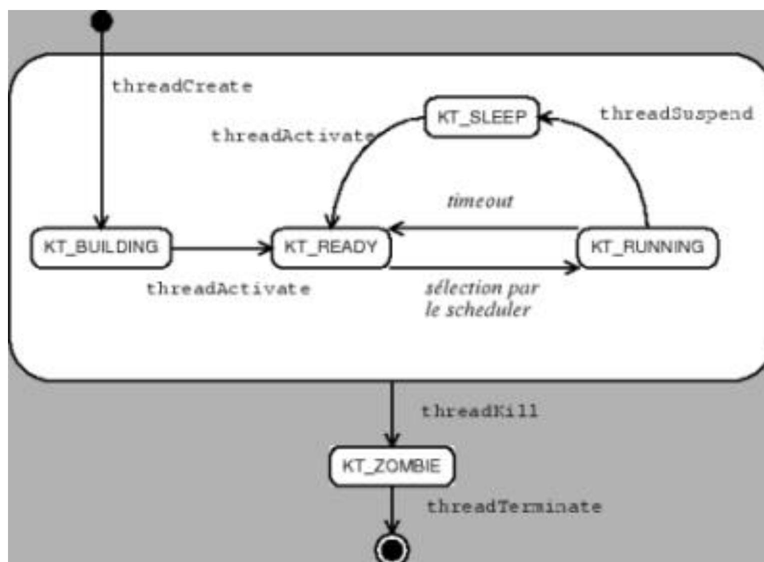
Anda dapat melakukannya dengan mencatat bahwa start() method dari sebuah applet dipanggil ketika applet tersebut pertama kali ditampilkan. Apabila user meninggalkan halaman web atau applet keluar dari tampilan, maka method stop() pada applet dipanggil (ini merupakan suatu keuntungan karena start() dan stop() keduanya terasosiasi dengan thread dan applet). Jika user kembali ke halaman web applet, kemudian start() method dipanggil kembali. Destroy() method dari sebuah applet dipanggil ketika applet tersebut dipindahkan dari cache-nya browser. Ini memungkinkan untuk mencegah sebuah applet berjalan ketika applet tersebut sedang tidak ditampilkan pada sebuah web browser dengan menggunakan stop() method dari applet yang

ditunda dan melaporkan eksekusi tersebut pada thread di applet start() method.

2.6.7. Keadaan *Thread*

Sebuah thread java dapat menjadi satu dari 4 kemungkinan keadaan:

1. new: sebuah thread pada keadaan ini ada ketika objek dari thread tersebut dibuat.
2. runnable: memanggil start() method untuk mengalokasikan memori bagi thread baru dalam JVM dan memanggil run() method untuk membuat objek.
3. block: sebuah thread akan diblok jika menampilkan sebuah kalimat pengeblokan. Contohnya: sleep() atau suspend().
4. dead: sebuah thread dipindahkan ke keadaan dead ketika run() method berhenti atau ketika stop() method dipanggil.



Gambar 2-24. Keadaan Thread.

2.6.8. *Thread* dan JVM

Pada penambahannya ke java program mengandung beberapa thread yang berbeda dari kontrol, disini ada beberapa thead yang sedang berjalan secara tidak sinkron untuk kepentingan dari penanganan sistem tingkatan JVM seperti managemen memori dan grafik kontrol. *Garbage Collector* mengevaluasi objek ketika

JVM untuk dilihat ketika mereka sedang digunakan. Jika tidak, maka itu akan kembali ke memori

dalam sistem.

2.6.9. JVM dan Sistem Operasi

Secara tipikal implementasi dari JVM adalah pada bagian atas terdapat host sistem operasi, pengaturan ini mengizinkan JVM untuk menyembunyikan detail implementasi dari sistem operasi dan menyediakan sebuah kekonsistenan, lingkungan yang abstrak tersebut mengizinkan program-program java untuk beroprasi pada berbagai sistem operasi yang mendukung sebuah JVM. Spesifikasi bagi JVM tidak mengidentifikasi bagaimana java thread dipetakan ke dalam sistem operasi.

2.6.10. Contoh Solusi *Multithreaded*

Pada bagian ini, kita memperkenalkan sebuah solusi multithreaded secara lengkap kepada masalah produser konsumer yang menggunakan penyampaian pesan. Kelas server pertama kali membuat sebuah mailbox untuk mengumpulkan pesan, dengan menggunakan kelas message queue kemudian dibuat produser dan konsumer threads secara terpisah dan setiap thread mereferensi ke dalam mailbox bersama. Thread produser secara bergantian antara tidur untuk sementara, memproduksi item, dan memasukkan item ke dalam mailbox. Konsumer bergantian antara tidur dan mengambil suatu item dari mailbox dan mengkonsumsinya. Karena receive() method dari kelas message queue adalah tanpa pengeblokan, konsumer harus mencek apakah pesan yang diambilnya tersebut adalah nol.

2.7. Penjadual CPU

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

2.7.1. Konsep Dasar

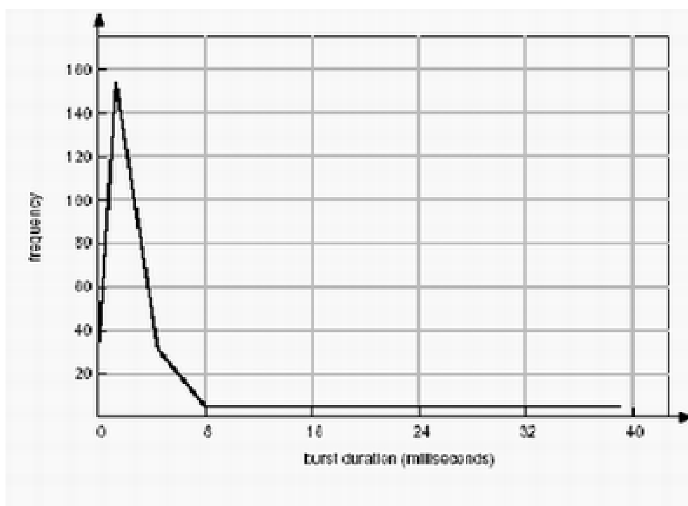
Tujuan dari multi programming adalah untuk mempunyai proses berjalan secara bersamaan, untuk memaksimalkan kinerja dari CPU. Untuk sistem uniprosesor, tidak pernah ada proses yang berjalan lebih dari satu. Bila ada proses yang lebih dari satu maka yang lain harus mengantri sampai CPU bebas. Ide dari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

porgamming sangat sederhana. Ketika sebuah proses dieksekusi yang lain harus menunggu sampai selesai. Di sistem komputer yang sederhana CPU akan banyak dalam posisi idle. Semua waktu ini sangat terbuang. Dengan multiprogamming kita mencoba menggunakan waktu secara produktif. Beberapa proses di simpan dalam memori dalam satu waktu. Ketika proses harus menuggu. Sistem operasi mengmbil CPU untuk memproses proses tersebut dan meninggalkan proses yang sedang dieksekusi.

Penjadual adalah fungsi dasar dari suatu sistem operasi. Hampir semua sumber komputer dijadual sebelum digunakan. CPU salah satu sumber dari komputer yang penting yang menjadi sentral dari sentral penjadual di sistem operasi.

2.7.1.1. Siklus Burst CPU-I/O

Keberhasilan dari penjadual CPU tergantung dari beberapa properti prosesor. Proses eksekusi mengandung siklus CPU ekskusi dan I/o Wait. Proses hanya akan bolak-balik dari dua state ini. Poses eksekusi dimulai dengan CPU Burst, setelah itu diikuti oleh I/O burst, dan dilakukan secara bergiliran.
Durasi dari CPU bust ini ditelah diukur secara ekstensif, walau pun mereka sangat berbeda dari proses ke proses. Mereka mempunyai frekeunsi kurva yang sama seperti yang diperlihatkan gambar dibawah ini.



Gambar 2-25. CPU Burst.

2.7.1.2. Penjadual CPU

Kapan pun CPU menjadi idle, sistem opersai harus memilih salah satu proses untuk masuk kedalam antrian ready (siap) untuk dieksekusi. Pemilihan tersebut dilakukan oleh penjadual short

term. Penjadual memilih dari sekian proses yang ada di memori yang sudah siap dieksekusi, den mengalokasikan CPU untuk mengeksekusinya

Penjadual CPU mungkin akan dijalankan ketika proses:

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

Penjadual dari no 1 sampai 4 non premptive sedangkan yang lain premptive. Dalam penjadual*nonpreemptive* sekali CPU telah dialokasikan untuk sebuah proses, maka tidak bisa di ganggu, penjadual model seperti ini digunakan oleh Windows 3.x; Windows 95 telah menggunakan penjadual preemptive.

2.7.1.3. *Dispatcher*

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

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

Dispatcher seharusnya secepat mungkin.

2.7.1.4. Kriteria Penjadual

Algoritma penjadual CPU yang berbeda mempunyai property yang berbeda. Dalam memilih algoritma yang digunakan untuk situasi tertentu, kita harus memikirkan properti yang berbeda untuk algoritma yang berbeda. Banyak kriteria yang dianjurkan utnuk membandingkan penjadual CPU algoritma. Kritria yang biasanya digunakan dalam memilih adalah:

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

2. Throughput: jika CPU sibuk mengeksekusi proses, jika begitu kerja telah dilaksanakan. Salah satu ukuran kerja adalah banyak proses yang diselesaikan per unit waktu, disebut througput. Untuk proses yang lama mungkin 1 proses per jam; untuk proses yang sebentar mungkin 10 proses perdetik.

3. Turnaround time: dari sudur pandang proses tertentu, kriteria yang penting adalah berapa lama untuk mengeksekusi proses tersebut. Interval dari waktu yang diizinkan dengan waktu yang dibutuhkan untuk menyelesaikan sebuah prose

disebut turn-around time. Trun around time adalah jumlah periode untuk menunggu untuk bisa ke memori, menunggu di ready queue, eksekusi di CPU, dan melakukan I/O.

4. Waiting time: algoritma penjadual CPU tidak mempengaruhi waktu untuk melaksanakan proses tersebut atau I/O; itu hanya mempengaruhi jumlah waktu yang dibutuhkan proses di antrian ready. Waiting time adalah jumlah periode menghabiskan di antrian ready.
5. Response time: di sistem yang interaktif, turnaround time mungkin bukan waktu yang terbaik untukkriteria. Sering sebuah proses bisa memproduksi output diawal, dan bisa meneruskan hasil yang baru sementara hasil yang sebelumnya telah diberikan ke user. Ukuran yang lain adalah waktu dari pengiriamn permintaan sampai respon yang pertama di berikan. Ini disebut response time, yaitu waktu untuk memulai memberikan respon, tetapi bukan waktu yang dipakai output untu respon tersebut.

Biasanya yang dilakukan adalah memaksimalkan CPU utilization dan throughput, dan minimalkan turnaround time, waiting time, dan response time dalam kasus tertentu kita

2.7.2. Algoritma Penjadual *First Come, First Served*

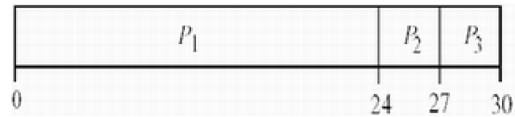
Penjadual CPU berurusan dengan permasalahan memutuskan proses mana yang akan dillaksanakan, oleh karena itu banyak bermacam algoritma penjadual, di seksi ini kita akan mendiskripsikan beberapa algoritma. Ini merupakan algoritma yang paling sederhana, dengan skema proses yang meminta CPU mendapat prioritas. Implementasi dari FCFS mudah diatasi dengan FIFO queue.

Contoh:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Gambar 2-26. Kedatangan Proses.

misal urutan kedatangan adalah P1, P2, P3 Gantt Chart untuk ini adalah:

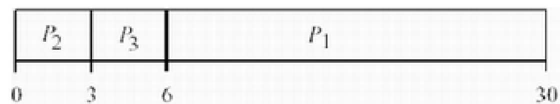


Gambar 2-28. Gannt Chart Kedatangan Proses I.

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

Gambar 2-27. Gannt Chart Kedatangan Proses II.

misal proses dibalik sehingga urutan kedatangan adalah P3, P2, P1.
Gantt chartnya adalah:



Gambar 2-30. Gannt Chart Kedatangan Proses III.

- Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3

Gambar 2-28. Gannt Chart Kedatangan Proses IV.

Dari dua contoh diatas bahwa kasus kedua lebih baik dari kasus pertama, karena pengaruh kedatangan disamping itu FCFS mempunyai kelemahan yaitu convoy effect dimana seandainya ada sebuah proses yang kecil tetapi dia mengantri dengan proses yang membutuhkan waktu yang lama mengakibatkan proses tersebut akan lama dieksekusi.

Penjadual FCFS algoritma adalah *nonpremptive*. Ketika CPU telah dialokasikan untuk sebuah proses, proses tetap menahan CPU sampai selssai. FCFS algortima jelas merupakan masalah bagi sistem time-sharing, dimana sangat penting untuk user mendapatkan pembagian CPU pada regular interval. Itu akan menjadi bencana untuk megizinkan satu proses pada CPU untuk waktu yang tidak terbatas

2.7.3. Penjadual *Shortest Job First*

Salah satu algoritma yang lain adalah Shortest Job First. Algoritma ini berkaitan dengan waktu setiap proses. Ketika CPU bebas proses yang mempunyai waktu terpendek untuk menyelesaikannya mendapat prioritas. Seandainya dua proses atau lebih mempunyai waktu yang sama maka FCFS algoritma digunakan untuk menyelsaikan masalah tersebut.

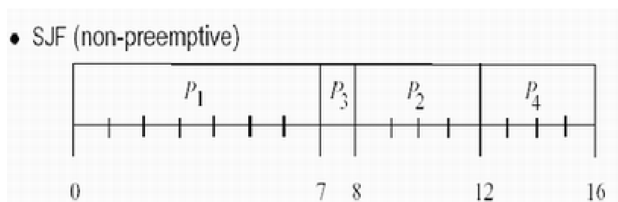
Ada dua skema dalam SJFS ini yaitu:

1. *nonpremptive*— ketika CPU memberikan kepada proses itu tidak bisa ditunda hingga selesai.
2. *premptive*— bila sebuah proses datang dengan waktu prose lebih rendah dibandingkan dengan waktu proses yang sedang dieksekusi oleh CPU maka proses yang waktunya lebih rendah mendapatkan prioritas. Skema ini disebut juga Short - Remaining Time First (SRTF).

Contoh:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Gambar 2-29. Kedatangan Proses.



Gambar 2-30. Gannt Chart SJF Non-Preemtive.

Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Gambar 2-31. Rata-rata Menunggu

SJF algoritma mungkin adalah yang paling optimal, karena ia memberikan rata-rata minimum waiting untuk kumpulan dari proses yang mengantri. Dengan mengeksekusi waktu yang paling

pendek baru yang paling lama. Akibatnya rata-rata waktu mnenuggu menurun.

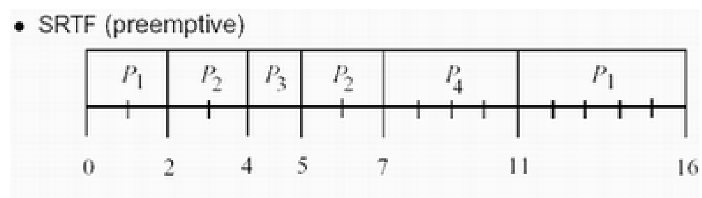
Hal yang sulit dengan SJF algoritma adalah mengethaui waku dari proses berikutnya. Untuk penjadual long term (lama) di sistem batch, kita bisa menggunakan panjang batas waktu proses yang user sebutkan ketika dia mengirim pekerjaan. Oleh karena itu sjf sering digunakan di penjadual long term. Walau pun SJF optimal tetapi ia tidak bisa digunakan untuk penjadual CPU short term. Tidak ada jalan untuk mengetahui panjang dari CPU burst berikutnya. Salah satu cara untuk mengimplementasikannya adalah dengan memprediksikan CPU burst berikutnya.

Contoh SJF premptive:

SJF algoritma mungkin adalah yang paling optimal, karena ia memberikan rata-rata minimum waiting untuk kumpulan dari proses yang mengantri.

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

Gambar 2-32. Kedatangan Proses.



Gambar 2-36. Gannt Chart SJF Preemtive.

Average waiting time = (9 + 1 + 0 + 2)/4 = 3

Gambar 2-33. Rata-rata Menunggu.

Kita lihat bahwa dengan premptive lebih baik hasilnya daripada non preemptive.

2.7.4. Penjadual Prioritas

Penjadualan SJF (*Shortest Job First*) adalah kasus khusus untuk algoritma penjadual Prioritas. Prioritas dapat diasosiasikan masing-masing proses dan CPU dialokasikan untuk proses dengan

prioritas tertinggi. Untuk proritas yang sama dilakukan dengan FCFS.

Ada pun algoritma penjadual prioritas adalah sebagai berikut:

- Setiap proses akan mempunyai prioritas (bilangan integer). Beberapa sistem menggunakan integer dengan urutan kecil untuk proses dengan prioritas rendah, dan sistem lain juga bisa menggunakan integer urutan kecil untuk proses dengan prioritas tinggi. Tetapi dalam teks ini diasumsikan bahwa integer kecil merupakan prioritas tertinggi.
- CPU diberikan ke proses dengan prioritas tertinggi (integer kecil adalah prioritas tertinggi).
- Dalam algoritma ini ada dua skema yaitu:
  1. Preemptive: proses dapat di interupsi jika terdapat prioritas lebih tinggi yang memerlukan CPU.
  2. Nonpreemptive: proses dengan prioritas tinggi akan mengganti pada saat pemakain time-slice habis.
- SJF adalah contoh penjadual prioritas dimana prioritas ditentukan oleh waktu pemakaian CPU berikutnya. Permasalahan yang muncul dalam penjadualan prioritas adalah indefinite blocking atau starvation.
- Kadang-kadang untuk kasus dengan prioritas rendah mungkin tidak pernah dieksekusi. Solusi untuk algoritma penjadual prioritas adalah aging
- Prioritas akan naik jika proses makin lama menunggu waktu jatah CPU.

2.7.5. Penjadual *Round Robin*

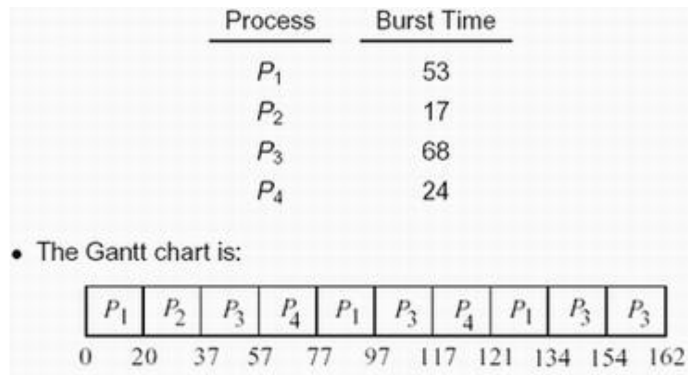
Algoritma *Round Robin* (RR) dirancang untuk sistem *time sharing*. Algoritma ini mirip dengan penjadual FCFS, namun preemption ditambahkan untuk switch antara proses. Antrian ready diperlakukan atau dianggap sebagai antrian sirkular. CPU menglilingi antrian ready dan mengalokasikan masing-masing proses untuk interval waktu tertentu sampai satu *time slice*/ *quantum*.

Berikut algritma untuk penjadual *Round Robin*:

- Setiap proses mendapat jatah waktu CPU (*time slice*/ *quantum*) tertentu Time slice/quantum umumnyaantara 10 - 100 milidetik.
  1. Setelah *time slice*/ *quantum* maka proses akan di-preempt dan dipindahkan ke antrian ready.
  2. Proses ini adil dan sangat sederhana.
- Jika terdapat n proses di "antrian ready" dan waktu quantum q (milidetik), maka:

1. Maka setiap proses akan mendapatkan 1/n dari waktu CPU.
2. Proses tidak akan menunggu lebih lama dari: (n-1)q *time units*.

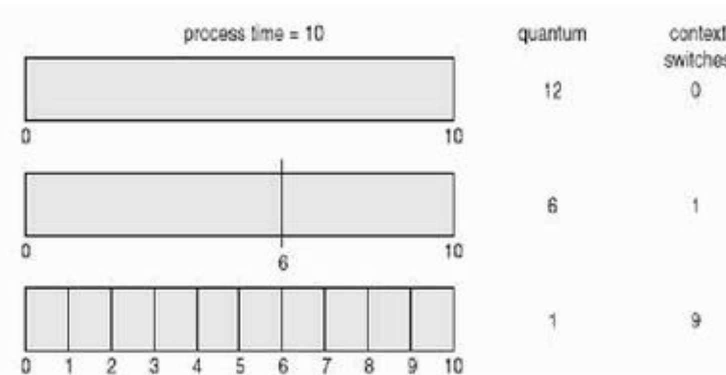
- Kinerja dari algoritma ini tergantung dari ukuran time quantum
  1. Time Quantum dengan ukuran yang besar maka akan sama dengan FCFS
  2. Time Quantum dengan ukuran yang kecil maka time quantum harus diubah ukurannya lebih besar dengan respek pada alih konteks sebaliknya akan memerlukan ongkos yang besar.



Gambar 2-34. Round Robin.

Tipikal: lebih lama waktu rata-rata turnaround dibandingkan SJF, tapi mempunyai response terhadap user lebih cepat.

Time Quantum Vs Alih Konteks



Gambar 2-35. Time Quantum dan Alih Konteks.

2.8. Penjadualan *Multiprocessor*

Multiprocessor membutuhkan penjadualan yang lebih rumit karena mempunyai banyak kemungkinan yang dicoba tidak seperti pada processor tunngal. Tapi saat ini kita hanya fokus pada processor yang homogen (sama) sesuai dengan fungsi masing-masing dari processor tersebut. Dan juga kita dapat menggunakan processor yang tersedia untuk menjalankan proses didalam antrian.

2.8.1. Penjadualan *Multiple Processor*

Diskusi kita sampai saat ini di permasalahan menjadualkan CPU di single prosesor. Jika multiple prosesor ada. Penjadualan menjadi lebih kompleks banyak kemungkinan telah dicoba dan telah kita lihat dengan penjadualan satu prosesor, tidak ada solusi yang terbaik. Pada kali ini kita hanya membahas secara sekilas tentang panjadualan di multiprosesor dengan syarat prosesornya identik.

Jika ada beberapa prosesor yang identik tersedia maka load sharing akan terjadi. Kita bisa menyediakan queue yang terpisah untuk setiap prosesor. Dalam kasus ini, bagaimana pun, satu prosesor bisa menjadi *idle* dengan antrian yang kosong sedangkan yang lain sangat sibuk. Untuk mengantisipasi hal ini kita menggunakan *ready queue* yang biasa. Semua proses pergi ke satu queue dan dijadualkan untuk prosesor yang bisa dipakai.

Dalam skema tersebut, salah satu penjadualan akan digunakan. Salah satu cara menggunakan symmetric multiprocessing (SMP). Dimana setiap prosesor menjadualkan diri sendiri. Setiap prosesor memeriksa raedy queue dan memilih proses yang akan dieksekusi.

Beberapa sistem membawa struktur satu langkah kedepan, dengan membawa semua keputusan penjadualan, I/O prosesing, dan aktivitas sistem yang lain ditangani oleh satu prosesor yang bertugas sebagai master prosesor. Prosesor yang lain mengeksekusi hanya user code yang disebut asymmetric multiprosessing jauh lebih mudah.

2.8.2. Penjadualan *Real Time*

Dalam bab ini, kita akan mendeskripsikan fasilitas penjadualan yang dibutuhkan untuk mendukung real time computing dengan bantuan sistem komputer.

Terdapat dua jenis real time computing: sistem hard real time dibutuhkan untuk menyelesaikan critical task dengan jaminan waktu tertentu. Secara umum, sebuah proses di kirim dengan sebuah pernyataan jumlah waktu dimana dibutuhkan untuk

menyelesaikan atau menjalankan I/O. Kemudian penjadual bisa menjamin proses untuk selesai atau menolak permintaan karena tidak mungkin dilakukan. Karena itu setiap operasi harus dijamin dengan waktu maksimum.

Soft real time computing lebih tidak ketat. Itu membutuhkan bahwa proses yang kritis menerima prioritas dari yang lain. Walau pun menambah fungsi soft real time ke sistem time sharing mungkin akan mengakibatkan pembagian sumber yang tidak adildan mengakibatkan delay yang lebih lama, atau mungkin pembatalan bagi proses tertentu, Hasilnya adalah tujuan secara umum sistem yang bisa mendukung multimedia, graphic berkecepatan tinggi, dan variasi tugas yang tidak bisa diterima di lingkungan yang tidak mendukunng soft real time computing

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

Untuk menjaga dispatch tetap rendah. Kita butuh agar system call untuk preemptible. Ada beberapa cara untuk mencapai tujuan ini. Satu untuk memasukkan preemption points di durasi yang lama system call, yang mana memeriksa apakah prioritas yang utama butuh untuk dieksekusi. Jika satu sudah, maka alih konteks mengambil alih; ketika high priority proses selesai, proses yang diinterupsi meneruskan dengan system call. Points premption bisa diganti hanya di lokasi yang aman di kernel — hanya kernel struktur tidak bisa dimodifikasi walau pun dengan preemption points, dispatch latency bisa besar, karena pada prakteknya untuk menambah beberapa preemption points untuk kernel.

Metoda yang lain untuk berurusan dengan preemption untuk membuat semua kernel preemptible. Karena operasi yang benar bisa dijamin, semua data kernel struktur dengan di proteksi. Dengan metode ini, kernel bisa selalu di preemptible, karena semua kernel bisa diupdate di proteksi.

Apa yang bisa diproteksi jika prioritas yang utama butuh untuk dibaca atau dimodifisikasi yang bisa dibutuhkan oleh yang lain, prioritas yang rendah? Prioritas yang tinggi harus menunggu menunggu untuk menyelesaikan prioritas yang rendah.

Fase konflik dari dispatch latency mempunyai dua komponen:

1. Preemption semua proses yang berjalan di kernel.
2. Lepas prioritas yang rendah untuk prioritas yang tinggi.

2.8.3. Penjadualan *Thread*

Di Bagian 2.5, kita mengenalkan threads untuk model proses, hal itu mengizinkan sebuah proses untuk mempunyai kontrol terhadap multiple threads. Lebih lanjut kita membedakan antara user-level dan kernel level threads. User level threads diatur oleh thread library. Untuk menjalankan di CPU, user level threads di mapping dengan asosiasi kernel level *thread*, walau pun mapping ini mungkin bisa *indirect* dan menggunakan *lightweight*.

2.9. Java *Thread* dan Algoritmanya

Penjadualan thread yang Runnable oleh Java Virtual Machine dilakukan dengan konsep preemptive dan mempunyai prioritas tertinggi. Dalam algoritma evaluasi ditentukan terlebih dahulu kriteria-kriterianya seperti utilisasinya dilihat dari segi waktu tunggu yang digunakan dan throughput yang disesuaikan dengan waktu turnaroundnya.

2.9.1. Penjadualan Java Thread

Java Virtual Machine menjadualkan thread menggunakan preemptive, berdasarkan prioritas algoritma penjadualan. Semua Java Thread diberikan sebuah prioritas dan Java Virtual Machine menjadualkan thread yang Runnable dengan menggunakan prioritas tertinggi saat eksekusi. Jika ada dua atau lebih thread yang Runnable yang mempunyai prioritas tertinggi, Java Virtual Machine akan menjadualkan thread tersebut menggunakan sebuah antrian secara FIFO.

2.9.1.1. Keunggulan Penjadualan Java *Thread*

1. Java *Virtual Machine* menggunakan prioritas preemtive berdasarkan algoritma penjadualan.
2. Semua thread Java mempunyai prioritas dan thread dengan proritas tertinggi dijadualkan untuk dieksekusi oleh Java Virtual Machine.
3. Jika terjadi dua thread dengan prioritas sama maka digunakan algoritma First In First Out.

Thread lain dijalankan bila terjadi hal-hal berikut ini:

- Thread yang sedang dieksekusi keluar dari status runnable misalnya diblok atau berakhir
- Thread dengan prioritas yang lebih tinggi dari thread yang sedang dieksekusi memasuki statusrunnable. Maka thread dengan prioritas yang lebih rendah ditunda eksekusinya dan digantikan oleh thread dengan prioritas lebih tinggi.

Time slicing tergantung pada implementasinya. Sebuah thread dapat memberi kontrol pada yield() method. Saat thread memberi sinyal pada CPU untuk mengontrol thread yang lain dengan prioritas yang sama maka thread tersebut dijadualkan untuk dieksekusi. Thread yang memberi kontrol pada CPU disebut Cooperative Multitasking.

2.9.1.2. Prioritas *Thread*

Java Virtual Machine memilih thread yang runnable dengan prioritas tertinggi. Semua thread java mempunyai prioritas dari 1 sampai 10. Prioritas tertinggi 10 dan berakhir dengan 1 sebagai prioritas terendah. Sedangkan prioritas normal adalah 5.

- Thread.MIN_PRIORITY = thread dengan prioritas terendah.
- Thread.MAX_PRIORITY = thread dengan prioritas tertinggi.
- Thread.NORM_PRIORITY = thread dengan prioritas normal.

Saat thread baru dibuat ia mempunyai prioritas yang sama dengan thread yang menciptakannya. Prioritas

thread dapat diubah dengan menggunakan setpriority() method.

2.9.1.3. Penjadualan *Round-Robin* dengan Java

```
public class Scheduler extends Thread {
    public Scheduler() {
            timeSlice = DEFAULT_TIME_SLICE;
            queue = new Circularlist();
     }

     public Scheduler(int quantum) {
            timeSlice = quantum;
            queue = new Circularlist();
     }

     public addThread(Thread t) {
            t.setPriority(2);
            queue.additem(t);
     }

    private void schedulerSleep() {
            try{
                 Thread.sleep(timeSlice );
         } catch (InterruptedException e){};
      }

      public void run(){
```

```
            Thread current;
            This.setpriority(6);
            while (true) {
               // get the next thread
               current = (Thread)qeue.getnext();
               if ( current != null) && (current.isAlive())) {
                         current.setPriority(4);
                         schedulerSleep();
                         current.setPriority(2)
              }
           }
      }

     private CircularList queue;
     private int timeSlice;
     private static final int DEFAULT_TIME_SLICE = 1000;
}

public class TesScheduler{
    public static void main()String args[]) {
            Thread.currentThread().setpriority(Thread.Max_Priority);
            Schedular CPUSchedular = new Scheduler ();
            CPUSchedular.start()
            TestThread t1 = new TestThread("Thread 1");
            t1.start()
            CpuSchedular.addThread(t1);
            TestThread t2 = new TestThread("Thread 2");
            t2.start()
            CpuSchedular.addThread(t2);
            TestThread t3 = new TestThread("Thread 1");
            t3.start()
            CpuSchedular.addThread(t3);
    }
}
```

Gambar 2-36. Round Robin.

2.9.2. Evaluasi Algoritma

Bagaimana kita memilih sebuah algoritma penjadualan CPU untuk sistem-sistem tertentu. Yang menjadipokok masalah adalah kriteria seperti apa yang digunakan untuk memilih sebuah algoritma. Untuk memilih suatu algoritma, pertama yang harus kita lakukan adalah menentukan ukuran dari suatu kriteria berdasarkan:

- Memaksimalkan penggunaan CPU dibawah maksimum waktu responnya yaitu 1 detik.
- Memaksimalkan throughput karena waktu turnaroundnya bergerak secara linier pada saat eksekusi proses.

2.9.2.1. Sinkronisasi dalam Java

Setiap objek dalam java mempunyai kunci yang unik yang tidak digunakan biasanya. Saat method dinyatakan sinkron, maka method dipanggil untuk mendapatkan kunci untuk objek tersebut. Saat kunci tersebut dipunyai thread yang lain maka thread tersebut diblok dan dimasukkan kedalam kumpulan kunci objek, misalnya:

```
public synchronized void enter(Object item) {
     while (count == BUFFER_SIZE)
         ;
     Thread.yeild();
  ++count;
     buffer[in] = item;
     in = (in+1) % BUFFER_SIZE;
}

public synchronized void remove (){
             Object item;
             while (count == 0)
               ;
             Thread.yeild();
             --count;
             item = buffer[out]
             out = (out+1) % BUFFER_SIZE;
             return item
}
```

Gambar 2-37. Sinkronisasi.

2.9.2.2. Metoda Wait() dan Notify()

Thread akan memanggil method wait() saat:

1. Thread melepaskan kunci untuk objek.
2. Status thread diblok.
3. Thread yang berada dalam status wait menunggu objek.

Thread akan memanggil method notify() saat: Thread yang dipilih diambil dari thread yang ada pada himpunan wait. Dengan cara:

1. Pindahkan thread yang dipilih dari wait set ke entry set.
2. Atur status dari thread yang dipilih dari blocked menjadi runnable.

2.9.2.3. Contoh Metoda Wait() dan Notify()

```
public synchronized void enter(Object item){
   while (count == BUFFER_SIZE) {
             try{
                wait();
```

```
        } catch (InterruptedException e) {}
    }

    // add an item to the buffer
    ++count;
    buffer[in] = item;
   in = (in+1) % BUFFER_SIZE;
   notify();
}

public synchronized void remove(Object item){
    while (count == 0) {
            try {
            wait();
            }
        catch (InterruptedException e) {}
    }

    // remove an item to the buffer
    --count;
    item = buffer[out];
    out = (out+1) % BUFFER_SIZE;
    notify();
    return item;
}
```

Gambar 2-38. Contoh Wait() dan Notify().

2.10. Kesimpulan

2.10.1. Proses

Sebuah proses adalah sebuah peristiwa adanya sebuah proses yang dapat dieksekusi. Sebagai sebuah eksekusi proses, maka hal tersebut membutuhkan perubahan keadaan. Keadaan dari sebuah proses dapat didefinisikan oleh aktivitas proses tertentu tersebut. Setiap proses mungkin menjadi satu dari beberapa state berikut, antara lain: *new*, *ready*, running, waiting, atau terminated. Setiap proses direpresentasikan ada sistem operasi berdasarkan proses-control-block (PCB)-nya.

Sebuah proses, ketika sedang tidak dieksekusi, ditempatkan pada antrian yang sama. Disini ada 2 kelas besar dari antrian dalam sebuah sistem operasi: permintaan antrian I/O dan ready queue. Ready queue memuat semua proses yang siap untuk dieksekusi dan yang sedang menunggu untuk dijalankan pada CPU. Setiap proses direpresentasikan oleh sebuah PCB, dan PCB tersebut dapat digabungkan secara bersamaan untuk mencatat sebuah ready queue. Penjadualan Long-term adalah pilihan dari proses-proses untuk diberi izin menjalankan CPU. Normalnya,

penjadualan long-term memiliki pengaruh yang sangat besar bagi penempatan sumber, terutama managemen memori. Penjadualan Short-term adalah pilihan dari satu proses dari ready queue.

Proses-proses pada sistem dapat dieksekusi secara berkelanjutan. Disini ada beberapa alasan mengapa proses tersebut dapat dieksekusi secara berkelanjutan: pembagian informasi, penambahan kecepatan komputasi, modularitas, dan kenyamanan atau kemudahan. Eksekusi secara berkelanjutan menyediakan sebuah mekanisme bagi proses pembuatan dan penghapusan.

Pengeksekusian proses-proses pada operating system mungkin dapat digolongkan menjadi proses independent dan kooperasi. Proses kooperasi harus memiliki beberapa alat untuk mendukung komunikasi antara satu dengan yang lainnya. Prinsipnya adalah ada dua rencana komplementer komunikasi: pembagian memori dan sistem pesan. Metode pembagian memori menyediakan proses komunikasi untuk berbagi beberapa variabel. Proses-proses tersebut diharapkan dapat saling melakukan tukar-menukar informasi seputar pengguna variabel yang terbagi ini. Pada sistem pembagian memori, tanggung jawab bagi penyedia komunikasi terjadi dengan programmer aplikasi; sistem operasi harus menyediakan hanya pembagian memori saja. Metode sistem pesan mengizinkan proses-proses untuk tukar-menukar pesan. Tanggung jawab bagi penyedia komunikasi ini terjadi dengan sistem operasi tersebut.

2.10.2. *Thread*

Thread adalah sebuah alur kontrol dari sebuah proses. Suatu proses yang multithreaded mengandung beberapa perbedaan alur kontrol dengan ruang alamat yang sama. Keuntungan dari multithreaded meliputi peningkatan respon dari user, pembagian sumber daya proses, ekonomis, dan kemampuan untuk mengambil keuntungan dari arsitektur multiprosesor. User level thread adalah thread yang tampak oleh programmer dan tidak diketahui oleh kernel. User level thread secara tipikal dikelola oleh sebuah library thread di ruang user. Kernel level thread didukung dan dikelola oleh kernel sistem operasi. Secara umum,

user level thread lebih cepat dalam pembuatan dan pengelolaan dari pada kernel thread. Ada tiga perbedaan tipe dari model yang berhubungan dengan user dan kernel thread.

- Model many to one: memetakan beberapa user level thread hanya ke satu buah kernel thread.
- Model one to one: memetakan setiap user thread ke dalam satu kernel thread. berakhir.

- Model many to many: mengizinkan pengembang untuk membuat user thread sebanyak mungkin, konkurensi tidak dapat tercapai karena hanya satu thread yang dapat dijadualkan oleh kernel dalam satu waktu.

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

seringkali mengganti keadaan thread itu sendiri.

2.10.3. Penjadualan CPU

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

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

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

2.11. Soal-soal Latihan

2.11.1. Proses

1. Sebutkan lima aktivitas sistem operasi yang merupakan contoh dari suatu managemen proses.
2. Definisikan perbedaan antara penjadualan short term, medium term dan long term.
3. Jelaskan tindakan yang diambil oleh sebuah kernel ketika alih konteks antar proses.
4. Informasi apa saja yang disimpan pada tabel proses saat alih konteks dari satu proses ke proses lain.
5. Di sistem UNIX terdapat banyak status proses yang dapat timbul (transisi) akibat event (eksternal) OS dan proses tersebut itu sendiri. Transisi state apa sajakah yang dapat ditimbulkan oleh proses itu sendiri. Sebutkan!
6. Apa keuntungan dan kekurangan dari:
   - • Komunikasi Simetrik dan asimetrik
   - • Automatic dan explicit buffering
   - • Send by copy dan send by reference
   - • Fixed-size dan variable sized messages
7. Jelaskan perbedaan short-term, medium-term dan long-term?
8. Jelaskan apa yang akan dilakukan oleh kernel kepada alih konteks ketika proses sedang berlangsung?
9. Beberapa single-user mikrokomputer sistem operasi seperti MS-DOS menyediakan sedikit atau tidak sama sekali arti dari pemrosesan yang konkuren. Diskusikan dampak yang paling mungkin ketika pemrosesan yang konkuren dimasukkan ke dalam suatu sistem operasi?
10. Perlihatkan semua kemungkinan keadaan dimana suatu proses dapat sedang berjalan, dan gambarkan diagram transisi keadaan yang menjelaskan bagaimana proses bergerak diantara state.
11. Apakah suatu proses memberikan 'issue' ke suatu disk I/O ketika, proses tersebut dalam 'ready' state, jelaskan?
12. Kernel menjaga suatu rekaman untuk setiap proses, disebut Proses Control Blocks (PCB). Ketika suatu proses sedang tidak berjalan, PCB berisi informasi tentang perlunya melakukan restart suatu proses dalam CPU. Jelaskan dua informasi yang harus dipunyai PCB.

2.11.2. *Thread*

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

3. Sebutkan dua perbedaan antara user level thread dan kernel thread. Saat kondisi bagaimana salah satu dari thread tersebut lebih baik
4. Jelaskan tindakan yang diambil oleh sebuah kernel saat alih konteks antara kernel level thread.
5. Sumber daya apa sajakah yang digunakan ketika sebuah thread dibuat? Apa yang membedakannya dengan pembentukan sebuah proses.
6. Tunjukkan tindakan yang diambil oleh sebuah thread library saat alih konteks antara user level thread.

2.11.3. Penjadualan CPU

1. Definisikan perbedaan antara penjadualan secara preemptive dan nonpreemptive!
2. Jelaskan mengapa penjadualan strict nonpreemptive tidak seperti yang digunakan di sebuah komputer pusat.
3. Apakah keuntungan menggunakan time quantum size di level yang berbeda dari sebuah antrian sistem multilevel?

Pertanyaan nomor 4 sampai dengan 5 dibawah menggunakan soal berikut:
Misal diberikan beberapa proses dibawah ini dengan panjang CPU burst (dalam milidetik)
Semua proses diasumsikan datang pada saat t=0

Tabel 2-1. Tabel untuk soal 4—5

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

4. Gambarkan 4 diagram Chart yang mengilustrasikan eksekusi dari proses-proses tersebut menggunakan FCFS, SJF, prioritas nonpreemptive dan round robin.
5. Hitung waktu tunggu dari setiap proses untuk setiap algoritma penjadualan.
6. Jelaskan perbedaan algoritma penjadualan berikut:
    - • FCFS
    - • Round Robin
    - • Antrian Multilevel feedback
7. Penjadualan CPU mendefinisikan suatu urutan eksekusi dari proses terjadual. Diberikan n buah proses yang akan dijadualkan dalam satu prosesor, berapa banyak kemungkinan penjadualan yang berbeda? berikan formula dari n.

8. Tentukan perbedaan antara penjadualan preemptive dan nonpreemptive (cooperative). Nyatakan kenapa nonpreemptive scheduling tidak dapat digunakan pada suatu komputer center. Di sistem komputer nonpreemptive, penjadualan yang lebih baik digunakan.

2.12. Rujukan

1. Avi Silberschatz, Peter Galvin, dan Greg Gagne, 2002, *Applied Operationg System Concepts*, 1stEd., John Wiley & Sons, Inc.

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

3. R.M. Samik-Ibrahim, 2001, Soal Mid Test 2001, Fakultas Ilmu Komputer, Universitas Indonesia.

4. *http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/Proses.PDF*

5. *http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/CPU-Scheduler.PDF*

6. *http://www.cs.nyu.edu/courses/spring02/v22.0202-002/lecture-03.html*

7. *http://www.risc.uni-linz.ac.at/people/schreine/papers/idimt97/multithread.gif*

8. *http://www.unet.univie.ac.at/aix/aixprggd/genprogc/figures/genpr68.jpg*

9. *http://www.unet.univie.ac.at/aix/aixprggd/genprogc/understanding_threads.htm*

10. *http://www.etnus.com/Support/docs/rel5/html/cli_guide/images/procs_n_threads8a.gif*

11. *http://www.etnus.com/Support/docs/rel5/html/cli_guide/procs_n_threads5.html*

12. *http://www.crackinguniversity2000.it/boooks/1575211025/ch6.htm*

13. *http://lass.cs.umass.edu/~shenoy/courses/fall01/labs/talab2.html*

14. *http://www.isbiel.ch/~myf/opsys1/Exercises/Chap4/Problems1.html*

15. *http://www.cee.hw.ac.uk/courses/5nm1/Exercises/2.html*

16. *http://www.cs.wisc.edu/~cao/cs537/midterm-answers1.txt*

2.13. Daftar Istilah

Algoritma

Algoritma FCFS

Algortima penjadualan

Algoritma Round Robin (RR)

Algoritma SJF

Alih Konteks

Antrian

Antrian Ready

CPU Burst

CPU utilization

Dispatcher

Dispatch Latency

Direct Mapping

Idle

Indirect Mapping

Interupsi

I/O

Java

Java Thread

Java Virtual Machine

Kernel

Lightweight

Multi Programming

Multiprocessor

New State

Nonpremptive

Penjadualan

Penjadualan CPU

Penjadualan CPU Long Term

Penjadualan CPU Short Term

Penjadualan FCFS (First Come First Serve)

Penjadualan Multiprocessor

Penjadualan Real Time

Penjadualan Round Robin

Penjadualan SJF (Shortest Job First)

Preemptive

Proses

Real Time Computing

Ready State

Response Time

Running State

Short-Remaining Time First (SRTF)

Sinkronisasi

Sistem

Sistem Batch

Sistem Operasi

Sistem Uniprosesor

Soft Real Time Computing

Starvation

State

Switching

Symmetric Multiprocessing(SMP)

Terminate

Thread

Throughput

Time Slicing

Time Units

Turnarround Time

Waiting State

Waiting Time

Bab 3

Sinkronisasi dan Deadlock

3.1. Sinkronisasi

Bab ini membicarakan proses-proses untuk saling berkordinasi. Bab ini juga akan menjawab pertanyaan-pertanyaan seperti, bagaimana proses bekerja dengan sumber daya yang dibagi-bagi.

Bagaimana memastikan hanya ada satu proses yang mengakses memori pada suatu saat? Bagaimana sinkronisasi benar-benar digunakan?

3.1.1. Latar Belakang

- Akses-akses yang dilakukan secara bersama-sama ke data yang sama, dapat menyebabkan data menjadi tidak konsisten.
- Untuk menjaga agar data tetap konsisten, dibutuhkan mekanisme-mekanisme untuk memastikan pemintaan ekseskusi dari proses yang bekerja.
- *Race Condition*: Situasi dimana beberapa proses mengakses dan memanipulasi data secara bersamaan. Nilai terakhir dari data bergantung dari proses mana yang selesai terakhir.
- Untuk menghindari *Race Condition*, proses-proses secara bersamaan harus disinkronisasikan.

3.1.1.1. Kasus Produsen-Konsumer

Dua proses berbagi sebuah buffer dengan ukuran yang tetap. Salah satunya produser, meletakkan informasi ke buffer yang lainnya. Konsumen mengambil informasi dari buffer. Ini juga dapat digeneralisasi untuk masalah yang memiliki m buah produsen dan n buah konsumen, tetapi kita hanya akan memfokuskan kasus dengan satu produsen dan satu konsumen karena diasumsikan dapat menyederhanakan solusi.

Masalah akan timbul ketika produsen ingin menaruh barang yang baru tetapi buffer sudah penuh. Solusi untuk produsen adalah istirahat (*sleep*) dan akan dibangunkan ketika konsumen telah mengambil satu atau lebih barang dari buffer. Biasanya jika konsumen ingin mengambil barang dari buffer dan melihat bahwa buffer sedang kosong, maka konsumen istirahat (*sleep*) sampai produsen meletakkan barang pada buffer dan membangunkan (*wake up*) consumer.

Pendekatan seperti ini terdengar cukup sederhana, tetapi hal ini dapat menggiring kita ke jenis masalah yang sama seperti *race condition* dengan spooler direktori.

Untuk mengetahui jumlah barang di buffer, kita membutuhkan sebuah variabel kita namakan count. Jika jumlah maksimum dairi barang yang dapat ditampung buffer adalah N, kode produser pertama kali akan mencoba untuk mengetahui apakah nilai count sama dengan nilai N. Jika itu terjadi maka produsen akan istirahat (*sleep*), tetapi jika nilai count tidak sama dengan N, produsen akan terus menambahkan barang dan menaikkan nilai count.

Sekarang mari kita kembali ke permasalahan race condition. Ini dapat terjadi karena akses ke count tidak dipaksakan. Situasi seperti itu mungkin dapat terjadi. Buffer sedang kosong dan konsumen baru saja membaca count untuk melihat apakah count bernilai 0. Pada saat itu, penjadual memutuskan untuk mengentikan proses konsumen sementara dan menjalakan produsen. Produsen memasukkan barang ke buffer, menaikkan nilai count, dan memberitahukan bahwa count sekarang bernilai 1. Pemikiran bahwa count baru saja bernilai 0 sehingga konsumen harus istirahat (*sleep*). Produsen memanggil fungsi *wake*

up untuk membangkitkan konsumen.

Sayangnya, konsumen secara logika belum istirahat. Jadi sinyal untuk membangkitkan konsumen, tidak dapat ditangkap oleh konsumen. Ketika konsumen bekerja berikutnya, konsumen akan memeriksa nilai count yang dibaca sebelumnya, dan mendapatkan nilai 0, kemudian konsumen istirahat (*sleep*) lagi. Cepat atau lambat produsen akan mengisi buffer dan juga pergi istirahat (*sleep*). Keduanya akan istirahat selamanya.

Inti permasalahannya disini adalah pesan untuk membangkitkan sebuah proses tidak tersampaikan. Jika pesan/ sinyal ini tersampaikan dengan baik, segalanya akan berjalan lancar.

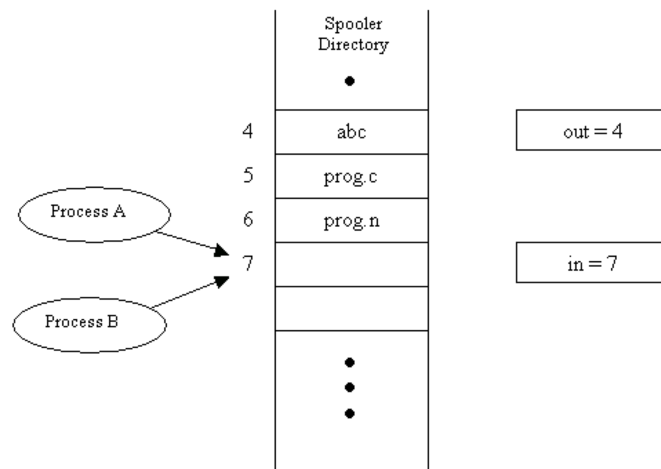
3.1.1.2. *Race Condition*

Race Condition adalah situasi di mana beberapa proses mengakses dan memanipulasi data bersama pada saat besamaan. Nilai akhir dari data bersama tersebut tergantung pada proses yang terakhir

selesai. Unutk mencegah *race condition*, proses-proses yang berjalan besamaan haus di disinkronisasi. Dalam beberapa sistem operasi, proses-proses yang berjalan bersamaan mungkin untuk membagi beberapa penyimpanan umum, masing-masing dapat melakukan proses baca (*read*) dan proses tulis (*write*). Penyimpanan bersama (*shared storage*) mungkin berada di memori utama atau berupa sebuah berkas bersama, lokasi dari memori bersama tidak merubah kealamian dari komunikasi atau masalah yang muncul. Untuk mengetahui bagaimana komunikasi antar proses bekerja, mari kita simak sebuah contoh sederhana, sebuah print spooler. Ketika sebuah proses ingin mencetak sebuah berkas, proses tersebut memasukkan nama berkas ke dalam sebuah spooler direktori yang khusus. Proses yang lain, printer daemon, secara periodik memeriksa untuk mengetahui jika ada banyak berkas yang akan dicetak, dan jika ada berkas yang sudah dicetak dihilangkan nama berkasnya dari direktori.

Bayangkan bahwa spooler direktori memiliki slot dengan jumlah yang sangat besar, diberi nomor 0, 1, 2, 3, 4,... masing-masing dapat memuat sebuah nama berkas. Juga bayangkan bahwa ada dua variabel bersama, out, penunjuk berkas berikutnya untuk dicetak, dan in, menunjuk slot kosong di direktori. Dua vaiabel tersebut dapat menamgami sebuah two-word berkas untuk semua proses. Dengan segera, slot 0, 1, 2, 3 kosong (berkas telah selesai dicetak), dan slot 4, 5, 6 sedang terisi (berisi nama dari berkas yang antre untuk dicetak). Lebih atau kurang secara besamaan, proses A dan B, mereka memutuskan untuk

antre untuk sebuah berkas untuk dicetak. Situasi seperti ini diperlihatkan oleh Gambar 3-1.



Gambar 3-1. Race Condition.

Dalam *Murphy's Law* kasus tesebut dapat terjadi. Proses A membaca in dan menyimpan nilai "7" di sebuah variabel lokal yang disebut next_free_slot. Sebuah clock interrupt terjadi dan CPU memutuskan bahwa proses A berjalan cukup lama, sehingga digantika oleh proses B. Proses B juga membaca in, dan juga mengambil nilai 7, sehingga menyimpan nama berkas di slot nomor 7 dan memperbaharui nilai in menjadi 8. Maka proses mati dan melakukan hal lain.

Akhirnya proses A berjalan lagi, dimulai dari tempat di mana proses tersebut mati. Hal ini terlihat dalam next_free_slot, ditemukan nilai 7 di sana, dan menulis nama berkas di slot nomor 7, menghapus nama berkas yang bau saja diletakkan oleh proses B. Kemudian proses A menghitung next_free_slot + 1, yang nilainya 8 dan memperbaharui nilai in menjadi 8. Direktori spooler sekarang secara internal konsisten, sehingga printer daemon tidak akan memberitahukan apa pun yang terjadi, tetapi poses B tidak akan mengambil output apa pun. Situasi seperti ini, dimana dua atau lebih proses melakukan proses reading atau writing beberapa shared data dan hasilnya bergantung pada ketepatan berjalan disebut *race condition*.

3.1.2. *Critical Section*

Bagaimana menghindari *race conditions*? Kunci untuk mencegah masalah ini dan di situasi yang lain yang melibatkan shared memori, shared berkas, and shared sumber daya yang lain adalah menemukan beberapa jalan untuk mencegah lebih dari satu proses untuk melakukan proses writing dan reading kepada shared data pada saat yang sama. Dengan kata lain kita memutuhkan *mutual exclusion*, sebuah jalan yang menjamin jika sebuah proses sedang menggunakan shared berkas, proses lain dikeluarkan dari pekerjaan yang sama. Kesulitan yang terjadi karena proses 2 mulai menggunakan variabel bersama sebelum proses 1 menyelesaikan tugasnya.

Masalah menghindari *race conditions* dapat juga diformulasikan secara abstrak. Bagian dari waktu, sebuah proses sedang sibuk melakukan perhitungan internal dan hal lain yang tidak enggiring ke kondisi *race conditions*. Bagaimana pun setiap kali sebuah proses mengakses shared memory atau shared berkas atau melakukan sesuatu yang kitis akan menggiring kepada *race conditions*. Bagian dari program dimana shaed memory diakses disebut *Critical Section* atau *Critical Region*.

Walau pun dapat mencegah *race conditions*, tapi tidak cukup untuk melakukan kerjasama antar proses secara pararel dengan baik dan efisien dalam menggunakan shared data. Kita butuh 4 kondisi agar menghasilkan solusi yang baik:

I. Tidak ada dua proses secara bersamaan masuk ke dalam citical section.
II. Tidak ada asumsi mengenai kecepatan atau jumlah cpu.
III. Tidak ada proses yang berjalan di luar critical secion yang dapat mengeblok proses lain.
IV. Tidak ada proses yang menunggu selamamya untuk masuk critical section.

Critical Section adalah sebuah segmen kode di mana sebuah proses yang mana sumber daya bersama diakses. Terdiri dari: *Entry Section*: kode yang digunakan untuk masuk ke dalam *critical section*

Critical Section: Kode di mana hanya ada satu proses yang dapat dieksekusi pada satu waktu

Exit Section: akhir dari *critical section*, mengizinkan proses lain

Remainder Section: kode istirahat setelah masuk ke *critical section*

Critical section harus melakukan ketiga aturan berikut:

Solusi yang diberikan harus memuaskan permintaaan berikut:

- *Mutual exclution*
- *eadlock free*
- *Starvation free*

Pendekatan yang mungkin untuk solusi proses sinkronisasi

i. Solusi Piranti lunak (Software solution)

- Tanpa Sinkronisasi.
- Dengan Sinkronisasi.
- Low-level primitives: *semaphore*
- High-level primitives: *monitors*

ii. Solusi Piranti Keras (Hardware solution)

3.1.2.1. *Mutual Exclusion*

Mutual Exclusion: Kondisi-kondisi untuk solusi

Tiga kondisi untuk menentukan *mutual Exclusion*

1. Tidak ada dua proses yang pada saat bersamaan berada di *critical region*.

2. Tidak ada proses yang berjalan diluar *critical region* yang bisa menghambat proses lain
3. Tidak ada proses yang tidak bisa masuk ke *critical region*

3.1.2.2. Solusi

Cara-cara memecahkan masalah

• Hanya dua proses, Po dan P1

• Struktur umum dari proses adalah Pi (proses lain Pj)

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

Gambar 3-2. Critical Section.

3.1.2.2.1. Algoritma 1

Disini kita akan mencoba membuat sebuah rangkaian solusi-solusi dari permasalahan yang makin meningkat kerumitannya.

Pada semua contoh, i adalah proses yang sedang berjalan, j adalah proses yang lain. Pada contoh ini *code*.

i. Shared variables

• int turn

Initially turn=0

• turn = i, *Pi can enter its critical section*

ii. Process Pi

```
do {
    while(turn!=1);
    critical section
    turn=j;
    remainder section
} while(1);
```

Gambar 3-3. Proses Pi.

iii. Memenuhi *mutual exclusion*, tapi bukan progress.

3.1.2.2.2. Algoritma 2

FLAG untuk setiap proses yang memberi STATE:

Setiap proses memantau suatu flag yang mengindikasikan ia ingin memasuki critical section. Dia memeriksa flag poses lain dan tidak akan memasuki critical section bila ada proses lain yang sedang masuk.

i. Shared variables

- boolean flag[2];
  initially flag [0] = flag [1] = false

- flag [i] = true , *Pi ready to enter its critical section*

ii. Process Pi

```
do {
      flag[i]:=true;
      while(turn!=1);
      critical section
      turn=j;
      remainder section
} while(1);
```

Gambar 3-4. Process Pi.

iii. Memenuhi *mutual exclusion*, tapi tidak memenuhi progess.

3.1.2.2.3. Algoritma 3

FLAG untuk meminta izin masuk:

- Setiap proses mengeset sebuah flag untuk meminta izin masuk. Lalu setiap proses mentoggle bit untuk mengizinkan yang lain untuk yang pertama
- Kode ini dijalankan untuk setiap proses i

```
Shared variables
F boolean flag[2];
initially flag[0] = flag[1] = false
F flag[i] = true;
```

Gambar 3-5. Kode.

Pi ready to enter its critical section

- Gabungan shared variables dari algorima 1 dan 2
- Process Pi

```
do {
      flag[i]:=true;
      turn = j;
      while(flag[j] and turn = j);
      critical section
      flag[i] = false;
      remainder section
} while(1);
```

Gambar 3-6. Process Pi.

- Memenuhi ketiga persyaratan, memecahkan persoalan *critical section* untuk kedua proses

3.1.2.2.4. Algoritma *Bakery*

Critical Section untuk n buah proses:

Sebelum memasukkan proses ke *critical section*, proses menerima sebuah nomor. Pemegang nomor terkecil masuk ke *critical section*. Jika ada dua proses atau lebih menerima nomor sama, maka proses dengan indeks terkecil yang dilayani terlebih dahulu untuk masuk ke critical section. Skema penomoran selalu naik secara berurut contoh: 1, 2, 3, 3, 3, 3, 4, 5,...

```
boolean choosing [n];
long long long int number [n];
/* 64 bit maybe okay for about 600 years */
Array structure elements are initiallized to false and 0
respectively
while (true) {
        choosing[i] = true;
        number[i] = max(number[0], ... [n-1]) + 1;
        choosing[i] = false;
        for (j = 0; j < n; j ++) {
            while (choosing[j]) {}
            while ((number[j] !=0) && ((number[j], j) <
            (number[i], i))) {}
        }
        number[i] = 0
}
```

Solves the critical-section problem
for n process

Gambar 3-7. Process Pi.

3.1.3. Solusi Hardware pada Sinkronisasi

Disabling Interrupts: Hanya untuk uni prosesor saja.

Atomic test and set: Returns parameter and sets parameter to true atomically.

```
while (test_and_set(lock));
/* critical section */
lock = false;
GET_LOCK: IF_CLEAR_THEN_SET_BIT_AND_SKIP (bit_address)
BRANCH GET_LOCK /* set failed */
                    /* set succeeded */
```

Gambar 3-8. Process Pi.

Harus hati-hati jika pendekatan ini untuk menyelesaikan *bounded-buffer* - harus menggunakan *round robin* - memerlukan kode yang dibuat di sekitar instruksi *lock*.

```
while (test_and_set(lock));
Boolean waiting[N];
int j; /* Takes on values from 0 to N - 1 */
Boolean key;
do {
        waiting[i] = TRUE;
        key = TRUE;
        while ( waiting[i] && key )
                key = test_and_set( lock ); /* Spin lock */
        waiting[i] = FALSE;

     /****** CRITICAL SECTION ********/
    j = ( i + 1 ) mod N;
    while ( ( j != i ) && ( ! waiting[ j ] ) )
    j = ( j + 1 ) % N;
    if ( j == i ) //Using Hardware
            lock = FALSE; //Test_and_set.
    else
            waiting[ j ] = FALSE;
    /******* REMAINDER SECTION *******/
} while (TRUE);
```

Gambar 3-9. Lock.

3.1.4. *Semaphore*

Jika kita ingin dapat melakukan proses tulis lebih rumit kita membutuhkan sebuah bahasa untuk melakukannya. Kita akhirnya medefinisikan semaphore yang kita asumsikan sebagai sebuah operasi atomik.

Semaphore adalah pendekatan yang diajukan oleh Djikstra, dengan prinsip bahwa dua proses atau lebih dapat bekerja sama dengan menggunakan penanda-penanda sederhana. Seperti proses dapat dipaksa berhenti pada suatu saat, sampai proses mendapatkan penanda tertentu itu. Sembarang kebutuhan koordinasi kompleks dapat dipenuhi dengan struktur penanda yang cocok untuk kebutuhan itu. Variabel khusus untuk penanda ini disebut semaphore.

Semaphore mempunyai dua sifat, yaitu:

i. Semaphore dapat diinisialisasi dengan nilai non-negatif.
ii. Terdapat dua operasi terhadap semaphore, yaitu Down dan Up. Usulan asli yang disampaikan Djikstra adalah operasi P dan V.

3.1.4.1. Operasi *Down*

Operasi ini menurunkan nilai semaphore, jika nilai semaphore menjadi non-positif maka proses yang mengeksekusinya diblocked.

```
Type Semaphore = Integer,

Procedure Down(Var: semaphore);
Begin

        s := s-1;
        if s <= 0 Then
        Begin
                Tempatkan antrian pada antrian untuk
        semaphore s
        Proses diblocked
        End;
End;
```

Gambar 3-10. Block.

Operasi Down adalah atomic, tak dapat diinterupsi sebelaum diselesaikan.Emnurunkan nilai, memeriksa nilai, menempatkan proses pada antrian dan memblocked sebagai instruksi tunggal. Sejak dimulai, tak ada proses alain yang dapat mengakses semaphore sampai operasi selesai atau diblocked.

3.1.4.2. Operasi *Up*

Operasi Up menakkan nilai semaphore. Jika satu proses atau lebih diblocked pada semaphore itu tak dapat menyelesaikan operasi Down, maka salah satu dipilih oleh system dan menyelesaikan operasi Down-nya. Urutan proses yang dipilih tidak ditentukan oleh Djikstra, dapat dipilih secara acak.

```
Type Semaphore = Integer,

Procedure Down(Var: semaphore);
Begin

        s := s + 1;
        if s <= 0 Then
        Begin
                Pindahkan satu proses P dari antrian untuk
        semaphore s
                Tempatkan proses P di senarai ready
        End;
End;
```

Gambar 3-11. Block.

Adanya semaphore mempermudah persoalan mutual exclusion. Skema penelesaian mutual exclusion mempunyai bagan sebagai berikut:

```
Cons N = 2;
Var S:semaphore;
Procedure enter_critical_section;
{
        mengerjakan kode-kode kritis
}

Procedure enter_noncritical_section;
{
        mengerjakan kode-kode tak kritis
}

ProcedureProses(i: integer);
Begin
        Repeat
        Down(s);
        Enter_critical_section;

        Up(s);
        Enter_noncritical_section;
        Forever
End;

Begin
    S:= 1;
    Parbegin
        Proses(0);
        Proses(1);
    ParEnd
End;
```

Gambar 3-12. Mutex.

Sebelum masuk *critical section*, proses melakukan Down. Bila berhasil maka proses masuk ke *critical section*. Bila tidak berhasil maka proses di-blocked atas semaphore itu. Proses yang diblocked akan dapat melanjutkan kembali bila proses yang ada di *critical section* keluar dan melakukan opersai up sehingga menjadikan proses yang diblocked ready dan melanjutkan sehingga opersi Down-nya berhasil.

3.1.5. Problem Klasik pada Sinkronisasi

Ada tiga hal yang selalu memjadi masalah pada proses sinkronisasi:

i. Problem *Bounded buffer*.
ii. Problem *Reades and Writer*.
iii. Problem *Dining Philosophers*.

3.1.5.1. Problem *Readers-Writers*

Problem lain yang terkenal adalah readers-writer problem yang memodelkan proses yang mengakses database. Sebagai contoh sebuah sistem pemesanan sebuah perusahaan penerbangan, dimana banyak proses berkompetisi berharap untuk membaca (*read*) dan menulis (*write*). Hal ini dapat diterima bahwa banyak proses membaca database pada saat yang sama, tetapi jika suatu proses sedang menulis database, tidak boleh ada proses lain yang mengakses database tersebut, termasuk membaca database tersebut.

Dalam solusi ini, pertama-tama pembaca mengakses database kemudian melakukan DOWN pada semaphore db.. Langkah selanjutnya readers hanya menaikkkan nilai sebuah counter. Hasil dari pembaca nilai counter diturunkan dan nilai terakhir dilakukan UP pada semaphore, mengizinkan memblok writer.

Misalkan selama sebuah reader menggunakan database, reader lain terus berdatangan. Karena ada dua reader pada saat bersamaan bukanlah sebuah masalah, maka reader yang kedua diterima, reader yang ketiga juga dapat diterima jika terus berdatangan reader-reader baru.

Sekarang misalkan writer berdatangan terus menerus. Writer tidak dapat diterima ke database karena writer hanya bisa mengakses data ke database secara ekslusif, jadi writer ditangguhkan. Nanti penambahan reader akan menunjukkan peningkatan. Selama paling tidak ada satu reader yang aktif, reader berikutnya jika datang akan diterima.

Sebagai konsekuensi dari strategi ini, selama terdapat suplai reader yang terus-menerus, mereka akan dilayani segera sesuai kedatanga mereka. Writer akan ditunda sampai tidak ada reader lagi. Jika sebuah reader baru tiba, katakan, setiap dua detik, dan masing-masing reader mendapatkan lima detik untuk melakukan tugasnya, writer tudak akan pernah mendapatkan kesempatan.

Untuk mencegah situasi seperti itu, program dapat ditulis agak sedikit berbeda: Ketika reader tiba dan writer menunggu, reader ditunda dibelakang writer yang justru diterima dengan segera. Dengan cara ini, writer tidak harus menunggu reader yang sedang aktif menyelesaikan pekerjaannya, tapi tidak perlu menunggu reader lain yang datang berturut-turut setelah itu.

3.1.5.2. Problem *Dining Philosopers*

Pada tahun 1965, Djikstra menyelesaikan sebuah masalah sinkronisasi yang beliau sebut dengan dining philisophers problem. Dining philosophers dapat diuraikan sebagai berikut:

Lima orang filosuf duduk mengelilingi sebuah meja bundar. Masing-masing filosof mempunyai sepiring spageti. Spageti-spageti tersebut sangat licin dan membutuhkan dua garpu untuk memakannya. Diantara sepiring spageti terdapat satu garpu.

Kehidupan para filosof terdiri dari dua periode, yaitu makan atau berpikir. Ketika seorang filosof lapar, dia berusaha untuk mendapatkan garpu kiri dan garpu kanan sekaligus. Jika sukses dalam mengambil dua garpu, filosof tersebut makan untuk sementara waktu, kemudian meletakkan kedua garpu dan melanjutkan berpikir.

Pertanyaan kuncinya adalah, dapatkah anda menulis program untuk masing-masing filosof yang melakukan apa yang harus mereka lakukan dan tidak pernah mengalami kebuntuan.

Prosedur take-fork menunggu sampai garpu-garpu yang sesuaididapatkan dan kemudian

menggunakannya. Sayangnya dari solusi ini ternyata salah. Seharusnya lima orang filosof mengambil garpu kirinya secara bersamaan. Tidak akan mungkin mereka mengambil garpu kanan mereka, dan akan terjadi deadlock.

Kita dapat memodifikasi program sehingga setelah mengambil garpu kiri, program memeriksa apakah garpu kanan meungkinkan untuk diambil. Jika garpu kanan tidak mungkin diambil, filosof tersebut meletakkan kembali garpu kirinya, menunggu untuk beberapa waktu, kemudia mengulangi proses yang sama. Usulan tersebut juga salah, walau pun dengan alasan yang berbeda. Dengan sedikit nasib buruk, semua filosof dapat memulai algoritma secara bersamaan, mengambil garpu kiri mereka, melihat garpu kanan mereka yang tidak mungkin untuk diambil, meletakkan kembali garpu kiri mereka, menunggu, mengambil garpu kiri mereka lagi secara bersamaan, dan begitu seterusnya. Situasi seperti ini dimana semua program terus berjalan secara tidak terbatas tetapi tidak ada perubahan/kemajuan yang dihasilkan disebut starvation.

Sekarang anda dapat berpikir "jika filosof dapat saja menunggu sebuah waktu acak sebagai pengganti waktu yang sama setelah tidak dapat mengambil garpu kiri dan kanan, kesempatan bahwa segala sesuatau akan berlanjut dalam kemandegan untuk beberapa jam adalah sangat kecil." Pemikiran seperti itu adalah benar,tapi beberapa aplikasi mengirimkan sebuah solusi yang selalu bekerja dan tidak ada kesalahan tidak seperti hsk nomor acak yang selalu berubah.

Sebelum mulai mengambil garpu, seorang filosof melakukan DOWN di mutex. Setelah menggantikan garpu dia harus melakukan UP di mutex. Dari segi teori, solusi ini cukup memadai. Dari segi praktek, solusi ini tetap memiliki masalah. Hanya ada

satu filosof yang dapat makan spageti dalam berbagai kesempatan. Dengan lima buah garpu, seharusnya kita bisa menyaksikan dua orang filosof makan spageti pada saat bersamaan.

Solusi yang diberikan diatas benar dan juga mengizinkan jumlah maksimum kegiatan paralel untuk sebuah jumlah filosf yang berubah-ubah ini menggunakan sebuah array, state, untuk merekam status seorang filosof apakah sedang makan (*eating*), berpikir (*think*), atau sedang lapar (*hungry*) karena sedang berusaha mengambil garpu. Seorang filosof hanya dapat berstatus makan (*eating*) jika tidak ada tetangganya yang sedang makan juga. Tetangga seorang filosof didefinisikan ole LEFT dan RIGHT.

Dengan kata lain, jika i = 2, maka tetangga kirinya (LEFT) = 1 dan tetangga kanannya (RIGHT) = 3. Program ini menggunakan sebuah array dari semaphore yang lapar (hungry) dapat ditahan jika garpu kiri atau kanannya sedang dipakai tetangganya. Catatan bahwa masing-masing proses menjalankan prosedur filosof sebagai kode utama, tetapi prosedur yang lain seperti take-forks, dan test adalah prosedur biasa dan bukan proses-proses yang terpisah.

3.1.6. Monitors

Solusi sinkronisasi ini dikemukakan oleh Hoare pada tahun 1974. Monitor adalah kumpulan prosedur, variabel dan struktur data di satu modul atau paket khusus. Proses dapat memanggil prosedur-prosedur kapan pun diinginkan. Tapi proses tak dapat mengakses struktur data internal dalam monitor secara langsung. Hanya lewat prosedur-prosedur yang dideklarasikan minitor untuk mengakses struktur internal.

Properti-properti monitor adalah sebagai berikut:

i. Variabel-variabel data lokal, hanya dapat diakses oleh prosedur-prosedur dala monitor dan tidak oleh prosedur di luar monitor.

ii. Hanya satu proses yang dapat aktif di monitor pada satu saat. Kompilator harus mengimplementasi ini(mutual exclusion).

iii. Terdapat cara agar proses yang tidak dapat berlangsung di-blocked. Menambahkan variabel-variabel kondisi, dengan dua operasi, yaitu Wait dan Signal.

iv. ait: Ketika prosedur monitor tidak dapat berkanjut (misal producer menemui buffer penuh) menyebabkan proses pemanggil diblocked dan mengizinkan proses lain masuk monitor.

v. Signal: Proses membangunkan partner-nya yang sedang diblocked dengan signal pada variabel kondisi yang sedang ditunggu partnernya.

vi. Versi Hoare: Setelah signal, membangunkan proses baru agar berjalan dan menunda proses lain. vii. Versi Brinch Hansen: Setelah melakukan signal, proses segera keluar dari monitor.

Dengan memaksakan disiplin hanya satu proses pada satu saat yang berjalan pada monitor, monitor menyediakan fasilitas mutual exclusion. Variabel-variabel data dalam monitor hanya dapat diakses oleh satu proses pada satu saat. Struktur data bersama dapat dilindungi dengan menempatkannya dalam monitor. Jika data pada monitor merepresentasikan sumber daya, maka monitor menyediakan fasilitas mutual exclusion dalam mengakses sumber daya itu.

3.2. *Deadlock*

Pada pembahasan di atas telah dikenal suatu istilah yang populer pada bagian *semaphores*, yaitu *deadlock*. Secara sederhana *deadlock* dapat terjadi dan menjadi hal yang merugikan, jika pada suatu saat ada suatu proses yang memakai sumber daya dan ada proses lain yang menunggunya. Bagaimanakah *deadlock* itu yang sebenarnya? Bagaimanakah cara penanggulangannya?

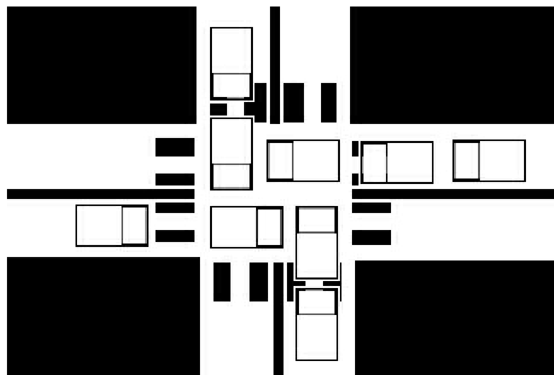
3.2.1. Latar Belakang

Misalkan pada suatu komputer terdapat dua buah program, sebuah *tape drive* dan sebuah *printer*. Program A mengontrol *tape* drive, sementara program B mengontrol *printer*. Setelah beberapa saat, program A meminta *printer*, tapi *printer* masih digunakan. Berikutnya, B meminta *tape drive*, sedangkan A masih mengontrol *tape drive*. Dua program tersebut memegang kontrol terhadap sumber daya yang dibutuhkan oleh program yang lain. Tidak ada yang dapat melanjutkan proses masing-masing sampai program yang lain memberikan sumber dayanya, tetapi tidak ada yang mengalah. Kondisi inilah yang disebut *Deadlock* atau pada beberapa buku disebut *Deadly Embrace Deadlock* yang mungkin dapat terjadi pada suatu proses disebabkan proses itu menunggu suatu kejadian tertentu yang tidak akan pernah terjadi. Dua atau lebih proses dikatakan berada dalam kondisi *deadlock*, bila setiap proses yang ada menunggu suatu kejadian yang hanya dapat dilakukan oleh proses lain dalam himpunan tersebut.

Terdapat kaitan antara *overhead* dari mekanisme koreksi dan manfaat dari koreksi *deadlock* itu sendiri. Pada beberapa kasus, *overhead* atau ongkos yang harus dibayar untuk membuat sistem bebas deadlock menjadi hal yang terlalu mahal dibandingkan jika mengabaikannya. Sementara pada kasus lain, seperti pada *real-time process control*, mengizinkan *deadlock* akan membuat sistem menjadi kacau dan membuat sistem tersebut tidak berguna.

Contoh berikut ini terjadi pada sebuah persimpangan jalan. Beberapa hal yang dapat membuat *deadlock* pada suatu persimpangan, yaitu:

- Terdapat satu jalur pada jalan.
- Mobil digambarkan sebagai proses yang sedang menuju sumber daya.
- Untuk mengatasinya beberapa mobil harus *preempt* (mundur).
- Sangat memungkinkan untuk terjadinya *starvation* (kondisi proses tak akan mendapatkan sumber daya).



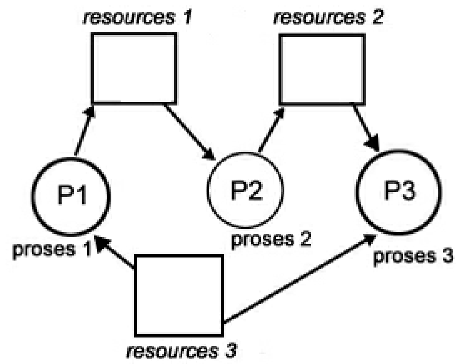
Gambar 3-13. Persimpangan.

3.2.2. *Resources-Allocation Graph*

Sebuah cara visual (matematika) untuk menentukan apakah ada *deadlock*, atau kemungkinan terjadinya. G = (V, E) Graf berisi *node and edge*. *Node* V terdiri dari proses-proses = {P1, P2, P3, ...} dan jenis *resource*. {R1, R2, ...} *Edge* E adalah (Pi, Rj) atau (Ri, Pj) Sebuah panah dari *process* ke *resource* menandakan proses meminta *resource*.

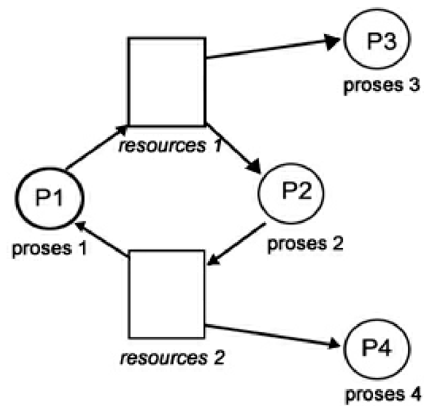
Sebuah panah dari *resource* ke *process* menunjukkan sebuah *instance* dari *resource* telah dtempatkan ke proses. *Process* adalah lingkaran, *resource* adalah kotak; titik-titik merepresentasikan

jumlah *instance* dari *resource* Dalam tipe. Meminta poin-poin ke kotak, perintah datang dari titik.



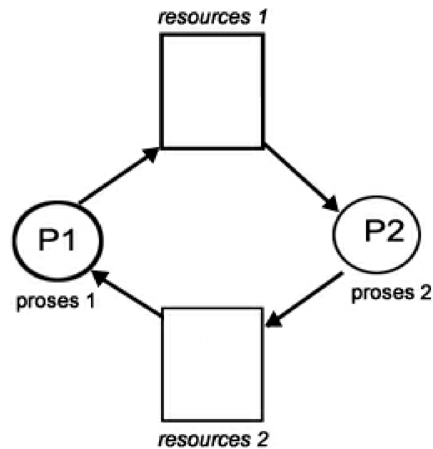
Gambar 3-14. Graph.

Jika graf tidak berisi lingkaran, maka tidak ada proses yang *deadlock*.
Jika membentuk lingkaran, maka:
i. jika tipe *resource* memiliki banyak *instance*, maka *deadlock* DAPAT ada.



Gambar 3-15. Non Deadlock.

ii. jika setiap tipe *resource* mempunyai satu *instance*, maka *deadlock* telah terjadi.



Gambar 3-16. Deadlock.

3.2.3. Model Sistem

Menurut *Coffman* dalam bukunya *"Operating System"* menyebutkan empat syarat bagi terjadinya *deadlock*, yaitu:

i. Mutual Exclusion
Suatu kondisi dimana setiap sumber daya diberikan tepat pada satu proses pada suatu waktu.

ii. Hold and Wait
Kondisi yang menyatakan proses-proses yang sedang memakai suatu sumber daya dapat meminta sumber daya yang lain.

iii. Non-pre-emptive
Kondisi dimana suatu sumber daya yang sedang berada pada suatu proses tidak dapat diambil secara paksa dari proses tersebut,sampai proses itu melepaskannya.

iv. Circular Wait
Kondisi yang menyatakan bahwa adanya rantai saling meminta sumber daya yang dimiliki oleh suatu proses oleh proses lainnya.

3.2.4. Strategi menghadapi *Deadlock*

Strategi untuk menghadapi *deadlock* dapat dibagi menjadi tiga pendekatan, yaitu:

i. Mengabaikan adanya *deadlock*.

ii. Memastikan bahwa *deadlock* tidak akan pernah ada, baik dengan metode. Pencegahan, dengan mencegah empat kondisi *deadlock* agar tidak akan pernah terjadi. Metode Menghindari *deadlock*, yaitu mengizinkan empat kondisi *deadlock*, tetapi menghentikan setiap proses yang kemungkinan mencapai *deadlock*.

iii. Membiarkan *deadlock* untuk terjadi, pendekatan ini membutuhkan dua metode yang saling mendukung, yaitu:
• Pendeteksian *deadlock*, untuk mengidentifikasi ketika *deadlock* terjadi.
• Pemulihan *deadlock*, mengembalikan kembali sumber daya yang dibutuhkan pada proses yang memintanya.

Dari penjabaran pendekatan diatas, terdapat empat metode untuk mengatasi *deadlock* yang akan terjadi, yaitu:

3.2.4.1. Strategi *Ostrich*

Pendekatan yang paling sederhana adalah dengan menggunakan strategi burung unta: masukkan kepala dalam pasir dan seolah-olah tidak pernah ada masalah sama sekali. Beragam pendapat muncul berkaitan dengan strategi ini. Menurut para ahli Matematika, cara ini sama sekali tidak dapat diterima dan semua keadaan *deadlock* harus ditangani. Sementara menurut para ahli Teknik, jika komputer lebih sering mengalami kerusakkan disebabkan oleh kegagalan *hardware*, *error* pada kompilator atau *bugs* pada sistem operasi. Maka ongkos yang dibayar untuk melakukan penanganan *deadlock* sangatlah besar dan lebih baik mengabaikan keadaan *deadlock* tersebut. Metode ini diterapkan pada sistem operasi UNIX dan MINIX.

3.2.5. Mencegah *Deadlock*

Metode ini merupakan metode yang paling sering digunakan. Metode Pencegahan dianggap sebagai solusi yang bersih dipandang dari sudut tercegahnya *deadlock*. Tetapi pencgahan akan mengakibatkan kinerja utilisasi sumber daya yang buruk.

Metode pencegahan menggunakan pendekatan dengan cara meniadakan empat syarat yang dapatmenyebabkan deadlock terjadi pada saat eksekusi Coffman (1971).

Syarat pertama yang akan dapat ditiadakan adalah *Mutual Exclusion*, jika tidak ada sumber daya yang secara khusus diperuntukkan bagi suatu proses maka tidak akan pernah terjadi *deadlock*. Namun jika membiarkan ada dua atau lebih proses

mengakses sebuah sumber daya yang sama akan menyebabkan *chaos*. Langkah yang digunakan adalah dengan *spooling* sumber daya, yaitu dengan mengantrikan *job-job* pada antrian dan akan dilayani satu-satu.

Beberapa masalah yang mungkin terjadi adalah:

i. Tidak semua dapat di-*spool*, tabel proses sendiri tidak mungkin untuk di-*spool*

ii. Kompetisi pada ruang disk untuk *spooling* sendiri dapat mengarah pada *deadlock*

Hal inilah yang menyebabkan mengapa syarat pertama tidak dapat ditiadakan, jadi *mutual exclusion* benar-benar tidak dapat dihilangkan.

Cara kedua dengan meniadakan kondisi *hold and wait* terlihat lebih menjanjikan. Jika suatu proses yang sedang menggunakan sumber daya dapat dicegah agar tidak dapat menunggu sumber daya yang lain, maka *deadlock* dapat dicegah. Langkah yang digunakan adalah dengan membuat proses agar meminta sumber daya yang mereka butuhkan pada awal proses sehingga dapat dialokasikan sumber daya yang dibutuhkan. Namun jika terdapat sumber daya yang sedang terpakai maka proses tersebut tidak dapat memulai prosesnya.

Masalah yang mungkin terjadi:

i. Sulitnya mengetahui berapa sumber daya yang dibutuhkan pada awal proses

ii. Tidak optimalnya pengunaan sumber daya jika ada sumber daya yang digunakan hanya beberapa waktu dan tidak digunakan tapi tetap dimiliki oleh suatu proses yang telah memintanya dari awal.

Meniadakan syarat ketiga *non preemptive* ternyata tidak lebih menjanjikan dari meniadakan syarat kedua, karena dengan meniadakan syarat ketiga maka suatu proses dapat dihentikan ditengah jalan. Hal ini tidak dimungkinkan karena hasil dari suatu proses yang dihentikan menjadi tidak baik.

Cara terakhir adalah dengan meniadakan syarat keempat *circular wait*. Terdapat dua pendekatan, yaitu:

i. Mengatur agar setiap proses hanya dapat menggunakan sebuah sumber daya pada suatu waktu, jika menginginkan sumber daya lain maka sumber daya yang dimiliki harus dilepas.

ii. Membuat penomoran pada proses-proses yang mengakses sumber daya. Suatu proses dimungkinkan untuk dapat

meminta sumber daya kapan pun, tetapi permintaannya harus dibuat terurut.

Masalah yang mungkin terjadi dengan mengatur bahwa setiap proses hanya dapat memiliki satu proses adalah bahwa tidak semua proses hanya membutuhkan satu sumber daya, untuk suatu proses yang kompleks dibutuhkan banyak sumber daya pada saat yang bersamaan. Sedangkan dengan penomoran masalah yang dihadapi adalah tidak terdapatnya suatu penomoran yang dapat memuaskan semua pihak.

Secara ringkas pendekatan yang digunakan pada metode pencegahan deadlock dan masalah-masalah yang menghambat-nya, terangkum dalam tabel dibawah ini.

Tabel 3-1. Tabel Deadlock

Syarat	**Langkah**	**Kelemahan**
Mutual Exclusion	*Spooling* sumber daya	Dapat menyebabkan *chaos*
Hold and Wait	Meminta sumber daya di awal	Sulit memperkirakan di awal dan tidak optimal
No Pre-emptive	Mengambil sumber daya di tengah proses	Hasil proses tidak akan baik
Circular Wait	Penomoran permintaan sumber daya	Tidak ada penomoran yang memuaskan semua pihak

3.2.6. Menghindari *Deadlock*

Pendekatan metode ini adalah dengan hanya memberi kesempatan ke permintaan sumber daya yang tidak mungkin akan menyebabkan *deadlock*. Metode ini memeriksa dampak pemberian akses pada suatu proses, jika pemberian akses tidak mungkin menuju kepada *deadlock*, maka sumber daya akan diberikan pada proses yang meminta. Jika tidak aman, proses yang meminta akan di-*suspend* sampai suatu waktu permintaannya aman untuk diberikan. Kondisi ini terjadi ketika setelah sumber daya yang sebelumnya dipegang oleh proses lain telah dilepaskan.

Kondisi aman yang dimaksudkan selanjutnya disebut sebagai *safe-state*, sedangkan keadaan yang tidak memungkinkan untuk diberikan sumber daya yang diminta disebut *unsafe-state*.

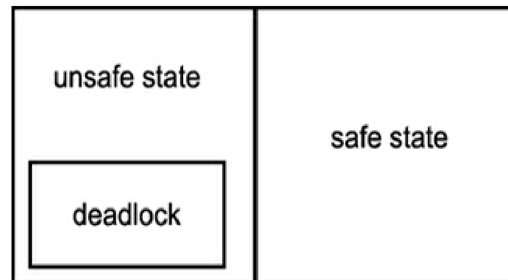
3.2.6.1. Kondisi Aman (*Safe state*)

Suatu keadaan dapat dinyatakan sebagai *safe state* jika tidak terjadi *deadlock* dan terdapat cara untuk memenuhi semua

permintaan sumber daya yang ditunda tanpa menghasilkan *deadlock*. Dengan cara mengikuti urutan tertentu.

3.2.6.2. Kondisi Tak Aman (*Unsafe state*)

Suatu *state* dinyatakan sebagai *state* tak selamat (*unsafe state*) jika tidak terdapat cara untuk memenuhi semua permintaaan yang saat ini ditunda dengan menjalankan proses-proses dengan suatu urutan.



Gambar 3-17. Safe.

3.2.7. Algoritma *Bankir*

Algoritma penjadualan ini diungkapkan oleh Dijkstra (1965) lebih dikenal dengan nama Algoritma Bankir. Model ini menggunakan suatu kota kecil sebagai percontohan dengan suatu bank sebagai sistem operasi, pinjaman sebagai sumber daya dan peminjam sebagai proses yang membutuhkan sumber daya.

Deadlock akan terjadi apabila terdapat seorang peminjam yang belum mengembalikan uangnya dan ingin meminjam kembali, padahal uang yang belum dikembalikan tadi dibutuhkan oleh peminjam lain yang juga belum mengembalikan uang pinjamannya.

Beberapa kelemahan algoritma Bankir Tanenbaum (1992), Stallings (1995) dan Deitel (1990) adalah sebagai berikut:

i. Sulit untuk mengetahui seluruh sumber daya yang dibutuhkan proses pada awal eksekusi.

ii. Jumlah proses yang tidak tetap dan berubah-ubah.

iii. Sumber daya yang tadinya tersedia dapat saja menjadi tidak tersedia kembali.

iv. Proses-proses yang dieksekusi haruslah tidak dibatasi oleh kebutuhan sinkronisasi antar proses.

v. Algoritma ini menghendaki memberikan semua permintaan selama waktu yang berhingga.

3.2.8. Mendeteksi *Deadlock* dan Memulihkan *Deadlock*

Metode ini mengunakan pendekatan dengan teknik untuk menentukan apakah *deadlock* sedang terjadi serta proses-proses dan sumber daya yang terlibat dalam deadlock tersebut. Setelah kondisi *deadlock* dapat dideteksi, maka langkah pemulihan dari kondisi *deadlock* dapat segera dilakukan. Langkah pemulihan tersebut adalah dengan memperoleh sumber daya yang diperlukan oleh proses-proses yang membutuhkannya. Beberapa cara digunakan untuk mendapatkan sumber daya yang diperlukan, yaitu dengan terminasi proses dan *pre-emption* (mundur) suatu proses. Metode ini banyak digunakan pada komputer *mainframe* berukuran besar.

3.2.8.1. Terminasi Proses

Metode ini akan menghapus proses-proses yang terlibat pada kondisi *deadlock* dengan mengacu pada beberapa syarat. Beberapa syarat yang termasuk dalam metode ini adalah, sebagai berikut:

- Menghapus semua proses yang terlibat dalam kondisi *deadlock* (solusi ini terlalu mahal).
- Menghapus satu persatu proses yang terlibat, sampai kondisi *deadlock* dapat diatasi (memakan banyak waktu).
- Menghapus proses berdasarkan prioritas, waktu eksekusi, waktu untuk selesai, dan kedalaman dari *rollback*.

3.2.8.2. *Resources Preemption*

Metode ini lebih menekankan kepada bagaimana menghambat suatu proses dan sumber daya, agar tidak terjebak pada *unsafe condition*.

Beberapa langkahnya, yaitu:

- Pilih salah satu - proses dan sumber daya yang akan di-*preempt*.
- *Rollback* ke *safe state* yang sebelumnya telah terjadi.
- Mencegah suatu proses agar tidak terjebak pada *starvation* karena metode ini.

3.3. Kesimpulan

Untuk mengatasi problem critical section dapat digunakan berbagai solusi software. Namun masalah yang akan timbul dengan solusi software adalah solusi software tidak mampu menangani masalah yang lebih berat dari critical section. Tetapi Semaphores mampu menanganinya, terlebih jika hardware yang digunakan mendukung maka akan memudahkan dalam menghadapi problem sinkronisasi.

Berbagai contoh klasik problem sinkronisasi berguna untuk mengecek setiap skema baru sinkronisasi. Monitor termasuk ke dalam level tertinggi mekanisme sinkronisasi yang berguna untuk mengkoordinir aktivitas dari banyak thread ketika mengakses data melalui pernyataan yang telah disinkronisasi

Kondisi *deadlock* akan dapat terjadi jika terdapat dua atau lebih proses yang akan mengakses sumber daya yang sedang dipakai oleh proses yang lainnya. Pendekatan untuk mengatasi *deadlock* dipakai tiga buah pendekatan, yaitu:

- Memastikan bahwa tidak pernah dicapai kondisi *deadlock*
- Membiarkan *deadlock* untuk terjadi dan memulihkannya
- Mengabaikan apa pun *deadlock* yang terjadi

Dari ketiga pendekatan diatas, dapat diturunkan menjadi empat buah metode untuk mengatasi *deadlock*, yaitu:

- Pencegahan *deadlock*
- Menghindari *deadlock*
- Mendeteksi *deadlock*
- Pemulihan *deadlock*

Namun pada sebagian besar Sistem Operasi dewasa ini mereka lebih condong menggunakan pendekatan untuk mengabaikan semua *deadlock* yang terjadi Silberschatz (1994) merumuskan sebuah strategi penanggulangan deadlock terpadu yang dapat disesuaikan dengan kondisi dan situasi yang berbeda, strateginya sendiri berbunyi:

1. Kelompokkan sumber daya kedalam kelas yang berbeda
2. Gunakan strategi pengurutan linear untuk mencegah kondisi *circular wait*yang nantinya akan mencegah deadlock diantara kelas sumber daya
3. Gunakan algoritma yang paling cocok untuk suatu kelas sumber daya yang berbeda satu dengan yang lain

3.4. Latihan

1. Proses dapat meminta berbagai kombinasi dari sumber daya dibawah ini: *CDROM*, *soundcard* dan *floppy*. Jelaskan tiga macam pencegahan deadlock skema yang meniadakan:
    - *Hold and Wait*
    - *Circular Wait*
    - *No Preemption*

2. Diasumsikan proses P0 memegang sumber daya R2 dan R3, meminta sumber daya R4; P1 menggunakan R4 dan meminta R1; P2 menggunakan R1 dan meminta R3 . Gambarkan *Wait-for Graph*. Apakah sistem terjebak dalam *deadlock*? Jika ya, tunjukkan proses mana yang menyebabkan *deadlock*. Jika tidak, tunjukkan urutan proses untuk selesai.
3. User x telah menggunakan 7 printer dan harus menggunakan 10 printer. User y telah menggunakan 1 printer dan akan memerlukan paling banyak 4 printer. User z telah menggunakan 2 printer dan akan menggunakan paling banyak 4 printer. Setiap user pada saat ini meminta 1 printer. Kepada siapakah OS akan memberikan grant printer tersebut dan tunjukkan "safe sequence" yang ada sehingga tidak terjadi deadlock.
4. Pernyataan manakah yang benar mengenai deadlock:
    1. Pencegahan deadlock lebih sulit dilakukan (implementasi) daripada menghindari deadlock.
    2. Deteksi deadlock dipilih karena utilisasi dari resources dapat lebih optimal.
    3. Salah satu prasyarat untuk melakukan deteksi deadlock adalah: hold and wait.
    4. Algoritma Banker’s (Djikstra) tidak dapat menghindari terjadinya deadlock.
    5. Suatu sistem jika berada dalam keadaan tidak aman: "unsafe", berarti telah terjadi deadlock.
5. User 1 sedang menggunakan x printers dan memerlukan total n printers. Kondisi umum adalah: y < -12, n < -12, x < -y, m < -n. State ini safe jika dan hanya jika:
    - i. x+n<-12 dan y+m<-12 dan x+m<-12
    - ii. x+n<-12 dan y+m<12 dan x+m<-12
    - iii. x+n<-12 atau(or) y+m<-12 dan x+m<-12
    - iv. x+m<-12
    - v. Semua statement diatas menjamin: safe state

3.5. Rujukan

3.5.1. Rujukan Sinkronisasi

Silberschatz, A., Gagne, G. dan Galvin, P., "*Applied Operating System Concept*", John Wiley and Sons Inc., 2000

Hariyanto, B.,"*Sistem Operasi*", Bandung: Informatika, Desember 1997

Tanenbaum, Andrew S., "*Modern Operating Systems*", Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1992

3.5.2. Rujukan *Deadlock*

Silberschatz, A., Gagne, G. dan Galvin, P., "*Applied Operating System Concept*", John Wiley and Sons Inc., 2000

Hariyanto, B.,"Sistem Operasi", Bandung: Informatika, Desember 1997

Tanenbaum, Andrew S., "*Modern Operating Systems*", Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1992

Coffman, E.G., Jr., M.J. Elphick dan A. Shoshani, "*System Deadlocks*", Computing surveys, Vol.3, No.2, June 1971

Deitel, H.M., "*Operating Systems*", 2nd Edition, Massachusetts: Addison-WesleyPublishing Company, 1990

Havender, J.W., "*Avoiding Deadlock in Multitasking Systems*", IBM Systems Journal, Vol.7, No.2, 1968. *97*

Bab 4

Memori

4.1. Latar Belakang

Memori merupakan inti dari sistem komputer modern. CPU mengambil instruksi dari memori sesuai yang ada pada program counter. Instruksi dapat berupa menempatkan/ menyimpan dari/ ke alamat di memori, penambahan, dan sebagainya. Dalam managemen memori ini, kita akan membahas bagaimana urutan alamat memori yang dibuat oleh program yang berjalan.

4.1.1. Pengikatan Alamat

Dalam banyak kasus, program akan berada dalam beberapa tahapan sebelum dieksekusi. Alamat-alamat yang dbutuhkan mungkin saja direpresentasikan dalam cara yang berbeda dalam tahapan-tahapan ini. Alamat dalam kode program masih berupa simbolik. Alamat ini akan diikat oleh kompilator ke alamat memori yang dapat diakses (misalkan 14 byte, mulai dari sebuah modul). Kemudian *linkage editor* dan *loader*, akan mengikat alamat fisiknya (misalkan 17014). Setiap pengikatan akan memetakan suatu ruang alamat ke lainnya.

Secara klasik, instruksi pengikatan dan data ke alamat memori dapat dilakukan dalam beberapa tahap:

- waktu *compile*: jika diketahui pada waktu compile, dimana proses ditempatkan di memori. Untuk kemudian kode absolutnya dapat di buat. Jika keumdian amat awalnya berubah, maka harus di compile ulang.
- waktu penempatan: Jika tidak diketahui dimana poses ditempatkan di memori, maka kompilator harus mmbuagt kode yang dapat dialokasikan. Dalam kasus pengikatan akan ditunda sampai waktu penempatan. Jika alamat awalnya berubah, kita hanya perlu menempatkan ulang kode, untuk menyesuaikan dengan perubahan.

- waktu eksekusi: Jika proses dapat dipindahkan dari suatu segmen memori ke lainnya selama dieksekusi. Pengikatan akan ditunda sampai *run-time*.

4.1.2. Ruang Alamat Fisik dan Logik

Alamat yang dibuat CPU akan merujuk ke sebuah alamat logik. Sedangkan alamat yang dilihat oleh memori adalah alamat yang dimasukkan ke register di memori, merujuk pada alamat fisik pada pengikatan alamat, waktu *compile* dan waktu penempatan mnghasilkan daerah dimana alamat logik dan alamat fisik sama. Sedangkan pada waktu eksekusi menghasilkan alamat fisik dan logik yang berbeda.

Kita biasanya menyebut alamat logik dengan alamat virtual. Kumpulan alamat logik yang dibuat oleh program adalah ruag alamat logik. Kumpulan alamat fisik yang berkoresponddensi dengan alamat logik sibut ruang alamat fisik. Pemetaan dari virtual ke alamat fisik dialkukan oleh *Memory-Management Unit* (MMU), yang merupakan sebuah perangkat keras.

Register utamanya disebut relocation-register. Nilai pada relocation register bertambah setiap alamat dibuat oleh proses pengguna, pada waktu yang sama alamat ini dikirim ke memori. Program pengguna tidak dapat langsung mengakses memori. Ketika ada program yang menunjuk ke alamat memori, kemudian mengoperasikannya, dan menaruh lagi di memori, akan di lokasikan awal oleh MMU, karena program pengguna hanya bernterkasi dengan alamat logik.

Konsep untuk memisahkan ruang alamat logik dan ruang alamat fisik, adalah inti dari managemen memori yang baik.

4.1.3. Penempatan Dinamis

Telah kita ketahui seluruh proses dan data berada memori fisik ketika di eksekusi. Ukuran dari memori fisik terbatas. Untuk mendapatkan utilisasi ruang memori yang baik, kita melakukan penempatan dinamis. Dengan penempatan dinamis, sebuah rutin tidak akan ditempatkan sampai dipanggil. Semua rutin diletakan di disk, dalam format yang dapat di lokasikan ulang. Program utama di tempatkan di memori dan dieksekusi. Jika sebuah rutin memanggil rutin lainnya, maka akan di cek dulu apakah rutin yang dipanggil ada di dalam memori atau tidak, jika tidak ada maka linkage loader dipanggil untuk menempatkan rutin yang diinginkan ke memori dan memperbaharui tabel alamat program untuk menyesuaikan perubahan. Kemudian kontrol diletakan pada rutin yang baru ditempatkan.

Keuntungan dari penempatan dinamis adalah rutin yang tidak digunakan tidak pernah ditempatkan. Metode ini berguna untuk kode dalam jumlah banyak, ketika muncul kasus-kasus yang tidak lazim, seperti rutin yang salah. Dalam kode yag besar, walau pun ukuran kode besar, tapi yang ditempatkan dapat jauh lebih kecil.

Penempatan Dinamis tidak didukung oleh sistem operasi. Ini adalah tanggung-jawab para pengguna untuk merancang program yang mengambil keuntungan dari metode ini. Sistem Operasi dapat membantu pembuat program dengan menyediakan libary rutin untuk mengimplementasi penempatan dinamis.

4.1.4. Perhubungan Dinamis dan Berbagi *Library*

Pada proses dengan banyak langkah, ditemukan juga perhubungan-perhubungan *library* yang dinamis. Beberapa sistem operasi hanya mendukung perhubungan yang dinamis, dimana sistem bahasa *library* diperlakukan seperti objek modul yang lain, dan disatukan oleh pemuat kedalam tampilan program biner.

Konsep perhubungan dinamis, serupa dengan konsep penempatan dinamis. Penempatan lebih banyak ditunda selama waktu eksekusi, dari pada lama penundaan oleh perhubungan dinamis. Keistimewaan ini biasanya digunakan dalam *library* sistem, seperti *library* bahasa sub-rutin. Tanpa fasilitas ini, semua program dalam sebuah sistem, harus mempunyai kopi dari libary bahasa mereka (atau setidaknya referensi rutin oleh program) termasuk dalam tampilan yang dapat dieksekusi. Kebutuhan ini sangat boros baik untuk disk, mau pun memori utama. Dengan penempatan dinamis, sebuah potongan dimasukkan kedalam tampilan untuk setiap rujukan *library* subrutin. Potongan ini adalah sebuah bagian kecil dari kode yang menunjukan bagaimana mealokasikan libary rutin di memori denga tepat, atau bagaimana menempatkan *library* jika rutin belum ada.

Ketika potongan ini dieksekusi, dia akan memeriksa dan melihat apakah rutin yang dibutuhkan sudah ada di memory. Jika rutin yang dibutuhkan tidak ada di memori, program akan menempatkannya ke memori. Jika rutin yang dibutuhkan ada dimemori, maka potongan akan mengganti dirinya dengan alamat dari rutin, dan mengeksekusi rutin. Demikianlah, berikutnya ketika segmentasi kode dicapai, rutin *library* dieksekusi secara langsung, dengan begini tidak ada biaya untuk penghubungan dinamis. Dalam skema ini semua proses yang menggunakan sebuah *library* bahasa, mengeksekusi hanya satu dari kopi kode *library*.

Fasilitas ini dapat diperluas menjadi pembaharuan *library* (seperti perbaikan bugs). Sebuah libary dapat ditempatkan lagi dengan versi yang lebih baru dan semua program yang merujuk ke *library*

akan secara otomatis menggunakan versi yang baru. Tanpa penempatan dinamis, semua program akan akan membutuhkan penempatan kembali, untuk dapat mengakses *library* yang baru. Jadi semua program tidak secara sengaja mengeksekusi yang baru, perubahan versi *library*, informasi versi dapat dimasukkan kedalam memori, dan setiap program menggunakan informasi versi untuk memutuskan versi mana yang akan digunakan dari kopi *library*. Sedikit perubahan akan tetap meggunakan nomor versi yang sama, sedangkan perubhan besar akan menambah satu versi seblumnya. Karenanya program yang dikompile dengan versi yang baru akan dipengaruhi dengan perubahan yang terdapat di dalamnya. Program lain yang berhubungan sebelum *library* baru diinstal, akan terus menggunakan *library* lama. Sistem ini juga dikenal sebagai berbagi *library*.

4.1.5. Lapisan Atas

Karena proses dapat lebih besar daripada memori yang dialokasikan, kita gunakan lapisan atas. Idenya untuk menjaga agar di dalam memori berisi hanya instruksi dan data yang dibutuhkan dalam satuan waktu. Ketika instruksi lain dibutuhkan instruksi akan dimasukkan kedalam ruang yang ditempati sebelumnya oleh oleh instruksi yang tidak lagi dibutuhkan.

Sebagai contoh, sebuah two-pass assembler. selama pass1 dibangun sebuah tabel simbol, kemudian selama pass2, akan membuat kode bahasa mesin. kita dapat mempartisi sebuah assembler menjadi kode pass1, kode pass2, dan simbol tabel. dan rutin biasa digunakan untuk kedua pass1 dan pass2.

Untuk menempatkan semuanya sekaligus, kita akan membutuh-kan 200K memori. Jika hanya 150K yang tersedia, kita tidak dapat menjalankan proses. Bagaimana pun perhatikan bahwa pass1 dan pass2 tidak harus berada di memori pada saat yang sama. Kita mendefinisikan dua lapisan atas. Lapisan atas A untuk pass1, tabel simbol dan rutin, lapisan atas 2 untuk simbol tabel, rutin, dan pass2.

Kita menambahkan sebuah driver lapisan atas (10K) dan mulai dengan lapisan atas A di memori. Ketika selesai pass1, lompat ke driver, dan membaca lapisan atas B kedalam memori, meniban lapisan atas A, dan mengirim kontrol ke pass2. Lapisan atas A butuh hanya 120K, dan B membutuhkan 150K memori. Kita sekarang dapat menjalankan assembler dalam 150K memori. Penempatan akan lebih cepat, karena lebih sedikit data yang ditransfer sebelum eksekusi dimulai. Jalan program akan lebih lambat, karena ekstra I/O dari kode lapisan atas B melalui kode lapisan atas A.

Seperti dalam penempatan dinamis, lapisan atas tidak membutuhkan dukungan tertentu dari sistem operasi. Implementasi dapat dilakukan secara lengkap oleh user dengan berkas struktur yang sederhana, membasa dari berkas ke memori, dan lompat dari memori tersebut, dan mengeksekusi instruksi yang baru dibaca. Sistem operasi hanya memperhatikan jika ada lebih banyak I/O dari biasanya.

Di sisi lain programmer harus mendesain program dengan struktur lapisan atas yang layak. Tugas ini membutuhkan pengetahuan yang komplit tentang struktur dari program, kode dan struktur data.

Pemakaian dari lapisan atas, dibatasi oleh mikrokomputer, dan sistem lain yang mempunyai batasan jumlah memori fisik, dan kurangnya dukungan perangkat keras, untuk teknik yang lebih maju. Teknik otomatis menjalankan program besar dalam dalam jumlah memori fisik yang terbatas, lebih diutamakan.

4.2. Penukaran (*Swap*)

Sebuah proses membutuhkan memori untuk dieksekusi. Sebuah proses dapat ditukar sementara keluar memori ke backing store (disk), dan kemudian dibawa masuk lagi ke memori untuk dieksekusi. Sebagai contoh, asumsi multiprogramming, dengan penjadualan algoritma CPU Round-Robin. Ketika kuantum habis, manager memori akan mulai menukar keluar proses yang selesai, dan memasukkan ke memori proses yang bebas. Sementara penjadualan CPU akan mangalokasikan waktu untuk proses lain di memori. Ketika tiap proses menghabiskan waktu kuantumnya, proses akan ditukar dengan proses lain.

Idealnya memori manager, dapat menukar proses-proses cukup cepat, sehingga selalu ada proses dimemori, siap dieksekusi, ketika penjadual CPU ingin menjadual ulang CPU. Besar kuantum juga harus cukup besar, sehingga jumlah perhitungan yang dilakukan antar pertukaran masuk akal. Variasi dari kebijakan *swapping* ini, digunakan untuk algoritma penjadualan berdasarkan prioritas. Jika proses yang lebih tinggi tiba, dan minta dilayani, memori manager dapat menukar keluar proses dengan prioritas yang lebih rendah, sehingga dapat memasukkan dan mengeksekusi proses dengan prioritas yang lebih tinggi. Ketika proses dengan prioritas lebih tinggi selesai, proses dengan prioritas yang lebih rendah, dapat ditukar masuk kembali, dan melanjutkan. Macam-macam pertukaran ini kadang disebut roll out, dan roll in.

Normalnya, sebuah proses yang ditukar keluar, akan dimasukkan kembali ke tempat memori yang sama dengan yang digunakan sebelumnya. Batasan ini dibuat oleh method pengikat alamat. Jika

pengikatan dilakukan saat assemble atau load time, maka proses tidak bisa dipindahkan ke lokasi yang berbeda. Jika menggunakan pengikatan waktu eksekusi, maka akan mungkin menukar proses kedalam tempat memori yang berbeda. Karena alamat fisik dihitung selama proses eksekusi.

Pertukaran membutuhkan sebuah backing store. Backing store biasanya adalah sebuah disk yang cepat. Cukup besar untuk mengakomodasi semua kopi tampilan memori. Sistem memelihara *ready queue* terdiri dari semua proses yang mempunyai tampilan memori yang ada di backing store, atau di memori dan siap dijalankan. Ketika penjadual CPU memutuskan untuk mengeksekusi sebuah proses, dia akan memanggil dispatcher, yang mengecek dan melihat apakah proses berikutnya ada diantrian memori. Jika proses tidak ada, dan tidak ada ruang memori yang kosong, *dispatcher* menukar keluar sebuah proses dan memaasukan proses yang diinginkan. Kemudian memasukkan ulang register dengan normal, dan mentransfer pengendali ke proses yang diinginkan.

Konteks waktu pergantian pada sistem swapping, lumayan tinggi. Untuk efisiensi kegunaan CPU, kita ingin waktu eksekusi untuk tiap proses lebih lama dari waktu pertukaran. Karenanya digunakan CPU penjadualan roun-robin, dimana kuantumnya harus lebih besar dari waktu pertukaran.

Perhatikan bahwa bagian terbesar dari waktu pertukaran, adalah waktu pengiriman. Total waktu pengiriman langsung didapat dari jumlah pertukaran memori.

Proses dengan kebutuhan memori dinamis, akan membutuhkan *system call* (meminta dan melepaskan memori), untuk memberi tahu sistem operasi tentang perubahan kebutuhan memori.

Ada beberapa keterbatasan *swapping*. Jika kita ingin menukar sebuah proses kita harus yakin bahwa proses sepenuhnya diam. Konsentrasi lebih jauh, jika ada penundaan I/O. Sebuah proses mungkin menunggu I/O, ketika kita ingin menukar proses itu untuk mengosongkan memori. Jika I/O secara asinkronus, mengakses memori dari I/O buffer, maka proses tidak bisa ditukar. Misalkan I/O operation berada di antrian, karena *device* sedang sibuk. Maka bila kita menukar keluar proses P1 dan memasukkan P2, mungkin saja operasi I/O akan berusaha masuk ke memori yang sekarang milik P2.

Dua solusi utama masalah ini adalah

1. Jangan pernah menukar proses yang sedang menunggu I/O.
2. Untuk mengeksekusi operasi I/O hanya pada *buffer* sistem operasi.

Secara umum, ruang pertukaran dialokasikan sebagai potongan disk, terpisah dari sistem berkas, sehingga bisa digunakan secepat mungkin.

Belakangan pertukaran standar pertukaran digunakan dibeberapa sistem. Ini membutuhkan terlalu banyak waktu untuk menukar dari pada untuk mengeksekusi untuk solusi managemen memori yang masuk akal. Modifikasi *swapping* digunakan dibanyak versi di UNIX. Pertukaran awalnya tidak bisa, tapi akan mulai bila banyak proses yang jalan dan menggunakan batas jumlah memori.

4.3. Alokasi Memori Yang Berdampingan

Memori biasanya dibagi menjadi dua bagian, yakni:

1. Sistem Operasi (*Operating System*).
2. Proses Pengguna (*User Processes*).

Sistem Operasi dapat dialokasikan pada memori bagian bawah (*low memory*) mau pun memori bagian atas (*high memory*). Hal ini tergantung pada letak vektor interupsi (*interrupt vector*) pada memori tersebut. Jika vektor interupsi lebih sering berada pada memori bawah, maka sistem operasi juga biasanya diletakkan pada memori bawah.

Memori memerlukan suatu perlindungan yang disebut dengan istilah *memory protection* yakni perlindungan memori terhadap:

1. Sistem operasi dari proses pengguna;
2. Proses pengguna yang satu dari proses pengguna lainnya.

Perlindungan memori tersebut dapat diakomadasikan menggunakan suatu register pengalokasian kembali (*relocation register*) dengan suatu register batasan (*limit register*).

Register batasan berisi jarak dari alamat logik (*logical address*), sementara register pengalokasian kembali berisi nilai dari alamat fisik (*physical address*) yang terkecil. Dengan adanya register pengalokasian kembali dan register batasan ini, mengakibatkan suatu alamat logik harus lebih kecil dari register batas an dan memori akan memetakan (*mapping*) alamat logik secara dinamik dengan menambah nilai dalam register pengalokasian kembali.

```
register batasan              register pengalokasian kembali
                                    |            |
|Prosesor|-->(alamat  logik)-->|<|->(ya)-->|+|-->(alamat  fisik)--
>|MAR|
                    |(no)
         perangkap: kesalahan pengalamatan
```

Gambar 4-1. Alokasi Kembali.

Sebagaimana telah diketahui, bahwa pengatur jadual prosesor (*CPU scheduler*) bertugas mengatur dan menyusun jadual dalam proses eksekusi proses yang ada. Dalam tugasnya, pengatur jadual prosesor akan memilih suatu proses yang telah menunggu di antrian proses (*process queue*) untuk dieksekusi. Saat memilih satu proses dari proses yang ada di antrian tersebut, *dispatcher* akan mengambil register pengalokasian kembali dan register batasan dengan nilai yang benar sebagai bagian dari skalar alih konteks.

Oleh karena setiap alamat yang ditentukan oleh prosesor diperiksa berlawanan dengan register-register ini, kita dapat melindungi sistem operasi dari program pengguna lainnya dan data dari pemodifikasian oleh proses yang sedang berjalan.

Metode yang paling sederhana dalam mengalokasikan memori ke proses-proses adalah dengan cara membagi memori menjadi partisi tertentu. Secara garis besar, ada dua metode khusus yang digunakan dalam membagi-bagi lokasi memori:

A. Alokasi partisi tetap (*Fixed Partition Allocation*) yaitu metode membagi memori menjadi partisi yang telah berukuran tetap.

   Kriteria-kriteria utama dalam metode ini antara lain:

   - Alokasi memori: proses p membutuhkan k unit memori.
   - Kebijakan alokasi yaitu "sesuai yang terbaik": memilih partisi terkecil yang cukup besar (memiliki ukuran = k).
   - Fragmentasi dalam (*Internal fragmentation*) yaitu bagian dari partisi tidak digunakan.
   - Biasanya digunakan pada sistem operasi awal (*batch*).
   - Metode ini cukup baik karena dia dapat menentukan ruang proses; sementara ruang proses harus konstan. Jadi sangat sesuai dengan parti si berukuran tetap yang dihasilkan metode ini.
   - etiap partisi dapat berisi tepat satu proses sehingga derajat dari pemrograman banyak *multiprogramming* dibatasi oleh jumlah partisi yang ada.
   - etika suatu partisi bebas, satu proses dipilih dari masukan antrian dan dipindahkan ke partisi tersebut.
   - Setelah proses berakhir (selesai), partisi tersebut akan tersedia (*available*) untuk proses lain.

B. Alokasi partisi variabel (*Variable Partition Allocation*) yaitu metode dimana sistem operasi menyimpan suatu tabel yang menunjukkan partisi memori yang tersedia dan yang terisi dalam bentuk s.

• Alokasi memori: proses p membutuhkan k unit memori.

• Kebijakan alokasi:

1. Sesuai yang terbaik: memilih lubang (*hole*) terkecil yang cukup besar untuk keperluan proses sehingga menghasilkan sisa lubang terkecil.
2. Sesuai yang terburuk: memilih lubang terbesar sehingga menghasil kan sisa lubang.
3. Sesuai yang pertama: memilih lubang pertama yang cukup besar untuk keperluan proses
   - Fragmentasi luar (*External Fragmentation*) yakni proses mengambil ruang, sebagian digunakan, sebagian tidak digunakan.
   - Memori, yang tersedia untuk semua pengguna, dianggap sebagai suatu blok besar memori yang disebut dengan lubang. Pada suatu saat memori memiliki suatu daftar set lubang (*free list holes*).
   - Saat suatu proses memerlukan memori, maka kita mencari suatu lubang yang cukup besar untuk kebutuhan proses tersebut.
   - Jika ditemukan, kita mengalokasikan lubang tersebut ke proses tersebut sesuai dengan kebutuhan, dan sisanya disimpan untuk dapat digunakan proses lain.
4. Suatu proses yang telah dialokasikan memori akan dimasukkan ke memori dan selanjutnya dia akan bersaing dalam mendapatkan prosesor untuk pengeksekusiannya.
5. Jika suatu proses tersebut telah selesai, maka dia akan melepaskan kembali semua memori yang digunakan dan sistem operasi dapat mengalokasi-kannya lagi untuk proses lainnya yang sedang menunggu di antrian masukan.
6. Apabila memori sudah tidak mencukupi lagi untuk kebutuhan proses, sistem operasi akan menunggu sampai ada lubang yang cukup untuk dialokasikan ke suatu proses dalam antrian masukan.
7. ika suatu lubang terlalu besar, maka sistem operasi akan membagi lubang tersebut menjadi dua bagian, dimana satu bagian untuk dialokasikan ke proses tersebut dan satu lagi dikembalikan ke set lubang lainnya.

8. Setelah proses tersebut selesai dan melepaskan memori yang digunakannya, memori tersebut akan digabungkan lagi ke set lubang.

Fragmentasi luar mempunyai kriteria antara lain:

- Ruang memori yang kosong dibagi menjadi partisi kecil.
- Ada cukup ruang memori untuk memenuhi suatu permintaan, tetapi memori itu tidak lagi berhubungan antara satu bagian dengan bagian lain (*contiguous*) karena telah dibagi-bagi.
- Kasus terburuk (*Worst case*): akan ada satu blok ruang memori yang kosong yang terbuang antara setiap dua proses.
- Aturan 50 persen: dialokasikan N blok, maka akan ada 0.5N blok yang hilang akibat fragmentasi sehingga itu berarti 1/3 memori akan tidak berguna.

Perbandingan kompleksitas waktu dan ruang tiga kebijakan alokasi memori.

	Waktu	Ruang
Sesuai yang Terbaik	Buruk	Baik
Sesuai yang Terburuk	Buruk	Buruk
Sesuai yang Pertama	Baik	Baik

Gambar 4-2. Alokasi Kembali.

Sesuai yang pertama merupakan kebijakan alokasi memori paling baik secara praktis.

4.4. Pemberian Halaman

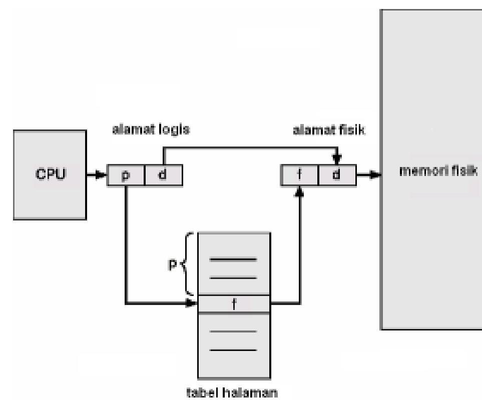
Solusi lain yang mungkin untuk permasalahan pemecahan luar adalah dengan membuat ruang alamat fisik dari sebuah proses menjadi tidak bersebelahan, jadi membolehkan sebuah proses untuk dialokasikan memori fisik bilamana nantinya tersedia. Satu cara mengimplementasikan solusi ini adalah melalui penggunaan dari skema pemberian halaman. Pemberian halaman mencegah masalah penting dari mengepaskan the ukuran bongkahan memori yang bervariasi ke dalam penyimpanan cadangan, yang mana diderita oleh kebanyakan dari skema managemen memori sebelumnya. Ketika beberapa pecahan kode dari data yang tersisa di memori utama perlu untuk di tukar keluar, harus ditemukan ruang di penyimpanan cadangan. Masalah pemecahan didiskusikan dengan kaitan bahwa meori utama juga lazim dengan penyimpanan cadangan, kecuali bahwa pengaksesanny lebih lambat, jadi kerapatan adalah tidak mungkin. Karena

keuntungannya pada metode-metode sebelumnya, pemberian halaman dalam berbagai bentuk biasanya digunakan pada banyak sistem operasi.

4.4.1. Metode Dasar

Memori fisik dipecah menjadi blok-blok berukuran tetap disebut sebagai *frame*. Memori logis juga dipecah menjadi blok-blok dengan ukuran yang sama disebut sebagai halaman. Ketika proses akan dieksekusi, halamannya akan diisi ke dalam *frames* memori mana saja yang tersedia dari penyimpanan cadangan. Penyimpanan cadangan dibagi-bagi menjadi blok-blok berukuran tetap yang sama besarnya dengan *frames* di memori.

Dukungan perangkat keras untuk pemberian halaman diilustrasikan pada gambar Gambar 4-3. Setiap alamat yang dihasilkan oleh CPU dibagi-bagi menjadi 2 bagian: sebuah nomor halaman (p) dan sebuah *offset* halaman (d). Nomor halaman digunakan sebagai indeks untuk tabel halaman. Tabel halaman mengandung basis alamat dari tiap-tiap halaman di memori fisik. Basis ini dikombinasikan dengan *offset* halaman untuk menentukan alamat memori fisik yang dikirim ke unit memori.



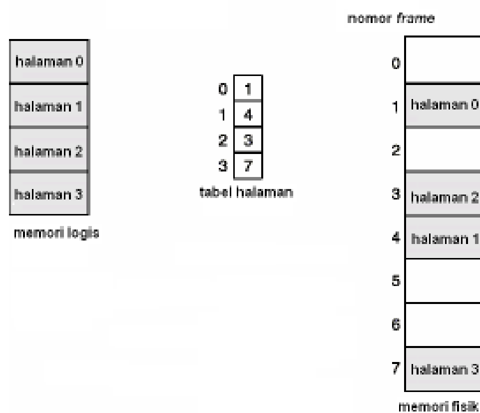
Gambar 4-3. Perangkat Keras Pemberian Halaman.

Ukuran halaman (seperti halnya ukuran *frame*) didefinisikan oleh perangkat keras. Khasnya ukuran dari sebuah halaman adalah pangkat 2 yang berkisar antara 512 byte dan 8192 byte per halamannya, tergantung dari arsitektur komputernya. Penentuan pangkat 2 sebagai ukuran halaman akan memudahkan penterjemahan dari memori logis ke nomor halaman dan *offset* halaman. Jika ukuran ruang dari memori logis adalah 2 pangkat m, dan ukuran sebuah halaman adalah 2 pangkat n unit pengalamatan (byte atau word), maka pangkat tinggi m-n bit dari

alamat logis manandakan *offset* dari halaman. Jadi, alamat logisnya adalah: dimana p merupakan index ke tabel halaman dan d adalah pemindahan dalam halaman.

Untuk konkritnya, walau kecil sekali, contoh, lihat memori Gambar 4-4. Menggunakan ukuran halaman 4 byte dan memori fisik 32 byte (8 halaman), kami menunjukan bagaimana pandangan pengguna terhadap memori dapat dipetakan kedalam memori fisik. Alamat logis 0 adalah halaman 0, *offset* 0.

Pemberian index menjadi tabel halaman, kita dapati bahwa halaman 0 berada pada frame 5. Jadi, alamat logis 0 memetakan ke alamat fisik 20 (=(5x4)+0). Alamat logis 3 (page 0, *offset* 3) memetakan ke alamat fisik 23 (=(5x4)+3). Alamat logis 4 adalah halaman 1, *offset*; menurut tabel halaman, halaman 1 dipetakan ke *frame* 6. Jadi, alamat logis 4 memetakan ke alamat fisik 24 (=(6x4)+0). Alamat logis 13 memetakan ke alamat fisik 9.



Gambar 4-4. Model pemberian halaman dari memori fisik dan logis.

Pembentukan pemberian halaman itu sendiri adalah suatu bentuk dari penampungan dinamis. Setiap alamat logis oleh perangkat keras untuk pemberian halaman dibatasi ke beberapa alamat fisik. Pembaca yang setia akan menyadari bahwa pemberian halaman sama halnya untuk menggunakan sebuah tabel dari basis register, satu untuk setiap *frame* di memori.

Ketika kita menggunakan skema pemberian halaman, kita tidak memiliki pemecah-mecahan luar: sembarang *frame* kosong dapat dialokasikan ke proses yang membutuhkan. Bagaimana pun juga kita mungkin mempunyai beberapa pemecahan di dalam. Mengingat bahwa *frame-frame* dialokasikan sebagai unit. Jika kebutuhan memori dari sebuah proses tidak menurun pada batas halaman, *frame* terakhir yang dialokasikan mungkin tidak sampai

penuh. Untuk contoh, jika halamannya 2048 byte, proses 72.766 byte akan membutuhkan 35 halaman tambah 1086 byte. Alokasinya menjadi 36 frame, menghasilkan fragmentasi internal dari 2048 - 1086 = 962 byte. Pada kasus terburuknya, proses akan membutuhkan n halaman tambah satu byte. Sehingga dialokasikan n + 1 *frame*, menghasilkan fragmentasi internal dari hampir semua *frame*. Jika ukuran proses tidak bergantung dari ukuran halaman, kita mengharapkan fragmentasi internal hingga rata-rata setengah halaman per prosesnya. Pertimbangan ini memberi kesan bahwa ukuran halaman yang kecil sangat diperlukan sekali. Bagaimana pun juga, ada sedikit pemborosan dilibatkan dalam masukan tabel halaman, dan pemborosan ini dikurangi dengan ukuran halaman meningkat. Juga disk I/O lebih efisien ketika jumlah data yang dipindahkan lebih besar. Umumnya, ukuran halaman bertambah seiring bertambahnya waktu seperti halnya proses, himpunan data, dan memori utama telah menjadi besar. Hari ini, halaman umumnya berukuran 2 atau 4 kilobyte.

Ketika proses tiba untuk dieksekusi, ukurannya yang diungkapkan di halaman itu diperiksa. Setiap pengguna membutuhkan satu *frame*. Jadi, jika proses membutuhkan n halaman, maka pasti ada n *frame* yang tersedia di memori. Jika ada n *frame* yang tersedia, maka mereka dialokasikan di proses ini. Halaman pertama dari proses diisi ke salah satu *frame* yang sudah teralokasi, dan nomor *frame*-nya diletakkan di tabel halaman untuk proses ini. Halaman berikutnya diisikan ke *frame* yang lain, dan nomor *frame*-nya diletakkan ke tabel halaman, dan begitu seterusnya (gambar Gambar 4-4).

Aspek penting dari pemberian halaman adalah pemisahan yang jelas antara pandangan pengguna tentang memori dan fisik memori sesungguhnya. Program pengguna melhiat memori sebagai satu ruang berdekatan yang tunggal, hanya mengandung satu program itu. Faktanya, program pengguna terpencar-pencar didalam memori fisik, yang juga menyimpan program lain. Perbedaan antara pandangan pengguna terhadap memori dan fisik memori sesungguhnya disetarakan oleh perangkat keras penterjemah alamat. Alamat logis diterjemahkan ke alamat fisik. Pemetaan ini tertutup bagi pengguna dan dikendalikan oleh sistem operasi. Perhatikan bahwa proses pengguna dalam definisi tidak dapat mengakses memori yang bukan haknya. Tidak ada pengalamatan memori di luar tabel halamannya, dan tabelnya hanya melingkupi halaman yang proses itu miliki.

Karena sistem operasi mengatur memori fisik, maka harus waspada dari rincian alokasi memori fisik: frame mana yang dialokasikan, frame mana yang tersedia, berapa banyak total frame yang ada, dan masih banyak lagi. Informasi ini umumnya

disimpan di struktur data yang disebut sebagai tabel frame. Tabel frame punya satu masukan untuk setiap fisik halaman frame, menandakan apakah yang terakhir teralokasi ataukah tidak, jika teralokasi maka kepada halaman mana dari proses mana.

Tambahan lagi sistem operasi harus waspada bahwa proses-proses pengguna beroperasi di ruang pengguna, dan semua logis alamat harus dipetakan untuk menghasilkan alamat fisik. Jika pengguna melakukan pemanggilan sistem (contohnya melakukan I/O) dan mendukung alamat sebagai parameter (contohnya penyangga), alamatnya harus dipetakan untuk menghasilkan alamat fisik yang benar. Sistem operasi mengatur salinan tabel halaman untuk tiap-tiap proses, seperti halnya ia mengatur salinan dari *counter* instruksi dan isi register. Salinan ini digunakan untuk menterjemahkan alamat fisik ke alamat logis kapan pun sistem operasi ingin memetakan alamat logis ke alamat fisik secara manual. Ia juga digunakan oleh *dispatcher* CPU untuk mendefinisikan tabel halaman perangkat keras ketika proses dialokasikan ke CPU. Oleh karena itu pemberian halaman meningkatkan waktu alih konteks.

4.4.2. Struktur Tabel Halaman

Setiap sistem operasi mempunyai metodenya sendiri untuk menyimpan tabel-tabel halaman. Sebagian besar mengalokasikan tabel halaman untuk setiap proses. Penunjuk ke tabel halaman disimpan dengan nilai register yang lain (seperti *counter* instruksi) di blok kontrol proses. Ketika pelaksana *dispatcher* mengatakan untuk memulai proses, maka harus disimpan kembali register-register pengguna dan mendefinisikan nilai tabel halaman perangkat keras yang benar dari tempat penyimpanan tabel halaman pengguna.

4.4.2.1. Dukungan Perangkat Keras

Implementasi perangkat keras dari tabel halaman dapat dilakukan dengan berbagai cara. Kasus sederhananya, tabel halaman diimplementasikan sebagai sebuah himpunan dari register resmi. Register ini harus yang bekecepatan tinggi agar penerjemahan alamat pemberian halaman efisien Setiap pengaksesan ke memori harus melalui peta pemberian halaman, jadi ke-efisienan adalah pertimbangan utama. Pelaksana (*dispatcher*) CPU mengisi kembali register-register ini, seperti halnya ia mengisi kembali register yang lain. Instruksi untuk mengisi atau mengubah register tabel halaman adalah, tentu saja diberi hak istimewa, sehingga hanya sistem operasi yang dapat mengubah peta memori. DEC PDP-11 adalah contoh arsitektur yang demikian. Alamatnya terdiri dari

16-bit, dan ukuran halamannya 8K. Jadi tabel halaman terdiri dari 8 masukan yang disimpan di register-register cepat.

4.4.2.2. Pemeliharaan

Proteksi memori dari suatu lingkungan berhalaman diselesaikan dengan bit-bit proteksi yang diasosiasikan dengan tiap-tiap *frame*. Normalnya, bit-bit ini disimpan di tabel halaman. Satu bit dapat menentukan halaman yang akan dibaca tulis atau baca saja. Setiap referensi ke memori menggunakan tabel halaman untuk menemukan nomor *frame* yang benar. Pada saat yang sama alamat fisik diakses, bit-bit proteksi dapat dicek untuk menguji tidak ada penulisan yang sedang dilakukan terhadap halaman yang boleh dibaca saja. Suatu usaha untuk menulis ke halaman yang boleh dibaca saja akan menyebabkan perangkat keras menangkapnya ke sistem operasi.

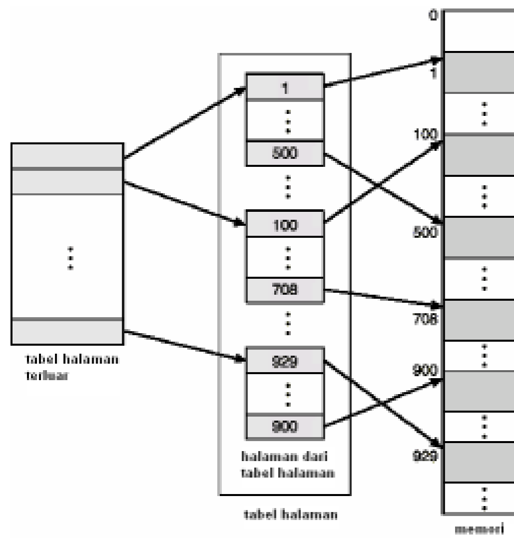
4.4.3. Pemberian Halaman Secara *Multilevel*

Banyak sistem komputer moderen mendukung ruang alamat logis yang sangat luas (2 pangkat 32 sampai 2 pangkat 64). Pada lingkungan seperti itu tabel halamanya sendiri menjadi sangat-sangat besat sekali. Untuk contoh, misalkan suatu sistem dengan ruang alamat logis 32-bit. Jika ukuran halaman di sistem seperti itu adalah 4K byte (2 pangkat 12), maka tabel halaman mungkin berisi sampai 1 juta masukan ((2^32)/(2^12)). Karena masing-masing masukan terdiri atas 4 byte, tiap-tiap proses mungkin perlu ruang alamat fisik sampai 4 megabyte hanya untuk tabel halamannya saja. Jelasnya, kita tidak akan mau

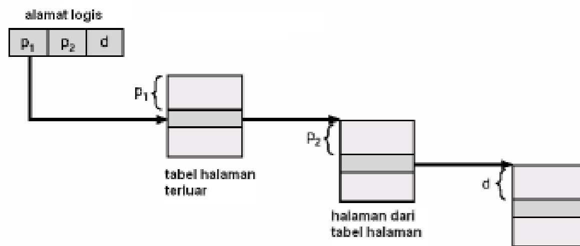
mengalokasi tabel halaman secara berdekatan di dalam memori. Satu solusi sederhananya adalah dengan membagi tabel halaman menjadi potongan-potongan yang lebih kecil lagi. Ada beberapa cara yang berbeda untuk menyelesaikan ini.

nomor halaman		*offset* halaman
p_i	p_2	d
10	10	12

Gambar 4-5. Alamat logis.



Gambar 4-6. Skema Tabel Halaman Dua Tingkat.

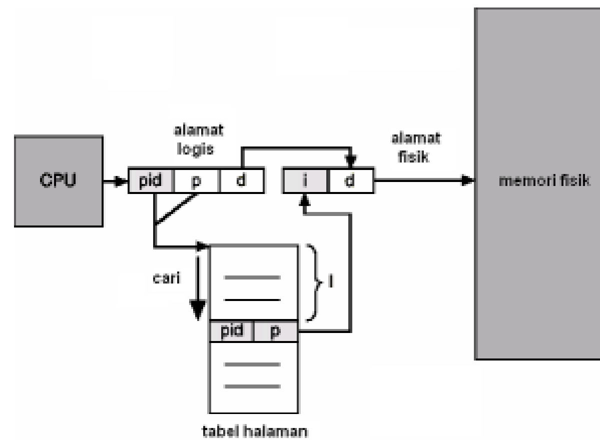


Gambar 4-7. Penterjemahan alamat untuk arsitektur pemberian halaman dua tingkat 32-bit logis.

4.4.3.1. Tabel Halaman yang Dibalik

Biasanya, setiap proses mempunyai tabel halaman yang diasosiasikan dengannya. Tabel halaman hanya punya satu masukan untuk setiap halaman proses tersebut sedang gunakan (atau satu slot untuk setiap alamat maya, tanpa meperhatikan validitas terakhir). Semenjak halaman referensi proses melalui alamat maya halaman, maka representasi tabel ini adalah alami. Sistem operasi harus menterjemahkan referensi ini ke alamat memori fisik. Semenjak tabel diurutkan berdasarkan alamat maya, sistem operasi dapat menghitung dimana pada tabel yang diasosiasikan dengan masukan alamat fisik, dan untuk menggunakan nilai tersebut secara langsung. Satu kekurangan

dari skema ini adalah masing-masing halaman mungkin mengandung jutaan masukan. Tabel ini mungkin memakan memori fisik dalam jumlah yang besar, yang mana dibutukan untuk tetap menjaga bagaimana memori fisik lain sedang digunakan.

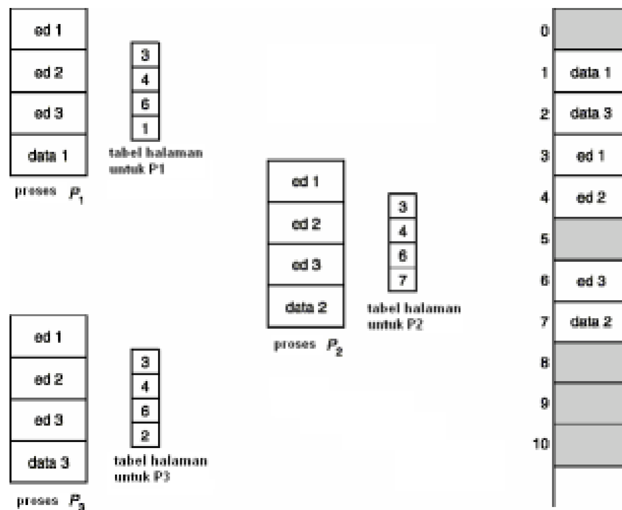


Gambar 4-8. Tabel halaman yang dibalik.

4.4.3.2. Berbagi Halaman

Keuntungan lain dari pemberian halaman adalah kemungkinan-nya untuk berbagi kode yang sama. Pertimbangan ini terutama sekali penting pada lingkungan yang berbagi waktu. Pertimbangkan sebuah sistem yang mendukung 40 pengguna, yang masing-masing menjalankan aplikasi pengedit teks. Jika editor teks tadi terdiri atas 150K kode dan 50K ruang data, kita akan membutuhkan 8000K untuk mendukung 40 pengguna. Jika kodenya dimasukan ulang, bagaimana pun juga dapat dibagi-bagi, seperti pada gambar Gambar 4-9. Disini kita lihat bahwa tiga halaman editor (masing-masing berukuran 50K; halaman ukuran besar digunakan untuk menyederhanakan gambar) sedang dibagi-bagi diantara tiga

proses. Masing-masing proses mempunyai halaman datanya sendiri.



Gambar 4-9. Berbagi kode pada lingkungan berhalaman.

Kode pemasukan kembali (juga disebut kode murni) adalah kode yang bukan *self-modifying*. Jika kodenya dimasukan kembali, maka ia tidak akan berubah selama eksekusi. Jadi, dua atau lebih proses dapat mengeksekusi kode yang sama pada saat bersamaan. Tiap-tiap proses mempunyai register salinannya sendiri dan penyimpan data untuk menahan data bagi proses bereksekusi. Data untuk dua proses yang berbeda akan bervariasi pada tiap-tiap proses.

Hanya satu salinan editor yang dibutuhkan untuk menyimpan di memori fisik. Setiap tabel halaman pengguna memetakan ke salinan fisik yang sama dari editor, tapi halaman-halaman data dipetakan ke *frame* yang berbeda. Jadi, untuk mendukung 40 pengguna, kita hanya membutuhkan satu salinannya editor (150K), ditambah 40 salinan 50K dari ruang data per pengguna. Total ruang yang dibutuhkan sekarang 2150K, daripada 8000K, penghematan yang signifikan.

Program-program lain pun juga dapat dibagi-bagi: *compiler*, *system window* database system, dan masih banyak lagi. Agar dapat dibagi-bagi, kodenya harus dimasukan kembali.

System yang menggunakan tabel halaman yang dibalik mempunyai kesulitan dalam mengimplementasikan berbagi memori. Berbagi memori biasanya diimplementasikan sebagai dua alamat maya yang dipetakan ke satu alamat fisik. Metode standar ini tidak dapat digunakan, bagaimana pun juga selama di situ hanya ada satu masukan halaman maya untuk setiap halaman fisik, jadi satu alamat fisik tidak dapat mempunyai dua atau lebih alamat maya yang dibagi-bagi.

4.5. Segmentasi

Salah satu aspek penting dari managemen memori yang tidak dapat dihindari dari pemberian halaman adalah pemisahan cara pandang pengguna dengan tentang bagaimana memori dipetakan dengan keadaan yang sebenarnya. Pada kenyataannya pemetaan tersebut memperbolehkan pemisahan antara memori logis dan memori fisik.

4.5.1. Metode Dasar

Bagaimanakah cara pandang pengguna tentang bagaimana memori dipetakan? Apakah pengguna menganggap bahwa memori dianggap sebagai sebuah kumpulan dari byte-byte, yang mana sebagian berisi instruksi dan sebagian lagi merupakan data, atau apakah ada cara pandang lain yang lebih layak digunakan? Ternyata programmer dari sistem tidak menganggap bahwa memori adalah sekumpulan *byte-byte* yang linear. Akan tetapi, mereka lebih senang dengan menganggap bahwa memori adalah sebagai kumpulan dari segmen-segmen yang berukuran beragam tanpa adanya pengurutan penempatan dalam memori fisik.

Ketika kita menulis suatu program, kita akan menganggapnya sebagai sebuah program dengan sekumpulan dari subrutin, prosedur, fungsi, atau variabel. mungkin juga terdapat berbagai macam struktur data seperti: tabel, *array*, *stack*, variabel, dsb. Tiap-tiap modul atau elemen-elemen dari data ini dapat di-referensikan dengan suatu nama, tanpa perlu mengetahui dimana alamat sebenarnya elemen-elemen tersebut disimpan di memori. dan kita juga tidak perlu mengetahui apakah terdapat urutan penempatan dari program yang kita buat. Pada kenyataannya, elemen-elemen yang terdapat pada sebuah segmen dapat ditentukan lokasinya dengan menambahkan *offset* dari awal alamat segmen tersebut.

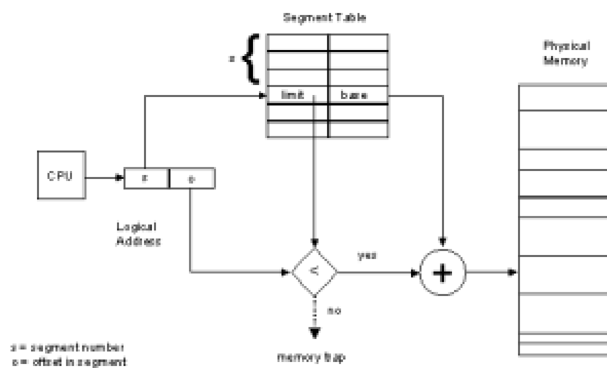
Segmentasi adalah sebuah bagian dari managemen memori yang mengatur pengalamatan dari memori yang terdiri dari segmen-segmen. *logical address space* adalah kumpulan dari segmen-segmen yang mana tiap-tiap segmen mempunyai nama dan panjang. alamat tersebut menunjukkan alamat dari segmen tersebut dan *offset*-nya didalam segmen-segmen tersebut. pengguna kemudian menentukan pengalamatan dari setiap segmen menjadi dua bentuk, nama segmen dan offset dari segmen tersebut (Hal ini berbeda dengan pemberian halaman, dimana pengguna hanya menentukan satu buah alamat, dimana pembagian alamat menjadi dua dilakukan oleh perangkat keras, semua ini tidak dapat dilihat oleh user).

Untuk kemudahan pengimplementasian, segmen-segmen diberi nomor dan direferensikan dengan menggunakan penomoran tersebut, daripada dengan menggunakan nama. maka, *logical address space* terdiri dari dua *tuple* yaitu: (nomor-segmen, offset) Pada umumnya, program dari pengguna akan dikompilasi, dan kompilator tersebut akan membuat segmen-segmen tersebut secara otomatis. Jika mengambil contoh kompilator dari Pascal, maka kemungkinan kompilator tersebut akan membuat beberapa segmen yang terpisah untuk

1. Variabel Global;
2. Prosedur dari pemanggilan stack, untuk menyimpan parameter dan pengembalian alamat;
3. Porsi dari kode untuk setiap prosedur atau fungsi; dan
4. Variabel lokal dari setiap prosedur dan fungsi.

4.5.2. Perangkat Keras

Walau pun pengguna sekarang dapat mengacu ke suatu objek yang berada di dalam program dengan menggunakan pengalamatan secara dua dimensi, akan tetapi, pada kenyataannya tetap saja pada memori fisik akan dipetakan ke dalam pengalamatan satu dimensi yang terdiri dari urutan dari *byte-byte*. Maka, kita harus mendefinisikan suatu implementasi untuk memetakan pengalamatan dua dimensi yang dilakukan oleh pengguna ke dalam pengalamatan satu dimensi yang terdapat di memori fisik. Pemetaan ini dapat di lakukan dengan menggunakan tabel segmen. Setiap anggota dari tabel segmen mempunyai basis dan limit yang akan menentukan letak dari segmen tersebut di dalam memori.



Gambar 4-10. Alamat Logis

Kegunaan tabel segmen dapat dilihat pada gambar Gambar 4-10 alamat logis terdiri dari dua bagian: bagian segmen, s, dan bagian offsetnya, d. Nomor dari segmen tersebut akan digunakan sebagai index di dalam tabel segmen. Offset dari d di alamat logis sebaiknya tidak melebihi limit dari alamat segmen, jika ini terjadi, maka sistem operasi sebaiknya dapat mengatasi hal ini, dengan melakukan *trap*.

4.5.3. Pemeliharaan dan Pembagian

Dengan dilakukannya pengelompokan antara segmen-segmen yang sama, maka pemeliharaan dari segmen tersebut dapat menjadi lebih mudah, walau pun didalam segmen tersebut sebagian berisi instruksi dan sebagian lagi berisi data. Dalam arsitektur modern, instruksi-instruksi yang digunakan tidak dapat diubah tanpa campur tangan pengguna, oleh karena itu, segmen yang berisi instruksi dapat diberi label *read only* atau hanya dapat dijalankan saja. Perangkat keras yang bertugas untuk melakukan pemetaan ke memori fisik akan melakukan pemeriksaan terhadap bit proteksi yang terdapat pada segmen, sehingga pengaksesan memori secara ilegal dapat dihindari, seperti suatu usaha untuk menulis ke area yang berstatus tidak boleh dimodifikasi.

Keuntungan lain dari segmentasi adalah menyangkut masalah pembagian penggunaan kode atau data. Setiap proses mempunyai tabel segmennya sendiri, dimana ini akan digunakan oleh *dispatcher* untuk menentukan tabel segmen dari perangkat keras yang mana akan digunakan ketika proses yang bersangkutan di eksekusi oleh CPU. Segmen akan berbagi ketika anggota dari elemen tabel segmen yang berasal dari dua proses yang berbeda menunjuk ke lokasi fisik yang sama. Pembagian tersebut terjadi pada level segmen, maka, informasi apa pun dapat dibagi jika didefinisikan pada level segmen. Bahkan beberapa segmen pun dapat berbagi, sehingga sebuah program yang terdiri dari beberapa segmen pun dapat saling berbagi pakai.

4.5.4. Fragmentasi

Penjadwalan jangka-panjang harus mencari dan mengalokasikan memori untuk semua segmen dari program pengguna. Situasi ini mirip dengan pemberian halaman kecuali bahwa segmen-segmen ini mempunyai panjang yang variabel; sedangkan pada halaman, semua mempunyai ukuran yang sama. maka, masalah yang dihadapi adalah pengalamatan memori secara dinamis, hal ini biasanya dapat diselesaikan dengan menggunakan algoritma *best-fit* atau algoritma *first-fit*.

Segmentasi dapat menyebabkan terjadi fragmentasi eksternal, ini terjadi ketika semua blok memori yang dapat dapat dialokasikan terlalu sedikit untuk mengakomodasi sebuah segmen. Dalam kasus ini, proses hanya harus menunggu sampai terdapat cukup tempat untuk menyimpan segmen tersebut di memori, atau, melakukan suatu pemampatan dapat digunakan untuk membuat ruang kosong dalam memori menjadi lebih besar. Karena segmentasi pada dasarnya adalah algoritma penempatan secara dinamis, maka kita dapat melakukan pemampatan memori kapan saja kita mau. Jika *CPU Scheduler* harus menunggu untuk satu proses, karena masalah pengalokasian memori, ini mungkin akan dilewati untuk mencari proses yang berprioritas lebih kecil untuk dieksekusi lebih dulu untuk membebaskan ruang kosong dalam memori.

Seberapa seriuskah masalah fragmentasi eksternal dalam segmentasi? Jawaban dari pertanyaan ini tergantung kepada besarnya rata-rata segmen yang tersimpan didalam memori. Jika ukuran rata-rata dari segmen menggunakan sedikit tempat di memori, maka fragmentasi eksternal yang dilakukan juga akan sedikit terjadi.

4.6. Segmentasi Dengan Pemberian Halaman

4.6.1. Pengertian

Metode segmentasi dan *paging* yang telah dijelaskan pada sub bab sebelumnya masing-masing memiliki keuntungan dan kerugian. Selain kedua metode itu ada metode pengaturan memori lain yang berusaha menggabungkan metode segmentasi dan paging. Metode ini disebut dengan segmentation with paging.

Dengan metode ini jika ukuran segmen melebihi ukuran memori utama maka segmen tersebut dibagi-bagi jadi ukuran-ukuran halaman yang sama ==> *paging*.

4.6.2. Kelebihan Segmentasi dengan Pemberian Halaman

Sesuai dengan definisinya yang merupakan gabungan dari segmentasi dan *paging*, maka metode ini memiliki keunggulan yang dimiliki baik oleh metode segmentasi mau pun yang dimiliki oleh *paging*.

Tetapi selain itu segmentasi dengan pemberian halaman ini juga memiliki beberapa kelebihan yang tidakdimiliki oleh kedua metode tersebut. Kelebihan-kelebihan segmentasi dengan pemberian halaman antara lain:

- Dapat dibagi.
- Proteksi.
- Tidak ada fragmentasi luar.
- Alokasi yang cepat.
- Banyak variasinya.
- Biaya kinerja yang kecil.

4.6.3. Perbedaan Segmentasi dan *Paging*

Ada beberapa perbedaan antara Segmentasi dan *Paging* diantaranya adalah:

1. Segmentasi melibatkan programer (programer perlu tahu teknik yang digunakan), sedangkan dengan *paging*, programer tidak perlu tahu teknik yang digunakan.
2. Pada segmentasi kompilasi dilakukan secara terpisah sedangkan pada *paging*, kompilasinya tidak terpisah.
3. Pada segmentasi proteksinya terpisah sedangkan pada *paging* proteksinya tidak terpisah.
4. Pada segmentasi ada *shared code* sedangkan pada *paging* tidak ada *shared code*.
5. Pada segmentasi terdapat banyak ruang alamat linier sedangkan pada *paging* hanya terdapat satu ruang alamat linier.
6. Pada segmentasi prosedur dan data dapat dibedakan dan diproteksi terpisah sedangkan pada *paging* prosedur dan data tidak dapat dibedakan dan diproteksi terpisah.
7. Pada segmentasi pengubahan ukuran tabel dapat dilakukan dengan mudah sedangkan pada *Paging* pengubahan ukuran tabel tidak dapat dilakukan dengan mudah.
8. Segmentasi digunakan untuk mengizinkan program dan data dapat dipecahkan jadi ruang alamat mandiri dan juga untuk mendukung sharing dan proteksi sedangkan *paging* digunakan untuk mendapatkan ruang alamat linier yang besar tanpa perlu membeli memori fisik lebih.

4.6.4. Pengimplementasian Segmentasi dengan Pemberian Halaman Intel i386

Salah satu contoh prosesor yang menggunakan metode segmentasi dengan pemberian halaman ini diantaranya adalah Intel i386. Jumlah maksimum segmen tiap proses adalah 16 K dan besar tiap segmen adalah 4 GB. Dan ukuran halamannya adalah 4 KB.

4.6.4.1. *Logical Address*

Ruang *logical address* dari suatu proses terbagi menjadi dua partisi yaitu:

1. Partisi I
   - Terdiri dari segmen berjumlah 8 K yang sifatnya pribadi atau rahasia terhadap proses tersebut.
   - Informasi tentang partisi ini disimpan didalam *Local Descriptor Table*.
2. Partisi II
   - Terdiri dari 8 K segmen yang digunakan bersama diantara *proses*-proses tersebut.
   - *Informasi* tentang partisi ini disimpan didalam *Global Descriptor Table*.

Tiap masukan atau entri pada *Local Descriptor Table* dan *Global Descriptor Table* terdiri dari 8 bit dengan informasi yang detil tentang segmen khusus termasuk lokasi dasar dan panjang segmen tersebut.

Logical address merupakan sepasang:

1. Selektor

Terdiri dari angka 16 bit:

Dimana s = jumlah segmen (13 bit)

g = mengindikasikan apakah segmen ada di *Global Descriptor Table* atau *Local Descriptor Table* (1 bit)

p= proteksi(2 bit)

s	g	p
13	1	2

2. *Offset*

Terdiri dari angka 32 bit yang menspesifikasikan lokasi suatu kata atau bita di dalam segmen tersebut.

Mesin memiliki 6 register segmen yang membiarkan 6 segmen dialamatkan pada suatu waktu oleh sebuah proses. Mesin meiliki register program mikro 8 bita untuk menampung *descriptor* yang bersesuaian baik dari *Global Descriptor Table* atau *Local Descriptor Table*. *Cache* ini membiarkan 386 menghindari membaca *descriptor* dari memori untuk tiap perujukan memori.

4.6.4.2. Alamat Fisik

Alamat fisik 386 panjangnya adalah 32 bit. Mula-mula register segmen menunjuk ke masukan atau entri di *Global Descriptor Table* atau *Local Descriptor Table*. Kemudian informasi dasar dan limit tentang segmen tersebut digunakan untuk mengeneralisasikan alamat linier. Limit itu digunakan untuk mengecek keabsahan alamat. Jika alamat tidak sah maka akan terjadi memori fault yang menyebabkan terjadinya trap pada sistem operasi. Sedangkan apabila alamat itu sah maka nilai dari offset ditambahkan kenilai dasar yang menghasilkan alamat linier 32 bit. Alamat inilah yang kemudian diterjemahkan ke alamat fisik.

Seperti dikemukakan sebelumnya tiap segmen dialamatkan dan tiap halaman 4 KB. Sebuah tabel halaman mungkin terdiri sampai satu juta masukan atau entri. Karena tiap entri terdiri dari 4 byte, tiap proses mungkin membutuhkan sampai 4 MB ruang alamat fisik untuk halaman tabel sendiri. Sudah jelas kalau kita tidak menginginkan untuk mengalokasi tabel halaman bersebelahan di memori utama. Solusi yang dipakai 386 adalah dengan menggunakan skema paging dua tingkat (two-level paging scheme). Alamat linier dibagi menjadi nomer halaman yang terdiri dari 20 bit dan *offset* halaman terdiri dari 12 bit. Karena kita *page* tabel halaman dibagi jadi 10 bit penunjuk halaman direktori dan 10 bit penunjuk tabel halaman sehingga *logical address* menjadi:

nomor halaman

offset halaman

p1	p2	d
10	10	12

4.7. Memori Virtual

Selama bertahun-tahun, pelaksanaan berbagai strategi managemen memori yang ada menuntut keseluruhan bagian proses berada di memori sebelum proses dapat mulai dieksekusi. Dengan kata lain, semua bagian proses harus memiliki alokasi sendiri pada memori fisiknya.

Pada nyatanya tidak semua bagian dari program tersebut akan diproses, misalnya:

1. Terdapat pernyataan-pernyataan atau pilihan yang hanya akan dieksekusi jika kondisi tertentu dipenuhi. Apabila kondisi tersebut tidak dipenuhi, maka pilihan tersebut tak akan pernah dieksekusi/ diproses. Contoh dari pilihan itu adalah: pesan-pesan error yang hanya akan muncul bila terjadi kesalahan dalam eksekusi program.
2. Terdapat fungsi-fungsi yang jarang digunakan, bahkan sampai lebih dari 100x pemakaian.
3. Terdapat pealokasian memori lebih besar dari yang sebenarnya dibutuhkan. Contoh pada: *array*, *list*, dan tabel.

Hal-hal di atas telah menurunkan optimalitasi utilitas dari ruang memori fisik. Pada memori berkapasitas besar, hal ini mungkin tidak menjadi masalah. Akan tetapi, bagaimana jika memori yang disediakan terbatas?

Salah satu cara untuk mengatasinya adalah dengan *overlay* dan *dynamic loading* . Namun hal ini menimbulkan masalah baru karena implementasinya yang rumit dan penulisan program yang akan memakan tempat di memori. Tujuan semula untuk menghemat memori bisa jadi malah tidak tercapai apabila program untuk *overlay* dan *dynamic loading* . malah lebih besar daripada program yang sebenarnya ingin dieksekusi.

Maka sebagai solusi untuk masalah-masalah ini digunakanlah konsep memori virtual.

4.7.1. Pengertian

Memori virtual merupakan suatu teknik yang memisahkan antara memori logis dan memori fisiknya. Teknik ini mengizinkan program untuk dieksekusi tanpa seluruh bagian program perlu ikut masuk ke dalam memori.

Berbeda dengan keterbatasan yang dimiliki oleh memori fisik, memori virtual dapat menampung program dalam skala besar, melebihi daya tampung dari memori utama yang tersedia. Prinsip dari memori virtual yang patut diingat adalah bahwa: "Kecepatan maksimum eksekusi proses di memori virtual dapat sama, tetapi tidak pernah melampaui kecepatan eksekusi proses yang sama di sistem tanpa menggunakan memori virtual."

Konsep memori virtual pertama kali dikemukakan Fotheringham pada tahun 1961 pada sistem komputer Atlas di Universitas Manchester, Inggris (Hariyanto, Bambang : 2001).

4.7.2. Keuntungan

Sebagaimana dikatakan di atas bahwa hanya sebagian dari program yang diletakkan di memori. Hal ini berakibat pada:

- Berkurangnya I/O yang dibutuhkan (lalu lintas I/O menjadi rendah). Misal, untuk program butuh membaca dari disk dan memasukkan dalam memory setiap kali diakses.
- Berkurangnya memori yang dibutuhkan (*space* menjadi lebih leluasa). Contoh, untuk program 10 MB tidak seluruh bagian dimasukkan dalam memori. Pesan-pesan *error* hanya dimasukkan jika terjadi *error*.
- Meningkatnya respon, sebagai konsekuensi dari menurunnya beban I/O dan memori.
- Bertambahnya jumlah *user* yang dapat dilayani. Ruang memori yang masih tersedia luas memungkinkan komputer untuk menerima lebih banyak permintaan dari *user*.

4.7.3. Implementasi

Gagasan dari memori virtual adalah ukuran gabungan program, data dan *stack* melampaui jumlah memori fisik yang tersedia. Sistem operasi menyimpan bagian-bagian proses yang sedang digunakan di memori utama (*main memory*) dan sisanya ditaruh di disk. Begitu bagian di disk diperlukan, maka bagian di memori yang tidak diperlukan akan disingkirkan (*swap-out*) dan diganti (*swap-in*) oleh bagian disk yang diperlukan itu.

Memori virtual diimplementasikan dalam sistem *multiprogramming*. Misalnya: 10 program dengan ukuran 2 Mb dapat berjalan di memori berkapasitas 4 Mb. Tiap program dialokasikan 256 KByte dan bagian-bagian proses di-*swap* masuk dan keluar memori begitu diperlukan. Dengan demikian, sistem *multiprogramming* menjadi lebih efisien.

Memori virtual dapat dilakukan melalui dua cara:

1. Permintaan pemberian halaman (*demand paging*).
2. Permintaan segmentasi (*demand segmentation*). Contoh: IBM OS/2. Algoritma dari permintaan segmentasi lebih kompleks, karenanya jarang diimplementasikan.

4.8. Permintaan Pemberian Halaman (*Demand Paging*)

Merupakan implementasi yang paling umum dari memori virtual.

Prinsip permintaan pemberian halaman (*demand paging*) hampir sama dengan sistem penomoran (*paging*) dengan menggunakan

swapping. Perbedaannya adalah *page* pada permintaan pemberian halaman tidak akan pernah di-*swap* ke memori sampai ia benar-benar diperlukan. Untuk itu diperlukan adanya pengecekan dengan bantuan perangkat keras mengenai lokasi dari *page* saat ia dibutuhkan.

4.8.1. Permasalahan pada *Page Fault*

Ada tiga kemungkinan kasus yang dapat terjadi pada saat dilakukan pengecekan pada *page* yang dibutuhkan, yaitu:

1. *Page* ada dan sudah berada di memori.
2. *Page* ada tetapi belum ditaruh di memori (harus menunggu sampai dimasukkan).
3. *Page* tidak ada, baik di memori mau pun di disk (invalid reference --> abort).

Saat terjadi kasus kedua dan ketiga, maka proses dinyatakan mengalami *page fault*.

4.8.2. Skema Bit Valid - Tidak Valid

Dengan meminjam konsep yang sudah pernah dijelaskan dalam Bab 9, maka dapat ditentukan page mana yang ada di dalam memori dan mana yang tidak ada di dalam memori.

Konsep itu adalah skema bit valid - tidak valid, di mana di sini pengertian "valid" berarti bahwa *page* legal dan berada dalam memori (kasus 1), sedangkan "tidak valid" berarti *page* tidak ada (kasus 3) atau *page* ada tapi tidak ditemui di memori (kasus 2).

Pengesetan bit:

Bit 1 -->

page berada di memori

Bit 0 -->

page tidak berada di memori.

(Dengan inisialisasi: semua bit

di-*set* 0).

Apabila ternyata hasil dari translasi, bit *page* = 0, berarti *page fault* terjadi.

4.8.2.1. Penanganan *Page Fault*

Prosedur penanganan *page fault* sebagaimana tertulis di buku *Operating System Concept 5th Ed* halaman 294 adalah sebagai berikut:

1. Cek tabel internal yang dilengkapi dengan PCB untuk menentukan valid atau tidaknya bit.
2. Apabila tidak valid, program akan di-*terminate* (interupsi oleh *illegal address trap*).
3. Memilih *frame* kosong (*free-frame*), misal dari *free-frame list*. Jika tidak ditemui ada *frame* yang kosong, maka dilakukan *swap-out* dari memori. *Frame* mana yang harus di-*swap-out* akan ditentukan oleh algoritma (lihat sub bab *Page Replacement*).
4. Menjadualkan operasi disk untuk membaca *page* yang diinginkan ke *frame* yang baru dialokasikan.
5. Ketika pembacaan komplit, tabel internal akan dimodifikasi dan *page* diidentifikasi ada di memori.
6. Mengulang instruksi yang tadi telah sempat diinterupsi. Jika tadi *page fault* terjadi saat instruksi di-*fetch*, maka akan dilakukan *fecthing* lagi. Jika terjadi saat operan sedang di-*fetch*, maka harus dilakukan *fetch* ulang, *decode*, dan *fetch* operan lagi.

4.8.2.2. Permasalahan Lain yang berhubungan dengan *Demand Paging*

Sebagaimana dilihat di atas, bahwa ternyata penanganan *page fault* menimbulkan masalah-masalah baru pada proses *restart instruction* yang berhubungan dengan arsitektur komputer.

Masalah yang terjadi, antara lain mencakup:

1. Bagaimana mengulang instruksi yang memiliki beberapa lokasi yang berbeda?
2. Bagaimana pengalamatan dengan menggunakan *special-addresing mode*, termasuk *autoincrement* dan *autodecrement mode*?
3. Bagaimana jika instruksi yang dieksekusi panjang (contoh: *block move*)?

Masalah pertama dapat diatasi dengan dua cara yang berbeda.

- komputasi *microcode* dan berusaha untuk mengakses kedua ujung dari blok, agar tidak ada modifikasi*page* yang sempat terjadi.
- memanfaatkan register sementara (*temporary register*) untuk menyimpan nilai yang sempat tertimpa/ termodifikasi oleh nilai lain.

Masalah kedua diatasi dengan menciptakan suatu *special-status register* baru yang berfungsi menyimpan nomor register dan banyak perubahan yang terjadi sepanjang eksekusi instruksi.

Sedangkan masalah ketiga diatasi dengan mengeset bit FPD (*first phase done*) sehingga *restart instruction* tidak akan dimulai dari awal program, melainkan dari tempat program terakhir dieksekusi.

4.8.2.3. Persyaratan Perangkat Keras

Pemberian nomor halaman melibatkan dukungan perangkat keras, sehingga ada persyaratan perangkat keras yang harus dipenuhi. Perangkat-perangkat keras tersebut sama dengan yang digunakan untuk *paging* dan *swapping*, yaitu:

- *Page-table*, menandai bit valid-tidak valid.
- *Secondary memory*, tempat menyimpan *page* yang tidak ada di memori utama.

Lebih lanjut, sebagai konsekuensi dari persyaratan ini, akan diperlukan pula perangkat lunak yang dapat mendukung terciptanya pemberian nomor halaman.

4.9. Pemindahan Halaman

Pada dasarnya, kesalahan halaman (*page fault*) sudah tidak lagi menjadi masalah yang terlalu dianggap serius. Hal ini disebabkan karena masing-masing halaman pasti akan mengalami paling tidak satu kali kesalahan dalam pemberian halaman, yakni ketika halaman ini ditunjuk untuk pertama kalinya.

Representasi seperti ini sebenarnya tidaklah terlalu akurat. Berdasarkan pertimbangan tersebut, sebenarnya proses-proses yang memiliki 10 halaman hanya akan menggunakan setengah dari jumlah seluruh halaman yang dimilikinya. Kemudian *demand paging* akan menyimpan I/O yang dibutuhkan untuk mengisi 5 halaman yang belum pernah digunakan. Kita juga dapat meningkatkan derajat *multiprogramming* dengan menjalankan banyak proses sebanyak 2 kali.

Jika kita meningkatkan derajat *multiprogramming*, itu sama artinya dengan melakukan *over-allocating* terhadap memori. Jika kita menjalankan 6 proses, dengan masing-masing mendapatkan 10 halaman, walau pun sebenarnya yang digunakan hanya 5 halaman, kita akan memiliki utilisasi CPU dan *throughput* yang lebih tinggi dengan 10 *frame* yang masih kosong.

Lebih jauh lagi, kita harus mempertimbangkan bahwa sistem memori tidak hanya digunakan untuk menangani pengalamatan suatu program. Penyangga *(buffer)* untuk I/O juga menggunakan sejumlah memori. Penggunaan ini dapat meningkatkan pemakaian algoritma dalam penempatan di memori.

Beberapa sistem mengalokasikan secara pasti beberapa persen dari memori yang dimilikinya untuk penyangga I/O, dimana keduanya, baik proses pengguna mau pun subsistem dari I/O saling berlomba untuk memanfaatkan seluruh sistem memori.

4.9.1. Skema Dasar

Pemindahan halaman mengambil pendekatan seperti berikut. Jika tidak ada *frame* yang kosong, kita mencari *frame* yang tidak sedang digunakan dan mengosongkannya. Kita dapat mengosongkan sebuah *frame* dengan menuliskan isinya ke ruang pertukaran *(swap space)*, dan merubah tabel halaman (juga tabel-tabel lainnya) untuk mengindikasikan bahwa halaman tesebut tidak akan lama berada di memori.

Sekarang kita dapat menggunakan *frame* yang kosong sebagai penyimpan halaman dari proses yang salah. Rutinitas pemindahan halaman:

1. Cari lokasi dari halaman yang diinginkan pada *disk*
2. Cari *frame* kosong:
    a. Jika ada *frame* kosong, gunakan.
    b. Jika tidak ada *frame* kosong, gunakan algoritma pemindahan halaman untuk menyeleksi *frame* yang akan digunakan.
    c. Tulis halaman yang telah dipilih ke disk, ubah tabel halaman dan tabel *frame*.
3. Baca halaman yang diinginkan kedalam *frame* kosong yang baru, ubah tabel halaman dan tabel *frame*.
4. Ulang dari awal proses pengguna.

Jika tidak ada *frame* yang kosong, pentransferan dua halaman (satu masuk, satu keluar) akan dilakukan. Situasi ini secara efektif akan menggandakan waktu pelayanan kesalahan halaman dan meningkatkan waktu akses efektif. Kita dapat mengurangi pemborosan ini dengan menggunakan bit tambahan. Masingmasing halaman atau *frame* mungkin memiliki bit tambahan yang diasosiasikan didalam perangkat keras.

Pemindahan halaman merupakan dasar dari *demand paging*. Yang menjembatani pemisahan antara memori lojik dan memori fisik. Dengan mekanisme seperti ini, memori virtual yang sangat besar

dapat disediakan untuk *programmer* dalam bentuk memori fisik yang lebih kecil. Dengan *nondemand paging*, alamat dari *user* dipetakan kedalam alamat fisik, jadi 2 set alamat dapat berbeda. Seluruh halaman dari proses masih harus berada di memori fisik. Dengan *demand paging*, ukuran dari ruang alamat logika sudah tidak dibatasi oleh memori fisik.

Kita harus menyelesaikan 2 masalah utama untuk mengimplementasikan *demand paging*. Kita harus mengembangkan algoritma pengalokasian *frame* dan algoritma pemindahan halaman. Jika kita memiliki banyak proses di memori, kita harus memutuskan berapa banyak *frame* yang akan dialokasikan ke masing-masing proses. Lebih jauh lagi, saat pemindahan halaman diinginkan, kita harus memilih *frame* yang akan dipindahkan. Membuat suatu algoritma yang tepat untuk menyelesaikan masalah ini adalah hal yang sangat penting.

Ada beberapa algoritma pemindahan halaman yang berbeda. Kemungkinan setiap Sistem Operasi memiliki skema pemindahan yang unik. Algoritma pemindahan yang baik adalah yang memiliki tingkat kesalahan halaman terendah.

Kita mengevaluasi algoritma dengan menjalankannya dalam *string* khusus di memori acuan dan menghitung jumlah kesalahan halaman. *String* dari memori acuan disebut *string* acuan *(reference string)*.

Sebagai contoh, jika kita memeriksa proses khusus, kita mungkin akan mencatat urutan alamat seperti dibawah ini:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

dimana pada 100 *bytes* setiap halaman, diturunkan menjadi *string* acuan seperti berikut:

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

Perlu diperhatikan bahwa selama jumlah *frame* meningkat, jumlah kesalahan halaman menurun.

Penambahan memori fisik akan meningkatkan jumlah *frame*.

4.9.2. Pemindahan Halaman Secara FIFO

Algoritma ini adalah algoritma paling sederhana dalam hal pemindahan halaman. Algoritma pemindahan FIFO *(First In First Out)* mengasosiasikan waktu pada saat halaman dibawa kedalam memori dengan masing-masing halaman. Pada saat halaman harus dipindahkan, halaman yang paling tua yang dipilih.

Sebagai contoh:

```
7  0  1  2  0  3  0  4  2  3  0  3  2  1  2  0  1  7  0  1
7  7  7  2     2  2  4  4  4  0        0  0        7  7  7
   0  0  0     3  3  3  2  2  2        1  1        1  0  0
      1  1     1  0  0  0  3  3        3  2        2  2  1
```

frame halaman

Gambar 4-11. String Acuan.

Dari contoh diatas, terdapat 15 kesalahan halaman. Algoritma FIFO mudah untuk dipahami dan diimplementasikan. Namun *performance*-nya tidak selalu bagus. Salah satu kekurangan dari algoritma FIFO adalah kemungkinan terjadinya anomali Beladi, dimana dalam beberapa kasus, tingkat kesalahan akan meningkat seiring dengan peningkatan jumlah *frame* yang dialokasikan.

4.9.3. Pemindahan Halaman Secara Optimal

Salah satu akibat dari upaya mencegah terjadinya anomali Beladi adalah algoritma pemindahan halaman secara optimal. Algoritma ini memiliki tingkat kesalahan halaman terendah dibandingkan dengan algoritma-algoritma lainnya. Algoritma ini tidak akan mengalami anomaly Belady. Konsep utama dari algoritma ini adalah mengganti halaman yang tidak akan digunakan untuk jangka waktu yang paling lama. Algoritma ini menjamin kemungkinan tingkat kesalahan terendah untuk jumlah *frame* yang tetap.

Sebagai contoh:

```
7  0  1  2  0  3  0  4  2  3  0  3  2  1  2  0  1  7  0  1
7  7  7  2     2     2           2        2              7
   0  0  0     0     4           0        0              0
      1  1     3     3           3        1              1
```

Gambar 4-12. String Acuan.

Dari contoh diatas, terdapat 9 kesalahan halaman. Dengan hanya 9 kesalahan halaman, algoritma optimal jauh lebih baik daripada algoritma FIFO.

Perlu disayangkan, algoritma optimal susah untuk diimplementasikan kedalam program, karena algoritma ini menuntut pengetahuan tentang *string* acuan yang akan muncul.

4.9.4. Pemindahan Halaman Secara LRU

Jika algoritma optimal sulit untuk dilakukan, mungkin kita dapat melakukan pendekatan terhadap algoritma tersebut. Jika kita

menggunakan waktu yang baru berlalu sebagai pendekatan terhadap waktu yang akan datang, kita akan memindahkan halaman yang sudah lama tidak digunakan dalam jangka waktu yang terlama. Pendekatan ini disebut algoritma LRU *(Least Recently Used)*.

Algoritma LRU mengasosiasikan dengan masing-masing halaman waktu dari halaman yang terakhir digunakan. Ketika halaman harus dipindahkan, LRU memilih halaman yang paling lama tidak digunakan pada waktu yang lalu. Inilah algoritma LRU, melihat waktu yang telah lalu, bukan waktu yang akan datang.

Sebagai contoh:

```
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
7 7 7 2   2   4 4 4 0     1   1     1
  0 0 0   0   0 0 3 3     3   0     0
    1 1   3   3 2 2 2     2   2     7
```

frame halaman

Gambar 4-13. String Acuan.

Dari contoh diatas, terdapat 12 kesalahan halaman. Meski pun algoritma ini menghasilkan 12 kesalahan halaman, algoritma ini masih lebih baik daripada algoritma FIFO, yang menghasilkan 15 kesalahan halaman. Untuk mengimplementasikan algoritma LRU, terdapat 2 implementasi yang dapat digunakan, yaitu dengan *counter* dan *stack*.

Selain algoritma optimal, algoritma LRU juga dapat terhindar dari anomali Beladi. Salah satu kelas dari algoritma pemindahan halaman adalah algoritma *stack*, yang juga tidak akan pernah mengalami anomali Beladi. Algoritma *stack* ini menyimpan nomor-nomor halaman pada *stack*. Kapan pun suatu halaman ditunjuk, halaman ini dikeluarkan dari *stack* dan diletakkan di blok paling atas dari *stack*. Dengan cara seperti ini, blok paling atas dari *stack* selalu berisi halaman yang baru digunakan, sedangkan blok terbawah dari *stack* selalu berisi halaman yang sudah lama tidak digunakan. Karena suatu halaman dalam *stack* dapat dikeluarkan meski pun berada ditengah-tengah *stack*, maka implementasi terbaik untuk algoritma ini adalah dengan daftar mata rantai ganda (*doubly linked list*), dengan kepala dan ekor sebagai penunjuk. Pendekatan ini sangat tepat untuk perangkat lunak atau implementasi kode mikro dari algoritma LRU. Sebagai contoh:

```
4 7 0 7 1 0 1 2 1 2 7 1 2
4 7 0 7 1 0 1 2 1 2 7 1 2
  4 7 0 7 1 0 1 2 1 2 7 1
    4  4 0 7 7 0 0 0 1  2 7
         4 4 4 7 7 7 0 0 0
                4 4 4 4 4 4
```

frame halaman.

Gambar 4-14. String Acuan.

4.9.5. Pemindahan Halaman Secara Perkiraan LRU

Hanya sedikit sistem komputer yang menyediakan perangkat lunak yang memberikan cukup dukungan terhadap algoritma pemindahan halaman secara LRU. Banyak sistem yang tidak menyediakan perangkat lunak yang memberikan dukungan terhadap algoritma LRU, sehingga terpaksa menggunakan algoritma lain, seperti FIFO. Banyak sistem menyediakan bantuan untuk menangani masalah ini, misalnya dengan bit acuan. Bit acuan untuk halaman diset oleh perangkat lunak kapan pun halaman tersebut ditunjuk. Bit acuan diasosiasikan dengan masing-masing isi dari tabel halaman.

Awalnya, seluruh bit dikosongkan oleh sistem operasi. Selama proses pengguna dijalankan, bit yang diasosiasikan ke masing-masing halaman acuan diset menjadi 1 oleh perangkat keras. Setelah beberapa waktu, kita dapat menentukan halaman mana yang sudah digunakan dan halaman mana yang belum digunakan dengan menguji bit-bit acuan. Informasi tersebut memberikan informasi penting untuk banyak algoritma pemindahan halaman yang memperkirakan halaman mana yang sudah lama tidak digunakan.

4.9.5.1. Algoritma *Additional-Reference-Bit*

Kita bisa mendapatkan informasi tambahan mengenai urutan dengan mencatat bit-bit acuan pada suatu interval yang tetap. Kita dapat menyimpan 8-bit byte untuk masing-masing halaman pada tabel di memori. Pada interval tertentu, pencatat waktu (*timer*) melakukan interupsi dengan mentransfer kontrol kepada sistem operasi. Sistem operasi mengubah bit acuan untuk masing-masing halaman kedalam bit *high-order* dari 8-bit byte ini dan membuang bit *low-order*. Register pengganti 8-bit ini berisi sejarah penggunaan halaman dalam periode 8 waktu terakhir.

Sebagai contoh, seandainya register pengganti berisi 00000000, maka itu berarti halaman sudah tidak digunakan dalam periode 8

waktu terakhir, halaman yang digunakan paling tidak 1 kali akan memiliki nilai register penggati 11111111.

4.9.5.2. Algoritma *Second-Chance*

Algoritma "*second-chance*" didasari oleh algoritma FIFO. Pada saat suatu halaman ditunjuk, kita akan menginspeksi bit acuannya. Jika bit acuan tersebut bernilai 0, kita memproses untuk membuang halaman ini. Jika bit acuan tersebut bernilai 1, kita berikan kesempatan kedua untuk halaman ini dan menyeleksi halaman FIFO selanjutnya.

Ketika suatu halaman mendapatkan kesempatan kedua, bit acuannya dikosongkan dan waktu tibanya direset menjadi saat ini. Karena itu, halaman yang mendapatkan kesempatan kedua tidak akan dipindahkan sampai seluruh halaman dipindahkan. Tambahan lagi, jika halaman yang digunakan cukup untuk menampung 1 set bit acuan, maka halaman ini tidak akan pernah dipindahkan.

4.9.5.3. Algoritma *Second-Chance* (Yang Diperbaiki)

Kita dapat memperbaiki kekurangan dari algoritma *second-chance* dengan mempertimbangkan 2 hal sekaligus, yaitu bit acuan dan bit modifikasi. Dengan 2 bit ini, kita akan mendapatkan 4 kemungkinan yang akan terjadi, yaitu:

- (0,0) tidak digunakan dan tidak dimodifikasi, bit terbaik untuk dipindahkan.
- (0,1) tidak digunakan tapi dimodifikasi, tidak terlalu baik untuk dipindahkan karena halaman ini perlu ditulis sebelum dipindahkan.
- (1,0) digunakan tapi tidak dimodifikasi, terdapat kemungkinan halaman ini akan segera digunakan lagi.
- (1,1) digunakan dan dimodifikasi, halaman ini mungkin akan segera digunakan lagi dan halaman ini perlu ditulis ke *disk* sebelum dipindahkan.

Algoritma ini digunakan dalam skema manajemen memori virtual Macintosh.

4.9.6. Dasar Perhitungan Pemindahan Halaman

Banyak algoritma-algoritma lain yang dapat digunakan untuk pemindahan halaman. Sebagai contoh, kita dapat menyimpan

counter dari nomor acuan yang sudah dibuat untuk masing-masing halaman, dan mengembangkan 2 skema dibawah ini:

ALGORITMA PEMINDAHAN HALAMAN LFU Algoritma LFU (*Least Frequently Used*)

menginginkan halaman dengan nilai terkecil untuk dipindahkan. Alasannya, halaman yang digunakan secara aktif akan memiliki nilai acuan yang besar.

ALGORITMA PEMINDAHAN HALAMAN MFU Algoritma MFU (*Most Frequently Used*) didasarkan pada argumen yang menyatakan bahwa halaman dengan nilai terkecil mungkin baru saja dimasukkan dan baru digunakan.

Kedua algoritma diatas tidaklah terlalu umum, hal ini disebabkan karena implementasi dari kedua algoritma diatas sangatlah mahal.

4.9.7. Algoritma *Page-Buffering*

Prosedur lain sering digunakan untuk menambah kekhususan dari algoritma pemindahan halaman. Sebagai contoh, pada umumnya sistem menyimpan *pool* dari *frame* yang kosong. Prosedur ini memungkinkan suatu proses mengulang dari awal secepat mungkin, tanpa perlu menunggu halaman yang akan dipindahkan untuk ditulis ke *disk* karena *frame*-nya telah ditambahkan kedalam *pool frame* kosong.

Teknik seperti ini digunakan dalam sistem VAX/ VMS, dengan algoritma FIFO. Ketika algoritma FIFO melakukan kesalahan dengan memindahkan halaman yang masih digunakan secara aktif, halaman tersebut akan dengan cepat diambil kembali dari penyangga *frame*-kosong, untuk melakukan hal tersebut tidak ada I/O yang dibutuhkan. Metode ini diperlukan oleh VAX karena versi terbaru dari VAX tidak mengimplementasikan bit acuan secara tepat.

4.10. Alokasi *Frame*

Terdapat masalah dalam alokasi *frame* dalam penggunaan memori virtual, masalahnya yaitu bagaimana kita membagi memori yang bebas kepada berbagai proses yang sedang dikerjakan? Jika ada sejumlah frame bebas dan ada dua proses, berapakah *frame* yang didapatkan tiap proses?

Kasus paling mudah dari memori virtual adalah sistem satu pemakai. Misalkan sebuah sistem mempunyai memori 128K dengan ukuran halaman 1K, sehingga ada 128 *frame*. Sistem operasinya menggunakan 35K sehingga ada 93 *frame* yang tersisa untuk proses tiap user. Untuk *pure demand paging*, ke-93 *frame*

tersebut akan ditaruh pada daftar *frame* bebas. Ketika sebuah proses user mulai dijalankan, akan terjadi sederetan *page fault*. Sebanyak 93 *page fault* pertama akan mendapatkan *frame* dari daftar *frame* bebas. Saat *frame* bebas sudah habis, sebuah algoritma pergantian halaman akan digunakan untuk memilih salah satu dari 93 halaman di memori yang diganti dengan yang ke 94, dan seterusnya. Ketika proses selesai atau diterminasi, sembilan puluh tiga *frame* tersebut akan disimpan lagi pada daftar *frame* bebas.

Terdapat macam-macam variasi untuk strategi sederhana ini, kita bisa meminta sistem operasi untuk mengalokasikan seluruh *buffer* dan ruang tabel-nya dari daftar *frame* bebas. Saat ruang ini tidak digunakan oleh sistem operasi, ruang ini bisa digunakan untuk mendukung paging dari user. Kita juga dapat menyimpan tiga *frame* bebas yang dari daftar *frame* bebas, sehingga ketika terjadi *page fault*, ada *frame* bebas yang dapat digunakan untuk *paging*. Saat pertukaran halaman terjadi, penggantinya dapat dipilih, kemudian ditulis ke disk, sementara proses user tetap berjalan.

Variasi lain juga ada, tetapi ide dasarnya tetap yaitu proses pengguna diberikan *frame* bebas yang mana saja. Masalah lain muncul ketika *demand paging* dikombinasikan dengan *multiprogramming*. Hal ini terjadi karena *multiprogramming* menaruh dua (atau lebih) proses di memori pada waktu yang bersamaan.

4.10.1. Jumlah Frame Minimum

Tentu saja ada berbagai batasan pada strategi kita untuk alokasi *frame*. Kita tidak dapat mengalokasikan lebih dari jumlah total *frame* yang tersedia (kecuali ada *page sharing*). Ada juga jumlah minimal *frame* yang dapat di alokasikan. Jelas sekali, seiring dengan bertambahnya jumlah *frame* yang dialokasikan ke setiap proses berkurang, tingkat *page fault* bertambah dan mengurangi kecepatan eksekusi proses.

Selain hal tersebut di atas, ada jumlah minimum *frame* yang harus dialokasikan. Jumlah minimum ini ditentukan oleh arsitektur set instruksi. Ingat bahwa ketika terjadi *page fault*, sebelum eksekusi instruksi selesai, instruksi tersebut harus diulang. Sehingga kita harus punya jumlah *frame* yang cukup untuk menampung semua halaman yang dirujuk oleh sebuah instruksi tunggal.

Jumlah minimum *frame* ditentukan oleh arsitektur komputer. Sebagai contoh, instruksi *move* pada PDP-11 adalah lebih dari satu kata untuk beberapa modus pengalamatan, sehingga instruksi tersebut bisa membutuhkan dua halaman. Sebagai tambahan, tiap operannya mungkin merujuk tidak langsung, sehingga total ada enam *frame*. Kasus terburuk untuk IBM 370 adalah instruksi MVC.

Karena instruksi tersebut adalah instruksi perpindahan dari penyimpanan ke penyimpanan, instruksi ini butuh 6 bit dan dapat memakai dua halaman. Satu blok karakter yang akan dipindahkan dan daerah tujuan perpindahan juga dapat memakai dua halaman, sehingga situasi ini membutuhkan enam *frame*.

Kesimpulannya, jumlah minimum *frame* yang dibutuhkan per proses tergantung dari arsitektur komputer tersebut, sementara jumlah maksimumnya ditentukan oleh jumlah memori fisik yang tersedia. Di antara kedua jumlah tersebut, kita punya pilihan yang besar untuk alokasi *frame*.

4.10.2. Algoritma Alokasi

Cara termudah untuk membagi m *frame* terhadap n proses adalah untuk memberikan bagian yang sama, sebanyak m/n *frame* untuk tiap proses. Sebagai contoh ada 93 *frame* tersisa dan 5 proses, maka tiap proses akanmendapatkan 18 *frame*. Frame yang tersisa, sebanyak 3 buah dapat digunakan sebagai *frame* bebas cadangan. Strategi ini disebut *equal allocation*.

Sebuah alternatif yaitu pengertian bahwa berbagai proses akan membutuhkan jumlah memori yang berbeda. Jika ada sebuah proses sebesar 10K dan sebuah proses basis data 127K dan hanya kedua proses ini yang berjalan pada sistem, maka ketika ada 62 *frame* bebas, tidak masuk akal jika kita memberikan masing-masing proses 31 *frame*. Proses pertama hanya butuh 10 *frame*, 21 *frame* lain akan terbuang percuma.

Untuk menyelesaikan masalah ini, kita menggunakan *proportional allocation*. Kita mengalokasikan memori yang tersedia kepada setiap proses tergantung pada ukurannya.

Let the size of the virtual memory for process pi be si, and define S = si.

Lalu, jika jumlah total dari *frame* yang tersedia adalah m, kita mengalokasikan proses ai ke proses pi, dimana ai mendekati

ai = si / S x m

Dalam kedua strategi ini, tentu saja, alokasi untuk setiap proses bisa bervariasi berdasarkan *multiprogramming level*-nya. Jika *multiprogramming level*-nya meningkat, setiap proses akan kehilangan beberapa *frame* guna menyediakan memori yang dibutuhkan untuk proses yang baru. Di sisi lain, jika *multiprogramming level*-nya menurun, *frame* yang sudah dialokasikan pada bagian process sekarang bisa disebar ke proses-proses yang masih tersisa.

Mengingat hal itu, dengan *equal* atau pun *proportional allocation*, proses yang berprioritas tinggi diperlakukan sama dengan proses yang berprioritas rendah. Berdasarkan definisi tersebut, bagaimanapun juga, kita ingin memberi memori yang lebih pada proses yang berprioritas tinggi untuk mempercepat eksekusi-nya, *to the detriment of low-priority processes*.

Satu pendekatan adalah menggunakan *proportional allocation scheme* dimana perbandingan *frame*-nya tidak tergantung pada ukuran relatif dari proses, melainkan lebih pada prioritas proses, atau tergantung kombinasi dari ukuran dan prioritas.

4.10.3. Alokasi Global lawan Local

Faktor penting lain dalam cara-cara pengalokasian *frame* ke berbagai proses adalah penggantian halaman. Dengan proses-proses yang bersaing mendapatkan *frame*, kita dapat mengklasifikasikan algoritma penggantian halaman kedalam dua kategori *broad*: Penggantian Global dan Penggantian Lokal.

Penggantian Global memperbolehkan sebuah proses untuk menyeleksi sebuah *frame* pengganti dari himpunan semua *frame*, meski pun *frame* tersebut sedang dialokasikan untuk beberapa proses lain; satu proses dapat mengambil sebuah *frame* dari proses yang lain. Penggantian Lokal mensyaratkan bahwa setiap proses boleh menyeleksi hanya dari himpunan *frame* yang telah teralokasi pada proses itu sendiri.

Untuk contoh, pertimbangkan sebuah skema alokasi dimana kita memperbolehkan proses berprioritas tinggi untuk meyeleksi *frame* dari proses berprioritas rendah untuk penggantian. Sebuah proses dapat menyeleksi sebuah pengganti dari *frame*-nya sendiri atau dari *frame-frame* proses yang berprioritas lebih rendah. Pendekatan ini memperbolehkan sebuah proses berprioritas tinggi untuk meningkatkan alokasi *frame*-nya pada expense proses berprioritas rendah.

Dengan strategi Penggantian Lokal, jumlah *frame* yang teralokasi pada sebuah proses tidak berubah. Dengan Penggantian Global, ada kemungkinan sebuah proses hanya menyeleksi *frame-frame* yang teralokasi pada proses lain, sehingga meningkatkan jumlah *frame* yang teralokasi pada proses itu sendiri (asumsi bahwa proses lain tidak memilih *frame* proses tersebut untuk penggantian).

Masalah pada algoritma Penggantian Global adalah bahwa sebuah proses tidak bisa mengontrol *page-fault*-nya sendiri. Himpunan halaman dalam memori untuk sebuah proses tergantung tidak hanya pada kelakuan paging dari proses tersebut, tetapi juga pada kelakuan *paging* dari proses lain. Karena itu, proses yang sama

dapat tampil berbeda (memerlukan 0,5 detik untuk satu eksekusi dan 10,3 detik untuk eksekusi berikutnya) *due to totally external circumstances*. Dalam Penggantian Lokal, himpunan halaman dalam memori untuk sebuah proses hanya dipengaruhi kelakuan paging proses itu sendiri.

Penggantian Lokal dapat menyembunyikan sebuah proses dengan membuatnya tidak tersedia bagi proses lain, menggunakan halaman yang lebih sedikit pada memori. Jadi, secara umum Penggantian Global menghasilkan sistem throughput yang lebih bagus, maka itu artinya metode yang paling sering digunakan.

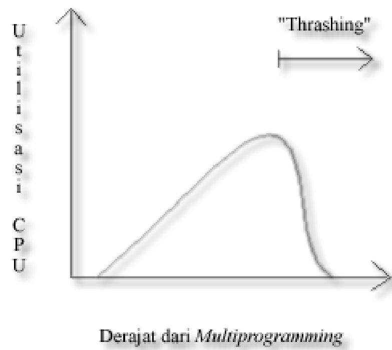
4.11. *Thrashing*

Jika suatu proses tidak memiliki *frame* yang cukup, walau pun kita memiliki kemungkinan untuk mengurangi banyaknya *frame* yang dialokasikan menjadi minimum, tetap ada halaman dalam jumlah besar yang memiliki kondisi aktif menggunakannya. Maka hal ini akan mengakibatkan kesalahan halaman. Pada kasus ini, kita harus mengganti beberapa halaman menjadi halaman yang dibutuhkan walau pun halaman yang diganti pada waktu dekat akan dibutuhkan lagi. Hal ini mengakibatkan kesalahan terus menerus.

Aktivitas yang tinggi dari *paging* disebut *thrashing*. Suatu proses dikatakan *thrashing* jika proses menghabiskan waktu lebih banyak untuk *paging* daripada eksekusi (proses sibuk untuk melakukan *swap-in swap-out*).

4.11.1. Penyebab *Thrashing*

Penyebab dari *thrashing* adalah utilisasi CPU yang rendah. Jika utilisasi CPU terlalu rendah, kita menambahkan derajat dari *multiprogramming* dengan menambahkan proses baru ke sistem. Sejalan dengan bertambahnya derajat dari *multiprogramming*, utilisasi CPU juga bertambah dengan lebih lambat sampai maksimumnya dicapai. Jika derajat dari *multiprogramming* ditambah terus menerus, utilisasi CPU akan berkurang dengan drastis dan terjadi *thrashing*. Untuk menambah utilisasi CPU dan menghentikan *thrashing*, kita harus mengurangi derajat dari *multiprogramming*.



Gambar 4-15. Derajat dari Multiprogramming.

Kita dapat membatasi efek dari *thrashing* dengan menggunakan algoritma penggantian lokal atau prioritas. Dengan penggantian lokal, jika satu proses mulai *thrashing*, proses tersebut tidak dapat mencuri *frame* dari proses yang lain dan menyebabkan proses tersebut tidak langsung mengalami *thrashing*. Jika proses *thrashing*, proses tersebut akan berada di antrian untuk melakukan *paging* yang mana hal ini memakan banyak waktu. Rata-rata waktu layanan untuk kesalahan halaman akan bertambah seiring dengan makin panjangnya rata-rata antrian untuk melakukan *paging*. Maka, waktu akses efektif akan bertambah walau pun untuk suatu proses yang tidak thrashing.

Untuk menghindari *thrashing*, kita harus menyediakan sebanyak mungkin *frame* sesuai dengan kebutuhan suatu proses. Cara untuk mengetahui berapa *frame* yang dibutuhkan salah satunya adalah dengan strategi *Working Set* yang akan dibahas pada bagian 10.5.2 yang mana strategi tersebut dimulai dengan melihat berapa banyak *frame* yang sesungguhnya digunakan oleh suatu proses. Ini merupakan model lokalitas dari suatu eksekusi proses.

Selama suatu proses dieksekusi, model lokalitas berpindah dari satu lokalitas ke lokalitas lainnya. Lokalitas adalah kumpulan halaman yang secara aktif digunakan bersama. Suatu program pada umumnya dibuat pada beberapa lokalitas, sehingga ada kemungkinan dapat terjadi *overlap*. Thrashing dapat muncul bila ukuran lokalitas lebih besar dari ukuran memori total.

4.11.2. Model *Working Set*

Model *Working Set* didasarkan pada asumsi lokalitas. Model ini menggunakan parameter Δ (delta) untuk mendefinisikan jendela

Working Set. Idenya adalah untuk menentukan Δ halaman yang dituju yang paling sering muncul. Kumpulan dari halaman dengan Δ halaman yang dituju yang paling sering muncul disebut *Working Set*. *Working Set* adalah pendekatan dari program lokalitas.

Contoh 4-1. Tabel Halaman

Jika terdapat tabel halaman yang dituju dengan isinya
1 3 5 7 2 5 8 9 dengan
Δ = 8,

Working Set pada waktu t1 adalah
{1, 2, 3, 5, 7, 8, 9}

Keakuratan *Working Set* tergantung pemilihan dari :

- Jika terlalu kecil, tidak akan dapat mewakilkan keseluruhan dari lokalitas.
- Jika terlalu besar, akan menyebabkan *overlap* beberapa lokalitas.
- Jika tidak terbatas, *Working Set* adalah kumpulan halaman sepanjang eksekusi proses.

Jika kita menghitung ukuran dari *Working Set*, WWSi, untuk setiap proses pada sistem, kita hitung dengan D = Δ WSSi, dimana D merupakan total *demand* untuk *frame*.

Jika total *demand* lebih dari total banyaknya *frame* yang tersedia (D > m), *thrashing* dapat terjadi karena beberapa proses akan tidak memiliki *frame* yang cukup. Jika hal tersebut terjadi, dilakukan satu pengeblokan dari proses-proses yang sedang berjalan.

Strategi *Working Set* menangani *thrashing* dengan tetap mempertahankan derajat dari *multiprogramming* setinggi mungkin.

Kesulitan dari model *Working Set* ini adalah menjaga *track* dari *Working Set*. Jendela *Working Set* adalah jendela yang bergerak. Suatu halaman berada pada *Working Set* jika halaman tersebut mengacu ke mana pun pada jendela *Working Set*. Kita dapat mendekati model *Working Set* dengan *fixed interval timer interrupt* dan *reference* bit.

Contoh: Δ = 10000 reference, *Timer interrupt* setiap 5000 reference.

Ketika kita mendapat *interrupt*, kita kopi dan hapus nilai *reference* bit dari setiap halaman.

Jika kesalahan halaman muncul, kita dapat menentukan *current reference* bit dan 2 pada bit memori untuk memutuskan apakah

halaman itu digunakan dengan 10000 ke 15000 reference terakhir.

Jika digunakan, paling sedikit satu dari bit-bit ini akan aktif. Jika tidak digunakan, bit ini akan menjadi tidak aktif.

Halaman yang memiliki paling sedikit 1 bit aktif, akan berada di *working-set*.

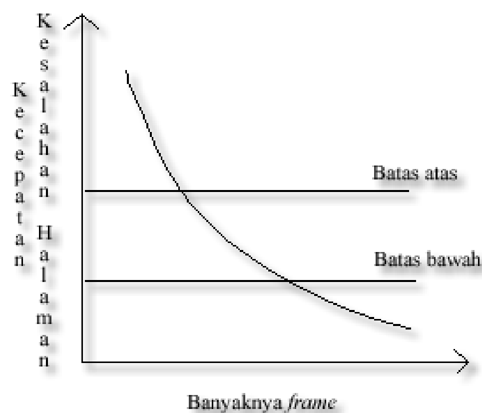
Hal ini tidaklah sepenuhnya akurat karena kita tidak dapat memberitahukan dimana pada interval 5000 tersebut, *reference* muncul. Kita dapat mengurangi ketidakpastian dengan menambahkan sejarah bit kita dan frekuensi dari *interrupt*.

Contoh: 20 bit dan interrupt setiap 1500 reference.

4.11.3. Frekuensi Kesalahan Halaman

Working-set dapat berguna untuk *prepaging*, tetapi kurang dapat mengontrol *thrashing*. Strategi menggunakan frekuensi kesalahan halaman mengambil pendekatan yang lebih langsung.

Thrashing memiliki kecepatan kesalahan halaman yang tinggi. Kita ingin mengontrolnya. Ketika terlalu tinggi, kita mengetahui bahwa proses membutuhkan *frame* lebih. Sama juga, jika terlalu rendah, maka proses mungkin memiliki terlalu banyak *frame*. Kita dapat menentukan batas atas dan bawah pada kecepatan kesalahan halaman seperti terlihat pada gambar berikut ini.



Gambar 4-16. Jumlah Frame.

Jika kecepatan kesalahan halaman yang sesungguhnya melampaui batas atas, kita mengalokasikan *frame* lain ke proses tersebut, sedangkan jika kecepatan kesalahan halaman di bawah batas bawah, kita pindahkan *frame* dari proses tersebut. Maka kita

dapat secara langsung mengukur dan mengontrol kecepatan kesalahan halaman untuk mencegah thrashing.

4.12. Contoh Pada Sistem Operasi

Pada bagian ini kita akan membahas beberapa contoh dalam penggunaan memori virtual.

4.12.1. Windows NT

Windows NT mengimplementasikan memori virtual dengan menggunakan demand paging melalui clustering. Clustering menanganani *page fault* dengan menambahkan tidak hanya page yang terkena fault, tetapi juga beberapa *page* yang ada dekat *page*tersebut. Saat proses pertama dibuat, dia diberikan *Working Set minimum* yaitu jumlah minimum *page* yang dijamin akan dimiliki oleh proses tersebut dalam memori. Jika memori yang cukup tersedia, proses dapat diberikan page sampai sebanyak *Working Set maximum*. Manager memori virtual akan menyimpan daftar dari *frame page* yang bebas. Terdapat juga sebuah nilai batasan yang diasosiasikan dengan daftar ini untuk mengindikasikan apakah memori yang tersedia masih mencukupi. Jika proses tersebut sudah sampai pada *Working Set maximum*-nya dan terjadi *page fault*, maka dia harus memilih page pengganti dengan menggunakan kebijakan penggantian page lokal FIFO.

Saat jumlah memori bebas jatuh di bawah nilai batasan, manager memori virtual menggunakan sebuah taktik yang dikenal sebagai *automatic working set trimming* untuk mengembalikan nilai tersebut di atas batasan. Hal ini bekerja dengan mengevaluasi jumlah page yang dialokasikan kepada proses. Jika proses telah mendapat alokasi *page* lebih besar daripada *Working Set minimum*-nya, manager memori virtual akan menggunakan algoritma FIFO untuk mengurangi jumlah *page*-nya sampai *working-set minimum*.

Jika memori bebas sudah tersedia, proses yang bekerja pada *working set minimum* dapat mendapatkan *page* tambahan.

4.12.2. Solaris 2

Dalam sistem operasi Solaris 2, jika sebuah proses menyebabkan terjadi *page fault*, kernel akan memberikan *page* kepada proses Tersebut dari daftar *page* bebas yang disimpan. Akibat dari hal ini adalah, kernel harus menyimpan sejumlah memori bebas. Terhadap daftar ini ada dua parameter yg disimpan yaitu *minfree*

dan *lotsfree*, yaitu batasan minimum dan maksimum dari memori bebas yang tersedia. Empat kali dalam tiap detiknya, kernel memeriksa jumlah memori yang bebas. Jika jumlah tersebut jatuh di bawah *minfree*, maka sebuah proses *pageout* akan dilakukan, dengan pekerjaan sebagai berikut. Pertama *clock* akan memeriksa semua *page* dalam memori dan mengeset bit referensi menjadi 0.

Saat berikutnya, *clock* kedua akan memeriksa bit referensi *page* dalam memori, dan mengembalikan bit yang masih di set ke 0 ke daftar memori bebas. Hal ini dilakukan sampai jumlah memori bebas melampaui parameter *lotsfree*. Lebih lanjut, proses ini dinamis, dapat mengatur kecepatan jika memori terlalu sedikit. Jika proses ini tidak bisa membebaskan memori, maka kernel memulai pergantian proses untuk membebaskan *page* yang dialokasikan ke proses-proses tersebut.

4.12.3. Linux

Seperti pada solaris 2, linux juga menggunakan variasi dari algoritma *clock*. Thread dari kernel linux (kswapd) akan dijalankan secara periodik (atau dipanggil ketika penggunaan memori sudah berlebihan).

Jika jumlah *page* yang bebas lebih sedikit dari batas atas *page* bebas, maka thread tersebut akan berusaha untuk membebaskan tiga *page*. Jika lebih sedikit dari batas bawah *page* bebas, thread tersebut akan berusaha untuk membebaskan 6 *page* dan 'tidur' untuk beberapa saat sebelum berjalan lagi. Saat dia berjalan, akan memeriksa mem_map, daftar dari semua page yang terdapat di memori. Setiap page mempunyai byte umur yang diinisialisasikan ke 3. Setiap kali *page* ini diakses, maka umur ini akan ditambahkan (hingga maksimum 20), setiap kali kswapd memeriksa *page* ini, maka umur akan dikurangi. Jika umur dari sebuah *page* sudah mencapai 0 maka dia bisa ditukar. Ketika kswapd berusaha membebaskan *page*, dia pertama akan membebaskan *page* dari *cache*, jika gagal dia akan mengurangi *cache* sistim berkas, dan jika semua cara sudah gagal, maka dia akan menghentikan sebuah proses.

Alokasi memori pada linux menggunakan dua buah alokasi yang utama, yaitu algoritma *buddy* dan *slab*.

Untuk algoritma *buddy*, setiap rutin pelaksanaan alokasi ini dipanggil, dia memeriksa blok memori berikutnya, jika ditemukan dia dialokasikan, jika tidak maka daftar tingkat berikutnya akan diperiksa.

Jika ada blok bebas, maka akan dibagi jadi dua, yang satu dialokasikan dan yang lain dipindahkan ke daftar yang di bawahnya.

4.13. Pertimbangan Lain

Pemilihan algoritma penggantian dan aturan alokasi adalah keputusan-keputusan utama yang kita buat untuk sistem pemberian halaman. Masih banyak pertimbangan lain.

4.13.1. Sebelum Pemberian Halaman

Sebuah ciri dari sistem *demand-paging* adalah adanya *page fault* yang terjadi saat proses dimulai. Situasi ini adalah hasil dari percobaan untuk mendapatkan tempat pada awalnya. Situasi yang sama mungkin muncul di lain waktu. Saat proses *swapped-out* dimulai kembali, seluruh halaman ada di *disk* dan setiap halaman harus dibawa masuk oleh *page-fault*-nya masing-masing. Sebelum pemberian halaman mencoba untuk mencegah tingkat tinggi dari *paging* awal. Stateginya adalah untuk membawa seluruh halaman yang akan dibutuhkan pada satu waktu ke memori.

Pada sistem yang menggunakan model *working-set*, sebagai contoh, kita tetap dengan setiap proses sebuah daftar dari halaman-halaman di *working-set*-nya. Jika kita harus menunda sebuah proses (karena menunggu I/O atau kekurangan *frame* bebas), kita mengingat *working-set* untuk proses itu. Saat proses itu akan melanjutkan kembali (I/O komplit atau *frame* bebas yang cukup), kita secara otomatis membawa kembali ke memori seluruh *working-set* sebelum memulai kembali proses tersebut.

Sebelum pemberian halaman bisa unggul di beberapa kasus. Pertanyaan sederhananya adalah apakah biaya untuk menggunakan sebelum pemberian halaman itu lebih rendah daripada biaya melayani *page-fault* yang berhubungan. Itu mungkin menjadi kasus dimana banyak halaman dibawa kembali ke memori dengan sebelum pemberian halaman tidak digunakan.

4.13.2. Ukuran Halaman

Para perancang sistem operasi untuk mesin yang ada kini jarang memiliki pilihan terhadap ukuran halaman. Bagaimana pun, saat mesin-mesin baru sedang dibuat, pemilihan terhadap ukuran halaman terbaik harus dibuat. Seperti yang kau mungkin harapkan, tidak ada sebuah ukuran halaman yang terbaik.

Namun, ada himpunan faktor-faktor yang mendukung ukuran-ukuran yang bervariasi. Ukuran-ukuran halaman selalu dengan pangkat 2, secara umum berkisar dari 4.096 (2^12) ke 4.194.304 (2^22) bytes

Bagaimana kita memilih sebuah ukuran halaman? Sebuah perhatian adalah ukuran dari tabel halaman.

Untuk sebuah memori virtual dengan ukuran 4 megabytes (2^22), akan ada 4.096 halaman 1.024 bytes, tapi hanya 512 halaman 8.192 bytes. Sebab setiap proses aktif harus memiliki salinan dari tabel halamannya, sebuah halaman yang besar diinginkan.

Di sisi lain, memori lebih baik digunakan dengan halaman yang lebih kecil. Jika sebuah proses dialokasikan di memori mulai dari lokasi 00000, melanjutkan sampai memiliki sebanyak yang dibutuhkan, itu mungkin tidak akan berakhir secara tepat di batas halaman. Kemudian, sebuah bagian dari halaman terakhir harus dialokasikan (sebab halaman-halaman adalah unit-unit dari alokasi) tapi tidak digunakan (pemecahan bagian dalam). Asumsikan ketergantungan antara ukuran proses dan ukuran halaman, kita dapat mengharapkan bahwa, dalam rata-rata, satu-setengah dari halaman terakhir dari setiap proses akan dibuang. Kehilangan ini hanya 256 bytes dari sebuah halaman 512 bytes, tapi akan 4.096 bytes dari halaman 8.192 bytes. Untuk meminimalkan pemecahan bagian dalam, kita membutuhkan ukuran halaman yang kecil.

Masalah lain adalah waktu yang dibutuhkan untuk membaca atau menulis halaman. Waktu I/O terdiri dari mencari, keterlambatan dan waktu pemindahan. Waktu transfer proporsional terhadap jumlah yang dipindahkan (yaitu, ukuran tabel). Sebuah fakta bahwa yang mungkin terasa janggal untuk ukuran tabel yang kecil. Ingat kembali dari Bab 2, bagaimana pun, keterlambatan dan waktu pencarian normalnya membuat waktu pemindahan menjadi kecil. Pada saat laju pemindahan 2 megabytes per detik, hanya menghabiskan 0.2 millidetik untuk memindahkan 512 bytes. Keterlambatan, di sisi lain, kira-kira 8 millidetik dan waktu pencarian 20 millidetik. Dari total waktu I/O (28.2 millidetik), untuk itulah, 1 persen dapat dihubungkan dengan pemindahan sebenarnya. Menggandakan ukuran halaman meningkatkan waktu I/O hingga 28.4 millidetik. Menghabiskan 28.4 millidetik untuk membaca halaman tunggal dari dari 1.024 bytes, tapi 56.4 millidetik untuk jumlah yang sama sebesar dua halaman masing-masing 512 bytes. Kemudian, keinginan untuk meminimalisir waktu I/O untuk ukuran halaman yang lebih besar

4.13.3. Tabel Halaman yang Dibalik

Kegunaan dari bentuk manajemen halaman adalah untuk mengurangi jumlah memori fisik yang dibutuhkan untuk melacak penerjemahan alamat *virtual-to-physical*. Kita menyelesaikan metode penghematan ini dengan membuat tabel yang memiliki hanya satu masukan tiap halaman memori fisik, terdaftar oleh pasangan (pengenal proses, nomor halaman).

Karena mereka tetap menjaga informasi tentang halaman memori virtual yang mana yang disimpan di setiap *frame* fisik, tabel

halaman yang terbalik mengurangi jumlah fisik memori yang dibutuhkan untuk menyimpan informasi ini. Bagaimana pun, tabel halaman yang dibalik tidak lagi mengandung informasi yang lengkap tentang alamat ruang *logical* dari sebuah proses, dan informasi itu dibutuhkan jika halaman yang direferensikan tidak sedang berada di memori. *Demand paging* membutuhkan informasi ini untuk memproses *page faults*. Agar informasi ini tersedia, sebuah tabel halaman luar (satu tiap proses) harus tetap dijaga. Setiap tabel tampak seperti tabel halaman tiap proses tradisional, mengandung informasi dimana setiap halaman *virtual* berada.

Tetapi, melakukan tabel halaman luar menegasikan kegunaan tabel halaman yang dibalik? Sejak tabel-tabel ini direferensikan hanya saat *page fault* terjadi, mereka tidak perlu untuk tersedia secara cepat. Namun, mereka masing-masing diberikan atau dikeluarkan halaman dari memori sesuai kebutuhan. Sayangnya, sebuah *page fault* mungkin sekarang muncul di manager memori virtual menyebabkan halaman lain *fault* seakan-akan halaman ditabel halaman luar perlu untuk mengalokasikan *virtual page* di bantuan penyimpanan. Ini merupakan kasus spesial membutuhkan penanganan di kernel dan delay di proses melihat halaman.

4.13.4. Struktur Program

Demand paging didesain untuk menjadi transparan kepada program pemakai. Di banyak kasus, pemakai sama sekali tidak mengetahui letak halaman di memori. Di kasus lain, bagaimana pun, kinerja sistem dapat ditingkatkan jika pemakai (atau kompilator) memiliki kesadaran akan *demand paging* yang mendasar Pemilihan yang hati-hati dari struktur data dan struktur permograman dapat meningkatkan *locality* dan karenanya menurunkan laju *page fault* dan jumlah halaman di himpunan kerja. Sebuah *stack* memiliki *locality* yang baik, sejak akses selalu dibuat di atas. Sebuah *hash table*, di sisi lain, didesain untuk menyebar referensi-referensi, menghasilkan *locality* yang buruk. Tentunya, referensi akan *locality* hanyalah satu ukuran dari efisiensi penggunaan struktur data. Faktor-faktor lain yang berbobot berat termasuk kecepatan pencarian, jumlah total dari referensi dan jumlah total dari halaman yang disentuh.

4.13.5. Penyambungan Masukan dan Keluaran

Saat *demand paging* digunakan, kita terkadang harus mengizinkan beberapa halaman untuk dikunci di memori. Salah satu situasi muncul saat I/O sering diimplementasikan oleh pemroses I/O yang terpisah.

Sebagai contoh, sebuah pengendali pita magnetik pada umumnya diberikan sejumlah bytes untuk memindahkan dan sebuah alamat memoro untuk *buffer*. Saat pemindahan selesai, CPU diinterupsi.

Kita harus meyakinkan urutan dari kejadian-kejadian berikut tidak muncul: Sebuah proses mengeluarkan permintaan I/O, dan diletakkan di antrian untuk I/O tersebut. Sementara itu, CPU diberikan ke proses-proses lain. Proses-proses ini menyebabkan kesalahan penempatan halaman, dan, menggunakan algoritma penggantian global, salah satu dari mereka menggantikan halaman yang mengandung *memory buffer* untuk proses yang menunggu. Halaman itu dikeluarkan. Kadang-kadang kemudian, saat permintaan I/O bergerak maju menuju ujung dari antrian *device*, I/O terjadi ke alamat yang telah ditetapkan. Bagaimana pun, *frame* ini sekarang sedang digunakan untuk halaman berbeda milik proses lain.

4.13.6. Pemrosesan Waktu Nyata

Diskusi-diskusi di bab ini telah dikonsentrasikan dalam menyediakan penggunaan yang terbaik secara menyeluruh dari sistem komputer dengan meningkatkan penggunaan memori. Dengan menggunakan memori untuk data yang aktif, dan memindahkan data yang tidak aktif ke *disk*, kita meningkatkan *throughput*. Bagaimana pun, proses *individual* dapat menderita sebagai hasilnya, sebab mereka sekarang dapat menyebabkan *page faults* tambahan selama eksekusi mereka.

Pertimbangkan sebuah proses atau *thread* waktu-nyata. Sebuah proses mengharapkan untuk memperoleh kendali CPU, dan untuk menjalankan penyelesaian dengan delay yang minimum. Memori virtual adalah saingan yang tepat untuk perhitungan waktu-nyata, sebab dapat menyebabkan delay jangka panjang, yang tidak diharapkan pada eksekusi sebuah proses saat halaman dibawa ke memori. Untuk itulah, sistem-sistem waktu-nyata hampir tidak memiliki memori virtual.

Pada kasus Solaris 2, para pengembang di Sun Microsystems ingin mengizinkan baik *time-sharing* dan perhitungan waktu nyata pada sebuah sistem. Untuk memecahkan masalah *page-fault*, mereka memiliki Solaris 2 mengizinkan sebuah proses untuk memberitahu bagian halaman mana yang penting untuk proses itu. Sebagai tambahan untuk mengizinkan petunjuk-petunjuk akan halaman yang digunakan, sistem operasi mengizinkan pemakai-pemakai yang berhak dapat mengunci halaman yang dibutuhkan di memori. Jika, disalah-gunakan, mekanisme ini dapat mengunci semua proses lain keluar dari sistem.

Adalah perlu untuk mengizinkan proses-proses waktu-nyata untuk dapat dibatasi *low-dispatch latency*

Bab 5

Sistem Berkas

5.1. Pengertian

Sistem berkas merupakan mekanisme penyimpanan *on-line* serta untuk akses, baik data mau pun program yang berada dalam Sistem Operasi. Terdapat dua bagian penting dalam sistem berkas, yaitu:

- kumpulan berkas, sebagai tempat penyimpanan data, serta
- struktur direktori, yang mengatur dan menyediakan informasi mengenai seluruh berkas dalam sistem.

Pada bab ini, akan dibahas tentang berbagai aspek dari berkas dan struktur, cara menangani proteksi berkas, cara mengalokasikan ruang pada disk, melacak lokasi data, serta meng-*interface* bagian-bagian lain dari sistem operasi ke penyimpanan sekunder.

5.2. Berkas

5.2.1. Konsep Dasar

Seperti yang telah kita ketahui, komputer dapat menyimpan informasi ke beberapa media penyimpanan yang berbeda, seperti *magnetic disks*, *magnetic tapes*, dan *optical disks*. Agar komputer dapat digunakan dengan nyaman, sistem operasi menyediakan sistem penyimpanan dengan sistematika yang seragam. Sistem Operasi mengabstraksi properti fisik dari media penyimpanannya dan mendefinisikan unit penyimpanan logis, yaitu berkas. Berkas dipetakan ke media fisik oleh sistem operasi. Media penyimpanan ini umumnya bersifat *non-volatile*, sehingga kandungan di dalamnya tidak akan hilang jika terjadi gagal listrik mau pun *system reboot*.

Berkas adalah kumpulan informasi berkait yang diberi nama dan direkam pada penyimpanan sekunder. Dari sudut pandang pengguna, berkas merupakan bagian terkecil dari penyimpanan logis, artinya data tidak dapat ditulis ke penyimpanan sekunder kecuali jika berada di dalam berkas. Biasanya berkas merepresentasikan program (baik *source* mau pun bentuk objek) dan data. Data dari berkas dapat bersifat numerik, alfabetik, alfanumerik, atau pun biner. Format berkas juga bisa bebas, misalnya berkas teks, atau dapat juga diformat pasti. Secara umum, berkas adalah urutan bit, byte, baris, atau catatan yang didefinisikan oleh pembuat berkas dan pengguna.

Informasi dalam berkas ditentukan oleh pembuatnya. Ada banyak beragam jenis informasi yang dapat disimpan dalam berkas. Hal ini disebabkan oleh struktur tertentu yang dimiliki oleh berkas, sesuai dengan jenisnya masing-masing. Contohnya:

- *Text file*; yaitu urutan karakter yang disusun ke dalam baris-baris.
- *Source file*; yaitu urutan *subroutine* dan fungsi, yang nantinya akan dideklarasikan.
- *Object file*; merupakan urutan byte yang diatur ke dalam blok-blok yang dikenali oleh *linker* dari sistem.
- *Executable file*; adalah rangkaian *code section* yang dapat dibawa loader ke dalam memori dandieksekusi.

5.2.2. Atribut Pada Berkas

Berkas diberi nama, untuk kenyamanan bagi pengguna, dan untuk acuan bagi data yang terkandung di dalamnya. Nama berkas biasanya berupa string atau karakter. Beberapa sistem membedakan penggunaan huruf besar dan kecil dalam penamaan sebuah berkas, sementara sistem yang lain menganggap kedua hal di atas sama.Ketika berkas diberi nama, maka berkas tersebut akan menjadi mandiri terhadap proses, pengguna, bahkan sistem yang membuatnya. Atribut berkas terdiri dari:

- *Nama*; merupakan satu-satunya informasi yang tetap dalam bentuk yang bisa dibaca oleh manusia (human-readable form)
- *Type*; dibutuhkan untuk sistem yang mendukung beberapa type berbeda
- *Lokasi*; merupakan pointer ke device dan ke lokasi berkas pada device tersebut
- *Ukuran (size)*; yaitu ukuran berkas pada saat itu, baik dalam byte, huruf, atau pun blok

- *Proteksi*; adalah informasi mengenai kontrol akses, misalnya siapa saja yang boleh membaca, menulis, dan mengeksekusi berkas
- *Waktu, tanggal dan identifikasi pengguna*; informasi ini biasanya disimpan untuk:
    1. pembuatan berkas,
    2. modifikasi terakhir yang dilakukan pada berkas, dan
    3. penggunaan terakhir berkas.

Data tersebut dapat berguna untuk proteksi, keamanan, dan monitoring penggunaan dari berkas.

Informasi tentang seluruh berkas disimpan dalam struktur direktori yang terdapat pada penyimpanan sekunder. Direktori, seperti berkas, harus bersifat *non-volatile*, sehingga keduanya harus disimpan pada sebuah *device* dan baru dibawa bagian per bagian ke memori pada saat dibutuhkan.

5.2.3. Operasi Pada Berkas

Sebuah berkas adalah jenis data abstrak. Untuk mendefinisikan berkas secara tepat, kita perlu melihat operasi yang dapat dilakukan pada berkas tersebut. Sistem operasi menyediakan *system calls* untuk membuat, membaca, menulis, mencari, menghapus, dan sebagainya. Berikut dapat kita lihat apa yang harus dilakukan sistem operasi pada keenam operasi dasar pada berkas.

- *Membuat sebuah berkas*: Ada dua cara dalam membuat berkas. Pertama, tempat baru di dalam sistem berkas harus di alokasikan untuk berkas yang akan dibuat. Kedua, sebuah direktori harus mempersiapkan tempat untuk berkas baru, kemudian direktori tersebut akan mencatat nama berkas dan lokasinya pada sistem berkas.
- *Menulis pada sebuah berkas*: Untuk menulis pada berkas, kita menggunakan *system call* beserta nama berkas yang akan ditulisi dan informasi apa yang akan ditulis pada berkas. Ketika diberi nama berkas, sistem mencari ke direktori untuk mendapatkan lokasi berkas. Sistem juga harus menyimpan penunjuk tulis pada berkas dimana penulisan berikut akan ditempatkan. Penunjuk tulis harus diperbaharui setiap terjadi penulisan pada berkas.
- *Membaca sebuah berkas*: Untuk dapat membaca berkas, kita menggunakan *system call* beserta nama berkas dan di blok memori mana berkas berikutnya diletakkan. Sama seperti

menulis, direktori mencari berkas yang akan dibaca, dan sistem menyimpan penunjuk baca pada berkas dimana pembacaan berikutnya akan terjadi. Ketika pembacaan dimulai, penunjuk baca harus diperbaharui.Sehingga secara umum, suatu berkas ketika sedang dibaca atau ditulis, kebanyakan sistem hanya mempunyai satu penunjuk, baca dan tulis menggunakan penunjuk yang sama, hal ini menghemat tempat dan mengurangi kompleksitas sistem.

- *Menempatkan kembali sebuah berkas*: Direktori yang bertugas untuk mencari berkas yang bersesuaian, dan mengembalikan lokasi berkas pada saat itu. Menempatkan berkas tidak perlu melibatkan proses I/O. Operasi sering disebut pencarian berkas.
- *Menghapus sebuah berkas*: Untuk menghapus berkas kita perlu mencari berkas tersebut di dalam direktori. Setelah ditemukan kita membebaskan tempat yang dipakai berkas tersebut (sehingga dapat digunakkan oleh berkas lain) dan menghapus tempatnya di direktori.
- *Memendekkan berkas*: Ada suatu keadaan dimana pengguna menginginkan atribut dari berkas tetap sama tetapi ingin menghapus isi dari berkas tersebut. Fungsi ini mengizinkan semua atribut tetap sama tetapi panjang berkas menjadi nol, hal ini lebih baik dari pada memaksa pengguna untuk menghapus berkas dan membuatnya lagi.

Enam operasi dasar ini sudah mencakup operasi minimum yang di butuhkan. Operasi umum lainnya adalah menyambung informasi baru di akhir suatu berkas, mengubah nama suatu berkas, dan lain-lain.

Operasi dasar ini kemudian digabung untuk melakukan operasi lainnya. Sebagai contoh misalnya kita menginginkan salinan dari suatu berkas, atau menyalin berkas ke peralatan I/O lainnya seperti *printer*, dengan cara membuat berkas lalu membaca dari berkas lama dan menulis ke berkas yang baru.

Hampir semua operasi pada berkas melibatkan pencarian berkas pada direktori. Untuk menghindari pencarian yang lama, kebanyakan sistem akan membuka berkas apabila berkas tersebut digunakan secara aktif. Sistem operasi akan menyimpan tabel kecil yang berisi informasi semua berkas yang dibuka yang disebut "tabel berkas terbuka". Ketika berkas sudah tidak digunakan lagi dan sudah ditutup oleh yang menggunakan, maka sistem operasi mengeluarkan berkas tersebut dari tabel berkas terbuka.

Beberapa sistem terkadang langsung membuka berkas ketika berkas tersebut digunakan dan otomatis menutup berkas tersebut jika program atau pemakainya dimatikan. Tetapi pada sistem lainnya terkadang membutuhkan pembukaan berkas secara tersurat dengan *system call* (*open*) sebelum berkas dapat digunakan.

Implementasi dari buka dan tutup berkas dalam lingkungan dengan banyak perngguna seperti UNIX, lebih rumit. Dalam sistem seperti itu pengguna yang membuka berkas mungkin lebih dari satu dan pada waktu yang hampir bersamaan. Umumnya sistem operasi menggunakan tabel internal dua level. Ada tabel yang mendata proses mana saja yang membuka berkas tersebut, kemudian tabel tersebut menunjuk ke tabel yang lebih besar yang berisi informasi yang berdiri sendiri seperti lokasi berkas pada disk, tanggal akses dan ukuran berkas. Biasanya tabel tersebut juga memiliki data berapa banyak proses yang

membuka berkas tersebut.

Jadi, pada dasarnya ada beberapa informasi yang terkait dengan pembukaan berkas yaitu:

- *Penunjuk Berkas*: Pada sistem yang tidak mengikutkan batas berkas sebagai bagian dari system call baca dan tulis, sistem tersebut harus mengikuti posisi dimana terakhir proses baca dan tulis sebagai penunjuk. Penunjuk ini unik untuk setiap operasi pada berkas, maka dari itu harus disimpan terpisah dari atribut berkas yang ada pada disk.
- *Penghitung berkas yang terbuka*: Setelah berkas ditutup, sistem harus mengosongkan kembali tabel berkas yang dibuka yang digunakan oleh berkas tadi atau tempat di tabel akan habis. Karena mungkin ada beberapa proses yang membuka berkas secara bersamaan dan sistem harus menunggu sampai berkas tersebut ditutup sebelum mengosongkan tempatnya di tabel. Penghitung ini mencatat banyaknya berkas yang telah dibuka dan ditutup, dan menjadi nol ketika yang terakhir membaca berkas menutup berkas tersebut barulah sistem dapat mengosongkan tempatnya di tabel.
- *Lokasi berkas pada disk*: Kebanyakan operasi pada berkas memerlukan sistem untuk mengubah data yang ada pada berkas. Informasi mengenai lokasi berkas pada disk disimpan di memori agar menghindari banyak pembacaan pada disk untuk setiap operasi.

Beberapa sistem operasi menyediakan fasilitas untuk memetakan berkas ke dalam memori pada sistem memori virtual. Hal tersebut

mengizinkan bagian dari berkas ditempatkan pada suatu alamat di memori virtual. Operasi baca dan tulis pada memori dengan alamat tersebut dianggap sebagai operasi baca dan tulis pada berkas yang ada di alamat tersebut. Menutup berkas mengakibatkan semua data yang ada pada alamat memori tersebut dikembalikan ke disk dan dihilangkan dari memori virtual yang digunakan oleh proses.

5.2.4. Jenis Berkas

Pertimbangan utama dalam perancangan sistem berkas dan seluruh sistem operasi, apakah sistem operasi harus mengenali dan mendukung jenis berkas. Jika suatu sistem operasi mengenali jenis dari berkas, maka ia dapat mengoperasikan berkas tersebut. Contoh apabila pengguna mencoba mencetak berkas yang merupakan kode biner dari program yang pasti akan menghasilkan sampah, hal ini dapat dicegah apabila sistem operasi sudah diberitahu bahwa berkas tersebut merupakan kode biner.

Teknik yang umum digunakan dalam implementasi jenis berkas adalah menambahkan jenis berkas dalam nama berkas. Nama dibagi dua, nama dan akhiran (ekstensi), biasanya dipisahkan dengan karakter titik. Sistem menggunakan akhiran tersebut untuk mengindikasikan jenis berkas dan jenis operasi yang dapat dilakukan pada berkas tersebut. Sebagai contoh hanya berkas yang berakhiran *.bat*, *.exe* atau *.com* yang bisa dijalankan (eksekusi). Program aplikasi juga menggunakan akhiran tersebut untuk mengenal berkas yang dapat dioperasikannya. Akhiran ini dapat ditimpa atau diganti jika diperbolehkan oleh sistem operasi.

Beberapa sistem operasi menyertakan dukungan terhadap akhiran, tetapi beberapa menyerahkan kepada aplikasi untuk mengatur akhiran berkas yang digunakan, sehingga jenis dari berkas dapat menjadi petunjuk aplikasi apa yang dapat mengoperasikannya.

Sistem UNIX tidak dapat menyediakan dukungan untuk akhiran berkas karena menggunakan angka ajaib yang disimpan di depan berkas untuk mengenali jenis berkas. Tidak semua berkas memiliki angka ini, jadi sistem tidak bisa bergantung pada informasi ini. Tetapi UNIX memperbolehkan akhiran berkas

tetapi hal ini tidak dipaksakan atau tergantung sistem operasi, kebanyakan hanya untuk membantu pengguna mengenali jenis isi dari suatu berkas.

Tabel 5-1. Tabel Jenis Berkas

Jenis berkas	Akhiran	Fungsi
executable	exe, com, bat, bin	program yang siap dijalankan
objek	obj, o	bahasa mesin, kode terkompilasi
kode asal (source code)	c, cc, pas, java, asm, a	kode asal dari berbagai bahasa
batch	bat, sh	perintah pada shell
text	txt, doc	data text, document
pengolah kata	wpd, tex, doc	format jenis pengolah data
library	lib, a, DLL	library untuk rutin program
print, gambar	ps, dvi, gif	format aSCII atau biner untuk dicetak
archive	arc, zip, tar	beberapa berkas yang dikumpulkan menjadi satu berkas. Terkadang dimampatkan untuk penyimpanan

5.2.5. Struktur Berkas

Kita juga dapat menggunakan jenis berkas untuk mengidentifikasi struktur dalam dari berkas. Berkas berupa source dan objek memiliki struktur yang cocok dengan harapan program yang membaca berkas tersebut. Suatu berkas harus memiliki struktur yang dikenali oleh sistem operasi. Sebagai contoh, sistem operasi menginginkan suatu berkas yang dapat dieksekusi memiliki struktur tertentu agar dapat diketahui dimana berkas tersebut akan ditempatkan di memori dan di mana letak instruksi pertama berkas tersebut.

Beberapa sistem operasi mengembangkan ide ini sehingga mendukung beberapa struktur berkas, dengan beberapa operasi khusus untuk memanipulasi berkas dengan struktur tersebut.

Kelemahan memiliki dukungan terhadap beberapa struktur berkas adalah: Ukuran dari sistem operasi dapat menjadi besar, jika sistem operasi mendefinisikan lima struktur berkas yang berbeda maka ia perlu menampung kode untuk yang diperlukan untuk mendukung semuanya. Setiap berkas harus dapat

menerapkan salah satu struktur berkas tersebut. Masalah akan timbul ketika terdapat aplikasi yang membutuhkan struktur informasi yang tidak didukung oleh sistem operasi tersebut.

Beberapa sistem operasi menerapkan dan mendukung struktur berkas sedikit struktur berkas. Pendekatan ini digunakan pada MS-DOS dan UNIX. UNIX menganggap setiap berkas sebagai urutan 8-bit byte, tidak ada interpretasi sistem operasi terhadap dari bit-bit ini. Skema tersebut menawarkan fleksibilitas tinggi tetapi dukungan yang terbatas. Setiap aplikasi harus menambahkan sendiri kode untuk menerjemahkan berkas masukan ke dalam struktur yang sesuai. Walau bagaimana pun juga sebuah sistem operasi harus memiliki minimal satu struktur berkas yaitu untuk berkas yang dapat dieksekusi sehingga sistem dapat memuat berkas dalam memori dan menjalankannya.

Sangat berguna bagi sistem operasi untuk mendukung struktur berkas yang sering digunakan karena akan menghemat pekerjaan pemrogram. Terlalu sedikit struktur berkas yang didukung akan mempersulit pembuatan program, terlalu banyak akan membuat sistem operasi terlalu besar dan pemrogram akan bingung.

5.2.6. Struktur Berkas Pada Disk

Menempatkan batas dalam berkas dapat menjadi rumit bagi sistem operasi. Sistem disk biasanya memiliki ukuran blok yang sudah ditetapkan dari ukuran sektor. Semua I/O dari disk dilakukan dalam satuan blok dan semua blok ('*physical record*') memiliki ukuran yang sama. Tetapi ukuran dari '*physical record*' tidak akan sama dengan ukuran '*logical record*'. Ukuran dari '*logical record*' akan bervariasi.Memuatkan beberapa '*logical record*' ke dalam '*physical record*' merupakan solusi umum dari masalah ini.

Sebagai contoh pada sistem operasi UNIX, semua berkas didefinisikan sebagai kumpulan byte. Setiap byte dialamatkan menurut batasnya dari awal berkas sampai akhir. Pada kasus ini ukuran '*logical record*' adalah 1 byte. Sistem berkas secara otomatis memuatkan byte-byte tersebut kedalam blok pada disk.

Ukuran '*logical record*', ukuran blok pada disk, dan teknik untuk memuatkannya menjelaskan berapa banyak '*logical record*' dalam tiap-tiap 'physical record'. Teknik memuatkan dapat dilakukan oleh aplikasi pengguna atau oleh sistem operasi.

Berkas juga dapat dianggap sebagai urutan dari beberapa blok pada disk. Konversi dari '*logical record*' ke 'physical record' merupakan masalah perangkat lunak. Tempat pada disk selalu berada pada blok, sehingga beberapa bagian dari blok terakhir yang ditempati berkas dapat terbuang. Jika setiap blok berukuran

512 byte, sebuah berkas berukuran 1.949 byte akan menempati empat blok (2.048 byte) dan akan tersisa 99 byte pada blok terakhir. Byte yang terbuang tersebut dipertahankan agar ukuran dari unit tetap blok bukan byte disebut fragmentasi dalam disk.

Semua sistem berkas pasti mempunyai fragmentasi dalam disk, semakin besar ukuran blok akan semakin besar fragmentasi dalam disknya.

5.2.7. Penggunaan Berkas Secara Bersama-sama

Konsistensi semantik adalah parameter yang penting untuk evaluasi sistem berkas yang mendukung penggunaan berkas secara bersama. Hal ini juga merupakan karakterisasi dari sistem yang menspesifikasi semantik dari banyak pengguna yang mengakses berkas secara bersama-sama. Lebih khusus, semantik ini seharusnya dapat menspesifikasi kapan

suatu modifikasi suatu data oleh satu pengguna dapat diketahui oleh pengguna lain.

Terdapat beberapa macam konsistensi semantik. Di bawah ini akan dijelaskan kriteria yang digunakan dalam UNIX.

Berkas sistem UNIX mengikuti konsistensi semantik:

- Penulisan ke berkas yang dibuka oleh pengguna dapat dilihat langsung oleh pengguna lain yang sedang mengakses ke berkas yang sama.
- Terdapat bentuk pembagian dimana pengguna membagi pointer lokasi ke berkas tersebut. Sehingga perubahan pointer satu pengguna akan mempengaruhi semua pengguna *sharing*nya.

5.3. Metode Akses

5.3.1. Akses Secara Berurutan

Ketika digunakan, informasi penyimpanan berkas harus dapat diakses dan dibaca ke dalam memori komputer. Beberapa sistem hanya menyediakan satu metode akses untuk berkas.

Pada sistem yang lain, contohnya IBM, terdapat banyak dukungan metode akses yang berbeda. Masalah pada sistem tersebut adalah memilih yang mana yang tepat untuk digunakan pada satu aplikasi tertentu.

Sequential Access merupakan metode yang paling sederhana. Informasi yang disimpan dalam berkas diproses berdasarkan urutan. Operasi dasar pada suatu berkas adalah tulis dan baca. Operasi baca membaca berkas dan meningkatkan pointer berkas

selama di jalur lokasi I/O. Operasi tulis menambahkan ke akhir berkas dan meningkatkan ke akhir berkas yang baru. Metode ini didasarkan pada tape model sebuah berkas, dan dapat bekerja pada kedua jenis *device* akses (urut mau pun acak).

5.3.2. Akses Langsung

Direct Access merupakan metode yang membiarkan program membaca dan menulis dengan cepat pada berkas yang dibuat dengan *fixed-length logical order* tanpa adanya urutan. Metode ini sangat berguna untuk mengakses informasi dalam jumlah besar. Biasanya database memerlukan hal seperti ini. Operasi berkas pada metode ini harus dimodifikasi untuk menambahkan nomor blok sebagai parameter.

Pengguna menyediakan nomor blok ke sistem operasi biasanya sebagai nomor blok relatif, yaitu indeks relatif terhadap awal berkas. Penggunaan nomor blok relatif bagi sistem operasi adalah untuk memutuskan lokasi berkas diletakkan dan membantu mencegah pengguna dari pengaksesan suatu bagian sistem berkas yang bukan bagian pengguna tersebut.

5.3.3. Akses Dengan Menggunakan Indeks

Metode ini merupakan hasil dari pengembangan metode *direct access*. Metode ini memasukkan indeks untuk mengakses berkas. Jadi untuk mendapatkan suatu informasi suatu berkas, kita mencari dahulu di indeks, lalu menggunakan pointer untuk mengakses berkas dan mendapatkan informasi tersebut. Namun metode ini memiliki kekurangan, yaitu apabila berkas-berkas besar, maka indeks berkas tersebut akan semakin besar. Jadi solusinya adalah dengan membuat 2 indeks, indeks primer dan indeks sekunder.

Indeks primer memuat pointer ke indeks sekunder, lalu indeks sekunder menunjuk ke data yang dimaksud.

5.4. Struktur Direktori

5.4.1. Operasi Pada Direktori

Operasi-operasi yang dapat dilakukan pada direktori adalah:

1. Mencari berkas, kita dapat menemukan sebuah berkas didalam sebuah struktur direktori. Karena berkas-berkas memiliki nama simbolik dan nama yang sama dapat mengindikasikan keterkaitan antara setiap berkas-berkas tersebut, mungkin kita berkeinginan untuk dapat menemukan seluruh berkas yang nama-nama berkas membentuk pola khusus.

2. Membuat berkas, kita dapat membuat berkas baru dan menambahkan berkas tersebut kedalam direktori.
3. Menghapus berkas, apabila berkas sudah tidak diperlukan lagi, kita dapat menghapus berkas tersebut dari direktori.
4. Menampilkan isi direktori, kita dapat menampilkan seluruh berkas dalam direktori, dan kandungan isi direktori untuk setiap berkas dalam daftar tersebut.
5. Mengganti nama berkas, karena nama berkas merepresentasikan isi dari berkas kepada user, maka user dapat merubah nama berkas ketika isi atau penggunaan berkas berubah. Perubahan nama dapat merubah posisi berkas dalam direktori.
6. Melintasi sistem berkas, ini sangat berguna untuk mengakses direktori dan berkas didalam struktur direktori.

5.4.2. Direktori Satu Tingkat

Ini adalah struktur direktori yang paling sederhana. Semua berkas disimpan di dalam direktori yang sama. Struktur ini tentunya memiliki kelemahan jika jumlah berkasnya bertambah banyak, karena tiap berkas mesti memiliki nama yang unik.

5.4.3. Direktori Dua Tingkat

Kelemahan yang ada pada direktori tingkat satu dapat diatas pada sistem direktori dua tingkat. Caranya ialah dengan membuat direktori secara terpisah. Pada direktori tingkat dua, setiap pengguna memiliki direktori berkas sendiri (UFD). Setiap UFD memiliki struktur yang serupa, tapi hanya berisi berkas-berkas dari seorang pengguna.

Ketika seorang pengguna *login*, *master* direktori berkas (MFD) dicari. Isi dari MFD adalah indeks dari nama pengguna atau nomor rekening, dan tiap entri menunjuk pada UFD untuk pengguna tersebut.

Ketika seorang pengguna ingin mengakses suatu berkas, hanya UFD-nya sendiri yang diakses. Jadi pada setiap UFD yang berbeda, boleh terdapat nama berkas yang sama.

5.4.4. Direktori Dengan Struktur *Tree*

Struktur direktori dua tingkat bisa dikatakan sebagai pohon dua tingkat. Sebuah direktori dengan struktur pohon memiliki sejumlah berkas atau subdirektori lagi. Pada penggunaan yang normal setiap pengguna memiliki direktorinya sendiri-sendiri.

Selain itu pengguna tersebut dapat memiliki subdirektori sendiri lagi.

Dalam struktur ini dikenal dua istilah, yaitu *path* relatif dan *path* mutlak. *Path* relatif adalah *path* yang dimulai dari direktori yang aktif. Sedangkan *path* mutlak adalah *path* yang dimulai dari direktori akar.

5.4.5. Direktori Dengan Struktur *Acyclic-Graph*

Jika ada sebuah berkas yang ingin diakses oleh dua pengguna atau lebih, maka struktur ini menyediakan fasilitas "*sharing*", yaitu penggunaan sebuah berkas secara bersama-sama. Hal ini tentunya berbeda dengan struktur pohon, dimana pada struktur tersebut penggunaan berkas atau direktori secara bersama-sama dilarang. Pada struktur "*Acyclic-Graph*", penggunaan berkas atau direktori secara bersama-sama diperbolehkan. Tapi pada umumnya struktur ini mirip dengan struktur pohon.

5.4.6. Direktori Dengan Struktur *Graph*

Masalah yang sangat utama pada struktur direktori "*Acyclic-Graph*" adalah kemampuan untuk memastikan tidak-adanya siklus. Jika pada struktur 2 tingkat direktori, seorang pengguna dapat membuat subdirektori, maka akan kita dapatkan direktori dengan struktur pohon. Sangatlah mudah untuk tetap mempertahankan sifat pohon setiap kali ada penambahan berkas atau subdirektori pada direktori dengan struktur pohon. Tapi jika kita menambahkan sambungan pada direktori dengan struktur pohon, maka akan kita dapatkan direktori dengan struktur *graph* sederhana.

Proses pencarian pada direktori dengan struktur "*Acyclic-Graph*", apabila tidak ditangani dengan baik (algoritma tidak bagus) dapat menyebabkan proses pencarian yang berulang dan menghabiskan banyak waktu. Oleh karena itu, diperlukan skema pengumpulan sampah ("*garbage-collection scheme*"). Skema ini menyangkut memeriksa seluruh sistem berkas dengan menandai tiap berkas yang dapat diakses.Kemudian mengumpulkan apa pun yang tidak ditandai sebagai tempat kosong. Hal ini tentunya dapat menghabiskan banyak waktu.

5.5. Proteksi Berkas

Ketika kita menyimpan informasi dalam sebuah sistem komputer, ada dua hal yang harus menjadi perhatian utama kita. Hal tersebut adalah:

1. Reabilitas dari sebuah sistem

Maksud dari reabilitas sistem adalah kemampuan sebuah sistem untuk melindungi informasi yangtelah disimpan agar terhindar dari kerusakan, dalam hal ini adalah perlindungan secara fisik pada sebuah berkas. Reabilitas sistem dapat dijaga dengan membuat cadangan dari setiap berkas secaramanual atau pun otomatis, sesuai dengan layanan yang dari sebuah sistem operasi. Reabilitas Sistemakan dibahas lebih lanjut pada Bagian 6.10.

2. Proteksi (Perlindungan) terhadap sebuah berkas

   Perlindungan terhadap berkas dapat dilakukan dengan berbagai macam cara. Pada bagian ini, kita akan membahas secara detil mekanisme yang diterapkan dalam melindungi sebuah berkas.

5.5.1. Tipe Akses Pada Berkas

Salah satu cara untuk melindungi berkas dalam komputer kita adalah dengan melakukan pembatasan akses pada berkas tersebut. Pembatasan akses yang dimaksudkan adalah kita, sebagai pemilik dari sebuah berkas, dapat menentukan operasi apa saja yang dapat dilakukan oleh pengguna lain terhadap berkas tersebut. Pembatasan ini berupa sebuah permission atau pun not permitted operation, tergantung pada kebutuhan pengguna lain terhadap berkas tersebut. Di bawah ini adalah beberapa operasi berkas yang dapat diatur aksesnya:

1. Read: Membaca dari berkas
2. Write: Menulis berkas
3. Execute: Meload berkas kedalam memori untuk dieksekusi.
4. Append: Menambahkan informasi kedalam berkas di akhir berkas.
5. Delete: Menghapus berkas.
6. List: Mendaftar properti dari sebuah berkas.
7. Rename: Mengganti nama sebuah berkas.
8. Copy: Menduplikasikan sebuah berkas.
9. Edit: Mengedit sebuah berkas.

Selain operasi-operasi berkas diatas, perlindungan terhadap berkas dapat dilakukan dengan mekanisme yang lain. Namun setiap mekanisme memiliki kelebihan dan kekurangan. Pemilihan mekanisme sangatlah tergantung pada kebutuhan dan spesifikasi sistem.

5.5.2. Akses List dan Group

Hal yang paling umum dari sistem proteksi adalah membuat akses tergantung pada identitas pengguna yang bersangkutan. Implementasi dari akses ini adalah dengan membuat daftar akses yang berisi keterangan setiap pengguna dan keterangan akses berkas dari pengguna yang bersangkutan. Daftar akses ini akan diperiksa setiap kali seorang pengguna meminta akses ke sebuah berkas. Jika pengguna tersebut memiliki akses yang diminta pada berkas tersebut, maka diperbolehkan untuk mengakses berkas tersebut. Proses ini juga berlaku untuk hal yang sebaliknya. Akses pengguna terhadap berkas akan ditolak, dan sistem operasi akan mengeluarkan peringatan *Protection Violation*.

Masalah baru yang timbul adalah panjang dari daftar akses yang harus dibuat. Seperti telah disebutkan, kita harus mendaftarkan semua pengguna dalam daftar akses tersebut hanya untuk akses pada satu berkas saja. Oleh karena itu, teknik ini mengakibatkan 2 konsekuensi yang tidak dapat dihindarkan:

1. Pembuatan daftar yang sangat panjang ini dapat menjadi pekerjaan yang sangat melelahkan sekaligus membosankan, terutama jika jumlah pengguna dalam sistem tidak dapat diketahui secara pasti.
2. Manajemen ruang *harddisk* yang lebih rumit, karena ukuran sebuah direktori dapat berubah-ubah, tidak memiliki ukuran yang tetap.

Kedua konsekuensi diatas melahirkan sebuah teknik daftar akses yang lebih singkat. Teknik inimengelompokkan pengguna berdasarkan tiga kategori:

1. Owner: User yang membuat berkas.
2. Group: Sekelompok pengguna yang memiliki akses yang sama terhadap sebuah berkas, atau men-share sebuah berkas.
3. Universe: Seluruh pengguna yang terdapat dalam sistem komputer.

Dengan adanya pengelompokkan pengguna seperti ini, maka kita hanya membutuhkan tiga *field* untukmelindungi sebuah berkas. Field ini diasosiasikan dengan 3 buah bit untuk setiap kategori. Dalam sistem UNIX dikenal bit rwx dengan bit r untuk mengontrol akses baca, bit w sebagai kontrol menulis dan bit x sebagai bit kontrol untuk pengeksekusian. Setiap *field* dipisahkan dengan *field separator*. Dibawah ini adalah contoh dari sistem proteksi dengan daftar akses pada sistem UNIX.

Tabel 5-2. Contoh sistem daftar akses pada UNIX

drwx	rwx	rwx	1	pbg	staff	512	Apr16 22.25	bekas.txt
owner	group	universe		group	owner	ukuran	waktu	Nama berkas

5.5.3. Pendekatan Sistem Proteksi yang Lain

Sistem proteksi yang lazim digunakan pada sistem komputer selain diatas adalah dengan menggunakan password (kata sandi) pada setiap berkas. Beberapa sistem operasi mengimplementasikan hal ini bukan hanya pada berkas, melainkan pada direktori. Dengan sistem ini, sebuah berkas tidak akan dapat diakses selain oleh pengguna yang telah mengetahui password untuk berkas tersebut. Akan tetapi, masalah yang muncul dari sistem ini adalah jumlah password yang harus diingat oleh seorang pengguna untuk mengakses berkas dalam sebuah sistem operasi. Masalah yang lain adalah keamanan password itu sendiri. Jika hanya satu password yang digunakan, maka kebocoran password tersebut merupakan malapetaka bagi pengguna yang bersangkutan. Sekali lagi, maka kita harus menggunakan password yang berbeda untuk setiap tingkatan yang berbeda.

5.6. Struktur Sistem Berkas

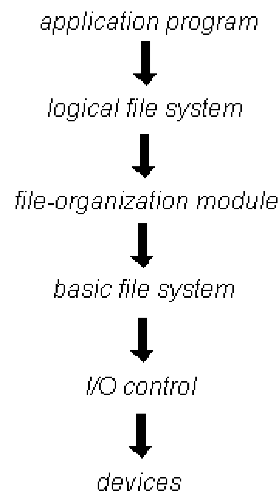
Disk menyediakan sebagian besar tempat penyimpanan dimana sistem berkas dikelola dikelola. Untuk meningkatkan efisiensi I/O, pengiriman data antara memori dan disk dilakukan dalam setiap blok. Setiap blok merupakan satu atau lebih sektor. Setiap disk memiliki ukuran yang berbeda-beda, biasanya berukuran 512 bytes. Disk memiliki dua karakteristik penting yang menjadikan disk sebagai media yang tepat untuk menyimpan berbagai macam berkas, yaitu:

- Disk tersebut dapat ditulis ulang di disk tersebut, hal ini memungkinkan untuk membaca, memodifikasi, dan menulis di disk tersebut.
- Dapat diakses langsung ke setiap blok di disk. Hal ini memudahkan untuk mengakses setiap berkas baik secara berurut mau pun tidak berurut, dan berpindah dari satu berkas ke berkas lain dengan hanya mengangkat head disk dan menunggu disk berputar.

5.6.1. Organisasi Sistem Berkas

Sistem operasi menyediakan sistem berkas agar data mudah disimpan, diletakkan dan diambil kembali dengan mudah. Terdapat dua masalah desain dalam membangun suatu sistem berkas. Masalah pertama adalah definisi dari sistem berkas. Hal ini mencakup definisi berkas dan atributnya, operasi ke berkas, dan struktur direktori dalam mengorganisasikan berkas-berkas. Masalah kedua adalah membuat algoritma dan struktur data yang memetakan struktur logikal sistem berkas ke tempat penyimpanan sekunder. Pada dasarnya sistem berkas tersusun atas beberapa tingkatan, yaitu (dari yang terendah):

- *I/O control*, terdiri atas *driver device* dan *interrupt handler*. *Driver device* adalah perantarakomunikasi antara sistem operasi dengan perangkat keras.
- *Basic file system*, diperlukan untuk mengeluarkan perintah generik ke *device driver* untuk baca dan tulis pada suatu blok dalam disk.
- *File-organization module*, informasi tentang alamat logika dan alamat fisik dari berkas tersebut. Modul ini juga mengatur sisa disk dengan melacak alamat yang belum dialokasikan dan menyediakan alamat tersebut saat user ingin menulis berkas ke dalam disk.
- • *Logical file system*, tingkat ini berisi informasi tentang simbol nama berkas, struktur dari direktori, dan proteksi dan sekuriti dari berkas tersebut.



Gambar 5-1. Lapisan pada sistem berkas.

5.6.2. *Mounting* Sistem Berkas

Seperti halnya sebuah berkas yang harus dibuka terlebih dahulu sebelum digunakan, sistem berkas harus di *mount* terlebih dahulu sebelum sistem berkas tersebut siap untuk memproses dalam sistem. Sistem operasi diberikan sebuah alamat mounting (*mount point*) yang berisi nama *device* yang bersangkutan dan lokasi dari *device* tersebut.

5.7. Metode Alokasi Berkas

Kemudahan dalam mengakses langsung suatu disk memberikan fleksibilitas dalam

mengimplementasikan sebuah berkas. Masalah utama dalam implementasi adalah bagaimana mengalokasikan berkas-berkas ke dalam disk, sehingga disk dapat terutilisasi dengan efektif dan berkas dapat diakses dengan cepat. Ada tiga metode utama, menurut buku "*Applied Operating System Concepts: First Edition*" oleh Avi Silberschatz, Peter Galvin dan Greg Gagne untuk mengalokasi ruang disk yang digunakan secara luas yaitu, *contiguous*, *linked*, dan *indexed*.

5.7.1. Alokasi Secara Berdampingan *(Contiguous Allocation)*

Metode ini menempatkan setiap berkas pada satu himpunan blok yang berurut di dalam disk. Alamat disk menyatakan sebuah urutan linier. Dengan urutan linier ini maka *head disk* hanya bergerak jika mengakses dari sektor terakhir suatu silinder ke sektor pertama silinder berikutnya. Waktu pencarian (*seek time*) dan banyak *disk seek* yang dibutuhkan untuk mengakses berkas yang di alokasi secara berdampingan ini sangat minimal. Contoh dari sistem operasi yang menggunakan *contiguous allocation* adalah IBM VM/ CMS karena pendekatan ini menghasilkan performa yang baik.

Contiguous allocation dari suatu berkas diketahui melalui alamat dan panjang disk (dalam unit blok) dari blok pertama. Jadi, misalkan ada berkas dengan panjang n blok dan mulai dari lokasi b maka berkas tersebut menempati blok b, b+1, b+2, ..., b+n-1. Direktori untuk setiap berkas mengindikasikan alamat blok awal dan panjang area yang dialokasikan untuk berkas tersebut. Terdapat dua macam cara untuk mengakses berkas yang dialokasi dengan metode ini, yaitu:

- *Sequential access*, sistem berkas mengetahui alamat blok terakhir dari disk dan membaca blok berikutnya jika diperlukan.
- *Direct access*, untuk akses langsung ke blok i dari suatu berkas yang dimulai pada blok b, dapat langsung mengakses blok b+i.

Kesulitan dari metode alokasi secara berdampingan ini adalah menemukan ruang untuk berkas baru. Masalah pengalokasian ruang disk dengan metode ini merupakan aplikasi masalah dari *dynamic storage-allocation* (alokasi tempat penyimpanan secara dinamik), yaitu bagaimana memenuhi permintaan ukuran n dari daftar ruang kosong. Strategi-strategi yang umum adalah *first fit* dan *best fit*. Kedua strategi tersebut mengalami masalah fragmentasi eksternal, dimana jika berkas dialokasi dan dihapus maka ruang kosong disk terpecah menjadi kepingan-kepingan kecil. Hal ini akan menjadi masalah ketika banyak kepingan kecil tidak dapat memenuhi permintaan karena kepingan-kepingan kecil tidak cukup besar untuk menyimpan berkas, sehingga terdapat banyak ruang yang terbuang.

Masalah yang lain adalah menentukan berapa banyak ruang yang diperlukan untuk suatu berkas. Ketika berkas dibuat, jumlah dari ruang berkas harus ditentukan dan dialokasikan. Jika ruang yang dialokasikan terlalu kecil maka berkas tidak dapat diperbesar dari yang telah dialokasikan. Untuk mengatasi hal ini ada dua kemungkinan. Pertama, program pengguna dapat diakhiri dengan pesan error yang sesuai. Lalu, pengguna harus mengalokasikan tambahan ruang dan menjalankan programnya lagi, tetapi hal ini *cost* yang dihasilkan lebih mahal. Untuk mengatasinya, pengguna dapat melakukan estimasi yang lebih terhadap ruang yang harus dialokasikan pada suatu berkas tetapi hal ini akan membuang ruang disk. Kemungkinan yang kedua adalah mencari ruang kosong yang lebih besar, lalu menyalin isi dari berkas ke ruang yang baru dan mengkosongkan ruang yang sebelumnya. Hal ini menghabiskan waktu yang cukup banyak. Walau pun jumlah ruang yang diperlukan untuk suatu berkas dapat diketahui, pengalokasian awal akan tidak efisien. Ukuran berkas yang bertambah dalam periode yang lama harus dapat dialokasi ke ruang yang cukup untuk ukuran akhirnya, walau pun ruang tersebut tidak akan digunakan dalam waktu yang lama. Hal ini akan menyebabkan berkas dengan jumlah fragmentasi internal yang besar.

Untuk menghindari hal-hal tersebut, beberapa sistem operasi memodifikasi skema metode alokasi secara berdampingan, dimana kepingan kecil yang berurut dalam ruang disk diinisialisasi terlebih dahulu, kemudian ketika jumlah ruang disk kurang besar, kepingan kecil yang berurut lainnya, ditambahkan pada alokasi awal. Kejadian seperti ini disebut perpanjangan. Fragmentasi internal masih dapat terjadi jika perpanjangan-perpanjangan ini terlalu besar dan fragmentasi eksternal masih menjadi masalah begitu perpanjangan-perpanjangan dengan ukuran yang bervariasi dialokasikan dan didealokasi.

5.7.2. Alokasi Secara Berangkai (*Linked Allocation*)

Metode ini menyelesaikan semua masalah yang terdapat pada contiguous allocation. Dengan metode ini, setiap berkas merupakan *linked list* dari blok-blok disk, dimana blok-blok disk dapat tersebar di dalam disk. Setiap direktori berisi sebuah penunjuk (*pointer*) ke awal dan akhir blok sebuah berkas. Setiap blok mempunyai penunjuk ke blok berikutnya. Untuk membuat berkas baru, kita dengan mudah membuat masukan baru dalam direktori. Dengan metode ini, setiap direktori masukan mempunyai penunjuk ke awal blok disk dari berkas. Penunjuk ini diinisialisasi menjadi *nil* (nilai penunjuk untuk akhir dari list) untuk menandakan berkas kosong. Ukurannya juga diset menjadi 0. Penulisan suatu berkas menyebabkan ditemukannya blok yang kosong melalui sistem manajemen ruang kosong (*free-space management system*), dan blok baru ini ditulis dan disambungkan ke akhir berkas. Untuk membaca suatu berkas, cukup dengan membaca blok-blok dengan mengikuti pergerakan penunjuk.

Metode ini tidak mengalami fragmentasi eksternal dan kita dapat menggunakan blok kosong yang terdapat dalam daftar ruang kosong untuk memenuhi permintaan pengguna. Ukuran dari berkas tidak perlu ditentukan ketika berkas pertama kali dibuat, sehingga ukuran berkas dapat bertambah selama masih ada blok-blok kosong.

Metode ini tentunya mempunyai kerugian, yaitu metode ini hanya dapat digunakan secara efektif untuk pengaksesan berkas secara sequential (*sequential-access file*). Untuk mencari blok ke-*i* dari suatu berkas, harus dimulai dari awal berkas dan mengikuti penunjuk sampai berada di blok ke-*i*. Setiap akses ke penunjuk akan membaca disk dan kadang melakukan pencarian disk (*disk seek*). Hal ini sangat tidak efisien untuk mendukung kemampuan akses langsung (*direct-access*) terhadap berkas yang menggunakan metode alokasi link. Kerugian yang lain dari metode ini adalah ruang yang harus disediakan untuk penunjuk. Solusi yang umum untuk masalah ini adalah mengumpulkan blok-blok persekutuan terkecil dinamakan *clusters* dan mengalokasikan *cluster-cluster* daripada blok. Dengan solusi ini maka, penunjuk menggunakan ruang disk berkas dengan persentase yang sangat kecil. Metode ini membuat *mapping* logikal ke fisikal blok tetap sederhana, tetapi meningkatkan *disk throughput* dan memperkecil ruang yang diperlukan untuk alokasi blok dan management daftar kosong (*free-list management*). Akibat dari pendekatan ini adalah meningkatnya fragmentasi internal, karena lebih banyak ruang yang terbuang jika sebuah *cluster* sebagian penuh daripada ketika sebuah blok sebagian penuh. Alasan *cluster* digunakan oleh

kebanyakan sistem operasi adalah kemampuannya yang dapat meningkatkan waktu akses disk untuk berbagai macam algoritma.

Masalah yang lain adalah masalah daya tahan metode ini. Karena semua berkas saling berhubungan dengan penunjuk yang tersebar di semua bagian disk, apa yang terjadi jika sebuah penunjuk rusak atau hilang. Hal ini menyebabkan berkas menyambung ke daftar ruang kosong atau ke berkas yang lain. Salah satu solusinya adalah menggunakan *linked list* ganda atau menyimpan nama berkas dan nomor relatif blok dalam setiap blok, tetapi solusi ini membutuhkan perhatian lebih untuk setiap berkas.

Variasi penting dari metode ini adalah penggunaan *file allocation table (FAT)*, yang digunakan oleh sistem operasi *MS-DOS* dan *OS/2*. Bagian awal disk pada setiap partisi disingkirkan untuk menempatkan tabelnya. Tabel ini mempunyai satu masukkan untuk setiap blok disk, dan diberi indeks oleh nomor blok. Masukkan direktori mengandung nomor blok dari blok awal berkas. Masukkan tabel diberi indeks oleh nomor blok itu lalu mengandung nomor blok untuk blok berikutnya dari berkas. Rantai ini berlanjut sampai blok terakhir, yang mempunyai nilai akhir berkas yang khusus sebagai masukkan tabel. Blok yang tidak digunakan diberi nilai 0. Untuk mengalokasi blok baru untuk suatu berkas hanya dengan mencari nilai 0 pertama dalam tabel, dan mengganti nilai akhir berkas sebelumnya dengan alamat blok yang baru. Metode pengalokasian FAT ini dapat menghasilkan jumlah pencarian *head disk* yang signifikan, jika berkas tidak di cache. *Head disk* harus bergerak dari awal partisi untuk membaca FAT dan menemukan lokasi blok yang ditanyakan, lalu menemukan lokasi blok itu sendiri. Kasus buruknya, kedua pergerakan terjadi untuk setiap blok. Keuntungannya waktu random akses meningkat, akibat dari *head disk* dapat mencari lokasi blok apa saja dengan membaca informasi dalam FAT.

5.7.3. Alokasi Dengan Indeks (*Indexed Allocation*)

Metode alokasi dengan berangkai dapat menyelesaikan masalah fragmentasi eksternal dan

pendeklarasian ukuran dari metode alokasi berdampingan. Bagaimana pun tanpa FAT, metode alokasi berangkai tidak mendukung keefisiensian akses langsung, karena penunjuk ke bloknya berserakan dengan bloknya didalam disk dan perlu didapatkan secara berurutan. Metode alokasi dengan indeks menyelesaikan masalah ini dengan mengumpulkan semua penunjuk menjadi dalam satu lokasi yang dinamakan blok indeks (*index block*). Setiap berkas mempunyai blok indeks, yang merupakan sebuah larik *array* dari alamat-alamat disk-blok. Direktori mempunyai alamat dari blok indeks. Ketika berkas

dibuat, semua penunjuk dalam blok indeks di set menjadi *nil*. Ketika blok ke-*i* pertama kali ditulis, sebuah blok didapat dari pengatur ruang kosong *free-space manager* dan alamatnya diletakkan ke dalam blok indeks ke-*i*. Metode ini mendukung akses secara langsung, tanpa mengalami fragmentasi eksternal karena blok kosong mana pun dalam disk dapat memenuhi permintaan ruang tambahan. Tetapi metode ini dapat menyebabkan ada ruang yang terbuang. Penunjuk yang berlebihan dari blok indeks secara umum lebih besar dari yang terjadi pada metode alokasi berangkai.

Mekanisme untuk menghadapi masalah berapa besar blok indeks yang diperlukan sebagai berikut:

- *Linked scheme*: untuk berkas-berkas yang besar, dilakukan dengan menyambung beberapa blok indeks menjadi satu.
- *Multilevel index*: sebuah varian dari representasi yang berantai adalah dengan menggunakan blok indeks level pertama menunjuk ke himpunan blok indeks level kedua, yang akhirnya menunjuk ke blok-blok berkas.
- *Combined scheme*: digunakan oleh sistem *BSD UNIX* yaitu dengan menetapkan 15 penunjuk dari blok indeks dalam blok indeksnya berkas. 12 penunjuk pertama menunjuk ke *direct blocks* yang menyimpan alamat-alamat blok yang berisi data dari berkas. 3 penunjuk berikutnya menunjuk ke *indirect blocks*. Penunjuk *indirect blok* yang pertama adalah alamat dari *single indirect block*, yang merupakan blok indeks yang berisi alamat-alamat blok yang berisi data. Lalu ada penunjuk *double indirect block* yang berisi alamat dari sebuah blok yang berisi alamat-alamat blok yang berisi penunjuk ke blok data yang sebenarnya.

5.7.4. Kinerja Sistem Berkas

Salah satu kesulitan dalam membandingkan performa sistem adalah menentukan bagaimana sistem tersebut akan digunakan. Sistem yang lebih banyak menggunakan akses sekuensial (berurutan) akan memakai metode yang berbeda dengan sistem yang lebih sering menggunakan akses random (acak).Untuk jenis akses apa pun, alokasi yang berdampingan hanya memerlukan satu akses untuk mendapatkan sebuah blok disk. Karena kita dapat menyimpan *initial address* dari berkas di dalam memori, maka alamat disk pada blok ke-*i* dapat segera dikalkulasi dan dibaca secara langsung.

Untuk alokasi berangkai *(linked list)*, kita juga dapat menyimpan alamat dari blok selanjutnya ke dalam memori, lalu membacanya secara langsung. Metode ini sangat baik untuk akses sekuensial,

namun untuk akses langsung, akses menuju blok ke-*i*kemungkinan membutuhkan pembacaan disk sebanyak *i* kali. Masalah ini mengindikasikan bahwa alokasi berangkai sebaiknya tidak digunakan untuk aplikasi yang membutuhkan akses langsung.

Oleh sebab itu, beberapa sistem mendukung akses langsung dengan menggunakan alokasi berdampingan (*contiguous allocation*), serta akses berurutan dengan alokasi berangkai. Untuk sistem-sistem tersebut, jenis akses harus dideklarasikan pada saat berkas dibuat. Berkas yang dibuat untuk akses sekuensial (berurutan) akan dirangkaikan dan tidak dapat digunakan untuk akses langsung. Berkas yang dibuat untuk akses langsung akan berdampingan dan dapat mendukung baik akses langsung mau pun akses berurutan, dengan mendeklarasikan jarak maksimum. Perhatikan bahwa sistem operasi harus mendukung struktur data dan algoritma yang sesuai untuk mendukung kedua metode alokasi di atas.

Alokasi dengan menggunakan indeks lebih rumit lagi. Jika blok indeks telah terdapat dalam memori, akses dapat dilakukan secara langsung. Namun, menyimpan blok indeks dalam memori memerlukan ruang (*space*) yang besar. Jika ruang memori tidak tersedia, maka kita mungkin harus membaca blok indeks terlebih dahulu, baru kemudian blok data yang diinginkan. Untuk indeks dua tingkat, pembacaan dua blok indeks mungkin diperlukan. Untuk berkas yang berukuran sangat besar, mengakses blok di dekat akhir suatu berkas akan membutuhkan pembacaan seluruh blok indeks agar dapat mengikuti rantai penunjuk sebelum blok data dapat dibaca. Dengan demikian, performa alokasi dengan menggunakan indeks ditentukan oleh: struktur indeks, ukuran berkas, dan posisi dari blok yang diinginkan.

Beberapa sistem mengkombinasikan alokasi berdampingan dengan alokasi indeks. Caranya adalah dengan menggunakan alokasi berdampingan untuk berkas berukuran kecil (3-4 blok), dan beralih secara otomatis ke alokasi indeks jika berkas semakin membesar.

5.8. Managemen Ruang Kosong (*Free Space*)

Semenjak hanya tersedia tempat yang terbatas pada disk maka sangat berguna untuk menggunakan kembali tempat dari berkas yang dihapus untuk berkas baru, jika dimungkinkan,karena pada media yang sekali tulis (media optik) hanya dimungkinkan sekali menulis dan menggunakannyanya kembali secara fisik tidak mungkin. Untuk mencatat tempat kosong pada disk, sistem mempunyai daftar tempat kosong (*free space list*). Daftar ini menyimpan semua blok disk yang kosong yang tidak dialokasikan pada sebuah berkas atau direktori. Untuk membuat berkas baru,

sistem mencari ke daftar tersebut untuk mencarikan tempat kosong yang di butuhkan, lalu tempat tersebut dihilangkan dari daftar. Ketika berkas dihapus, alamat berkas tadi ditambahkan pada daftar.

5.8.1. Menggunakan Bit Vektor

Seringnya daftar raung kosong diimplementasikan sebagai bit map atau bit vektor. Tiap blok direpresentasikan sebagai 1 bit. Jika blok tersebut kosong maka isi bitnya 1 dan jika bloknya sedang dialokasikan maka isi bitnya 0. Sebagai contoh sebuah disk dimana blok 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 dan 27 adalah kosong, dan sisanya dialokasikan.

Bit mapnya akan seperti berikut:

001111001111110001100000011100000...

Keuntungan utama dari pendekatan ini adalah relatif sederhana dan efisien untuk mencari blok pertama yang kosong atau berturut-turut n blok yang kosong pada disk. Banyak komputer yang menyediakan instruksi manipulasi bit yang dapat digunakan secara efektif untuk tujuan ini. Sebagai contohnya, dari keluarga prosesor Intel dimulai dari 80386 dan keluarga Motorola dimulai dari 68020 (prosesor yang ada di PC dan Macintosh) mempunyai instruksi yang mengembalikan jarak di *word* dari bit pertama dengan nilai 1. Sistem operasi Apple Macintosh menggunakan metode bit vektor untuk mengalokasikan tempat pada disk. Dalam hal ini perangkat keras mendukung perangkat lunak tetapi bit vektor tidak efisien kecuali seluruh vektor disimpan dalam memori utama (dan ditulis di disk untuk kebutuhan pemulihan). Menyimpan dalam memori utama dimungkinkan untuk disk yang kecil pada mikro komputer, tetapi tidak untuk disk yang besar. Sebuah *disk* 1,3 GB dengan 512-byte blok akan membutuhkan *bit map* sebesar 332K untuk mencatat blok yang kosong.

5.8.2. *Linked List*

Pendekatan lain adalah untuk menghubungkan semua blok yang kosong, menyimpan *pointer* ke blok pertama yang kosong di tempat yang khusus pada disk dan menyimpannya di memori. Blok pertama ini menyimpan *pointer* ke blok kosong berikutnya dan seterusnya. Pada contoh sebelumnya kita akan menyimpan *pointer* ke blok ke 2 sebagai blok kosong pertama, blok 2 akan menyimpan *pointer* ke blok 3, yang akan menunjuk ke blok 4 dan seterusnya. Bagaimana pun metode ini tidak efisien karena untuk *traverse* daftar tesebut kita perlu membaca tiap blok yang membutuhkan waktu I/O. Untungnya traverse ini tidak sering

digunakan. Umumnya, sistem operasi membutuhkan blok kosong untuk mengalokasikan blok tersebut ke berkas, maka blok pertama pada daftar ruang kosong digunakan.

5.8.3. *Grouping*

Modifikasi lainnya adalah dengan menyimpan alamat dari n blok kosong pada blok kosong pertama. Pada n-1 pertama dari blok-blok ini adalah kosong. Blok terakhir menyimpan alamat n blok kosong lainnya dan seterusnya. Keuntungannya dari implementasi seperti ini adalah alamat dari blok kosong yang besar sekali dapat ditemukan dengan cepat, tidak seperti pendekatan standar *linked-list*.

5.8.4. *Counting*

Pendekatan lain adalah dengan mengambil keuntungan dari fakta bahwa beberapa blok yang berkesinambungan akan dialokasikan atau dibebaskan secara simultan. Maka dari itu dari pada menyimpan daftar dari banyak alamat disk, kita dapat menyimpan alamat dari blok kosong pertama dan jumlah dari blok kosong yang berkesinambungan yang mengikuti blok kosong pertama. Tiap isi dari daftar menyimpan alamat disk dan penghitung (*counter*). Meski pun setiap isi membutuhkan tempat lebih tetapi secara keseluruhan daftar akan lebih pendek, selama *count* lebih dari satu.

5.9. Implementasi Direktori

Pemilihan dalam algoritma alokasi direktori dan manajemen direktori mempunyai efek yang besar dalam efisiensi, performa, dan kehandalan dari sistem berkas.

5.9.1. *Linear List*

Metode paling sederhana dalam mengimplementasikan sebuah direktori adalah dengan menggunakan *linear list* dari nama berkas dengan penunjuk ke blok data. *Linear list* dari direktori memerlukan pencarian searah untuk mencari suatu direktori didalamnya. Metode sederhana untuk di program tetapi memakan waktu lama ketika dieksekusi. Untuk membuat berkas baru kita harus mencari di dalam direktori untuk meyakinkan bahwa tidak ada berkas yang bernama sama. Lalu kita tambahkan sebuah berkas baru pada akhir direktori. Untuk menghapus sebuah berkas, kita mencari berkas tersebut dalam direktori, lalu

melepaskan tempat yang dialokasikan untuknya. Untuk menggunakan kembali suatu berkas dalam direktori kita dapat melakukan beberapa hal. Kita dapat menandai berkas tersebut sebagai tidak terpakai (dengan menamainya secara khusus, seperti nama yang kosong, atau bit terpakai atau tidak yang ditambahkan pada berkas), atau kita dapat menambahkannya pada daftar direktori bebas. Alternatif lainnya kita dapat menyalin ke tempat yang dikosongkan pada direktori. Kita juga bisa menggunakan linked list untuk mengurangi waktu untuk menghapus berkas. Kelemahan dari linear list ini adalah percarian searah untuk mencari sebuah berkas. Direktori yang berisi informasi sering digunakan, implementasi yang lambat pada cara aksesnya akan menjadi perhatian pengguna. Faktanya, banyak sistem operasi mengimplementasikan '*software cache*' untuk menyimpan informasi yang paling sering digunakan. Penggunaan '*cache*' menghindari pembacaan informasi berulang-ulang pada disk. Daftar yang telah diurutkan memperbolehkan pencarian biner dan mengurangi waktu rata-rata pencarian.

Bagaimana pun juga penjagaan agar daftar tetap terurut dapat merumitkan operasi pembuatan dan penghapusan berkas, karena kita perlu memindahkan sejumlah direktori untuk mengurutkannya. *Tree* yang lebih lengkap dapat membantu seperti B-tree. Keuntungan dari daftar yang terurut adalah kita dapatkan daftar direktori yang terurut tanpa pengurutan yang terpisah.

5.9.2. *Hash Table*

Struktur data lainnya yang juga digunakan untuk direktori berkas adalah *hash table*. Dalam metode ini linear list menyimpan direktori, tetapi struktur data hash juga digunakan. *Hash table* mengambil nilai yang dihitung dari nama berkas dan mengembalikan sebuah penunjuk ke nama berkas yang ada di-*linear list*. Maka dari itu dapat memotong banyak biaya pencarian direktori. Memasukkan dan menghapus berkas juga lebih mudah dan cepat. Meski demikian beberapa aturan harus dibuat untuk mncegah tabrakan, situasi dimana dua nama berkas pada *hash* mempunyai tempat yang sama. Kesulitan utama dalam *hash table* adalah ukuran tetap dari *hash table* dan ketergantungan dari fungsi *hash* dengan ukuran *hash table*. Sebagai contoh, misalkan kita membuat suatu *linear-probing hash table* yang dapat menampung 64 data. Fungsi *hash* mengubah nama berkas menjadi nilai dari 0 sampai 63. Jika kita membuat berkas ke 65 maka ukuran tabel *hash* harus diperbesar sampai misalnya 128 dan kita membutuhkan suatu fungsi *hash* yang baru yang dapat memetakan nama berkas dari jangkauan 0 sampai 127, dan kita

harus mengatur data direktori yang sudah ada agar memenuhi fungsi *hash* yang baru.

Sebagai alternatif dapat digunakan *chained-overflow hash table*, setiap *hash table* mempunyai daftar yang terkait (*linked list*) dari pada nilai individual dan kita dapat mengatasi tabrakan dengan menambah tempat pada daftar terkait tersebut. Pencarian dapat menjadi lambat, karena pencarian nama memerlukan tahap pencarian pada daftar terkait. Tetapi operasi ini lebih cepat dari pada pencarian linear terhadap seluruh direktori.

5.10. Efisiensi dan Unjuk Kerja

Setelah kita membahas alokasi blok dan pilihan manajemen direktori maka dapat dibayangkan bagaimana efek mereka dalam keefisiensian dan unjuk kerja penggunaan disk. Hal ini dikarenakan disk selalu menjadi "bottle-neck" dalam unjuk kerja sistem.

5.10.1. Efisiensi

Disk dapat digunakan secara efisien tergantung dari teknik alokasi disk serta algoritma pembentukan direktori yang digunakan. Contoh, pada UNIX, direktori berkas dialokasikan terlebih dahulu pada partisi. Walau pun disk yang kosong pun terdapat beberapa persen dari ruangnya digunakan untuk direktori tersebut. Unjuk kerja sistem berkas meningkat akibata dari pengalokasian awal dan penyebaran direktori ini pada partisi. Sistem berkas UNIX melakukan ini agar blok-blok data berkas selalu dekat dengan blok direktori berkas sehingga waktu pencariannya berkurang.

Ada pula keefesiensian pada ukuran penunjuk yang digunakan untuk mengakses data. Masalahnya dalam memilih ukuran penunjuk adalah merencanakan efek dari perubahan teknologi. Masalah ini diantisipasi dengan menginisialisasi terlebih dahulu sistem berkasnya dengan alasan keefisiensian.

Pada awal, banyak struktur data dengan panjang yang sudah ditentu kan dan dialokasi pada ketika sistem dijalankan. Ketika tabel proses penuh maka tidak ada proses lain yang dapat dibuat. Begitu juga dengan tabel berkas ketika penuh, tidak ada berkas yang dapat dibuka. Hal ini menyebabkan sistem gagal melayani permintaan pengguna. Ukuran tabel-tabel ini dapat ditingkatkan hanya dengan mengkompilasi ulang kernel dan boot ulang sistemnya. Tetapi sejak dikeluarkannya Solaris 2, hampir setiap struktur kernel dialokasikan secara dinamis sehingga menghapus batasan buatan pada unjuk kerja sistem.

5.10.2. Kinerja

Ketika metode dasar disk telah dipilih, maka masih ada beberapa cara untuk meningkatkan unjuk kerja. Salah satunya adalah dengan mengguna kan cache, yang merupakan memori lokal pada pengendali disk, dimana cache cukup besar untuk menampung seluruh track pada satu waktu. Beberapa sistem mengatur seksi terpisah dari memori utama untuk disk-cache, yang diasumsikan bahwa blok-blok disimpan karena mereka akan digunakan dalam waktu dekat. Ada juga sistem yang menggunakan memori fisik yang tidak digunakan sebagai penyangga yang dibagi atas sistem halaman (paging) dan sistem disk-blok cache. Suatu sistem melakukan banyak operasi I/O akan menggunakan sebagian banyak memorinya sebagai blok cache, dimana suatu sistem mengeksekusi banyak program akan menggunakan sebagian besar memori-nya untuk ruang halaman.

Beberapa sistem mengoptimalkan disk-cache nya dengan menggunakan berbagai macam algoritma penempatan ulang (replacement algorithms), tergantung dari macam tipe akses dari berkas. Pada akses yang sekuen sial dapat dioptimasi dengan teknik yang dikenal dengan nama free-behind dan read-ahead. Free-behind memindahkan sebuah blok dari penyangga secepatnya ketika blok berikutnya diminta. Hal ini dilakukan karena blok sebelumnya tidak lagi digunakan sehingga akan membuang ruang yang ada di penyangga. Sedangkan dengan read ahead, blok yang diminta dan beberapa blok berikutnya dibaca dan ditempatkan pada cache. Hal ini dilakukan karena kemungkinan blok-blok berikutnya akan diminta setelah blok yang sedang diproses. Hal ini juga mem beri dampak pada waktu yang digunakan akan lebih cepat.

Metode yang lain adalah dengan membagi suatu seksi dari memori untuk disk virtual atau RAM disk. Pada RAM disk terdapat operasi-operasi standar yang terdapat pada disk, tetapi semua operasi tersebut terjadi di dalam suatu seksi memori, bukan pada disk. Tetapi, RAM disk hanya berguna untuk penyimpanan sementara, karena jika komputer di boot ulang atau listrik mati maka isi dalam RAM disk akan terhapus.

Perbedaan antara RAM disk dan disk cache adalah dalam masalah siapa yang mengendalikan disk tersebut. RAM disk dikendalikan oleh peng guna sepenuhnya, sedangkan disk cache dikendalikan oleh sistem operasi.

5.11. *Recovery*

Karena semua direktori dan berkas disimpan di dalam memori utama dan disk, maka kita perlu memastikan bahwa kegagalan

pada sistem tidak menyebabkan hilangnya data atau data menjadi tidak konsiten.

5.11.1. Pemeriksaan Rutin

Informasi direktori pada memori utama pada umumnya lebih up to date daripada informasi yang terdapat di disk dikarenakan penulisan dari informasi direktori cached ke disk tidak langsung terjadi pada saat setelah peng-update-an terjadi. Consistency checker membandingkan data yang terdapat di struktur direktori dengan blok-blok data pada disk, dan mencoba memperbaiki semua ketidak konsistensian yang terjadi akibat crash-nya komputer. Algoritma pengalokasian dan management ruang kosong menentukan tipe dari masalah yang ditemukan oleh checker dan seberapa sukses dalam memperbaiki masalah-masalah tersebut.

5.11.2. *Back Up and Restore*

Karena kadang-kadang magnetik disk gagal, kita harus memastikan bahwa datanya tidak hilang selamanya. Karena itu, kita menggunakan program sistem untuk mem-back up data dari disk ke alat penyimpanan yang lain seperti floopy disk, magnetic tape, atau optical disk. Pengembalian berkas-berkas yang hilang hanya masalah menempatkan lagi data dari back up data yang telah dilakukan.

Untuk meminimalisir penyalinan, kita dapat menggunakan informasi dari setiap masukan direktori berkas. Umpamanya, jika program back up mengetahui bahwa back up terakhir dari berkas sudah selesai dan penulisan terakhir pada berkas dalam direktori menandakan berkas tidak terjadi perubahan maka berkas tidak harus disalin lagi. Penjadualan back up yang umum sebagai berikut:

- Hari 1: Salin ke tempat penyimpanan back up semua berkas dari disk, disebut sebuah full backup.
- Hari 2: Salin ke tempat penyimpanan lain semua berkas yang berubah sejak hari 1, disebut incremental backup.
- Hari 3: Salin ke tempat peyimpanan lain semua berkas yang berubah sejak hari 2.
- Hari N: salin ke tempat penyimpanan lain semua berkas yang berubah sejak hari N-1, lalu kembali ke hari 1.

Keuntungan dari siklus backup ini adalah kita dapat menempatkan kembali berkas mana pun yang tidaksengaja

terhapus pada waktu siklus dengan mendapatkannya dari back up hari sebelumnya. Panjang dari siklus disetujui antara banyaknya tempat penyimpanan backup yang diperlukan dan jumlah hari kebelakang dari penempatan kembali dapat dilakukan.

Ada juga kebiasaan untuk mem-backup keseluruhan dari waktu ke waktu untuk disimpan selamanya daripada media backupnya digunakan kembali. Ada bagusnya menyimpan backup-backup permanent ini di lokasi yang jauh dari backup yang biasa, untuk menghindari kecelakaan seperti kebakaran dan lain-lain. Dan jangan menggunakan kembali media backup terlalu lama karena media tersebut akan rusak jika terlalu sering digunakan kembali.

5.12. Macam-macam Sistem Berkas

5.12.1. Sistem Berkas Pada Windows

Direktori dan Berkas

Sistem operasi Windows merupakan sistem operasi yang telah dikenal luas. Sistem operasi ini sangat memudahkan para penggunanya dengan membuat struktur direktori yang sangat *user-friendly*. Para pengguna Windows tidak akan menemui kesulitan dalam menggunakan sistem direktori yang telah dibuat oleh Microsoft. Windows menggunakan sistem *drive letter* dalam merepresentasikan setiap partisi dari *disk*. Sistem operasi secara otomatis akan terdapat dalam partisi pertama yang diberi label *drive* C.

Sistem operasi Windows dibagi menjadi dua keluarga besar, yaitu keluarga Windows 9x dan keluarga Windows NT (New Technology).

Direktori yang secara otomatis dibuat dalam instalasi Windows adalah:

1. Direktori C:\WINDOWS

   Direktori ini berisikan sistem dari Windows. Dalam direktori ini terdapat pustaka-pustaka yang diperlukan oleh Windows, *device driver*, *registry*, dan program-program esensial yang dibutuhkan oleh Windows untuk berjalan dengan baik.

2. Direktori C:\Program Files

   Direktori ini berisikan semua program yang diinstal ke dalam sistem operasi. Semua program yang diinstal akan menulis *entry* ke dalam *registry* agar program tersebut dapat dijalankan dalam sistem Windows.

3. Direktori C:\My Documents

Direktori ini berisikan semua dokumen yang dimiliki oleh pengguna sistem.

Sistem operasi Windows dapat berjalan diatas beberapa macam sistem berkas. Setiap sistem berkas memiliki keunggulan dan kekurangan masing-masing. Semua keluarga Windows yang berbasis Windows NT dapat mendukung sistem berkas yang digunakan oleh keluarga Windows 9x, namun hal tersebut tidak berlaku sebaliknya.

Sistem Berkas yang terdapat dalam sistem operasi Windows adalah:

1. FAT 16: Sistem berkas ini digunakan dalam sistem operasi DOS dan Windows 3.1
2. FAT 32: Sistem ini digunakan oleh keluarga Windows 9x.
3. NTFS: Merupakan singkatan dari *New Technology File System*. Sistem berkas ini adalah sistem berkas berbasis *journaling* dan dapat digunakan hanya pada keluarga Windows NT. Keunggulan dari sistem berkas ini adalah fasilitas *recovery* yang memungkinkan dilakukannya penyelamatan data saat terjadi kerusakan pada sistem operasi.

5.12.2. Sistem Berkas pada UNIX (dan turunannya)

Ketika kita *login* ke UNIX, kita akan ditempatkan di direktori *root* kita. Direktori *root* kita dikenal sebagai direktori *home* kita dan dispesifikasi dengan *environment variable* yang dinamakan HOME. *Environment variable* ini menentukan karakteristik dari *shell* kita dan interaksi pengguna dengan *shell* tersebut. *Environment variable* yang umum adalah variabel PATH, yang mendefinisikan dimana *shell* akan mencari ketika perintah dari pengguna. Untuk melihat daftar *environment variable*, gunakan saja perintah printenv. Sedangkan untuk mengatur *environment variable*, gunakan setenv.

Ada beberapa direktori yang umum terdapat dalam instalasi UNIX:

1. Direktori "/" (*root*)

Direktori ini terletak pada level teratas dari struktur direktori UNIX. Biasanya direktori *root* ini

diberi tanda / atau *slash*. Direktori ini biasanya hanya terdiri dari direktori-direktori lainnya yang terletak pada level dibawah level direktori *root*. Berkas-berkas dapat disimpan pada direktori *root* tetapi usahakan tidak menyimpan berkas-berkas biasa sehingga direktori ini tetap terjaga keteraturannya.

Perubahan penamaan direktori-direktori yang ada pada direktori *root* akan menyebabkan sebagian besar dari sistem menjadi tidak berguna. Karena sebagian besar dari direktori-direktori ini berisi fungsi-fungsi yang sifatnya kritikal yang dimana sistem operasi dan semua aplikasi memerlukan direktori-direktori ini dengan nama yang sudah diberikan pada awal instalasi. Tetapi kita bisa membuat direktori lain pada level ini. Direktori *home* juga bisa ditemukan pada level ini hasil pembuatan oleh administrator sistem.

2. Direktori "/bin"

Direktori ini berisi program-program yang esensial agar sistem operasi dapat bekerja dengan benar. Dalam direktori ini dapat ditemukan perintah-perintah navigasi, program-program *shell*, perintah pencarian dan lain-lainnya. bin adalah singkatan dari kata *binary*. Di UNIX, sebuah *binary* adalah berkas yang dapat dieksekusi. Sebagian besar dari perintah dalam UNIX merupakan *binary*, perintah-perintah tersebut merupakan program-program kecil yang dapat dieksekusi oleh pengguna.

Ada beberapa perintah yang disebut perintah *built-in* dimana fungsi mereka dikendalikan oleh program *shell* sehingga mereka tidak beroperasi sebagai *binary* yang terpisah.

Terkadang direktori *bin* terhubung ke direktori lain yang dinamakan /usr/bin. Direktori /usr/bin biasanya adalah lokasi sebenarnya dari *binary-binary* pengguna disimpan.Dalam hal ini, /bin adalah gerbang untuk mencapai /usr/bin.

3. Direktori "/dev"

Direktori ini berisi berkas-berkas alat atau alat I/O. Sistem UNIX menganggap semua hal sebagai berkas. Hal-hal seperti monitor, CD-ROM, printer dan lain-lainnya dianggap hanya sebagai berkas saja oleh sistem operasi. Jika UNIX memerlukan perangkat-perangkat tersebut maka UNIX akan mencarinya ke direktori *dev*.

4. Direktori "/etc"

Direktori yang dibaca et-see ini berisi beberapa konfigurasi berkas pengguna dan sistem, dan berkas yang ditunjuk sistem sebagai operasi normal seperti berkas kata sandi, pesan untuk hari ini, dan lain-lainnya.

5. Direktori "/lib"

Direktori ini berisi pustaka-pustaka (*libraries*) yang dibagi (*shared*). Pustaka ini adalah rutin

perangkat lunak (*software routines*) yang digunakan lebih dari satu bagian dari sistem operasi. Ketika kita menginstalasi perangkat lunak yang baru maka ada pustaka-pustaka baru yang ditambahkan ke direktori lib. Jika pada waktu berusaha menjalankan aplikasi terdapat pesan *error*, hal ini diakibatkan ada pustaka yang hilang dari direktori lib. Aplikasi-aplikasi di UNIX biasanya memeriksa lib ketika menginstalasi untuk memeriksa apakah pustaka-pustaka yang diperlukan oleh aplikasi sudah tersedia atau belum. Jika sudah tersedia, UNIX biasanya tidak menimpa pustaka tersebut.

6. Direktori "/sbin"

   Direktori ini berisi *binary-binary* juga seperti pada direktori *bin*.Tetapi,bedanya adalah *binary-binary* pada direktori ini berhubungan dengan fungsi-fungsi sistem administrasi pada sistem operasi UNIX. *Binary-binary* ini bukan yang biasa digunakan oleh pengguna tetapi digunakan agar komputer dapat beroperasi secara efisien.

7. Direktori "/usr"

   Direktori ini terdiri dari banyak direktori seperti pada direktori *root*. Direktori ini berisi berkas-berkas yang dapat diakses oleh para pengguna biasa. Struktur dari direktori ini mirip dengan struktur direktori "/". Beberapa direktori yang terdapat dalam direktori ini berhubungan dengan direktori yang ada di direktori /.

8. Direktori "/var"

   Direktori ini berisi data yang bermacam-macam (vary). Perubahan data dalam sistem yang aktif sangatlah cepat. Data-data seperti ini ada dalam waktu yang singkat. Karena sifatnya yang selalu berubah tidak memungkinkan disimpan dalam direktori seperti "/etc". Oleh karena itu, data-data seperti ini disimpan di direktori var.

5.12.3. Perbandingan antara Windows dan UNIX

Sistem berkas UNIX berbeda dengan sistem berkas Windows (DOS) karena sistem berkas UNIX lebih hebat dan mudah diatur daripada Windows (DOS). Penamaan dalam UNIX dan Windows berbeda.Karena sistem Windows ingin memudahkan pengguna maka sistem mereka mengubah nama menjadi nama yang lebih mudah bagi para pengguna. Contohnya adalah nama *folder* dalam adalah perubahan dari *directory* yang masih digunakan oleh UNIX. Penggunaan *back slash* (\) digunakan untuk memisahkan direktori-direktori dalam Windows, tetapi hal ini tidak ada dalam

UNIX. Sistem UNIX menggunakan *case sensitive*, yang artinya nama suatu berkas yang sama jika dibaca, tetapi penulisan namanya berbeda dalam hal ada satu file yang menggunakan huruf kapital dalam penamaan dan satu tidak akan berbeda dalam UNIX. Contohnya ada berkas bernama berkasdaku.txt dan BerkasDaku.txt, jika dibaca nama berkasnya sama tetapi dalam UNIX ini merupakan dua berkas yang jauh berbeda. Jika berkas-berkas ini berada di sistem Windows, mereka menunjuk ke berkas yang sama yang berarti Windows tidak *case sensitive*.

Hal lain yang membedakan sistem berkas UNIX dengan Windows adalah UNIX tidak menggunakan *drive letter* seperti C:, D: dalam Windows. Tetapi semua partisi dan drive ekstra di *mount* didalam sub-direktori di bawah direktori *root*. Jadi pengguna tidak harus bingung di *drive letter* mana suatu berkas berada sehingga seluruh sistem seperti satu sistem berkas yang berurutan dari direktori root menurun secara hierarki.

5.12.4. Macam-macam Sistem Berkas di UNIX

Secara garis besar, sistem berkas di sistem UNIX terbagi menjadi dua, yaitu sistem berkas dengan fasilitas *journaling* dan yang tidak memiliki fasilitas tersebut. Dibawah ini adalah beberapa sistem berkas yang digunakan dalam sistem UNIX pada umumnya:

1. EXT2
2. EXT3
3. JFS (*Journaling File System*)
4. ReiserFS
5. Dan Lain-lain.

5.13. Kesimpulan

Sistem berkas merupakan mekanisme penyimpanan on-line serta untuk akses, baik data mau pun program yang berada dalam Sistem Operasi. Terdapat dua bagian penting dalam sistem berkas, yaitu:

1. Kumpulan berkas, sebagai tempat penyimpanan data, serta
2. Struktur direktori, yang mengatur dan menyediakan informasi mengenai seluruh berkas dalamsistem.

Berkas adalah kumpulan informasi berkait yang diberi nama dan direkam pada penyimpanan sekunder.

Atribut berkas terdiri dari:

1. Nama; merupakan satu-satunya informasi yang tetap dalam bentuk yang bisa dibaca oleh manusia (human-readable form)
2. Type; dibutuhkan untuk sistem yang mendukung beberapa type berbeda
3. Lokasi; merupakan pointer ke device dan ke lokasi berkas pada device tersebut
4. Ukuran (size); yaitu ukuran berkas pada saat itu, baik dalam byte, huruf, atau pun blok
5. Proteksi; adalah informasi mengenai kontrol akses, misalnya siapa saja yang boleh membaca, menulis, dan mengeksekusi berkas
6. Waktu, tanggal dan identifikasi pengguna; informasi ini biasanya disimpan untuk:

   - pembuatan berkas
   - modifikasi terakhir yang dilakukan pada berkas, dan
   - modifikasi terakhir yang dilakukan pada berkas, dan
   - modifikasi terakhir yang dilakukan pada berkas, dan
   - penggunaan terakhir berkas

Operasi Pada Berkas

1. Membuat sebuah berkas.
2. Menulis pada sebuah berkas.
3. Membaca sebuah berkas.
4. Menempatkan kembali sebuah berkas.
5. Menghapus sebuah berkas.
6. Memendekkan berkas.

Metode Akses

1. Akses Berurutan.
2. Akses Langsung.
3. Akses menggunakan Indeks.

Operasi Pada Direktori

1. Mencari berkas.
2. Membuat berkas.
3. Menghapus berkas.
4. Menampilkan isi direktori.
5. Mengganti nama berkas.

6. Melintasi sistem berkas.

Macam-macam Direktori

1. Direktori Satu Tingkat
2. Direktori Dua Tingkat.
3. Direktori Dengan Struktur "Tree".
4. Direktori Dengan Struktur "Acyclic-Graph".
5. Direktori Dengan Struktur Graph.

Metode Alokasi Berkas

1. Alokasi Secara Berdampingan (*Contiguous Allocation*).
2. Alokasi Secara Berangkai (*Linked Allocation*).
3. Alokasi Dengan Indeks (*Indexed Allocation*).

Manajemen Free Space

1. Menggunakan Bit Vektor.
2. Linked List.
3. Grouping.
4. Counting.

Implementasi Direktori

1. Linear List.
2. Hash Table.

Sistem Berkas pada Windows

Direktori yang secara otomatis dibuat dalam instalasi Windows adalah:

1. Direktori C:\WINDOWS
2. Direktori C:\Program Files
3. Direktori C:\My Documents

Sistem Berkas yang terdapat dalam sistem operasi Windows adalah:

1. FAT 16

   Sistem berkas ini digunakan dalam sistem operasi DOS dan Windows 3.1

2. FAT 32

   Sistem ini digunakan oleh keluarga Windows 9x

3. NTFS

Merupakan singkatan dari New Technology File System. Sistem berkas ini adalah sistem berkas berbasis journaling dan dapat digunakan hanya pada keluarga Windows NT. Keunggulan dari sistem berkas ini adalah fasilitas recovery yang memungkinkan dilakukannya penyelamatan data saat terjadi kerusakan pada sistem operasi.

Sistem Berkas pada UNIX (dan turunannya)

Ada beberapa direktori yang umum terdapat dalam instalasi UNIX:

1. Direktori /root.
2. Direktori /bin.
3. Direktori /dev.
4. Direktori /etc.
5. Direktori /lib.
6. Direktori /sbin.
7. Direktori /usr.
8. Direktori /var.

Macam-macam Sistem Berkas di UNIX

1. EXT2.
2. EXT3.
3. JFS (Journaling File System).
4. ReiserFS.
5. Dan Lain-lain.

5.14. Soal-Soal Sistem Berkas

1. Sebutkan macam-macam atribut pada berkas!
2. Operasi apa sajakah yang dapat diterapkan pada sebuah berkas?
3. Sebutkan informasi yang terkait dengan pembukaan berkas!
4. Sebutkan dan jelaskan metode alokasi pada sistem berkas!
5. Sebutkan dan jelaskan operasi pada direktori?
6. Sebutkan dan Jelaskan tentang tipe akses pada berkas?
7. Sebutkan dan jelaskan bagaimana cara mengatur free space?
8. Bagaimanakah implementasi dari sebuah direktori dalam disk
9. Sebutkan keunggulan dari sistem berkas dalam UNIX dengan sistem berkas pada WINDOWS?

10. Bagaimanakah langkah-langkah dalam proses back-up?

Bab 6

I/O dan Disk

6.1. Perangkat Keras *I/O*

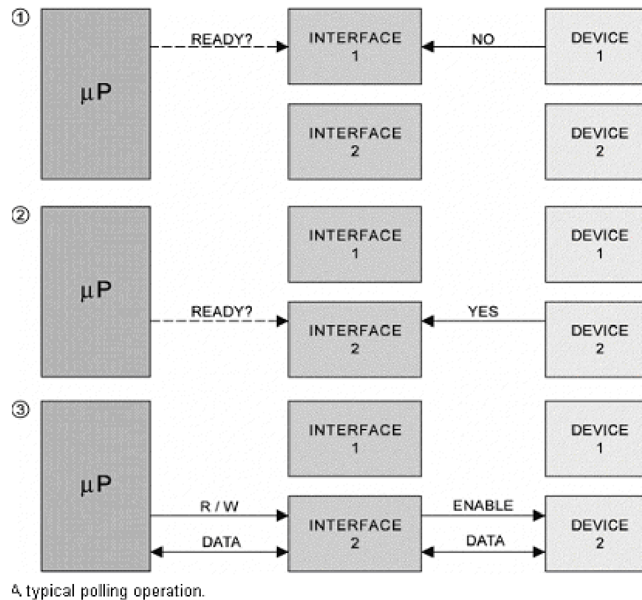
Secara umum, terdapat beberapa jenis seperti *device* penyimpanan (*disk*, *tape*), *transmission device* (*network card, modem*), dan *human-interface device* (*screen, keyboard, mouse*). *Device* tersebut dikendalikan oleh instruksi *I/O*. Alamat-alamat yang dimiliki oleh *device* akan digunakan oleh *direct I/O instruction* dan *memory-mapped I/O*.

Beberapa konsep yang umum digunakan ialah *port, bus (daisy chain/ shared direct access)*, dan *controller* (*host adapter*). *Port* adalah koneksi yang digunakan oleh *device* untuk berkomunikasi dengan mesin. *Bus* adalah koneksi yang menghubungkan beberapa *device* menggunakan kabel-kabel. *Controller* adalah alat-alat elektronik yang berfungsi untuk mengoperasikan *port, bus, dan device*.

Langkah yang ditentukan untuk *device* adalah *command-ready, busy*, dan *error*. *Host* mengeset *command-ready* ketika perintah telah siap untuk dieksekusi oleh *controller*. *Controller* mengeset *busy* ketika sedang mengerjakan sesuatu, dan men *clear busy* ketika telah siap untuk menerima perintah selanjutnya. *Error* diset ketika terjadi kesalahan.

6.1.1. *Polling*

Busy-waiting/ polling adalah ketika *host* mengalami *looping* yaitu membaca status register secara terus-menerus sampai status *busy* di-clear. Pada dasarnya *polling* dapat dikatakan efisien. Akan tetapi polling menjadi tidak efisien ketika setelah berulang-ulang melakukan *looping*, hanya menemukan sedikit *device* yang siap untuk men-*service*, karena CPU *processing* yang tersisa belum selesai.



Gambar 6-1. Polling Operation.

6.1.2. Interupsi

6.1.2.1. Mekanisme Dasar Interupsi

Ketika CPU mendeteksi bahwa sebuah *controller* telah mengirimkan sebuah sinyal ke *interrupt request line* (membangkitkan sebuah interupsi), CPU kemudian menjawab interupsi tersebut (juga disebut menangkap interupsi) dengan menyimpan beberapa informasi mengenai *state* terkini CPU--contohnyanilai instruksi *pointer*, dan memanggil *interrupt handler* agar *handler* tersebut dapat melayani *controller* atau alat yang mengirim interupsi tersebut.

6.1.2.2. Fitur Tambahan pada Komputer Modern

Pada arsitektur komputer modern, tiga fitur disediakan oleh CPU dan *interrupt controller* (pada perangkat keras) untuk dapat menangani interrupsi dengan lebih bagus. Fitur-fitur ini antara lain adalah kemampuan menghambat sebuah proses *interrupt handling* selama prosesi berada dalam *critical state*, efisiensi penanganan interupsi sehingga tidak perlu dilakukan polling untuk mencari *device* yang mengirimkan interupsi, dan fitur yang ketiga adalah adanya sebuah konsep *multilevel* interupsi sedemikian rupa sehingga terdapat prioritas dalam penanganan

interupsi (diimplementasikan dengan *interrupt priority level system*).

6.1.2.3. *Interrupt Request Line*

Pada peranti keras CPU terdapat kabel yang disebut *interrupt request line*, kebanyakan CPU memiliki dua macam *interrupt request line*, yaitu *nonmaskable interrupt* dan *maskable interrupt*. *Maskable interrupt* dapat dimatikan/ dihentikan oleh CPU sebelum pengeksekusian deretan *critical instruction (critical instruction sequence)* yang tidak boleh diinterupsi. Biasanya, interrupt jenis ini digunakan oleh *device controller* untuk meminta pelayanan CPU.

6.1.2.4. *Interrupt Vector* dan *Interrupt Chaining*

Sebuah mekanisme interupsi akan menerima alamat *interrupt handling routine* yang spesifik dari sebuah set, pada kebanyakan arsitektur komputer yang ada sekarang ini, alamat ini biasanya berupa sekumpulan bilangan yang menyatakan *offset* pada sebuah tabel (biasa disebut *interrupt vector*). Tabel ini menyimpan alamat-alamat *interrupt handler* spesifik di dalam memori. Keuntungan dari pemakaian vektor adalah untuk mengurangi kebutuhan akan sebuah *interrupt handler* yang harus mencari semua kemungkinan sumber interupsi untuk menemukan pengirim interupsi.

Akan tetapi, *interrupt vector* memiliki hambatan karena pada kenyataannya, komputer yang ada memiliki *device* (dan *interrupt handler*) yang lebih banyak dibandingkan dengan jumlah alamat pada *interrupt vector*. Karena itulah, digunakanlah teknik *interrupt chaining* dimana setiap elemen dari *interrupt vector* menunjuk/ merujuk pada elemen pertama dari sebuah daftar *interrupt handler*. Dengan teknik ini, *overhead* yang dihasilkan oleh besarnya ukuran tabel dan inefisiensi dari penggunaan sebuah *interrupt handler* (fitur pada CPU yang telah disebutkan sebelumnya) dapat dikurangi, sehingga keduanya menjadi kurang lebih seimbang.

6.1.2.5. Penyebab Interupsi

Interupsi dapat disebabkan berbagai hal, antara lain *exception, page fault*, interupsi yang dikirimkan oleh *device controllers*, dan *system call Exception* adalah suatu kondisi dimana terjadi sesuatu/ dari sebuah operasi didapat hasil tertentu yang dianggap khusus sehingga harus mendapat perhatian lebih, contoh nya pembagian dengan 0 (nol), pengaksesan alamat memori yang *restricted* atau

bahkan tidak valid, dan lain-lain. *System call* adalah sebuah fungsi pada aplikasi (perangkat lunak) yang dapat mengeksekusikan instruksi khusus berupa *software interrupt* atau *trap*.

6.1.3. DMA

6.1.3.1. Definisi

DMA adalah sebuah prosesor khusus (*special purpose processor*) yang berguna untuk menghindari pembebanan CPU utama oleh program *I/O* (PIO).

Gambar 6-2. DMA Interface.

6.1.3.2. Transfer DMA

Untuk memulai sebuah transfer DMA, *host* akan menuliskan sebuah DMA *command block* yang berisi *pointer* yang menunjuk ke sumber transfer, *pointer* yang menunjuk ke tujuan/ destinasi transfer, dan jumlah *byte* yang ditransfer, ke memori. CPU kemudian menuliskan alamat *command block* ini ke DMA *controller*, sehingga DMA *controller* dapat kemudian mengoperasikan *bus* memori secara langsung dengan menempatkan alamat-alamat pada *bus* tersebut untuk melakukan transfer tanpa bantuan CPU.

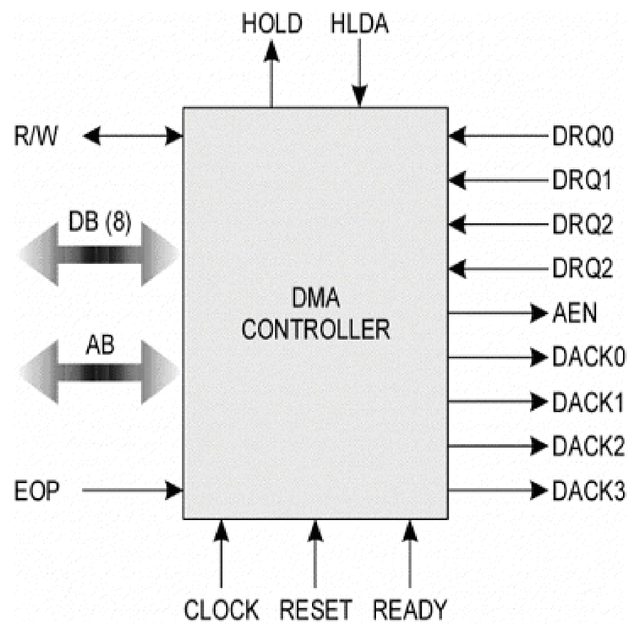
Tiga langkah dalam transfer DMA:

1. Prosesor menyiapkan DMA transfer dengan menyedia kan data-data dari *device*, operasi yang akan ditampilkan, alamat memori yang menjadi sumber dan tujuan data, dan banyaknya byte yang di transfer.
2. DMA *controller* memulai operasi (menyiapkan bus, menyediakan alamat, menulis dan membaca data), sampai seluruh blok sudah di transfer.

3. DMA *controller* meng-interupsi prosesor, dimana selanjutnya akan ditentukan tindakan berikutnya.

Pada dasarnya, DMA mempunyai dua metode yang berbeda dalam mentransfer data. Metode yang pertama adalah metode yang sangat baku dan simple disebut HALT, atau *Burst Mode* DMA, karena DMA *controller* memegang kontrol dari sistem bus dan mentransfer semua blok data ke atau dari memori pada *single burst*. Selagi transfer masih dalam progres, sistem mikroprosessor di-set *idle*, tidak melakukan instruksi operasi untuk menjaga internal register. Tipe operasi DMA seperti ini ada pada kebanyakan komputer.

Metode yang kedua, mengikut-sertakan DMA *controller* untuk memegang kontrol dari sistem bus untuk jangka waktu yang lebih pendek pada periode dimana mikroprosessor sibuk dengan operasi internal dan tidak membutuhkan akses ke sistem bus. Metode DMA ini disebut *cycle stealing mode*. *Cycle stealing* DMA lebih kompleks untuk diimplementasikan dibandingkan HALT DMA, karena DMA controller harus mempunyai kepintaran untuk merasakan waktu pada saat sistem bus terbuka.



Gambar 6-3. DMA Controller.

6.1.3.3. *Handshaking*

Proses *handshaking* antara DMA *controller* dan *device controller* dilakukan melalui sepasang kabel yang disebut DMA-*request* dan DMA-*acknowledge*. Device controller mengirimkan sinyal melalui DMA-*request* ketika akan mentransfer data sebanyak satu *word*. Hal ini kemudian akan mengakibatkan DMA *controller* memasukkan alamat-alamat yang dinginkan ke kabel alamat memori, dan mengirimkan sinyal melalui kabel DMA-*acknowledge*. Setelah sinyal melalui kabel DMA-*acknowledge* diterima, *device controller* mengirimkan data yang dimaksud dan mematikan sinyal pada DMA-*request*.

Hal ini berlangsung berulang-ulang sehingga disebut *handshaking*. Pada saat DMA *controller* mengambil alih memori, CPU sementara tidak dapat mengakses memori (dihalangi), walau pun masih dapat mengaksees data pada cache primer dan sekunder. Hal ini disebut *cycle stealing*, yang walau pun memperlambat komputasi CPU, tidak menurunkan kinerja karena memindahkan pekerjaan data transfer ke DMA *controller* meningkatkan performa sistem secara keseluruhan.

6.1.3.4. Cara-cara Implementasi DMA

Dalam pelaksanaannya, beberapa komputer menggunakan memori fisik untuk proses DMA , sedangkan jenis komputer lain menggunakan alamat virtual dengan melalui tahap "penerjemahan" dari alamat memori virtual menjadi alamat memori fisik, hal ini disebut *direct virtual-memory address* atau DVMA.

Keuntungan dari DVMA adalah dapat mendukung transfer antara dua *memory mapped device* tanpa intervensi CPU.

6.2. *Interface* Aplikasi I/O

Ketika suatu aplikasi ingin membuka data yang ada dalam suatu disk, sebenarnya aplikasi tersebut harus dapat membedakan jenis disk apa yang akan diaksesnya. Untuk mempermudah pengaksesan, sistem operasi melakukan standarisasi cara pengaksesan pada peralatan I/O. Pendekatan inilah yang dinamakan *interface* aplikasi I/O.

Interface aplikasi I/O melibatkan abstraksi, enkapsulasi, dan *software layering*. Abstraksi dilakukan dengan membagi-bagi detail peralatan-peralatan I/O ke dalam kelas-kelas yang lebih umum. Dengan adanya kelas-kelas yang umum ini, maka akan lebih mudah untuk membuat fungsi-fungsi standar (*interface*)

untuk mengaksesnya. Lalu kemudian adanya *device driver* pada masing-masing peralatan I/O, berfungsi untuk enkapsulasi perbedaan-perbedaan yang ada dari masing-masing anggota kelas-kelas yang umum tadi. *Device driver* mengenkapsulasi tiap - tiap peralatan I/O ke dalam masing-masing 1 kelas yang umum tadi (*interface* standar). Tujuan dari adanya lapisan *device driver* ini adalah untuk menyembunyikan perbedaan-perbedaan yang ada pada *device controller* dari subsistem I/O pada kernel.

Karena hal ini, subsistem I/O dapat bersifat independen dari *hardware*.

Karena subsistem I/O independen dari *hardware* maka hal ini akan sangat menguntungkan dari segi pengembangan *hardware*. Tidak perlu menunggu vendor sistem operasi untuk mengeluarkan support code untuk *hardware-hardware* baru yang akan dikeluarkan oleh vendor *hardware*.

6.2.1. Peralatan *Block* dan Karakter

Peralatan block diharapkan dapat memenuhi kebutuhan akses pada berbagai macam disk drive dan juga peralatan *block* lainnya. *Block device* diharapkan dapat memenuhi/mengerti perintah baca, tulis dan juga perintah pencarian data pada peralatan yang memiliki sifat *random-access*.

Keyboard adalah salah satu contoh alat yang dapat mengakses *stream*-karakter. *System call* dasar dari *interface* ini dapat membuat sebuah aplikasi mengerti tentang bagaimana cara untuk mengambil dan menuliskan sebuah karakter. Kemudian pada pengembangan lanjutannya, kita dapat membuat *library* yang dapat mengakses data/pesan per-baris.

6.2.2. Peralatan Jaringan

Karena adanya perbedaan dalam kinerja dan pengalamatan dari jaringan I/O, maka biasanya sistem operasi memiliki *interface* I/O yang berbeda dari baca, tulis dan pencarian pada disk. Salah satu yang banyak digunakan pada sistem operasi adalah *interface socket*.

Socket berfungsi untuk menghubungkan komputer ke jaringan. *System call* pada *socket* interface dapat memudahkan suatu aplikasi untuk membuat *local socket*, dan menghubungkannya ke *remote socket*. Dengan menghubungkan komputer ke socket, maka komunikasi antar komputer dapat dilakukan.

6.2.3. Jam dan *Timer*

Adanya jam dan timer pada *hardware* komputer, setidaknya memiliki tiga fungsi, memberi informasi waktu saat ini, memberi informasi lamanya waktu sebuah proses, sebagai trigger untuk suatu operasi pada suatu waktu. Fungsi fungsi ini sering digunakan oleh sistem operasi. Sayangnya, *system call* untuk pemanggilan fungsi ini tidak di-standarisasi antar sistem operasi *Hardware* yang mengukur waktu dan melakukan operasi *trigger* dinamakan *programmable interval timer*. Dia dapat di set untuk menunggu waktu tertentu dan kemudian melakukan interupsi. Contoh penerapannya ada pada *scheduler*, dimana dia akan melakukan interupsi yang akan memberhentikan suatu proses pada akhir dari bagian waktunya.

Sistem operasi dapat mendukung lebih dari banyak *timer request* daripada banyaknya jumlah *hardware timer*. Dengan kondisi seperti ini, maka kernel atau *device driver* mengatur list dari interupsi dengan urutan yang duluan datang yang duluan dilayani.

6.2.4. *Blocking* dan *Nonblocking* I/O

Ketika suatu aplikasi menggunakan sebuah blocking *system call*, eksekusi aplikasi itu akan

diberhentikan untuk sementara. aplikasi tersebut akan dipindahkan ke *wait queue*. Dan setelah *system call* tersebut selesai, aplikasi tersebut dikembalikan ke *run queue*, sehingga pengeksekusian aplikasi tersebut akan dilanjutkan. *Physical action* dari peralatan I/O biasanya bersifat *asynchronous*. Akan tetapi, banyak sistem operasi yang bersifat *blocking*, hal ini terjadi karena *blocking application* lebih mudah dimengerti dari pada *nonblocking application*.

6.3. *Kernel I/O Subsystem*

Kernel menyediakan banyak service yang berhubungan dengan *I/O*. Pada bagian ini, kita akan mendeskripsikan beberapa service yang disediakan oleh kernel *I/O subsystem*, dan kita akan membahas bagaimana caranya membuat infrastruktur *hardware* dan *device-driver*. Service yang akan kita bahasadalah *I/O scheduling, buffering, caching, spooling, reservasi device, error handling*.

6.3.1. *I/O Scheduling*

Untuk menjadualkan sebuah set permintaan *I/O*, kita harus menetukan urutan yang bagus untuk mengeksekusi permintaan tersebut. *Scheduling* dapat meningkatkan kemampuan sistem secara keseluruhan, dapat membagi device secara rata di antara

proses-proses, dan dapat mengurangi waktu tunggu rata-rata untuk menyelesaikan *I/O*. Ini adalah contoh sederhana untuk menggambarkan definisi di atas. Jika sebuah *arm* disk terletak di dekat permulaan disk, dan ada tiga aplikasi yang memblokir panggilan untuk membaca untuk disk tersebut. Aplikasi 1 meminta sebuah blok dekat akhir disk, aplikasi 2 meminta blok yang dekat dengan awal, dan aplikasi 3 meminta bagian tengah dari disk. Sistem operasi dapat mengurangi jarak yang harus ditempuh oleh *arm* disk dengan melayani aplikasi tersebut dengan urutan 2, 3, 1. Pengaturan urutan pekerjaan kembali dengan cara ini merupakan inti dari *I/O* scheduling. Sistem operasi mengembangkan implementasi scheduling dengan menetapkan antrian permintaan untuk tiap device. Ketika sebuah aplikasi meminta sebuah *blocking* sistem *I/O*, permintaan tersebut dimasukkan ke dalam antrian untuk device tersebut. *Scheduler I/O* mengatur urutan antrian untuk meningkatkan efisiensi dari sistem dan waktu respon rata-rata yang harus dialami oleh aplikasi. Sistem operasi juga mencoba untuk bertindak secara adil, seperti tidak ada aplikasi yang menerima service yang buruk, atau dapat seperti memberi prioritas service untuk permintaan penting yang ditunda. Contohnya, pemintaan dari subsistem mungkin akan mendapatkan prioritas lebih tinggi daripada permintaan dari aplikasi. Beberapa algoritma scheduling untuk disk *I/O* akan dijelaskan ada bagian *Disk Scheduling*.

Satu cara untuk meningkatkan efisiensi *I/O* subsistem dari sebuah komputer adalah dengan mengatur operasi *I/O*. Cara lain adalah dengan menggunakan tempat penyimpanan pada memori utama atau pada disk, melalui teknik yang disebut *buffering, caching, dan spooling*.

6.3.2. *Buffering*

Buffer adalah area memori yang menyimpan data ketika mereka sedang dipindahkan antara dua *device* atau antara *device* dan aplikasi. *Buffering* dilakukan untuk tiga buah alasan. Alasan pertama adalah untuk men-*cope* dengan kesalahan yang terjadi karena perbedaan kecepatan antara produsen dengan konsumen dari sebuah *stream data*. Sebagai contoh, sebuah file sedang diterima melalui modem dan ditujukan ke media penyimpanan di *hard disk*. Kecepatan modem tersebut kira-kira hanyalah 1/1000 daripada *hard disk*. Jadi *buffer* dibuat di dalam memori utama untuk mengumpulkan jumlah *byte* yang diterima dari modem. Ketika keseluruhan data di *buffer* sudah sampai, buffer tersebut dapat ditulis ke disk dengan operasi tunggal. Karena penulisan disk tidak terjadi dengan instan dan modem masih memerlukan tempat untuk menyimpan data yang berdatangan, maka dipakai 2 buah *buffer*. Setelah modem memenuhi *buffer* pertama, akan

terjadi request untuk menulis di disk. Modem kemudian mulai memenuhi *buffer* kedua sementara *buffer* pertama dipakai untuk penulisan ke disk. Pada saat modem sudah memenuhi *buffer* kedua, penulisan ke disk dari *buffer* pertama seharusnya sudah selesai, jadi modem akan berganti kembali memenuhi *buffer* pertama dan *buffer* kedua dipakai untuk menulis. Metode *double buffering* ini membuat pasangan ganda antara produsen dan konsumen sekaligus mengurangi kebutuhan waktu di antara mereka.

Alasan kedua dari *buffering* adalah untuk menyesuaikan device-device yang mempunyai perbedaan dalam ukuran transfer data. Hal ini sangat umum terjadi pada jaringan komputer, dimana *buffer* dipakai secara luas untuk fragmentasi dan pengaturan kembali pesan-pesan yang diterima. Pada bagian pengirim, sebuah pesan yang besar akan dipecah ke paket-paket kecil. Paket-paket tersebut dikirim melalui jaringan, dan penerima akan meletakkan mereka di dalam *buffer* untuk disusun kembali.

Alasan ketiga untuk *buffering* adalah untuk mendukung *copy semantics* untuk aplikasi *I/O*. Sebuah contoh akan menjelaskan apa arti dari *copy semantics*. Jika ada sebuah aplikasi yang mempunyai *buffer* data yang ingin dituliskan ke disk. Aplikasi tersebut akan memanggil sistem penulisan, menyediakan pointer ke *buffer*, dan sebuah *integer* untuk menunjukkan ukuran bytes yang ingin ditulis. Setelah pemanggilan tersebut, apakah yang akan terjadi jika aplikasi tersebut merubah isi dari *buffer*, dengan *copy semantics*, keutuhan data yang ingin ditulis sama dengan data waktu aplikasi ini memanggil sistem untuk menulis, tidak tergantung dengan perubahan yang terjadi pada *buffer*. Sebuah cara sederhana untuk sistem operasi untuk menjamin *copy semantics* adalah membiarkan sistem penulisan untuk mengkopi data aplikasi ke dalam *buffer* kernel sebelum mengembalikan kontrol kepada aplikasi. Jadi penulisan ke disk dilakukan pada *buffer* kernel, sehingga perubahan yang terjadi pada *buffer* aplikasi tidak akan membawa dampak apa-apa. Mengcopy data antara *buffer* kernel data aplikasi merupakan sesuatu yang umum pada sistem operasi, kecuali *overhead* yang terjadi karena operasi ini karena *clean semantics*. Kita dapat memperoleh efek yang sama yang lebih efisien dengan memanfaatkan virtual-memori mapping dan proteksi *copy-on-wire* dengan pintar.

6.3.3. *Caching*

Sebuah *cache* adalah daerah memori yang cepat yang berisikan data kopian. Akses ke sebuah kopian yang di-*cached* lebih efisien daripada akses ke data asli. Sebagai contoh, instruksi-instruksi dari proses yang sedang dijalankan disimpan ke dalam disk, dan

ter-*cached* di dalam memori *physical*, dan kemudian dicopy lagi ke dalam *cache secondary and primary* dari CPU. Perbedaan antara sebuah *buffer* dan ache adalah *buffer* dapat menyimpan satu-satunya informasi datanya sedangkan sebuah cache secara definisi hanya menyimpan sebuah data dari sebuah tempat untuk dapat diakses lebih cepat.

Caching dan *buffering* adalah dua fungsi yang berbeda, tetapi terkadang sebuah daerah memori dapat digunakan untuk keduanya. sebagai contoh, untuk menghemat *copy semantics* dan membuat scheduling *I/O* menjadi efisien, sistem operasi menggunakan *buffer* pada memori utama untuk menyimpan data.

Buffer ini juga digunakan sebagai cache, untuk meningkatkan efisiensi *I/O* untuk file yang digunakan secara bersama-sama oleh beberapa aplikasi, atau yang sedang dibaca dan ditulis secara berulang-ulang.

Ketika kernel menerima sebuah permintaan file *I/O*, kernel tersebut mengakses *buffer cache*untuk melihat apakah daerah memori tersebut sudah tersedia dalam memori utama. Jika iya, sebuah *physical disk I/O* dapat dihindari atau tidak dipakai. penulisan disk juga terakumulasi ke dalam *buffer cache* selama beberapa detik, jadi transfer yang besar akan dikumpulkan untuk mengefisiensikan *schedule* penulisan. Cara ini akan menunda penulisan untuk meningkatkan efisiensi *I/O* akan dibahas pada bagian *Remote File Access*.

6.3.4. *Spooling* dan *Reservasi Device*

Sebuah spool adalah sebuah *buffer* yang menyimpan *output* untuk sebuah *device*, seperti printer, yang tidak dapat menerima *interleaved data streams*. Walau pun printer hanya dapat melayani satu pekerjaan pada waktu yang sama, beberapa aplikasi dapat meminta printer untuk mencetak, tanpa harus mendapatkan hasil *output* mereka tercetak secara bercampur. Sistem operasi akan menyelesaikan masalah ini dengan meng-*intercept* semua output kepada printer. Tiap *output* aplikasi sudah di-*spooled* ke disk file yang berbeda. Ketika sebuah aplikasi selesai mengeprint, sistem spooling akan melanjutkan ke antrian berikutnya. Di dalam beberapa sistem operasi, *spooling* ditangani oleh sebuah sistem proses daemon. Pada sistem operasi yang lain, sistem ini ditangani oleh *in-kernel thread*. Pada kedua kasus, sistem operasi menyediakan *interface*kontrol yang membuat *users and system administrator* dapat menampilkan antrian tersebut, untuk mengenyahkan antrian-antrian yang tidak diinginkan sebelum mulai di-print.

Untuk beberapa *device*, seperti *drive tape*dan printer tidak dapat me-*multiplex* permintaan *I/O* dari beberapa aplikasi. *Spooling*

merupakan salah satu cara untuk mengatasi masalah ini. Cara lain adalah dengan membagi koordinasi untuk *multiple concurrent* ini. Beberapa sistem operasi menyediakan dukungan untuk akses *device* secara eksklusif, dengan mengalokasikan proses ke *device idle* dan membuang *device* yang sudah tidak diperlukan lagi. Sistem operasi lainnya memaksakan limit suatu file untuk menangani *device* ini. Banyak sistem operasi menyediakan fungsi yang membuat proses untuk menangani koordinat *exclusive* akses diantara mereka sendiri.

6.3.5. *Error Handling*

Sebuah sistem operasi yang menggunakan *protected memory* dapat menjaga banyak kemungkinan *error* akibat *hardware* mau pun aplikasi. *Devices* dan transfer *I/O* dapat gagal dalam banyak cara, bisa karena alasan *transient*, seperti overloaded pada *network*, mau pun alasan *permanen* yang seperti kerusakan yang terjadi pada *disk controller*. Sistem operasi seringkali dapat mengkompensasikan untuk kesalahan *transient*. Seperti, sebuah kesalahan baca pada disk akan mengakibatkan pembacaan ulang kembali dan sebuah kesalahan pengiriman pada *network* akan mengakibatkan pengiriman ulang apabila protokolnya diketahui. Akan tetapi untuk kesalahan *permanent*, sistem operasi pada umumnya tidak akan bisa mengembalikan situasi seperti semula.

Sebuah ketentuan umum, yaitu sebuah sistem *I/O* akan mengembalikan satu *bit* informasi tentang status panggilan tersebut, yang akan menandakan apakah proses tersebut berhasil atau gagal. Sistem operasi pada UNIX menggunakan *integer* tambahan yang dinamakan *errno* untuk mengembalikan kode kesalahan sekitar 1 dari 100 nilai yang mengindikasikan sebab dari kesalahan tersebut. Akan tetapi, beberapa perangkat keras dapat menyediakan informasi kesalahan yang detail, walau pun banyak sistem operasi yang tidak mendukung fasilitas ini.

6.3.6. *Kernel Data Structure*

Kernel membutuhkan informasi *state* tentang penggunakan komponen *I/O*. Kernel menggunakan banyak struktur yang mirip untuk melacak koneksi jaringan, komunikasi karakter-*device*, dan aktivitas *I/O* lainnya.

UNIX menyediakan akses sistem file untuk beberapa entiti, seperti file *user*, *raw devices*, dan alamat tempat proses. Walau pun tiap entiti ini didukung sebuah operasi baca, *semantics*-nya berbeda untuk tiap entiti. Seperti untuk membaca file user, kernel perlu memeriksa *buffer cache* sebelum memutuskan apakah akan melaksanakan *I/O* disk. Untuk membaca sebuah *raw disk*, kernel

perlu untuk memastikan bahwa ukuran permintaan adalah kelipatan dari ukuran sektor disk, dan masih terdapat di dalam batas sektor. Untuk memproses citra, cukup perlu untuk mengkopi data ke dalam memori. UNIX mengkapsulasikan perbedaan-perbedaan ini di dalam struktur yang uniform dengan menggunakan teknik *object oriented*.

Beberapa sistem operasi bahkan menggunakan metode *object oriented* secara lebih extensif. Sebagai contoh, Windows NT menggunakan implementasi *message-passing* untuk *I/O*. Sebuah permintaan *I/O* akan dikonversikan ke sebuah pesan yang dikirim melalui kernel kepada *I/O manager* dan kemudian ke *device driver*, yang masing-masing bisa mengubah isi pesan. Untuk output, isi *message* adalah data yang akan ditulis. Untuk input, *message* berisikan *buffer* untuk menerima data. Pendekatan *message-passing* ini dapat menambah *overhead*, dengan perbandingan dengan teknik prosedural yang men-*share* struktur data, tetapi akan mensederhanakan struktur dan *design* dari sistem *I/O* tersebut dan menambah fleksibilitas.

Kesimpulannya, subsistem *I/O* mengkoordinasi kumpulan-kumpulan service yang banyak sekali, yang tersedia dari aplikasi mau pun bagian lain dari kernel. Subsistem I/O mengawasi:

1. Manajemen nama untuk file dan *device*.
2. Kontrol akses untuk file dan *device*.
3. Kontrol operasi, contoh: model yang tidak dapat dikenali.
4. Alokasi tempat sistem file.
5. Alokasi *device*.
6. *Buffering, caching, spooling*.
7. *I/O scheduling*
8. Mengawasi status *device*, *error handling*, dan kesalahan dalam *recovery*.
9. Konfigurasi dan utilisasi *driver device*.

6.4. Penanganan Permintaan I/O

Di bagian sebelumnya, kita mendeskripsikan *handshaking* antara *device driver* dan *device controller*, tapi kita tidak menjelaskan bagaimana Sistem Operasi menyambungkan permintaan aplikasi untuk menyiapkan jaringan menuju sektor disk yang spesifik.

Sistem Operasi yang modern mendapatkan fleksibilitas yang signifikan dari tahapan-tahapan tabel *lookup* di jalur diantara permintaan dan *physical device controller*. Kita dapat mengenalkan *device* dan *driver* baru ke komputer tanpa harus meng-*compile* ulang kernelnya. Sebagai fakta, ada beberapa sistem operasi yang

mampu untuk me-*load device drivers* yang diinginkan. Pada waktu *boot*, sistem mula-mula meminta bus piranti keras untuk menentukan *device* apa yang ada, kemudian sistem me-*load* ke dalam *driver* yang sesuai; baik sesegera mungkin, mau pun ketika diperlukan oleh sebuah permintaan *I/O*.

UNIX Sistem V mempunyai mekanisme yang menarik, yang disebut *streams*, yang membolehkan aplikasi untuk men-*assemble pipeline* dari kode *driver* secara dinamis. Sebuah *stream* adalah sebuah koneksi *full duplex* antara sebuah *device driver* dan sebuah proses *user-level*. *Stream* terdiri atas sebuah *stream head* yang merupakan antarmuka dengan *user process*, sebuah *driver end* yang mengontrol *device*, dan nol atau lebih *stream modules* diantara mereka. *Modules* dapat didorong ke stream untuk menambah fungsionalitas di sebuah *layered fashion*. Sebagai gambaran sederhana, sebuah proses dapat membuka sebuah alat port serial melalui sebuah *stream*, dan dapat mendorong ke sebuah modul untuk memegang edit input. *Stream* dapat digunakan untuk interproses dan komunikasi jaringan. Faktanya, di Sistem V, mekanisme soket diimplementasikan dengan stream.

Berikut dideskripsikan sebuah *lifecycle* yang tipikal dari sebuah permintaan pembacaan blok.

1. Sebuah proses mengeluarkan sebuah *blocking read system call* ke sebuah file deskriptor dari berkas yang telah dibuka sebelumnya.
2. Kode *system-call* di kernel mengecek parameter untuk kebenaran. Dalam kasus input, jika data telah siap di buffer cache, data akan dikembalikan ke proses dan permintaan *I/O* diselesaikan.
3. Jika data tidak berada dalam *buffer cache*, sebuah *physical I/O* akan bekerja, sehingga proses akan dikeluarkan dari antrian jalan (*run queue*) dan diletakkan di antrian tunggu (*wait queue*) untuk alat, dan permintaan *I/O* pun dijadwalkan. Pada akhirnya, subsistem *I/O* mengirimkan permintaan ke *device driver*. Bergantung pada sistem operasi, permintaan dikirimkan melalui *call* subrutin atau melalui pesan *in-kernel*.
4. *Device driver* mengalokasikan ruang *buffer* pada kernel untuk menerima data, dan menjadwalkan *I/O*. Pada akhirnya, *driver* mengirim perintah ke *device controller* dengan menulis ke register *device control*.
5. *Device controller* mengoperasikan piranti keras *device* untuk melakukan transfer data.
6. *Driver* dapat menerima status dan data, atau dapat menyiapkan transfer DMA ke memori kernel. Kita mengasumsikan bahwa transfer diatur oleh sebuah *DMA controller*, yang meggunakan interupsi ketika transfer selesai.

7. *Interrupt handler* yang sesuai menerima interupsi melalui tabel vektor-interupsi, menyimpan sejumlah data yang dibutuhkan, menandai *device driver*, dan kembali dari interupsi.
8. *Device driver* menerima tanda, menganalisa permintaan *I/O* mana yang telah diselesaikan, menganalisa status permintaan, dan menandai subsistem *I/O* kernel yang permintaannya telah terselesaikan.
9. Kernel mentransfer data atau mengembalikan kode ke ruang alamat dari proses permintaan, dan memindahkan proses dari antrian tunggu kembali ke antrian siap.
10. Proses tidak diblok ketika dipindahkan ke antrian siap. Ketika penjadwal (*scheduler*)

    mengembalikan proses ke CPU, proses meneruskan eksekusi pada penyelesaian dari *system call*.

6.5. Kinerja *I/O*

6.5.1. Pengaruh *I/O* pada Kinerja

I/O sangat berpengaruh pada kinerja sebuah sistem komputer. Hal ini dikarenakan *I/O* sangat menyita CPU dalam pengeksekusian *device driver* dan penjadwalan proses, demikian sehingga alih konteks yang dihasilkan membebani CPU dan cache perangkat keras. Selain itu, *I/O* juga memenuhi bus memori saat mengkopi data antara *controller* dan *physical memory*, serta antara buffer pada kernel dan *application space data*. Karena besarnya pengaruh *I/O* pada kinerja komputer inilah bidang pengembangan arsitektur komputer sangat memperhatikan masalah-masalah yang telah disebutkan diatas.

6.5.2. Cara Meningkatkan Efisiensi *I/O*

1. Menurunkan jumlah alih konteks.
2. Mengurangi jumlah pengkopian data ke memori ketika sedang dikirimkan antara *device* dan aplikasi.
3. Mengurangi frekuensi interupsi, dengan menggunakan ukuran transfer yang besar, *smart controller*, dan *polling*.
4. Meningkatkan *concurrency* dengan *controller* atau *channel* yang mendukung DMA.

5. Memindahkan kegiatan processing ke perangkat keras, sehingga operasi kepada *device controller* dapat berlangsung bersamaan dengan CPU.
6. Menyeimbangkan antara kinerja CPU, *memory subsystem, bus*, dan *I/O*.

6.5.3. Implementasi Fungsi I/O

Pada dasarnya kita mengimplementasikan algoritma *I/O* pada level aplikasi. Hal ini dikarenakan kode aplikasi sangat fleksible, dan *bugs* aplikasi tidak mudah menyebabkan sebuah sistem *crash*. Lebih lanjut, dengan mengembangkan kode pada level aplikasi, kita akan menghindari kebutuhan untuk *reboot* atau *reload device driver* setiap kali kita mengubah kode. Implementasi pada level aplikasi juga bisa sangat tidak efisien. Tetapi, karena *overhead* dari alih konteks dan karena aplikasi tidak bisa mengambil keuntungan dari struktur data kernel internal dan fungsionalitas dari kernel (misalnya, efisiensi dari kernel *messaging, threading dan locking*.

Pada saat algoritma pada level aplikasi telah membuktikan keuntungannya, kita mungkin akan mengimplementasikannya di kernel. Langkah ini bisa meningkatkan kinerja tetapi perkembangannya dari kerja jadi lebih menantang, karena besarnya kernel dari sistem operasi, dan kompleksnya sistem sebuah perangkat lunak. Lebih lanjut , kita harus men-*debug* keseluruhan dari implementasi *in-kernel* untuk menghindari korupsi sebuah data dan sistem *crash*.

Kita mungkin akan mendapatkan kinerja yang optimal dengan menggunakan implementasi yang special pada perangkat keras, selain dari *device* atau *controller*. Kerugian dari implementasi perangkat keras termasuk kesukaran dan biaya yang ditanggung dalam membuat kemajuan yang lebih baik dalam mengurangi *bugs*, perkembangan waktu yang maju dan fleksibilitas yang meningkat. Contohnya, RAID *controller* pada perangkat keras mungkin tidak akan menyediakan sebuah efek pada kernel untuk mempengaruhi urutan atau lokasi dari individual *block reads* dan *write*, meski pun kernel tersebut mempunyai informasi yang spesial mengenai *workload* yang dapat mengaktifkan kernel untuk meningkatkan kinerja dari *I/O*.

6.6. Struktur Disk

Disk menyediakan penyimpanan sekunder bagi sistem komputer modern. *Magnetic tape* sebelumnya digunakan sebagai media penyimpanan sekunder, tetapi waktu aksesnya lebih lambat dari

disk. Oleh karena itu, sekarang *tape* digunakan terutama untuk *backup*, untuk penyimpanan informasi yang tidak sering, sebagai media untuk mentransfer infromasi dari satu sistem ke sistem yang lain, dan untuk menyimpan sejumlah data yang terlalu besar untuk sistem disk.

Disk drive modern dialamatkan sebagai suatu array satu dimensi yang besar dari blok lojik, dimana blok lojik merupakan unit terkecil dari transfer. Ukuran dari blok lojik biasanya adalah 512 *bytes*, walau pun sejumlah disk dapat diformat di level rendah (*low level formatted*) untuk memilih sebuah ukuran blok lojik yang berbeda, misalnya 1024 *bytes*.

Array satu dimensi dari blok lojik dipetakan ke bagian dari disk secara sekuensial. Sektor 0 adalah sektor pertama dari trek pertama di silinder paling luar (*outermost cylinder*). Pemetaan kemudian memproses secara berurutan trek tersebut, kemudian melalui trek selanjutnya di silinder tersebut, dan kemudian sisa silinder dari yang paling luar sampai yang paling dalam.

Dengan menggunakan pemetaan, kita dapat minimal dalam teori mengubah sebuah nomor blok logikal ke sebuah alamat disk yang bergaya lama (*old-style disk address*) yang terdiri atas sebuah nomor silinder, sebuah nomor trek di silinder tersebut, dan sebuah nomor sektor di trek tersebut. Dalam prakteknya, adalah sulit untuk melakukan translasi ini, dengan 2 alasan. Pertama, kebanyakan disk memiliki sejumlah sektor yang rusak, tetapi pemetaan menyembunyikan hal ini dengan mensubstitusikan dengan sektor yang dibutuhkan dari mana-mana di dalam disk. Kedua, jumlah dari sektor per trek tidaklah konstan. Semakin jauh sebuah trek dari tengah disk, semakin besar panjangnya, dan juga semakin banyak sektor yang dipunyainya. Oleh karena itu, disk modern diatur menjadi zona-zona silinder. Nomor sektor per trek adalah konstan dalam sebuah zona. Tetapi seiring kita berpindah dari zona dalam ke zona luar, nomor sektor per trek bertambah. Trek di zona paling luar tipikalnya mempunyai 40 persen sektor lebih banyak daripada trek di zona paling dalam.

Nomor sektor per trek telah meningkat seiring dengan peningkatan teknologi disk, dan adalah lazim untuk mempunyai lebih dari 100 sektor per trek di zona yang lebih luar dari disk. Dengan analogi yang sama, nomor silinder per disk telah meningkat, dan sejumlah ribuan silinder adalah tak biasa.

6.7. Penjadualan Disk

Salah satu tanggung jawab sistem operasi adalah menggunakan *hardware* dengan efisien. Khusus untuk *disk drives*, efisiensi yang dimaksudkan di sini adalah dalam hal waktu akses yang cepat dan

aspek *bandwidth disk*. Waktu akses memiliki dua komponen utama yaitu waktu pencarian dan waktu rotasi disk. Waktu pencarian adalah waktu yang dibutuhkan *disk arm* untuk menggerakkan *head* ke bagian silinder *disk* yang mengandung sektor yang diinginkan. Waktu rotasi *disk* adalah waktu tambahan yang dibutuhkan untuk menunggu rotasi atau perputaran *disk*, sehingga sektor yang diinginkan dapat dibaca oleh *head*. Pengertian *Bandwidth* adalah total jumlah *bytes* yang ditransfer dibagi dengan total waktu antara permintaan pertama sampai seluruh bytes selesai ditransfer. Untuk meningkatkan kecepatan akses dan bandwidth, kita dapat melakukan penjadualan pelayanan atas permintaan *I/O* dengan urutan yang tepat.

Sebagaimana kita ketahui, jika suatu proses membutuhkan pelayanan *I/O* dari atau menuju *disk*, maka proses tersebut akan melakukan *system call* ke sistem operasi. Permintaan tersebut membawa informasi-informasi antara lain:

1. Apakah operasi *input* atau *output*.
2. Alamat *disk* untuk proses tersebut.
3. Alamat memori untuk proses tersebut
4. Jumlah *bytes* yang akan ditransfer

Jika *disk drive* beserta *controller* tersedia untuk proses tersebut, maka proses akan dapat dilayani dengan segera. Jika ternyata *disk drive* dan *controller* tidak tersedia atau sedang sibuk melayani proses lain,maka semua permintaan yang memerlukan pelayanan *disk* tersebut akan diletakkan pada suatu antrian penundaan permintaan untuk *disk* tersebut. Dengan demikian, jika suatu permintaan telah dilayani, maka sistem operasi memilih permintaan tertunda dari antrian yang selanjutnya akan dilayani.

6.7.1. Penjadualan FCFS

Bentuk paling sederhana dalam penjadualan *disk* adalah dengan sistem antrian (queue) atau *First Come First Served* (FCFS). Algoritma ini secara intrinsik bersifat adil, tetapi secara umum algoritma ini pada kenyataannya tidak memberikan pelayanan yang paling cepat. Sebagai contoh, antrian permintaan pelayanan *disk* untuk proses *I/O* pada blok dalam silinder adalah sebagai berikut: 98, 183, 37, 122, 14, 124, 65, 67. Jika *head* pada awalnya berada pada 53, maka *head* akan bergerak dulu dari 53 ke 98, kemudian 183, 37, 122, 14, 124, 65, dan terakhir 67, dengan total pergerakan *head* sebesar 640 silinder.

Permasalahan dengan menggunakan penjadualan jenis ini dapat diilustrasikan dengan pergerakan dari 122 ke 14 dan kembali lagi ke 124. Jika permintaan terhadap silinder 37 dan 14 dapat

dikerjakan/ dilayani secara bersamaan, baik sebelum mau pun setelah permintaan 122 dan 124, maka pergerakan total *head* dapat dikurangi secara signifikan, sehingga dengan demikian pendayagunaan akan meningkat.

6.7.2. Penjadualan SSTF

Sangat beralasan jika kita menutup semua pelayanan pada posisi *head* saat ini, sebelum menggerakkan *head* ke tempat lain yang jauh untuk melayani suatu permintaan. Asumsi ini mendasari algoritma penjadualan kita yang kedua yaitu *shortest-seek-time-first* (SSTF). Algoritma ini memilih permintaan dengan berdasarkan waktu pencarian atau *seek time* paling minimum dari posisi *head* saat itu. Karena waktu pencarian meningkat seiring dengan jumlah silinder yang dilewati oleh *head*, maka SSTF memilih permintaan yang paling dekat posisinya di *disk* terhadap posisi *head* saat itu.

Perhatikan contoh antrian permintaan yang kita sajikan pada penjadualan FCFS, permintaan paling dekat dengan posisi *head* saat itu (53) adalah silinder 65. Jika kita enuhi permintaan 65, maka yang terdekat berikutnya adalah silinder 67. Dari 67, silinder 37 letaknya lebih dekat ke 67 dibandingkan silinder 98, jadi 37 dilayani duluan. Selanjutnya, dilanjutkan ke silinder 14, 98, 122, 124, dan terakhir adalah 183. Metode penjadualan ini hanya menghasilkan total pergerakan *head* sebesar 236 silinder -- kira-kira sepertiga dari yang dihasilkan penjadualan FCFS. Algoritma SSTF ini memberikan peningkatan yang cukup signifikan dalam hal pendayagunaan atau *performance* sistem.

Penjadualan SSTF merupakan salah satu bentuk dari penjadualan *shortest-job-first* (SJF), dan karena itu maka penjadualan SSTF juga dapat mengakibatkan *starvation* pada suatu saat tertentu. Kita ketahui bahwa permintaan dapat datang kapan saja. Anggap kita memiliki dua permintaan dalam antrian, yaitu untuk silinder 14 dan 186. Selama melayani permintaan 14, kita anggap ada permintaan baru yang letaknya dekat dengan 14. Karena letaknya lebih dekat ke 14, maka permintaan ini akan dilayani dulu sementara permintaan 186 menunggu gilirannya. Jika kemudian berdatangan lagi permintaan-permintaan yang letaknya lebih dekat dengan permintaan terakhir yang dilayani jika dibandingkan dengan 186, maka permintaan 186 bisa saja menunggu sangat lama. Kemudian jika ada lagi permintaan yang lebih jauh dari 186, maka juga akan menunggu sangat lama untuk dapat dilayani.

Walau pun algoritma SSTF secara substansial meningkat jika dibandingkan dengan FCFS, tetapi algoritma SSTF ini tidak optimal. Seperti contoh diatas, kita dapat menggerakkan *head* dari

53 ke 37, walau pun bukan yang paling dekat, kemudian ke 14, sebelum menuju 65, 67, 98, 122, dan 183. Strategi ini dapat mengurangi total gerakan *head* menjadi 208 silinder.

6.7.3. Penjadualan SCAN

Pada algoritma SCAN, pergerakan *disk arm* dimulai dari salah satu ujung *disk*, kemudian bergerak menuju ujung yang lain sambil melayani permintaan setiap kali mengunjungi masing-masing silinder. Jika telah sampai di ujung *disk*, maka *disk arm* bergerak berlawanan arah, kemudian mulai lagi melayani permintaan-permintaan yang muncul. Dalam hal ini *disk arm* bergerak bolak-balik melalui *disk*.

Kita akan menggunakan contoh yang sudah dibarikan diatas. Sebelum melakukan SCAN untuk melayani permintaan-permintaan 98, 183, 37, 122, 14, 124, 65, dan 67, kita harus mengetahui terlebih dahulu pergerakan *head* sebagai langkah awal dari 53. Jika *disk arm* bergerak menuju 0, maka *head* akan melayani 37 dan kemudian 14. Pada silinder 0, *disk arm* akan bergerak berlawanan arah dan bergerak menuju ujung lain dari *disk* untuk melayani permintaan 65, 67, 98, 122, 124, dan 183. Jika permintaan terletak tepat pada *head* saat itu, maka akan dilayani terlebih dahulu, sedangkan permintaan yang datang tepat dibelakang *head* harus menunggu dulu *head* mencapai ujung *disk*, berbalik arah, baru kemudian dilayani.

Algoritma SCAN ini disebut juga algoritma lift/ elevator, karena kelakuan *disk arm* sama seperti elevator dalam suatu gedung, melayani dulu orang-orang yang akan naik ke atas, baru kemudian berbalik arah untuk melayani orang-orang yang ingin turun ke bawah.

Kelemahan algoritma ini adalah jika banyak permintaan terletak pada salah satu ujung *disk*, sedangkan permintaan yang akan dilayani sesuai arah *arm disk* jumlahnya sedikit atau tidak ada, maka mengapa permintaan yang banyak dan terdapat pada ujung yang berlawanan arah dengan gerakan *disk arm* saat itu tidak dilayani duluan? Ide ini akan mendasari algoritma penjadualan berikut yang akan kita bahas.

6.7.4. Penjadualan C-SCAN

Circular-SCAN adalah varian dari algoritma SCAN yang sengaja didesain untuk menyediakan waktu tunggu yang sama. Seperti halnya SCAN, C-SCAN akan menggerakkan *head* dari satu ujung *disk* ke ujung lainnya sambil melayani permintaan yang terdapat selama pergerakan tersebut. Tetapi pada saat *head* tiba pada salah satu ujung, maka *head* tidak berbalik arah dan melayani permintaan-permintaan, melainkan akan kembali ke ujung *disk*

asal pergerakannya. Jika *head* mulai dari ujung 0, maka setelah tiba di ujung disk yang lainnya, maka *head* tidak akan berbalik arah menuju ujung 0, tetapi langsung bergerak ulang dari 0 ke ujung satunya lagi.

6.7.5. Penjadualan LOOK

Perhatikan bahwa SCAN dan C-SCAN menggerakkan *disk arm* melewati lebar seluruh *disk*. Pada kenyataanya algoritma ini tidak diimplementasikan demikian (pergerakan melewati lebar seluruh *disk*). Pada umumnya, *arm disk* bergerak paling jauh hanya pada permintaan terakhir pada masing-masin arah pergerakannya. Kemudian langsung berbalik arah tanpa harus menuju ujung *disk*. Versi SCAN dan C-SCAN yang berprilaku seperti ini disebut LOOK SCAN dan LOOK C-SCAN, karena algoritma ini melihat dulu permintaan-permintaan sebelum melanjutkan arah pergerakannya.

6.7.6. Pemilihan Algoritma Penjadualan *Disk*

Dari algoritma-algoritma diatas, bagaimanakah kita memilih algoritma terbaik yang akan digunakan? SSTF lebih umum dan memiliki prilaku yang lazim kita temui. SCAN dan C-SCAN memperlihatkan kemampuan yang lebih baik bagi sistem yang menempatkan beban pekerjaan yang berat kepada *disk*, karena algoritma tersebut memiliki masalah *starvation* yang paling sedikit. Untuk antrian permintaan tertentu, mungkin saja kita dapat mendefinisikan urutan akses dan pengambilan data dari *disk* yang optimal, tapi proses komputasi membutuhkan penjadualan optimal yang tidak kita dapatkan pada SSTF atau SCAN.

Dengan algoritma penjadualan yang mana pun, kinerja sistem sangat tergantung pada jumlah dan tipe permintaan. Sebagai contoh, misalnya kita hanya memiliki satu permintaan, maka semua algoritma penjadualan akan dipaksa bertindak sama, karena algoritma-algoritma tersebut hanya punya satu pilihan dari mana menggerakkan *disk head*: semuanya berprilaku seperti algoritma penjadualan FCFS.

Perlu diperhatikan pula bahwa pelayanan permintaan *disk* dapat dipengaruhi pula oleh metode alokasi file. Sebuah program yang membaca alokasi file secara terus menerus mungkin akan membuat beberapa permintaan yang berdekatan pada *disk*, menyebabkan pergerakan *head* menjadi terbatas. File yang memiliki link atau indeks, dilain pihak, mungkin juga memasukkan blok-blok yang tersebar luas pada *disk*, menyebabkan pergerakan *head* yang sangat besar.

Lokasi blok-blok indeks dan *directory* juga tidak kalah penting. Karena file harus dibuka sebelum digunakan, proses pembukaan file membutuhkan pencarian pada struktur *directory*, dengan demikian *directory* akan sering diakses. Kita anggap catatan *directory* berada pada awal silinder, sedangkan data file berada pada silinder terakhir. Pada kasus ini, *disk head* harus bergerak melewati sepanjang lebar *disk*. Membuat tempat penyimpanan sementara dari blok-blok indeks dan *directory* ke dalam memori dapat membantu mengurangi pergerakan *disk arm*, khususnya untuk permintaan membaca *disk*.

Karena kerumitan inilah, maka algoritma penjadualan *disk* harus ditulis dalam modul terpisah dari sistem operasi, jadi dapat saling mengganti dengan algoritma lain jika diperlukan. Baik SSTF mau pun LOOK keduanya merupakan pilihan yang paling masuk akal sebagai algoritma yang paling dasar.

6.8. Managemen Disk

6.8.1. Memformat Disk

Sebuah disk magnetik yang baru sebenarnya hanyalah sebuah *slate* kosong yang berupa piringan magnetik untuk menyimpan sesuatu. Sebelum disk tersebut dapat menyimpan data, harus dilakukan proses *low-level formatting*/ *physical formatting*, yaitu membagi disk menjadi beberapa sektor dan mengisinya dengan struktur data tertentu (biasanya *header, area data*, dan *trailer*) agar dapat dibaca dan ditulis oleh *disk controller*.

Salah satu informasi yang dibutuhkan oleh *disk controller* adalah *error-correcting code* (ECC). Disebut seperti itu karena jika terdapat satu atau dua bit data yang *corrupt, controller* dapat mengidentifikasi bit mana yang berubah dan mengoreksi nya. Proses ini otomatis dilakukan oleh *controller* setiap membaca atau menulis pada disk.

Low-level formatting berfungsi agar pihak manufaktur dapat mengetes disk dan menginisialisasi mapping dari lojikal nomor blok ke pendeteksi sektor kosong. Semakin besar ukuran sektor yang diformat, semakin sedikit sektor yang dapat diisi pada masing-masing *track* dan semakin sedikit *header* dan *trailer* yang ditulis pada setiap *track*. Hal ini berarti ruang yang dapat digunakan untuk data semakin besar.

Agar disk dapat menggunakan suatu berkas, sistem operasi membutuhkan untuk menyimpan struktur datanya pada disk. Langkah pertama adalah membagi disk menjadi satu/ lebih silinder (*partition*), sehingga sistem operasi dapat memperlakukannya sebagai disk yang terpisah. Langkah kedua adalah logical formatting, atau membuat sistem berkas. Pada

langkah ini, sistem operasi menyimpan struktur data yang telah diinisialisasi ke disk.

Raw I/O adalah *array* pada blok lojikal yang memiliki kemampuan untuk menggunakan suatu partisi disk tanpa struktur data dari sistem berkas. Dengan partisi *raw* ini, untuk beberapa aplikasi tertentu akan lebih efisien dari segi penyimpanan. Tetapi kebanyakan aplikasi akan berjalan lebih baik dengan servis sistem berkas biasa.

6.8.2. *Boot Block*

Ketika pertama kali menjalankan komputer, dibutuhkan program yang sudan diinisialisasi, yaitu *bootstrap*. Yang diinisialisasi adalah segala aspek sistem, dari *CPU register* sampai *device controller* dan isi dari *main memory*, kemudian menjalankan sistem operasi. Untuk itu *bootstrap* mencari *kernel* sistem operasi pada disk, me-*load*-nya ke memori, dan menggunakan alamat yang telah diinisialisasi untuk mulai menjalankan sistem operasi.

Hampir semua komputer menyimpan *bootstrap* pada *Read-Only Memory* (ROM). Alasannya karena ROM tidak membutuhkan inisialisasi dan berada pada lokasi yang tetap dimana prosesor tetap dapat mengeksekusinya ketika komputer baru dinyalakan/ di-reset. Kelebihan lainnya karena ROM *read-only*, ia tidak dapat terkena virus. Tetapi masalah yang timbul adalah jika kita mengubah kode *bootstrap* berarti mengubah *chip* ROM juga. Untuk mengatasinya, sistem menyimpan *bootstrap loader* di ROM, yang hanya berfungsi untuk memasukkan seluruh program *bootstrap* dari disk. *Boot blocks* adalah suatu partisi untuk menyimpan seluruh program *bootstrap*. *Boot disk* atau *system disk* adalah disk yang memiliki partisi *boot*.

6.8.3. *Bad Blocks*

Bad blocks adalah satu/lebih sektor yang rusak pada suatu disk. Pada disk sederhana, *bad blocks* diatasi secara manual. Untuk disk yang lebih kompleks seperti disk SCSI, *bad blocks* diatasi dengan *sector sparing* atau *forwarding*, yaitu *controller* dapat mengganti sektor yang rusak dengan sebuah sektor yang terpisah. Alternatif lainnya adalah mengganti sektor tersebut dengan cara *sector slipping*.

Mengganti blok yang rusak bukan sepenuhnya merupakan proses yang otomatis, karena data-data yang tersimpan sebelum nya akan terhapus.

6.9. Penanganan *Swap-Space*

Penanganan (*management*) *swap-space* (tempat pertukaran; tetapi karena istilah *swap-space* sudah umum dipakai, maka untuk seterusnya kita tetap memakai istilah *swap-space*) adalah salah satu dari *low-level* task pada sebuah sistem operasi. Memori Virtual menggunakan *disk space* sebagai perpanjangan (atau *space* tambahan) dari memori utama. Karena kecepatan akses disk lebih lambat daripada kecepatan akses memori, menggunakan *swap-space* akan mengurangi performa sistem secara signifikan. Tujuan utama dari perancangan dan implementasi *swap-space* adalah untuk menghasilkan kinerja memori virtual yang optimal. Dalam sub-bab ini, kita akan membicarakan bagaimana *swap-space* digunakan, dimana letak *swap-space* pada disk, dan bagaimana penanganan *swap-space*.

6.9.1. Penggunaan *Swap-Space*

Penggunaan *swap-space* pada berbagai macam sistem operasi berbeda-beda, tergantung pada algoritma *memory management* yang diimplementasikan. Sebagai contoh, sistem yang mengimplementasikan *swapping* mungkin akan menggunakan *swap-space* untuk menyimpan (dan mengerjakan) sebuah proses, termasuk segmen kode dan datanya. Sistem yang menggunakan *paging* hanya akan menyimpan *page* (atau "halaman " proses) yang sudah dikeluarkan dari memori utama. Besarnya *swap-space* yang dibutuhkan sebuah sistem bermacam-macam, tergantung dari banyaknya *physical memory* (RAM, seperti EDO DRAM, SDRAM, RD RAM), memori virtual yang disimpan di *swap-space*, dan caranya memori virtual digunakan. Besarnya bervariasi, antara beberapa megabytes sampai ratusan megabytes atau lebih.

Beberapa sistem operasi, seperti UNIX, menggunakan *swap-space* sebanyak yang diperlu kan. *Swap-space* ini biasanya disimpan dalam beberapa *disk* yang terpisah, jadi beban yang diterima oleh sistem I/O dari *paging* dan *swapping* bisa didistribusikan ke berbagai I/O *device* pada sistem.

Harap dicatat bahwa menyediakan *swap-space* yang berlebih lebih aman daripada kekurangan *swap-space*, karena bila kekurangan maka ada kemungkinan sistem terpaksa menghentikan sebuah atau lebih proses atau bahkan membuat sistem menjadi *crash*. *Swap-space* yang berlebih memang membuang *disk space* yang sebenarnya bisa digunakan untuk menyimpan berkas (*file*), tapi setidaknya tidakmenimbulkan resiko yang lain.

6.9.2. Lokasi *Swap-Space*

Ada dua tempat dimana *swap-space* bisa berada: *swap-space* bisa diletakkan pada partisi yang sama dengan sistem operasi, atau

pada partisi yang berbeda. Apabila *swap-space* yang dipakai hanya berupa sebuah berkas yang besar di dalam sistem berkas, maka sistem berkas yang dipakai bisa digunakan untuk membuat, menamakan, dan mengalokasikan tempat *swap-space*. Maka dari itu, pendekatan seperti ini mudah untuk diimplementasikan. Sayangnya, juga tidak efisien. Menelusuri struktur direktori dan struktur data alokasi disk memakan waktu, dan berpotensi untuk mengakses disk lebih banyak dari yang diperlukan. Fragmentasi eksternal bisa membuat *swapping* lebih lama dengan memaksakan pencarian sekaligus banyak (*multiple seeks*) ketika sedang membaca atau menulis sebuah proses. Kita bisa meningkatkan performa dengan meng-*cache* informasi lokasi blok pada *physical memory*, dan dengan menggunakan aplikasi khusus untuk mengalokasikan blok-blok yang *contiguous* (tidak terputus) untuk berkas *swap*-nya, dengan waktu tambahan untuk menelusuri struktur data *file-system* masih tetap ada.

Metode yang lebih umum adalah untuk membuat *swap-space* di partisi yang terpisah. Tidak ada sistem *file* atau struktur direktori di dalam partisi ini. Justru sebuah *swap-space storage manager* yang terpisah digunakan untuk mengalokasikan dan melepaskan blok-blok yang digunakan. *Manager* ini menggunakan algoritma yang dioptimalkan untuk kecepatan, daripada efisiensi tempat. Fragmentasi internal mungkin akan meningkat, tetapi ini bisa diterima karena data dalam *swap-space* biasanya umurnya lebih singkat daripada data-data di sistem *file*, dan *swap area*-nya diakses lebih sering.

Pendekatan ini membuat besar *swap-space* yang tetap selagi mempartisi *disk*. Menambah jumlah *swap-space* bisa dilakukan hanya me lalui mempartisi ulang *disk* (dimana juga termasuk memindahkan atau menghancurkan dan mengembalikan partisi *file-system* lainnya dari *backup*), atau dengan menambahkan *swap-space* di tempat lain.

Beberapa sistem operasi cukup fleksibel dan bisa *swapping* baik di partisi mentah (*raw*, belum di-format) dan di *file-system*. Contohnya Solaris 2. *Policy* dan implementasinya terpisah, sehingga administrator mesinnya (komputernya) bisa memutus kan mana yang akan digunakan. Pertimbangannya adalah antara kemudahan alokasi dan pengelolaan *file-system*, dan performa dari *swapping* pada partisi yang *raw*.

6.9.3. Pengelolaan *Swap-Space*

Untuk mengilustrasikan metode-metode yang digunakan untuk mengelola *swap-space*, kita sekarang akan mengikuti evolusi dari *swapping* dan *paging* pada GNU/ Linux. Seperti yang akan dibahas sepenuhnya pada Bab 7, GNU/ Linux memulai dengan implemen

tasi *swapping* yang menyalin seluruh proses antara daerah disk yang *contiguous* (tidak terputus) dan memori. UNIX berevolusi menjadi kombinasi dari *swapping* dan *paging* dengan tersedianya *hardware* untuk *paging*.

Dalam 4.3BSD, *swap-space* dialokasikan untuk proses ketika sebuah proses dimulai. Tempat yang cukup disediakan untuk menampung program, yang juga dikenal sebagai halaman-halaman teks (*text pages*) atau segmen teks, dan segmen data dari proses itu. Alokasi dini tempat yang dibutuhkan dengan cara seperti ini umumnya mencegah sebuah proses untuk kehabisan *swap-space* selagi proses itu dikerjakan. Ketika proses mulai, teks di dalamnya di-*page* dari file system. Halaman-halaman (*pages*) ini akan ditaruh di *swap* bila perlu, dan dibaca kembali dari sana, jadi sistem *file* akan diakses sekali untuk setiap *text page*. Halaman-halaman dari segmen data dibaca dari sistem *file*, atau dibuat (bila belum sebelumnya), dan ditaruh di *swap space* dan di-*page* kembali bila perlu. Satu contoh optimisasi (sebagai contoh, ketika dua pengguna menggunakan editor yang sama) adalah proses-proses dengan text page yang identik membagi halaman-halaman (*pages*) ini, baik di memori mau pun di *swap-space*.

Dua peta *swap* untuk setiap proses digunakan oleh kernel untuk melacak penggunaan *swap-space*. Segmen teks besarnya tetap, maka *swap space* yang dialokasikan sebesar 512K setiap potong (*chunks*), kecuali untuk potongan terakhir, yang menyimpan sisa halaman-halaman (*pages*) tadi, dengan kenaikan (*increments*) sebesar 1K.

Peta swap dari Segmen data lebih rumit, karena segmen data bisa mem besar setiap saat. Petanya sendiri besarnya tetap, tapi menyimpan a lamat-alamat swap untuk blok-blok yang besarnya bervariasi. Misalkan ada index i, bla-bla-bla, dengan besar maksimun 2 megabytes. Data struktur ini ditunjukkan oleh gambar 13.8. (Besar minimum dan maksi mum blok bervariasi, dan bisa diubah ketika me-reboot sistem.) Ketika sebuah proses mencoba untuk memperbesar segmen datanya melebihi blok yang dialokasikan di tempat swap, sistem operasi mengalokasikan blok lain lagi, dua kali besarnya yang pertama. Skema ini menyebab kan proses-proses yang kecil menggunakan blok-blok kecil. Ini juga meminimalisir fragmentasi. Blok-blok dari proses yang besar bisa di temukan dengan cepat, dan peta swap tetap kecil.

Pada Solaris 1 (SunOS 4), para pembuatnya membuat perubahan pada me tode standar UNIX untuk meningkatkan efisiensi dan untuk mencermin kan perubahan teknologi. Ketika sebuah proses berjalan, halaman-hala man (pages) dari segmen teks dibawa kembali dari sistem berkas, diak ses di memori utama, dan dibuang bila diputuskan untuk di-pageout. A kan lebih efisien untuk membaca ulang sebuah halaman (page) dari sis tem berkas

daripada menaruhnya di swap-space dan membacanya ulang dari sana. Lebih banyak lagi perubahan pada Solaris 2. Perubahan terbesar ada lah Solaris 2 mengalokasikan swap-space hanya ketika sebuah halaman (page) dipaksa keluar dari memori, daripada ketika halaman (page) da ri memori virtual pertama kali dibuat. Perubahan ini memberikan per forma yang lebih baik pada komputer-komputer modern, yang sudah mem punyai memori lebih banyak daripada komputer-komputer dengan sistem yang sudah lama, dan lebih jarang melakukan paging.

6.10. Kehandalan Disk

Disk memiliki resiko untuk mengalami kerusakan. Kerusakan ini dapat berakibat turunnya performa atau pun hilangnya data. Meski pun terdapat *backup* data, tetap saja ada kemungkinan data yang hilang karena adanya perubahan setelah terakhir kali data di-*backup*. Karenanya reliabilitas dari suatu disk harus dapat terus ditingkatkan.

Berikut adalah beberapa macam penyebab terjadinya hilangnya data:

1. Ketidaksengajaan dalam menghapus.

   Bisa saja pengguna secara tidak sengaja menghapus suatu berkas, hal ini dapat dicegah seminimal mungkin dengan cara melakukan *backup* data secara reguler.

2. Hilangnya tenaga listrik

   Hilangnya tenaga listrik dapat mengakibatkan adanya *corrupt* data.

3. Blok rusak pada disk.

   Rusaknya blok pada disk dapat saja disebabkan dari umur disk tersebut. Seiring dengan waktu, banyaknya blok pada disk yang rusak dapat terus terakumulasi. Blok yang rusak pada disk, tidak akan dapat dibaca.

4. Rusaknya Disk.

   Bisa saja karena suatu kejadian disk rusak total. Sebagai contoh, dapat saja disk jatuh atau pun ditendang ketika sedang dibawa.

5. *System Corrupt*.

   Ketika komputer sedang dijalankan, bisa saja terjadi OS *error*, program *error*, dan lain sebagainya. Hal ini tentu saja dapat menyebabkan hilangnya data.

Berbagai macam cara dilakukan untuk meningkatkan kinerja dan juga reliabilitas dari disk. Biasanya untuk meningkatkan kinerja,

dilibatkan banyak disk sebagai satu unit penyimpanan. Tiap-tiap blok data dipecah ke dalam beberapa subblok, dan dibagi-bagi ke dalam disk-disk tersebut. Ketika mengirim data disk-disk tersebut bekerja secara pararel. Ditambah dengan sinkronisasi pada rotasi masing-masing disk, maka kinerja dari disk dapat ditingkatkan. Cara ini dikenal sebagai RAID (*Redundant Array of Independent Disks*). Selain masalah kinerja RAID juga dapat meningkatkan reabilitas dari disk dengan jalan melakukan redundansi data.

Salah satu cara yang digunakan pada RAID adalah dengan *mirroring* atau *shadowing*, yaitu dengan membuat duplikasi dari tiap-tiap disk. Pada cara ini, berarti diperlukan media penyimpanan yang dua kali lebih besar daripada ukuran data sebenarnya. Akan tetapi, dengan cara ini pengaksesan disk yang dilakukan untuk membaca dapat ditingkatkan dua kali lipat. Hal ini dikarenakan setengah dari permintaan membaca dapat dikirim ke masing-masing disk. Cara lain yang digunakan pada RAID adalah *block interleaved parity*. Pada cara ini, digunakan sebagian kecil dari disk untuk penyimpanan *parity block*. Sebagai contoh, dimisalkan terdapat 10 disk pada array. Karenanya setiap 9 data *block* yang disimpan pada array, 1 parity *block* juga akan disimpan. Bila terjadi kerusakan pada salah satu *block* pada disk maka dengan adanya informasi pada *parity block* ini, ditambah dengan data *block* lainnya, diharapkan kerusakan pada disk tersebut dapat ditanggulangi, sehingga tidak ada data yang hilang. Penggunaan *parity block* ini juga akan menurunkan kinerja sama seperti halnya pada *mirroring*. Pada *parity block* ini, tiap kali *subblock* data ditulis, akan terjadi perhitungan dan penulisan ulang pada *parity block*.

6.11. Implementasi *Stable-Storage*

Pada bagian sebelumnya, kita sudah membicarakan mengenai write-ahead log, yang membutuhkan ketersediaan sebuah storage yang stabil. Berdasarkan definisi, informasi yang berada di dalam stable storage tidak akan pernah hilang. Untuk mengimplementasikan storage seperti itu, kita perlu mereplikasi informasi yang dibutuhkan ke banyak peralatan storage (biasanya disk-disk) dengan failure modes yang independen. Kita perlu mengkoordinasikan penulisan update-update dalam sebuah cara yang menjamin

bila terjadi kegagalan selagi meng-update tidak akan membuat semua kopi yang ada menjadi rusak, dan bila sedang recover dari sebuah kegagalan, kita bisa memaksa semua kopi yang ada ke dalam keadaan yang bernilai benar dan konsisten, bahkan bila ada kegagalan lain yang terjadi ketika sedang recovery.

Untuk selanjutnya, kita akan membahas bagaimana kita bisa mencapai kebutuhan kita.

Sebuah disk write menyebabkan satu dari tiga kemungkinan:

1. Successful completion.

   Data disimpan dengan benar di dalam disk.

2. Partial failure.

   Kegagalan terjadi di tengah-tengah transfer, menyebabkan hanya bebe rapa sektor yang diisi dengan data yang baru, dan sektor yang diisi ketika terjadi kegagalan menjadi rusak.

3. Total failure.

   Kegagalan terjadi sebelum disk write dimulai, jadi data yang sebe lumnya ada pada disk masih tetap ada.

Kita memerlukan, kapan pun sebuah kegagalan terjadi ketika sedang me nulis ke sebuah blok, sistem akan mendeteksinya dan memanggil sebuah prosedur recovery untuk me-restore blok tersebut ke sebuah keadaan yang konsisten. Untuk melakukan itu, sistem harus menangani dua blok physical untuk setiap blok logical. Sebuah operasi output dieksekusi seperti berikut:

1. Tulis informasinya ke blok physical yang pertama.
2. Ketika penulisan pertama berhasil, tulis informasi yang sama ke blok physical yang kedua.
3. Operasi dikatakan berhasil hanya jika penulisan kedua berhasil.

Pada saat recovery dari sebuah kegagalan, setiap pasang blok physi cal diperiksa. Jika keduanya sama dan tidak terdeteksi adanya kesa lahan, tetapi berbeda dalam isi, maka kita mengganti isi dari blok yang pertama dengan isi dari blok yang kedua. Prosedur recovery se perti ini memastikan bahwa sebuah penulisan ke stable storage akan sukses atau tidak ada perubahan sama sekali.

Kita bisa menambah fungsi prosedur ini dengan mudah untuk memboleh kan penggunaan dari kopi yang banyak dari setiap blok pada stable storage. Meski pun sejumlah besar kopi semakin mengurangi kemungkin an untuk terjadinya sebuah kegagalan, maka biasanya wajar untuk men simulasi stable storage hanya dengan dua kopi. Data di dalam stable storage dijamin aman kecuali sebuah kegagalan menghancurkan semua ko pi yang ada.

6.12. *Tertiary-Storage Structure*

Ciri-ciri Tertiary-Storage Structure:

• Biaya produksi lebih murah.
• Menggunakan *removable media*.
• Data yang disimpan bersifat permanen.

6.12.1. Macam-macam *Tertiary-Strorage Structure*

6.12.1.1. *Floppy Disk*

Gambar 6-4. Floppy Disk.

Floopy disk adalah fleksible disk yang tipis, dilapisi material yang bersifat magnet, dan ditutupi oleh plastik.
Ciri-ciri:
• Umumnya mempunyai kapasitas antara 1-2 MB.
• Kemampuan akses hampir seperti *hardisk*.

6.12.1.2. *Magneto-optic disk*

Gambar 6-5. Magneto Optic.

Magneto-optic Disk adalah Piringan *optic* yang keras dilapisi oleh material yang bersifat magnet, kemudian dilapisi pelindung dari plastik atau kaca yang berfungsi untuk menahan *head* yang hancur.

Drive ini mempunyai medan magnet. Pada suhu kamar, medan magnet terlalu kuat dan terlalu lemah untuk memagnetkan satu *bit* ke disk. Untuk menulis satu *bit*, *disk head* akan mengeluarkan sinar laser ke permukaan disk. Sinar laser ini ditujukan pada *spot* yang kecil. *Spot* ini adalah tempat yang ingin kita tulis satu *bit*. *Spot* yang ditembak sinar laser menjadi rentan terhadap medan magnet sehingga menulis satu *bit* dapat dilakukan baik pada saat medan magnet kuat mau pun lemah.

Magneto-optic disk head berjarak lebih jauh dari permukaan disk daripada *magnetic disk head*. Walau pun demikian, *drive* tetap dapat membaca *bit*, yaitu dengan bantuan sinar laser (disebut *Kerr effect*).

6.12.1.3. *Optical Disk*

Gambar 6-6. Optical Disk.

Disk ini tidak menggunakan sifat magnet, tetapi menggunakan bahan khusus yang dimodifikasi menggunakan sinar laser. Setelah dimodifikasi dengan dengan sinar laser pada *disk* akan terdapat *spot* yang gelap atau terang. *Spot* ini menyimpan satu *bit*.

Optical-disk teknologi terbagi atas:

1. *Phase-change disk*, dilapisi oleh material yang dapat membeku menjadi *crystalline* atau *amorphous state*. Kedua state ini memantulkan sinar laser dengan kekuatan yang berbeda. *Drive* menggunakan sinar laser pada kekuatan yang berbeda untuk mencairkan dan membekukan *spot* di disk sehingga *spot* berubah antara *crystalline* atau *amorphous state*.

2. *Dye-polimer disk*, merekam data dengan membuat *bump*. Disk dilapisi plastik yang mengandung *dye* yang dapat menyerap sinar laser. Sinar laser membakar *spot* yang kecil sehingga *spot* membengkak dan membentuk *bump*. Sinar laser juga dapat menghangatkan *bump* sehingga *spot* menjadi lunak dan *bump* menjadi datar.

6.12.1.4. WORM Disk (*Write Once, Read Many Times*)

Gambar 6-7.Worm Disk.

WORM adalah Aluminium film yang tipis dilapisi oleh piringan plastik atau kaca pada bagian atas dan
bawahnya.
Untuk menulis *bit*, *drive* tersebut menggunakan sinar laser untuk membakar *hole* yang kecil pada
aluminium. *Hole* ini tidak dapat diubah seperti sebelumnya. Oleh karena itu, *disk* hanya dapat ditulis
sekali.
Ciri-ciri:
• Data hanya dapat ditulis sekali.
• Data lebih tahan lama dan dapat dipercaya.
• *Read Only disk*, seperti *CD-ROM* dan *DVD* yang berasal dari pabrik sudah berisi data.

6.12.1.5. *Tapes*

Gambar 6-8. Tape.

Walau pun harga *tape drive* lebih mahal daripada *disk drive*, harga *tape cartridge* lebih murah daripada *disk cartridge* apabila dilihat dari kapasitas yang sama. Jadi, untuk penggunaan yang lebih ekonomis lebih baik digunakan *tape*. *Tape drive* dan *disk drive* mempunyai *transfer rate* yang sama. Tetapi, *random access tape* lebih lambat daripada disk karena *tape* menggunakan operasi *forward* dan *rewind*.

Seperti disebutkan diatas, *tape* adalah media yang ekonomis apabila media yang ingin digunakan tidak membutuhkan kemampuan *random access*, contoh: *backup* data dari data disk, menampung data yang besar. *Tape* digunakan oleh *supercomputer center* yang besar untuk menyimpan data yang besar. Data ini gunakan oleh badan penelitian ilmiah dan perusahaan komersial yang besar.

Pemasangan *tape* yang besar menggunakan *robotic tape changers*. *robotic tape changers* memindahkan beberapa *tape* antara beberapa *tape drive* dan beberapa *slot* penyimpanan yang berada di dalam *tape library*. *library* yang menyimpan beberapa tape disebut *tape stacker*. *library* yang menyimpan ribuan tape disebut *tape silo*.

Robotic tape library mengurangi biaya penyimpanan data. *File* yang ada di disk dapat dipindahkan ke *tape* dengan tujuan mengurangi biaya penyimpanan. Apabila *file* itu ingin digunakan, maka komputer akan memindahkan *file* tadi ke disk.

6.12.2. Masalah-Masalah yang Berkaitan Dengan Sistem Operasi

1. Tugas terpenting dari sistem operasi adalah mengatur *physical devices* dan menampilkan abstarksi mesin virtual dari aplikasi (*Interface aplikasi*).
2. Untuk *hardisk*, OS menyediakan dua abstaksi, yaitu:
   - *Raw device* = *array* dari beberapa data blok.
   - *File sistem* = sistem operasi mengantrikan dan menjadwalkan beberapa permintaan *interleaved* yang berasal dari beberapa aplikasi.

6.12.3. *Interface Aplikasi*

Kebanyakan sistem operasi menangani *removable media* hampir sama dengan *fixed disk*, yaitu *cartridge* di *format* dan dibuat *file sistem* yang kosong pada disk.

Tapes ditampilkan sebagai media *raw storage* dan aplikasi tidak membuka file pada *tape*, tetapi *tapes* dibuka kesemuanya sebagai *raw device*. Biasanya *tape drive* disediakan untuk penggunaan khusus dari suatu aplikasi sampai aplikasi berakhir atau menutup *tape drive*. Penggunaan khusus ini dikarenakan *random access tape* membutuhkan waktu yang lama. Jadi, *interleaving random access* oleh *tape* oleh beberapa aplikasi akan menyebabkan *thrashing*.

Sistem operasi tidak menyediakan *file system* sehingga aplikasi harus memutuskan bagaimana cara menggunakan *array* dari blok-blok. Sebagai contoh, program yang mem-*backup hardisk* ke *tape* akan mendaftar nama file dan kapasitas file pada permulaan *tape*. Kemudian, program meng-*copy* data file ke *tape*.

Setiap aplikasi mempunyai caranya masing-masing untuk mengatur *tape* sehingga *tape* yang telah penuh terisi data hanya dapat digunakan oleh program yang membuatnya.

Operasi dasar *tape drive* berbeda dengan operasi dasar disk drive. Contoh operasi dasar *tape drive*:

- Operasi *locate* berfungsi untuk menetapkan posisi *tape head* ke sebuah *logical blok*. Operasi ini mirip operasi yang ada di disk, yaitu: operasi *seek*. Operasi *seek* berfungsi untuk menetapkan posisi semua *track*.
- Operasi *read position* berfungsi memberitahu posisi *tape head* dengan menunjukkan nomor *logical blok*.
- Operasi *space* berfungsi memindahkan posisi *tape head*. Misalnya operasi *space* -2 akan memindahkan posisi *tape head* sejauh dua blok ke belakang.

Kapasitas blok ditentukan pada saat blok ditulis. Apabila terdapat area yang rusak pada saat blok ditulis, maka area yang rusak itu tidak dipakai dan penulisan blok dilanjutkan setelah daerah yang rusak tersebut.

Tape drive "append-only" devices, maksudnya adalah apabila kita meng-*update* blok yang ada di tengah berarti kita akan menghapus semua data sebelumnya pada blok tersebut. Oleh karena itu, *meng-update* blok tidak diperbolehkan.

Untuk mencegah hal tadi digunakan tanda *EOT* (*end-of-tape*). Tanda EOT ditaruh setelah sebuah blok ditulis. *Drive* menolak ke lokasi sebelum tanda EOT, tetapi *drive* tidak menolak ke lokasi tanda EOT kemudian *drive* mulai menulis data. Setelah selesai menulis data, tanda EOT ditaruh setelah blok yang baru ditulis tadi.

6.12.4. Penamaan Berkas

Menamakan berkas pada *removable media* cukup sulit terutama pada saat kita menulis data pada *removable cartridge* pada suatu komputer, kemudian menggunakan *cartridge* ini pada komputer yang lain. Jika jenis komputer yang digunakan sama dan jenis *cartridge* yang digunakan sama, maka permasalahannya adalah mengetahui isi dan *data layout* dari *cartridge*. Tetapi, bila jenis komputer yang digunakan dan jenis *drive* yang digunakan berbeda, maka berbagai masalah akan muncul. Apabila hanya jenis *drive* yang digunakan sama, komputer yang berbeda menyimpan *bytes* dengan berbagai cara dan juga menggunakan *encoding* yang berbeda untuk *binary number* atau huruf.

Pada umumnya sistem operasi sekarang tidak memperdulikan masalah penamaan *space* pada *removable media*. Hal ini tergantung kepada aplikasi dan user bagaimana cara mengakses dan menterjemahkan data. Tetapi, beberapa jenis *removable media* (contoh: CDs) distandarkan cara menggunakannya untuksemua jenis komputer.

6.12.5. Managemen Penyimpanan Hirarkis

Managemen Penyimpanan Hirarkis (*Hierachical Storage management*) menjelaskan *storage hierarchy* antara *primary memory* dan *secondary storage* untuk membentuk *tertiary storage*. *Tertiary storage* biasanya diimplementasikan sebagai *jukebox* dari *tapes* atau *removable media*.

Walau pun *tertiary storage* dapat memepergunakan sistem *virtual-memory*, cara ini tidak baik. Karena pengambilan data dari *jukebox* membutuhkan waktu yang agak lama. Selain itu diperlukan waktu yang agak lama untuk *demand paging* dan untuk bentuk lain dari penggunaan *virtual-memory*.

File yang kapasitasnya kecil dan sering digunakan disimpan di disk. Sedangkan file yang kapasitasnya besar, sudah lama, dan tidak aktif akan diarsipkan di *jukebox*. Pada beberapa sistem *file-archiving*, *directory entry* untuk file selalu ada, tetapi isi file tidak berada di *secondary storage*. Jika aplikasi mencoba membuka file, pemanggilan *open system* akan ditunda sampai isi file dikirim dari *tertiary storage*. Ketika isi file sudah ada di *secondary storage*, operasi *open* dikembalikan ke aplikasi.

Hierachical Storage management biasanya ditemukan pada pusat *supercomputing* dan *installasi* besar lainnya yang mempunyai data yang besar.

6.13. Rangkuman

6.13.1. *I/O*

Dasar dari elemen perangkat keras yang terkandung pada *I/O* adalah *bus*, *device controller*, dan *I/O* itu sendiri. Kinerja kerja pada data yang bergerak antara device dan memori utama di jalankan oleh CPU, di program oleh *I/O* atau mungkin DMA *controller*. Modul kernel yang mengatur *device* adalah *device driver*. *System-call interface* yang disediakan aplikasi dirancang untuk menghandle beberapa dasar kategori dari perangkat keras, termasuk *block devices, character devices, memory mapped files, network sockets* dan *programmed interval timers*.

Subsistem *I/O* kernel menyediakan beberapa servis. Diantaranya adalah *I/O schedulling, buffering, spooling, error handling* dan *device reservation*. Salah satu servis dinamakan *translation*, untuk membuat koneksi antara perangkat keras dan nama file yang digunakan oleh aplikasi.

I/O system calls banyak dipakai oleh CPU, dikarenakan oleh banyaknya lapisan dari perangkat lunak antara *physical device* dan aplikasi. Lapisan ini mengimplikasikan *overhead* dari alih konteks untuk melewati *kernel's protection boundary*, dari sinyal dan *interrupt handling* untuk melayani *I/O devices*.

6.13.2. Disk

Disk drives adalah *major secondary-storage I/O device* pada kebanyakan komputer. Permintaan untuk disk *I/O* digenerate oleh sistem file dan sistem virtual memori. Setiap permintaan menspesifikasikan alamat pada disk untuk dapat direferensikan pada *form* di *logical block number*.

Algoritma *disk schedulling* dapat meningkatkan efektifitas *bandwith*, *average response time*, dan *variance response time*. Algoritma seperti SSTF, SCAN, C-SCAN, LOOK dan C-LOOK didesain untuk membuat perkembangan dengan menyusun ulang antrian disk untuk meningkatkan total waktu pencarian.

Kinerja dapat rusak karena *external fragmentation*. Satu cara untuk menyusun ulang disk untuk mengurangi fragmentasi adalah untuk *back up* dan *restore* seluruh disk atau partisi. Blok-blok dibaca dari lokasi yang tersebar, me-*restore* tulisan mereka secara berbeda. Beberapa sistem mempunyai kemampuan untuk men-*scan* sistem file untuk mengidentifikasi file terfragmentasi, lalu menggerakan blok-blok mengelilingi untuk meningkatkan fragmentasi. Men-defragmentasi file yang sudah di fragmentasi (tetapi hasilnya kurang optimal) dapat secara signifikan

meningkatkan kinerja, tetapi sistem ini secara umum kurang berguna selama proses defragmentasi sedang berjalan. Sistem operasi me-*manage* blok-blok pada disk. Pertama, disk baru di format secara *low level* untuk menciptakan sektor pada perangkat keras yang masih belum digunakan. Lalu, disk dapat di partisi dan sistem file diciptakan, dan blok-blok boot dapat dialokasikan. Terakhir jika ada blok yang terkorupsi, sistem harus mempunyai cara untuk me-*lock out* blok tersebut, atau menggantikannya dengan cadangan.

Tertiary storage di bangun dari disk dan *tape drives* yang menggunakan media yang dapat dipindahkan. Contoh dari *tertiary storage* adalah *magnetic tape*, *removable magnetic*, dan *magneto-optic disk*.

Untuk *removable disk*, sistem operasi secara general menyediakan servis penuh dari sistem file *interface*, termasuk *space management* dan *request-queue schedulling*. Untuk tape, sistem operasi secara general hanya menyediakan *interface* yang baru. Banyak sistem operasi yang tidak memiliki *built-in support* untuk *jukeboxes*. *Jukebox support* dapat disediakan oleh *device driver*.

6.14. Soal Latihan

Perangkat Keras I/O

1. Gambarkan diagram dari Interrupt Driven I/O Cycle.

2. Sebutkan langkah-langkah dari transfer DMA!

3. Apakah perbedaan dari polling dan interupsi?

4. Apa hubungan arsitektur kernel yang di-thread dengan implemen tasi interupsi?

Interface Aplikasi I/O

1. Kenapa dibutuhkan interface pada aplikasi I/O?

2. Apa tujuan adanya device driver? Berikan contoh keuntungan yang kita dapatkan dengan adanya hal

ini!

Kernel I/O Subsystem

1. Apakah yang dimaksud dengan proses *pooling*? (jelaskan dengan jelas)

2. Mengapa diperlukan proses *pooling*?

3. Apakah yang dimaksud dengan *buffer*?

4. Jelaskan dengan singkat mengenai *I/O Scheduling*!

Penanganan Permintaan I/O

1. Apakah kegunaan dari Streams pada Sistem V UNIX?
2. Jelaskan lifecycle dari permintaan pembacaan blok!

Performa I/O

1. Gambarkan bagan mengenai komunikasi antar komputer
2. Bagaimana cara meningkatkan efisiensi performa I/O
3. Jelaskan mengenai implementasi dari fungsi I/O

Struktur Disk

1. Sebutkan bagian-bagian dari disk
2. Apa keuntungan penggunaan pemetaan pada disk?

Penjadualan Disk

1. Buatlah dengan pemikiran Anda sendiri, strategi penjadualan disk yang tepat dan efisien menurut Anda
2. Menurut Anda, diantara algoritma-algoritma penjadualan disk diatas manakah yang paling cepat,
3. Manakah yang paling efisien (hemat/tidak mahal), dan manakah yang paling lambat dan tidak efisien? Jelaskan!

Managemen Disk

1. Bagaimana cara disk SCSI me-recovery blok yang rusak? Jelaskan selengkap mungkin!

Penanganan *Swap-Space*

1. Bagaimana penanganan swap space pada disk?
2. Bagaimana pengelolaan *swap space* pada disk?

Reabilitas Disk

1. Terangkan bagaimana RAID dapat meningkatkan reabilitas dari disk?
2. Adakah batas waktu hidup suatu disk? Jika ada, berapa lama? Jika tidak, kenapa?

Implementasi *Stable-Storage*

1. Sebutkan kemungkinan-kemungkinan dari disk write!
2. Bagaimanakah suatu operasi output dieksekusi?

Tertiary-Storage Structure

1. Sebutkan kelebihan tertiary storage structure?
2. Apakah kegunaan EOT pada tapes? Jelaskan cara kerjanya?
3. Jelaskan tugas sistem operasi terhadap *tertiary-storage structure*?

6.15. Rujukan

1. Applied Operating System Concept, Silberschatz, Galvin, Gagne, 1999
2. *RAID and Data Protection Solutions for Linux* (http://linas.org/linux/raid.html)
3. *DMA Interface* (http://www.eso.org/projects/iridt/irace/aboutirace.html)
4. *I/O Transfer Method* (http://www.ebiz.com.pk/pakistan/dma.doc)

6.16. Daftar Istilah

I/O = I/O (Input/Output)

hardware -> perangkat keras

device = device

storage device -> device penyimpanan

disk = disk

transmission = transmission

processor -> prosesor

human-interface device = human-interface device

instruction -> instruksi

direct I/O instruction = direct I/O instruction

memory-mapped I/O = memory-mapped I/O

port = port (perangkat keras)

bus = bus (perangkat keras)

daisy chain = daisy chain

shared direct access = shared direct access

controller = controller

host adapter = host adapter

command-ready =command-ready

busy = busy

error = error

host = host

polling = polling

looping = looping

status register -> register status

service = service

CPU processing = CPU processing

Interrupt -> Interupsi

request line = request line

pointer = pointer

interrupt handler/ing = interrupt handler/ing

interrupt controller = interrupt controller

critical state = critical state, efisiensi

interrupt priority level system = interrupt priority level system

interrupt request line = interrupt request line

nonmaskable interrupt = nonmaskable interrupt

maskable interrupt = maskable interrupt

critical instruction sequence = critical instruction sequence

interrupt vector = interrupt vector

interrupt chaining = interrupt chaining

offset = offset

overhead = overhead

exception = exception

page fault = page fault

system call = system call

software interrupt = software interrupt

trap = trap

DMA = Direct Memory Access

command block = command block

transfer destination -> destinasi transfer

address -> alamat (istilah komputer dalam penunjukkan lokasi)

block -> blok

burst mode = burst mode

single burst = single burst

microprocessor -> mikroprosesor

idle = idle
cycle stealing mode = cycle stealing mode
handshaking = handshaking
DMA request = DMA request
DMA acknowledge = DMA acknowledge
memory-address -> alamat memori
cycle stealing = cycle stealing
virtual address -> alamat virtual
physical memory -> memori fisik
performance -> performa
device driver = device driver
memory bus -> bus memori
controller = controller
physical memory = physical memory
application space data = application space data
context switch = alih konteks
device = device
interrupt -> interupsi
smart controller = smart controller
polling = polling
concurrency = concurrency
channel = channel
memory subsystem = memory subsystem
bus = bus
application code = kode aplikasi
bugs = bugs
reboot = reboot
reload = reload
overhead = overhead
internal kernel -> kernel internal
messaging = messaging
threading = threading
locking = locking
debug = debug
crash = crash

block reads = block reads
write = write
workload = workload
secondary storage -> penyimpanan sekunder
magnetic tape = magnetic tape
tape = tape
backup = backup
disk drive = disk drive
logic block -> blok lojik
bytes = bytes
low level formatted = low level formatted
logical block number -> nomor blok lojikal
disk address -> alamat disk
sector -> sektor
hardware = hardware
disk drives = disk drives
bandwith disk = bandwith disk
seek time -> waktu pencarian
disk arm = disk arm
head = head
disk = disk
bandwith = bandwith
bytes = bytes
input = input
output = output
controller = controller
memory address = alamat memori
First-come First-serve = First-com First-serve
shortest-seek-time-first = shortest-seek-time-first
shortest-job-first = shortest-job-first
starvation = starvation
schedulling -> penjadwalan
disk arm = disk arm
Circular-SCAN = Circular-SCAN
variance -> varian

index -> indeks
directory = directory
disk head = disk head
magnetic disk = disk magnetik
slate = slate
low-level formatting = low-level formatting
physical formatting = physical formatting
trailer = trailer
disk controller = disk controller
partition = partition
I/O = I/O
logical block -> blok lojikal
raw I/O = raw I/O
main memory = memori utama
bootstrap = boostrap
boot disk = boot disk
bad blocks = bad blocks
sector slipping = sector slipping
interface = interface
I/O Application -> aplikasi I/O
software layering = software layering
device driver = device driver
layer -> lapisan
disk drive = disk drive
block device = block device
random-access = random-access
stream character -> karakter stream
library = library
network device -> peralatan jaringan
interface socket = interface socket
local socket = local socket
remote socket = remote socket
clock -> jam
timer = timer
trigger = trigger

programmable interval timer = programmable interval timer
scheduler = scheduler
timer request = timer request
hardware timer = hardware timer
blocking (application) = blocking (application)
nonblocking (application) = nonblocking (application)
wait queue = wait queue
run queue = run queue
physical action = physical action
asynchronous = asynchronous

Rujukan

Buku

[Coffmanea1997] E G Coffman, Jr., M J Elphick, dan A Shoshani, 1971, *System Deadlocks*, Computing Surveys, Vol.3, No.2.

[Deitel1990] H M Deitek, 1990, *Operating Systems*, Massachusetts, Addison-Wesley, 2nd ed.

[Hariyanto1997] B Hariyanto, 1997, *Sistem Operasi*, Informatika, Bandung.

[Havender1968] J W Havender, 1968, *Avoiding Deadlock in Multitasking Systems*, IBM Systems Journal, Vol.7, No.2.

[Samik-Ibrahim2001] Rahmat Samik-Ibrahim, 2001, *Ujian Mid Test 2001*, Fakultas Ilmu Komputer, Universitas Indonesia.

[Silberschatz2000] Avi Silberschatz, Peter Galvin, dan Grag Gagne, 2000, *Applied Operating Systems: First Edition*, Edisi Pertama, John Wiley & Sons.

[Stallings2001] William Stallings, 2001, *Operating Systems*, Fourth Edition, Prentice Hall.

[Tanenbaum1992] Andrew S Tanenbaum, 1992, *Modern Operating Systems*, Englewood Cliffs, New Jersey.

[Walsh2002] Norman Walsch dan Leonard Muellner, Bob Stayton, 1999, 2000, 2001, 2002, *DocBook: The Definitive Guide*, Version 2.0.7, O'Reilly.

URLs

[DMA] *DMA Interface*, *DMA Interface* (http://www.eso.org/projects/iridt/irace/aboutirace.html) .

[History] *History*, *History of Linux* (http://www.ragib.hypermart.net/linux/) .

[IO] *I/O Transfer Method*, *I/O Transfer Method* (http://www.ebiz.com.pk/pakistan/dma.doc) .

[RAID] *RAID*, *RAID and Data Protection Solutions for Linux* (http://linas.org/linux/raid.html) .

[TUX] *TUX*, *The story behind tux* (http://www.wired.com/news/culture/0,1284,42209.html) .

[FIXME1] *FIXME1*, *http://www.risc.uni-linz.ac.at/people/schreine/papers/idimt97/multithread.gif*

(http://www.risc.uni-linz.ac.at/people/schreine/papers/idimt97/multithread.gif) .

[FIXME2] *FIXME2*, *http://www.unet.univie.ac.at/aix/aixprggd/genprogc/figures/genpr68.jpg*

(http://www.unet.univie.ac.at/aix/aixprggd/genprogc/figures/genpr68.jpg) .

[FIXME3] *FIXME3*, *http://www.unet.univie.ac.at/aix/aixprggd/genprogc/understanding_threads.htm*

(http://www.unet.univie.ac.at/aix/aixprggd/genprogc/understanding_threads.htm) .

[FIXME4] *FIXME4*, *http://www.freeos.com/articles/4051/* (http://www.freeos.com/articles/4051/) .

[FIXME5] *FIXME5*, *http://www.pdrs.com.au/pdrs/linuxfs.htm*

(http://www.pdrs.com.au/pdrs/linuxfs.htm) .

[FIXME6] *FIXME6*, *http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html*

(http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html) .

[FIXME7] *FIXME7*, *http://www.csc.uvic.ca/~mcheng/360/notes/NOTES2.html*

(http://www.csc.uvic.ca/~mcheng/360/notes/NOTES2.html) .

[FIXME8] *FIXME8*, *http://www.sao.nrc.ca/imsb/rcsg/documents/basic/node30.html*

(http://www.sao.nrc.ca/imsb/rcsg/documents/basic/node30.html) .

[FIXME9] *FIXME9*, *http://www.isbiel.ch/~myf/opsys1/Exercises/Chap4/Problems1.html*

(http://www.isbiel.ch/~myf/opsys1/Exercises/Chap4/Problems1.html) .

[FIXME10] *FIXME10*, *http://www.chipcenter.com/circuitcellar/march02/c0302dc4.htm*

(http://www.chipcenter.com/circuitcellar/march02/c0302dc4.htm) .

[FIXME11] *FIXME11*,

http://www.etnus.com/Support/docs/rel5/html/cli_guide/images/procs_n_threads8a.gif

(http://www.etnus.com/Support/docs/rel5/html/cli_guide/images/procs_n_threads8a.gif) .

[FIXME12] *FIXME12*, *http://www.etnus.com/Support/docs/rel5/html/cli_guide/procs_n_threads5.html*

(http://www.etnus.com/Support/docs/rel5/html/cli_guide/procs_n_threads5.html) .

[FIXME13] *FIXME13*, *http://www.linuxhq.com/guides/* (http://www.linuxhq.com/guides/) .

[FIXME14] *FIXME14*, *http://www.linuxhq.com/guides/TLK/tlk-toc.html* (http://www.linuxhq.com/guides/TLK/tlk-toc.html) .

[FIXME15] *FIXME15*, *http://www.osdata.com/kind/history.htm* (http://www.osdata.com/kind/history.htm) .

[FIXME16] *FIXME16*, *http://www.redhat.com/docs/manuals/linux/RHL-7.3-Manual/ref-guide* (http://www.redhat.com/docs/manuals/linux/RHL-7.3-Manual/ref-guide) .

[FIXME17] *FIXME17*, *http://www.redhat.com/docs/manuals/linux/RHL-7.3-Manual/ref-guide/ch-proc.html* (http://www.redhat.com/docs/manuals/linux/RHL-7.3-Manual/ref-guide/ch-proc.html) .

[FIXME18] *FIXME18*, *http://ctdp.tripod.com/os/linux/howlinuxworks/* (http://ctdp.tripod.com/os/linux/howlinuxworks/) .

[FIXME19] *FIXME19*, *http://ctdp.tripod.com/os/linux/howlinuxworks/linux_hlprocess.html* (http://ctdp.tripod.com/os/linux/howlinuxworks/linux_hlprocess.html) .

[FIXME20] *FIXME20*, *http://cs-pub.bu.edu/fac/richwest/cs591_w1/notes/linux_process_mgt.PDF* (http://cs-pub.bu.edu/fac/richwest/cs591_w1/notes/linux_process_mgt.PDF) .

[FIXME21] *FIXME21*, *http://telemann.coda.cs.cmu.edu/doc/talks/linuxvfs/* (http://telemann.coda.cs.cmu.edu/doc/talks/linuxvfs/) .

[FIXME22] *FIXME22*, *http://web.mit.edu/tytso/www/linux/ext2intro.html* (http://web.mit.edu/tytso/www/linux/ext2intro.html) .

[FIXME23] *FIXME23*, *http://www.cs.wm.edu/~dsn/444F02/lectures/linux_l1.pdf* (http://www.cs.wm.edu/~dsn/444F02/lectures/linux_l1.pdf) .

[FIXME24] *FIXME24*, *http://www.imm.dtu.dk/courses/02220/OS/OH/week7.pdf* (http://www.imm.dtu.dk/courses/02220/OS/OH/week7.pdf) .

[FIXME25] *FIXME25*, *http://www.cs.nyu.edu/courses/spring02/v22.0202-002/lecture-03.html*

(http://www.cs.nyu.edu/courses/spring02/v22.0202-002/lecture-03.html) .

[FIXME26] *FIXME26*, *http://www.mcsr.olemiss.edu/unixhelp/concepts/history.html*

(http://www.mcsr.olemiss.edu/unixhelp/concepts/history.html) .

[FIXME27] *FIXME27*, *http://www.cs.panam.edu/fox/CSCI4334/ch3.ppt*

(http://www.cs.panam.edu/fox/CSCI4334/ch3.ppt) .

[FIXME28] *FIXME28*, *http://lass.cs.umass.edu/~shenoy/courses/fall01/labs/talab2.html*

(http://lass.cs.umass.edu/~shenoy/courses/fall01/labs/talab2.html) .

[FIXME29] *FIXME29*, *http://www.cis.umassd.edu/~rbalasubrama/*

(http://www.cis.umassd.edu/~rbalasubrama/) .

[FIXME30] *FIXME30*, *http://www.cs.umd.edu/projects/shrug/ppt/5-Oct-2001.ppt*

(http://www.cs.umd.edu/projects/shrug/ppt/5-Oct-2001.ppt) .

[FIXME31] *FIXME31*, *http://legion.virginia.edu/presentations/sc2000/sld001.htm*

(http://legion.virginia.edu/presentations/sc2000/sld001.htm) .

[FIXME32] *FIXME32*, *http://www.cs.wisc.edu/~cao/cs537/midterm-answers1.txt*

(http://www.cs.wisc.edu/~cao/cs537/midterm-answers1.txt) .

[FIXME33] *FIXME33*, *http://www.cs.wpi.edu/~cs502/s99/* (http://www.cs.wpi.edu/~cs502/s99/) .

[FIXME34] *FIXME34*, *http://cs-www.cs.yale.edu/homes/avi/os-book/osc/slide-dir/*

(http://cs-www.cs.yale.edu/homes/avi/os-book/osc/slide-dir/) .

[FIXME35] *FIXME35*, *http://www.hardware.fr/articles/338/page1.html*

(http://www.hardware.fr/articles/338/page1.html) .

[FIXME36] *FIXME36*, *http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/CPU-Scheduler.PDF*

(http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/CPU-Scheduler.PDF) .

[FIXME37] *FIXME37*, *http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/Proses.PDF*

(http://www.cs.ui.ac.id/kuliah/IKI20230/materi/week4/Proses.PDF) .

[FIXME38] *FIXME38*, *http://opensource.ucc.ie/icse2002/SchachOffutt.pdf*

(http://opensource.ucc.ie/icse2002/SchachOffutt.pdf) .

[FIXME39] *FIXME39, http://www.ignou.ac.in/virtualcampus/adit/course/index-tr1.htm* (http://www.ignou.ac.in/virtualcampus/adit/course/index-tr1.htm) .

[FIXME40] *FIXME40, http://www.cs.technion.ac.il/~hagit/OSS98* (http://www.cs.technion.ac.il/~hagit/OSS98) .

[FIXME41] *FIXME41, http://www.crackinguniversity2000.it/boooks/1575211025/ch6.htm* (http://www.crackinguniversity2000.it/boooks/1575211025/ch6.htm) .

[FIXME42] *FIXME42, http://www.science.unitn.it/~fiorella/guidelinux/tlk/node5.html* (http://www.science.unitn.it/~fiorella/guidelinux/tlk/node5.html) .

[FIXME43] *FIXME43, http://www.science.unitn.it/~fiorella/guidelinux/tlk/node94.html* (http://www.science.unitn.it/~fiorella/guidelinux/tlk/node94.html) .

[FIXME44] *FIXME44, http://home.earthlink.net/~jknapka/linux-mm/vmoutline.html* (http://home.earthlink.net/~jknapka/linux-mm/vmoutline.html) .

[FIXME45] *FIXME45, http://kernelbook.sourceforge.net/* (http://kernelbook.sourceforge.net/) .

[FIXME46] *FIXME46, http://www.techrescue.net/guides/insthware.asp* (http://www.techrescue.net/guides/insthware.asp) .

[FIXME47] *FIXME47, http://agt.buka.org/concept.html* (http://agt.buka.org/concept.html) .

[FIXME48] *FIXME48, http://kos.enix.org/pub/greenwald96synergy.pdf* (http://kos.enix.org/pub/greenwald96synergy.pdf) .

[FIXME49] *FIXME49, Situs GNU* (http://www.gnu.org/) .

[FIXME50] *FIXME50, http://www.kernel.org/* (http://www.kernel.org/) .

[FIXME51] *FIXME51, http://www.kernelnewbies.org/* (http://www.kernelnewbies.org/) .

[FIXME52] *FIXME52, http://www.kernelnewbies.org/documents/* (http://www.kernelnewbies.org/documents/) .

[FIXME53] *FIXME53, http://www.reiserfs.org/* (http://www.reiserfs.org/) .

[FIXME54] *FIXME54, http://en.tldp.org/guides.html* (http://en.tldp.org/guides.html) .

[FIXME55] *FIXME55, http://www.tldp.org/HOWTO/Linux+XFS-HOWTO/* (http://www.tldp.org/HOWTO/Linux+XFS-HOWTO/) .

[FIXME56] *FIXME56, http://sdn.vlsm.org/share/LDP/intro/* (http://sdn.vlsm.org/share/LDP/intro/) .

[FIXME57] *FIXME57, http://sdn.vlsm.org/share/LDP/lkmpg/* (http://sdn.vlsm.org/share/LDP/lkmpg/)

[FIXME58] *FIXME58*, *http://en.tldp.org/LDP/intro-linux/Intro-Linux.pdf*
(http://en.tldp.org/LDP/intro-linux/Intro-Linux.pdf) .

[FIXME59] *FIXME59*, *http://en.tldp.org/LDP/lki/lki.pdf*
(http://en.tldp.org/LDP/lki/lki.pdf) .
[FIXME60] *FIXME60*, *http://en.tldp.org/LDP/lkmpg/lkmpg.pdf*
(http://en.tldp.org/LDP/lkmpg/lkmpg.pdf) .
[FIXME61] *FIXME61*,
http://www.cee.hw.ac.uk/courses/5nm1/Exercises/2.htm
(http://www.cee.hw.ac.uk/courses/5nm1/Exercises/2.htm) .
[FIXME54] *FIXME54*,
http://www.cs.wits.ac.za/~adi/courses/linuxadmin/content/module2doc.html
(http://www.cs.wits.ac.za/~adi/courses/linuxadmin/content/module2doc.html) .