

Sistem Operasi

Bahan Kuliah IKI-20230

**Gabungan Kelompok Kerja 21–28 Semester Genap
2002/2003 dan 41–49 Semester Ganjil 2003/2004 Mata
Kuliah Sistem Operasi**

Sistem Operasi: Bahan Kuliah IKI-20230

oleh Gabungan Kelompok Kerja 21–28 Semester Genap 2002/2003 dan 41–49 Semester Ganjil 2003/2004 Mata Kuliah Sistem Operasi

\$Revision: 1.7.0.0 \$ Edisi

Diterbitkan 11 November 2003

Hak Cipta © 2003 oleh Gabungan Kelompok Kerja 21–28 Semester Genap 2002/2003 dan 41–49 Semester Ganjil 2003/2004 Mata Kuliah Sistem Operasi.

Silakan menyalin, mengedarkan, dan/ atau, memodifikasi bagian dari dokumen – \$Revision: 1.7.0.0 \$ – yang dikarang oleh Gabungan Kelompok Kerja 21–28 Semester Genap 2002/2003 dan 41–49 Semester Ganjil 2003/2004 Mata Kuliah Sistem Operasi, sesuai dengan ketentuan "*GNU Free Documentation License* versi 1.1" atau versi selanjutnya dari FSF (*Free Software Foundation*); tanpa bagian "*Invariant*", tanpa teks "*Front-Cover*", dan tanpa teks "*Back-Cover*". Lampiran A ini> berisi salinan lengkap dari lisensi tersebut. Ketentuan ini **TIDAK** berlaku untuk bagian dan/ atau kutipan yang bukan dikarang oleh Gabungan Kelompok Kerja 21–28 Semester Genap 2002/2003 dan 41–49 Semester Ganjil 2003/2004 Mata Kuliah Sistem Operasi.

Catatan Revisi

Revisi 1.7 17-11-2003 Revised by: Kelompok 49

Versi rilis alfa buku OS

Revisi 1.5 17-11-2003 Revised by: Kelompok 49

Penggabungan pertama seluruh pekerjaan kelompok 41 sampai kelompok 48. Masih ada beberapa gambar yang belum lengkap. Rujukan

Revisi 1.4 08-11-2003 Revised by: Kelompok 49

Pengubahan template versi 1.3 dengan template yang baru yang akan digunakan dalam versi 1.4-2.0

Revisi 1.3.0.5 12-11-2003 Revised by: RMS46

Revisi ini diedit oleh Rahmat M. Samik-Ibrahim: dipilah sesuai dengan sub-pokok bahasan yang ada.

Revisi 1.3 30-09-2003 Revised by: RMS46

Revisi ini diedit oleh Rahmat M. Samik-Ibrahim: melanjutkan perbaikan tata letak dan pengindeksan.

Revisi 1.2 17-09-2003 Revised by: RMS46

Revisi ini diedit oleh Rahmat M. Samik-Ibrahim: melanjutkan perbaikan.

Revisi 1.1 01-09-2003 Revised by: RMS46

Revisi ini diedit oleh Rahmat M. Samik-Ibrahim: melakukan perbaikan struktur SGML, tanpa terlalu banyak mengubah isi buku.

Revisi 1.0 27-05-2003 Revised by: RMS46

Kompilasi ulang, serta melakukan sedikit perapihan.

Revisi 0.21.4 05-05-2003 Revised by: Kelompok 21

Perapihan berkas dan penambahan entity.

Revisi 0.21.3 29-04-2003 Revised by: Kelompok 21

Perubahan dengan menyempurnakan nama file.

Revisi 0.21.2 24-04-2003 Revised by: Kelompok 21

Merubah Kata Pengantar.

Revisi 0.21.1 21-04-2003 Revised by: Kelompok 21

Menambahkan Daftar Pustaka dan Index.

Revisi 0.21.0 26-03-2003 Revised by: Kelompok 21

Memulai membuat tugas kelompok kuliah Sistem Operasi.

Persembahan

Buku ini dipersembahkan *dari* Gabungan Kelompok Kerja 21–28 Semester Genap 2002/2003 dan 41–49 Semester Ganjil 2003/2004 Mata Kuliah Sistem Operasi, *oleh* Gabungan Kelompok Kerja 21–28 Semester Genap 2002/2003 dan 41–49 Semester Ganjil 2003/2004 Mata Kuliah Sistem Operasi, *untuk* siapa saja yang ingin mempelajari Sistem Operasi. Tim penyusun buku ini ialah sebagai berikut:

Kelompok 21 (Koordinator)

Dhani Yuliarso, Fernan, Hanny Faristin, Melanie Tedja, Paramanandana D.M., Widya Yuwanda.

Kelompok 22

Budiono Wibowo, Agus Setiawan, Baya U.H.S., Budi A. Azis Dede Junaedi, Heriyanto, Muhammad Rusdi.

Kelompok 23

Indra Agung, Ali Khumaidi, Arifullah, Baihaki A.S., Christian K.F. Daeli, Eries Nugroho, Eko Seno P., Habrar, Haris Sahlan.

Kelompok 24

Adzan Wahyu Jatmiko, Agung Pratomo, Dedy Kurniawan, Samiaji Adisasmito, Zidni Agni.

Kelompok 25

Nasrullah, Amy S. Indrasari, Ihsan Wahyu, Inge Evita Putri, Muhammad Faizal Ardhi, Muhammad Zaki Rahman, N. Rifka N. Liputo, Nelly, Nur Indah, R. Ayu P., Sita A.R.

Kelompok 26

Rakhmad Azhari, Adhe Aries, Adityo Pratomo, Aldiantoro Nugroho, Framadhan A., Pelangi, Satrio Baskoro Y.

Kelompok 27

Teuku Amir F.K., Alex Hendra Nilam, Anggraini W., Ardini Ridhatillah, R. Ferdy Ferdian, Ripta Ramelan, Suluh Legowo, Zulkiffi.

Kelompok 28

Christiono H, Arief Purnama L.K., Arman Rahmanto, Fajar, Muhammad Ichsan, Rama P. Tardan, Unedo Sanro Simon.

Kelompok 41

Ahmad Furqan S K., Aristo, Obeth M S.

Kelompok 42

Puspita K S, Retno Amelia, Susi R, Sutia H.

Kelompok 43

Agus Setiawan, Adhita Amanda, Afaf M, Alisa Dewayanti, Andung J Wicaksono, Dian Wulandari L, Gunawan, Jefri Abdullah, M Gantino, Prita I.

Kelompok 44

Arnold W, Antonius H, Irene, Theresia B, Ilham W K, Imelda T, Dessy N, Alex C.

Kelompok 45

Bima Satria T, Adrian Dwitomo, Alfa Rega M, Bobby, Diah Astuti W, Dian Kartika P, Pratiwi W, S Budianti S, Satria Graha, Siti Mawaddah, Vita Amanda.

Kelompok 46

Josef, Arief Aziz, Bimo Widhi Nugroho, Chrysta C P, Dian Maya L, Monica Lestari P, Muhammad Alaydrus, Syntia Wijaya Dharma, Wilmar Y Igenesjz, Yenni R

Kelompok 47

Bayu Putera, Enrico, Ferry Haris, Franky, Hadyan Andika, Ryan Loanda, Satriadi, Setiawan A, Siti P Wulandari, Tommy Khoerniawan, Wadiyono Valens, William Hutama.

Kelompok 48

Amir Murtako, Dwi Astuti A, M Abdushshomad E, Mauldy Laya, Novarina Azli, Raja Komkom S.

Kelompok 49 (Koordinator)

Fajran Iman Rusadi, Carroline D Puspa.

Daftar Isi

Kata Pengantar	i
1. Konsep Dasar Perangkat Komputer	1
2. Konsep Dasar Sistem Operasi	2
3. Proses dan Penjadwalan.....	3
4. Sinkronisasi dan <i>Deadlock</i>	4
5. Managemen Memori	5
5.1. Manajemen Memori.....	5
5.1.1. Latar Belakang.....	5
5.1.2. Pemberian Alamat	5
5.1.3. Ruang Alamat Logika & Fisik	5
5.1.4. Pemanggilan Dinamis.....	6
5.1.5. Penghubungan Dinamis dan <i>Library</i> Bersama	7
5.1.6. <i>Overlays</i>	7
5.2. <i>Swap</i> dan Alokasi Memori.....	9
5.2.1. <i>Swapping</i>	9
5.2.2. Pengalokasian Memori	11
5.2.2.1. <i>Contiguous Memory Allocation</i>	11
5.2.2.2. <i>Fragmentasi</i>	13
5.3. Pemberian Halaman	14
5.3.1. Metode Dasar.....	14
5.3.2. Keuntungan dan Kerugian Pemberian Halaman.....	15
5.4. Struktur Page Table	15
5.4.1. Page Table.....	17
5.4.2. Pemberian Page Secara <i>Multilevel</i>	19
5.4.3. Page Table secara <i>Inverted</i>	19
5.4.4. Berbagi Page.....	20
5.5. Segmentasi	20
5.5.1. Arsitektur Segmentasi	20
5.5.2. Saling Berbagi dan Proteksi	21
5.5.3. Alokasi yang Dinamis	21
5.5.4. Masalah dalam Segmentasi	21
5.5.5. Segmentasi dengan <i>paging</i>	22
5.5.6. Penggunaan Segmentasi	22
5.6. Pengantar Memori Virtual; Demand Paging	24
5.6.1. Pengertian	25
5.6.2. Keuntungan.....	25
5.6.3. Implementasi	25
5.6.4. Demand Paging	26
5.6.4.1. Permasalahan pada Demand Paging	26
5.6.4.2. Skema Bit Valid - Tidak Valid	26
5.6.4.3. Penanganan <i>Page Fault</i>	27
5.6.4.4. Apa yang terjadi pada saat <i>page-fault</i> ?.....	27
5.6.4.5. Kinerja Demand Paging.....	28
5.6.4.6. Permasalahan Lain yang berhubungan dengan Demand Paging	28

5.6.4.7. Persyaratan Perangkat Keras.....	29
5.7. Aspek Demand Paging : Pembuatan Proses.....	29
5.7.1. Copy-On-Write.....	29
5.7.2. Memory-Mapped Files	30
5.8. Konsep Dasar <i>Page Replacement</i>	31
5.8.1. Konsep Dasar.....	32
5.9. Algoritma <i>Page Replacement</i>	34
5.9.1. Algoritma FIFO (<i>First In First Out</i>).....	34
5.9.2. Algoritma Optimal.....	35
5.9.3. Algoritma LRU (<i>Least Recently Used</i>)	36
5.9.4. Algoritma Perkiraan LRU	37
5.9.5. Algoritma <i>Counting</i>	38
5.9.6. Algoritma <i>Page Buffering</i>	38
5.10. Strategi Alokasi <i>Frame</i>	39
5.10.1. Alokasi <i>Frame</i>	39
5.10.1.1. Jumlah <i>Frame</i> Minimum.....	39
5.10.1.2. Algoritma Alokasi.....	39
5.10.1.3. Alokasi Global lawan Lokal	40
5.10.2. <i>Thrashing</i>	41
5.10.2.1. Penyebab <i>Thrashing</i>	41
5.10.2.2. Membatasi Efek <i>Thrashing</i>	42
5.10.2.3. Model <i>Working Set</i>	42
5.10.2.4. Frekuensi <i>Page-Fault</i>	43
5.11. Pertimbangan Lain	44
5.11.1. <i>Prepaging</i>	44
5.11.2. Ukuran <i>page</i>	45
5.11.3. <i>TLBreach</i>	46
5.11.4. <i>page table</i> yang Dibalik.....	46
5.11.5. Struktur Program	47
5.11.6. <i>I/O Interlock</i>	47
5.11.7. Pemrosesan <i>Real Time</i>	47
5.11.8. Contoh pada Sistem Operasi	48
5.11.9. Windows NT.....	48
5.11.10. Solaris 2.....	48
5.11.11. Linux.....	48
6. Sistem Berkas	50
7. I/O.....	51
8. Studi Kasus: GNU/Linux	52
A. GNU Free Documentation License	53
A.1. PREAMBLE	53
A.2. APPLICABILITY AND DEFINITIONS	53
A.3. VERBATIM COPYING.....	54
A.4. COPYING IN QUANTITY	54
A.5. MODIFICATIONS.....	55
A.6. COMBINING DOCUMENTS	56
A.7. COLLECTIONS OF DOCUMENTS	56
A.8. AGGREGATION WITH INDEPENDENT WORKS.....	57

A.9. TRANSLATION	57
A.10. TERMINATION.....	57
A.11. FUTURE REVISIONS OF THIS LICENSE.....	57
A.12. How to use this License for your documents	58

Daftar Gambar

5-1. Memory Management Unit	6
5-2. <i>Two-Pass Assembler</i>	8
5-3.	12
5-4. Penerjemahan Halaman	14
5-5. Struktur MMU	16
5-6. Skema Page Table.....	17
5-7. Skema Page Table 2 tingkat.....	18
5-8. Arsitektur Segmentasi	20
5-9. Segmentasi dengan <i>paging</i>	22
5-10. Penggunaan Segmentasi dengan <i>paging</i> pada MULTICS.....	23
5-11. Penggunaan Segmentasi dengan <i>paging</i> pada INTEL 30386	24
5-12. Bagan proses <i>memory-mapped files</i>	31
5-13. Kondisi yang memerlukan <i>Page Replacement</i>	32
5-14. <i>Page Replacement</i>	32
5-15. Contoh Algoritma FIFO	34
5-16. Contoh Algoritma Optimal.....	35
5-17. Contoh Algoritma LRU	37
5-18. Derajat <i>Multiprogramming</i>	41
5-19. Kecepatan <i>page-fault</i>	44

Kata Pengantar

Buku ini merupakan hasil karya Gabungan Kelompok Kerja 21–28 Semester Genap 2002/2003 dan 41–49 Semester Ganjil 2003/2004 Mata Kuliah Sistem Operasi Fakultas Ilmu Komputer Universitas Indonesia (Fasilkom UI). Sebagai tim penyusun, kami sangat senang atas peluncuran buku ini. Penulisan buku ini bertujuan untuk mengatasi kelangkaan bahan kuliah berbahasa Indonesia, serta diharapkan akan dimanfaatkan sebagai rujukan oleh para peserta kuliah angkatan berikutnya.

Kami menyadari bahwa ini masih banyak kekurangannya. Silakan menyampaikan kritik/ tanggapan/ usulan anda ke <write03 AT yahoo03groups DOT com>.

Bab 1. Konsep Dasar Perangkat Komputer

Bab 2. Konsep Dasar Sistem Operasi

Bab 3. Proses dan Penjadwalan

Bab 4. Sinkronisasi dan *Deadlock*

Bab 5. Manajemen Memori

5.1. Manajemen Memori

5.1.1. Latar Belakang

Memori adalah pusat kegiatan pada sebuah komputer, karena setiap proses yang akan dijalankan, harus melalui memori terlebih dahulu. CPU mengambil instruksi dari memori sesuai yang ada pada *Program Counter*. Instruksi dapat berupa menempatkan / menyimpan dari / ke alamat di memori, penambahan, dan sebagainya. Tugas sistem operasi adalah mengatur peletakan banyak proses pada suatu memori. Memori harus dapat digunakan dengan baik, sehingga dapat memuat banyak proses dalam suatu waktu. Dalam manajemen memori ini, kita akan membahas bagaimana urutan alamat memori yang dibuat oleh program yang berjalan.

5.1.2. Pemberian Alamat

Sebelum masuk ke memori, suatu proses harus menunggu. Hal ini disebut *Input Queue*. Proses-proses ini akan berada dalam beberapa tahapan sebelum dieksekusi. Alamat-alamat yang dibutuhkan mungkin saja direpresentasikan dalam cara yang berbeda dalam tahapan- tahapan ini. Alamat dalam kode program masih berupa simbolik. Alamat ini akan diikat oleh kompilator ke alamat memori yang dapat diakses. Kemudian *linkage editor* dan *loader*, akan mengikat alamat fisiknya. Setiap pengikatan akan memetakan suatu ruang alamat ke lainnya.

Penjilidan alamat dapat terjadi pada 3 saat, yaitu :

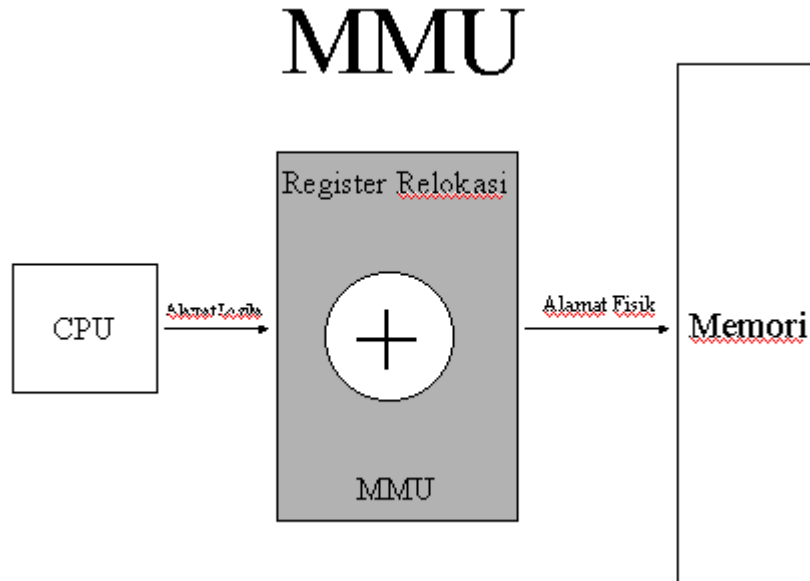
- **Waktu *compile*** : Jika diketahui pada waktu *compile*, dimana proses ditempatkan di memori. Untuk kemudian kode absolutnya dapat dibuat. Jika kemudian alamat awalnya berubah, maka harus di-*compile* ulang.
- **Waktu pemanggilan** : Jika tidak diketahui dimana proses ditempatkan di memori, maka kompilator harus membuat kode yang dapat dialokasikan. Dalam kasus pengikatan akan ditunda sampai waktu pemanggilan. Jika alamat awalnya berubah, kita hanya perlu menempatkan ulang kode, untuk menyesuaikan dengan perubahan.
- **Waktu eksekusi** : Jika proses dapat dipindahkan dari suatu segmen memori ke lainnya selama dieksekusi. Pengikatan akan ditunda sampai *run-time*.

5.1.3. Ruang Alamat Logika & Fisik

Alamat Logika adalah alamat yang dibentuk di CPU, disebut juga alamat virtual. Alamat fisik adalah alamat yang terlihat oleh memori. Waktu *compile* dan waktu pemanggilan menghasilkan daerah dimana alamat logika dan alamat fisik sama. Sedangkan pada waktu eksekusi menghasilkan alamat fisik dan logika yang berbeda. Kumpulan alamat logika yang dibuat oleh program adalah ruang alamat logika.

Kumpulan alamat fisik yang berkorespondensi dengan alamat logika disebut ruang alamat fisik. Untuk mengubah dari alamat logika ke alamat fisik diperlukan suatu perangkat keras yang bernama MMU (*Memory Management Unit*).

Gambar 5-1. Memory Management Unit



ini merupakan skema dari MMU

Register utamanya disebut register relokasi. Nilai pada register relokasi bertambah setiap alamat dibuat oleh proses pengguna, pada waktu yang sama alamat ini dikirim ke memori. Program pengguna tidak dapat langsung mengakses memori. Ketika ada program yang menunjuk ke alamat memori, kemudian mengoperasikannya, dan menaruh lagi di memori, akan di lokasikan awal oleh MMU, karena program pengguna hanya berinteraksi dengan alamat logika. Pengubahan dari alamat logika ke alamat fisik adalah pusat dari manajemen memori.

5.1.4. Pemanggilan Dinamis

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

Keuntungan dari pemanggilan dinamis adalah *routine* yang tidak digunakan tidak pernah dipanggil. Metode ini berguna untuk kode dalam jumlah banyak, ketika muncul kasus-kasus yang tidak lazim, seperti *routine* yang salah. Dalam kode yang besar, walaupun ukuran kode besar, tapi yang dipanggil dapat jauh lebih kecil.

Pemanggilan Dinamis tidak memerlukan bantuan sistem operasi. Ini adalah tanggung jawab para pengguna untuk merancang program yang mengambil keuntungan dari metode ini. Sistem operasi dapat membantu pembuat program dengan menyediakan kumpulan data *routine* untuk mengimplementasi pemanggilan dinamis.

5.1.5. Penghubungan Dinamis dan *Library* Bersama

Pada proses dengan banyak langkah, ditemukan juga penghubungan-penghubungan *library* yang dinamis, dimana menghubungkan semua *routine* yang ada di *library*. Beberapa sistem operasi hanya mendukung penghubungan yang statis, dimana seluruh *routine* yang ada dihubungkan ke dalam suatu ruang alamat. Setiap program memiliki salinan dari seluruh *library*. Konsep penghubungan dinamis, serupa dengan konsep pemanggilan dinamis. Pemanggilan lebih banyak ditunda selama waktu eksekusi, dari pada lama penundaan oleh penghubungan dinamis. Keistimewaan ini biasanya digunakan dalam sistem kumpulan *library*, seperti *library* bahasa *sub-routine*. Tanpa fasilitas ini, semua program dalam sebuah sistem, harus mempunyai salinan dari *library* bahasa mereka (atau setidaknya referensi *routine* oleh program) termasuk dalam tampilan yang dapat dieksekusi. Kebutuhan ini sangat boros baik untuk *disk*, maupun memori utama. Dengan pemanggilan dinamis, sebuah potongan dimasukkan ke dalam tampilan untuk setiap rujukan *library sub-routine*. Potongan ini adalah sebuah bagian kecil dari kode yang menunjukkan bagaimana mealokasikan *library routine* di memori dengan tepat, atau bagaimana menempatkan *library* jika *routine* belum ada.

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

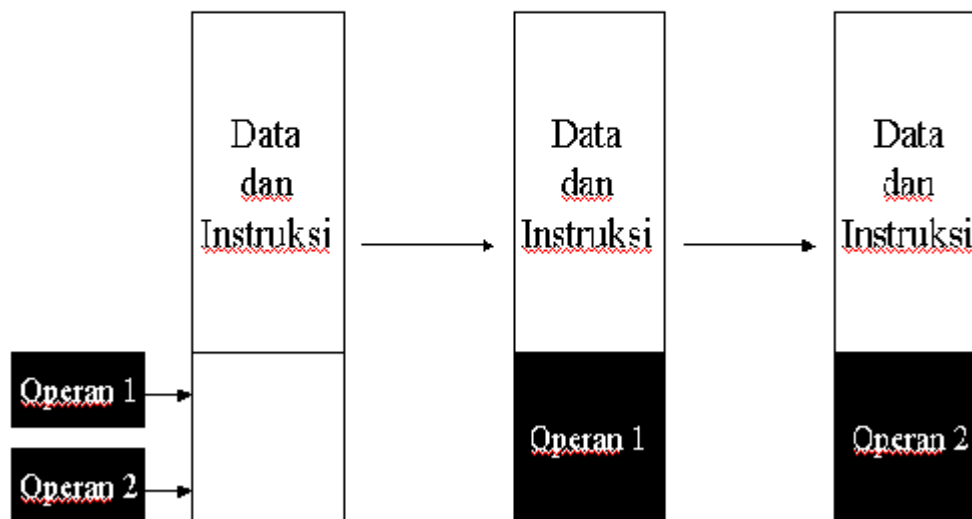
Fasilitas ini dapat diperluas menjadi pembaharuan *library*. Sebuah kumpulan data dapat ditempatkan lagi dengan versi yang lebih baru dan semua program yang merujuk ke *library* akan secara otomatis menggunakan versi yang baru. Tanpa pemanggilan dinamis, semua program akan akan membutuhkan pemanggilan kembali, untuk dapat mengakses *library* yang baru. Jadi semua program tidak secara sengaja mengeksekusi yang baru, perubahan versi *library*, informasi versi dapat dimasukkan ke dalam memori, dan setiap program menggunakan informasi versi untuk memutuskan versi mana yang akan digunakan dari salinan *library*. Sedikit perubahan akan tetap menggunakan nomor versi yang sama, sedangkan perubahan besar akan menambah satu versi sebelumnya. Karenanya program yang di *compile* 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*. Jadi seluruh *library* yang ada dapat digunakan bersama-sama. Sistem seperti ini membutuhkan bantuan sistem operasi.

5.1.6. Overlays

Overlays berguna untuk memasukkan suatu proses yang membutuhkan memori lebih besar dari yang tersedia. Idennya untuk menjaga agar di dalam memori berisi hanya instruksi dan data yang dibutuhkan dalam satuan waktu. *Routine* -nya dimasukkan ke memori secara bergantian.

Gambar 5-2. *Two-Pass Assembler*

Perakit dengan 2 operan



ini merupakan skema dari *two-Pass Assembler*

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

Untuk menempatkan semuanya sekaligus, kita akan membutuhkan 200K memori. Jika hanya 150K yang tersedia, kita tidak dapat menjalankan proses. Bagaimana pun perhatikan bahwa pass1 dan pass2 tidak harus berada di memori pada saat yang sama. Kita mendefinisikan dua *overlays*. *Overlays A* untuk pass1, tabel simbol dan *routine*, *overlays 2* untuk simbol tabel, *routine*, dan pass2.

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

Seperti dalam pemanggilan dinamis, *overlays* tidak membutuhkan bantuan dari sistem operasi. Implementasi dapat dilakukan secara lengkap oleh *user* dengan berkas struktur yang sederhana, membaca dari berkas ke memori, dan pindah dari memori tersebut, dan mengeksekusi instruksi yang baru dibaca. Sistem operasi hanya memperhatikan jika ada lebih banyak I/O dari biasanya.

Di sisi lain pemrogram harus merancang program dengan struktur *overlays* yang layak. Tugas ini membutuhkan pengetahuan yang lengkap tentang struktur dari program, kode dan struktur data.

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

5.2. Swap dan Alokasi Memori

5.2.1. Swapping

Sebuah proses harus berada di dalam memori untuk dapat dieksekusi. Sebuah proses, bagaimanapun juga, dapat di- *swap* sementara keluar memori ke sebuah *backing store (disk)*, dan kemudian dibawa masuk lagi ke memori untuk melanjutkan pengeksekusian. Sebagai contoh, asumsikan sebuah *multiprogramming environment*, dengan penjadualan algoritma penjadualan CPU *round-robin*. Ketika kuantum habis, pengatur memori akan mulai men- *swap* proses yang telah selesai, dan memasukkan proses yang lain ke dalam memori yang sudah bebas. Sementara di saat yang bersamaan, penjadual CPU akan mengalokasikan waktu untuk proses lain di dalam memori. Ketika waktu kuantum setiap proses sudah habis, proses tersebut akan di- *swap* dengan proses lain. Idealnya, *memory manager*, dapat melakukan *swapping* proses-proses tersebut dengan cukup cepat sehingga beberapa proses akan selalu berada di dalam memori dan siap untuk dieksekusi saat penjadual CPU hendak menjadual CPU. Lama kuantum pun harus cukup besar sehingga jumlah komputasi yang dilakukan selama terjadi *swap* cukup masuk akal.

Variasi dari kebijakan swapping ini, digunakan untuk algoritma penjadualan berbasis prioritas. Jika proses dengan prioritas lebih tinggi tiba dan meminta layanan, *memory manager* dapat men- *swap* keluar proses-proses yang prioritasnya rendah, sehingga proses-proses yang prioritasnya lebih tinggi tersebut dapat dieksekusi. Setelah proses-proses yang memiliki prioritas lebih tinggi tersebut selesai dieksekusi, proses-proses dengan prioritas rendah dapat di- *swap* kembali ke dalam memori dan dilanjutkan eksekusinya. Cara ini disebut juga dengan metode *roll out, roll in*.

Pada umumnya, proses yang telah di- *swap* keluar akan di- *swap* kembali menempati ruang memori yang sama dengan yang ditempatinya sebelum proses tersebut keluar dari memori. Pembatasan ini dinyatakan menurut metode pemberian alamat. Apabila pemberian alamat dilakukan pada saat *assembly time* atau *load time*, maka proses tersebut tidak dapat dipindahkan ke lokasi memori lain. Tetapi apabila pemberian alamat dilakukan pada saat *execution time*, maka proses tersebut dapat di- *swap* kembali ke dalam ruang memori yang berbeda, karena alamat fisiknya dihitung pada saat *execution time*.

Swapping membutuhkan sebuah *backing store*. *Backing store* pada umumnya adalah sebuah *fast disk*, dan harus cukup untuk menampung salinan dari seluruh *memory image* untuk semua *user*, dan harus mendukung akses langsung terhadap *memory image* tersebut. Sistem mengatur *ready queue* yang berisikan semua proses yang *memory image*-nya berada di memori dan siap untuk dijalankan. Saat sebuah penjadual CPU ingin menjalankan sebuah proses, ia akan memeriksa apakah proses yang

mengantri di *ready queue* tersebut sudah berada di dalam memori tersebut atau belum. Apabila belum, penjadual CPU akan melakukan *swap out* terhadap proses-proses yang berada di dalam memori sehingga tersedia tempat untuk memasukkan proses yang hendak dieksekusi tersebut. Setelah itu *register* dikembalikan seperti semula dan proses yang diinginkan akan dieksekusi.

Waktu *context-switch* dalam sebuah sistem yang melakukan *swapping* pada umumnya cukup tinggi. Untuk mendapatkan gambaran mengenai waktu *context-switch*, akan diilustrasikan sebuah contoh. Misalkan ada sebuah proses sebesar 1 MB, dan media yang digunakan sebagai *backing store* adalah sebuah *hard disk* dengan kecepatan transfer 5 MBps. Waktu yang dibutuhkan untuk mentransfer proses 1 MB tersebut dari atau ke dalam memori adalah:

$$1000 \text{ KB} / 5000 \text{ KBps} = 1/5 \text{ detik} = 200 \text{ milidetik}$$

Apabila diasumsikan *head seek* tidak dibutuhkan dan rata-rata waktu latensi adalah 8 milidetik, satu proses *swapping* memakan waktu 208 milidetik. Karena kita harus melakukan proses *swapping* sebanyak 2 kali, (memasukkan dan mengeluarkan dari memori), maka keseluruhan waktu yang dibutuhkan adalah 416 milidetik.

Untuk penggunaan CPU yang efisien, kita menginginkan waktu eksekusi kita relatif panjang apabila dibandingkan dengan waktu *swap* kita. Sehingga, misalnya dalam penjadualan CPU menggunakan metode *round robin*, kuantum yang kita tetapkan harus lebih besar dari 416 milidetik.

Bagian utama dari waktu *swap* adalah waktu transfer. Besar waktu transfer berhubungan langsung dengan jumlah memori yang di-*swap*. Jika kita mempunyai sebuah computer dengan *main memory* 128 MB dan sistem operasi memakan tempat 5 MB, besar proses *user* maksimal adalah 123 MB. Bagaimanapun juga, proses *user* pada kenyataannya dapat berukuran jauh lebih kecil dari angka tersebut. Bahkan terkadang hanya berukuran 1 MB. Proses sebesar 1 MB dapat di-*swap* hanya dalam waktu 208 milidetik, jauh lebih cepat dibandingkan men-*swap* proses sebesar 123 MB yang akan menghabiskan waktu 24.6 detik. Oleh karena itu, sangatlah berguna apabila kita mengetahui dengan baik berapa besar memori yang dipakai oleh proses *user*, bukan sekedar dengan perkiraan saja. Setelah itu, kita dapat mengurangi besar waktu *swap* dengan cara hanya men-*swap* hanya proses-proses yang benar-benar membutuhkannya. Agar metode ini bisa dijalankan dengan efektif, *user* harus menjaga agar sistem selalu memiliki informasi mengenai perubahan kebutuhan memori. Oleh karena itu, proses yang membutuhkan memori dinamis harus melakukan *system call* (*request memory* dan *release memory*) untuk memberikan informasi kepada sistem operasi akan perubahan kebutuhan memori.

Swapping dipengaruhi oleh banyak faktor. Jika kita hendak men-*swap* suatu proses, kita harus yakin bahwa proses tersebut siap. Hal yang perlu diperhatikan adalah kemungkinan proses tersebut sedang menunggu I/O. Apabila I/O secara *asynchronous* mengakses memori *user* untuk I/O *buffer*, maka proses tersebut tidak dapat di-*swap*. Bayangkan apabila sebuah operasi I/O berada dalam antrian karena *device* I/O-nya sedang sibuk. Kemudian kita hendak mengeluarkan proses P1 dan memasukkan proses P2. Operasi I/O mungkin akan berusaha untuk memakai memori yang sekarang seharusnya akan ditempati oleh P2. Cara untuk mengatasi masalah ini adalah:

1. Hindari men-*swap* proses yang sedang menunggu I/O.
2. Lakukan eksekusi operasi I/O hanya di *buffer* sistem operasi.

Hal tersebut akan menjaga agar transfer antara *buffer* sistem operasi dan proses memori hanya terjadi saat si proses di-*swap in*.

Pada masa sekarang ini, proses *swapping* secara dasar hanya digunakan di sedikit sistem. Hal ini dikarenakan *swapping* menghabiskan terlalu banyak waktu *swap* dan memberikan waktu eksekusi yang

terlalu kecil sebagai solusi dari manajemen memori. Akan tetapi, banyak sistem yang menggunakan versi modifikasi dari metode *swapping* ini.

Salah satu sistem operasi yang menggunakan versi modifikasi dari metode *swapping* ini adalah UNIX. *Swapping* berada dalam keadaan non-aktif, sampai apabila ada banyak proses yang berjalan yang menggunakan cukup besar memori. *Swapping* akan berhenti lagi apabila jumlah proses yang berjalan sudah berkurang.

Pada awal pengembangan komputer pribadi, tidak banyak perangkat keras (atau sistem operasi yang memanfaatkan perangkat keras) yang dapat mengimplementasikan memori manajemen yang baik, melainkan digunakan untuk menjalankan banyak proses berukuran besar dengan menggunakan versi modifikasi dari metode *swapping*. Salah satu contoh yang baik adalah Microsoft Windows 3.1, yang mendukung eksekusi proses berkesinambungan. Apabila suatu proses baru hendak dijalankan dan tidak terdapat cukup memori, proses yang lama perlu dimasukkan ke dalam *disk*. Sistem operasi ini, bagaimanapun juga, tidak mendukung *swapping* secara keseluruhan karena yang lebih berperan menentukan kapan proses *swapping* akan dilakukan adalah *user* dan bukan penjadual CPU. Proses-proses yang sudah dikeluarkan akan tetap berada di luar memori sampai *user* memilih proses yang hendak dijalankan. Sistem-sistem operasi Microsoft selanjutnya, seperti misalnya Windows NT, memanfaatkan fitur *Memory Management Unit*.

5.2.2. Pengalokasian Memori

5.2.2.1. Contiguous Memory Allocation

Main memory harus dapat melayani baik sistem operasi maupun proses *user*. Oleh karena itu kita harus mengalokasikan pembagian memori seefisien mungkin. Salah satunya adalah dengan cara ***contiguous memory allocation***. *Contiguous memory allocation* berarti alamat memori diberikan kepada proses secara berurutan dari kecil ke besar. Keuntungan menggunakan *contiguous memory allocation* dibandingkan menggunakan *non-contiguous memory allocation* adalah:

1. Sederhana
2. Cepat
3. Mendukung proteksi memori

Sedangkan kerugian dari menggunakan *contiguous memory allocation* adalah apabila tidak semua proses dialokasikan di waktu yang sama, akan menjadi sangat tidak efektif sehingga mempercepat habisnya memori.

Contiguous memory allocation dapat dilakukan baik menggunakan sistem partisi banyak, maupun menggunakan sistem partisi tunggal. Sistem partisi tunggal berarti alamat memori yang akan dialokasikan untuk proses adalah alamat memori pertama setelah pengalokasian sebelumnya. Sedangkan sistem partisi banyak berarti sistem operasi menyimpan informasi tentang semua bagian memori yang tersedia untuk dapat diisi oleh proses-proses (disebut *hole*). Sistem partisi banyak kemudian dibagi lagi menjadi sistem partisi banyak tetap, dan sistem partisi banyak dinamis. Hal yang membedakan keduanya adalah untuk sistem partisi banyak tetap, memori dipartisi menjadi blok-blok yang ukurannya tetap yang ditentukan dari awal. Sedangkan sistem partisi banyak dinamis artinya memori dipartisi menjadi bagian-bagian dengan jumlah dan besar yang tidak tentu. Untuk selanjutnya, kita akan memfokuskan pembahasan pada sistem partisi banyak.

Sistem operasi menyimpan sebuah tabel yang menunjukkan bagian mana dari memori yang memungkinkan untuk menyimpan proses, dan bagian mana yang sudah diisi. Pada intinya, seluruh memori dapat diisi oleh proses *user*. Saat sebuah proses datang dan membutuhkan memori, CPU akan mencari *hole* yang cukup besar untuk menampung proses tersebut. Setelah menemukannya, CPU akan mengalokasikan memori sebanyak yang dibutuhkan oleh proses tersebut, dan mempersiapkan sisanya untuk menampung proses-proses yang akan datang kemudian (seandainya ada).

Saat proses memasuki sistem, proses akan dimasukkan ke dalam *input queue*. Sistem operasi akan menyimpan besar memori yang dibutuhkan oleh setiap proses dan jumlah memori kosong yang tersedia, untuk menentukan proses mana yang dapat diberikan alokasi memori. Setelah sebuah proses mendapat alokasi memori, proses tersebut akan dimasukkan ke dalam memori. Setelah proses tersebut dimatikan, proses tersebut akan melepas memori tempat dia berada, yang mana dapat diisi kembali oleh proses lain dari *input queue*.

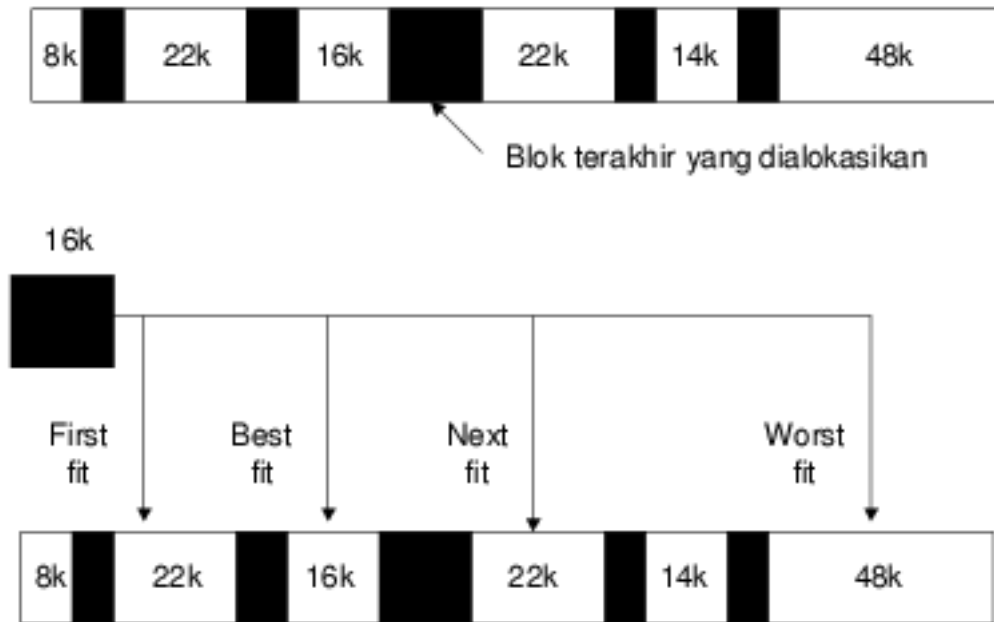
Sistem operasi setiap saat selalu memiliki catatan jumlah memori yang tersedia dan *input queue*. Sistem operasi dapat mengatur *input queue* berdasarkan algoritma penjadwalan yang digunakan. Memori dialokasikan untuk proses sampai akhirnya kebutuhan memori dari proses selanjutnya tidak dapat dipenuhi (tidak ada *hole* yang cukup besar untuk menampung proses tersebut). Sistem operasi kemudian dapat menunggu sampai ada blok memori cukup besar yang kosong, atau dapat mencari proses lain di *input queue* yang kebutuhan memorinya memenuhi jumlah memori yang tersedia.

Pada umumnya, kumpulan *hole-hole* dalam berbagai ukuran tersebar di seluruh memori sepanjang waktu. Apabila ada proses yang datang, sistem operasi akan mencari *hole* yang cukup besar untuk menampung memori tersebut. Apabila *hole* yang tersedia terlalu besar, akan dipecah menjadi 2. Satu bagian akan dialokasikan untuk menerima proses tersebut, sementara bagian lainnya tidak digunakan dan siap menampung proses lain. Setelah proses selesai, proses tersebut akan melepas memori dan mengembalikannya sebagai *hole-hole*. Apabila ada 2 *hole* yang kecil yang berdekatan, keduanya akan bergabung untuk membentuk *hole* yang lebih besar. Pada saat ini, sistem harus memeriksa apakah ada proses yang menunggu yang dapat dimasukkan ke dalam ruang memori yang baru terbentuk tersebut.

Hal ini disebut **Permasalahan *storage-allocation* dinamis**, yakni bagaimana memenuhi permintaan sebesar n dari kumpulan *hole-hole* yang tersedia. Ada berbagai solusi untuk mengatasi hal ini, yaitu:

1. *First fit* : Mengalokasikan *hole* pertama ditemukan yang besarnya mencukupi. Pencarian dimulai dari awal.
2. *Best fit* : Mengalokasikan *hole* dengan besar minimum yang mencukupi permintaan.
3. *Next fit* : Mengalokasikan *hole* pertama ditemukan yang besarnya mencukupi. Pencarian dimulai dari akhir pencarian sebelumnya.
4. *Worst fit* : Mengalokasikan *hole* terbesar yang ada.

Gambar 5-3.



Memilih yang terbaik diantara keempat metode diatas adalah sepenuhnya tergantung kepada *user* , karena setiap metode memiliki kelebihan dan kekurangan masing-masing. Menggunakan *best fit* dan *worst fit* berarti kita harus selalu memulai pencarian hole dari awal, kecuali apabila *hole-hole* sudah disusun berdasarkan ukuran. Metode *worst fit* akan menghasilkan sisa *hole* yang terbesar, sementara metode *best fit* akan menghasilkan sisa *hole* yang terkecil.

5.2.2.2. Fragmentasi

Fragmentasi adalah munculnya *hole-hole* yang tidak cukup besar untuk menampung permintaan dari proses. Fragmentasi dapat berupa fragmentasi internal maupun fragmentasi eksternal. Fragmentasi ekstern muncul apabila jumlah keseluruhan memori kosong yang tersedia memang mencukupi untuk menampung permintaan tempat dari proses, tetapi letaknya tidak berkesinambungan atau terpecah menjadi beberapa bagian kecil sehingga proses tidak dapat masuk. Sedangkan fragmentasi intern muncul apabila jumlah memori yang diberikan oleh penjadual CPU untuk ditempati proses lebih besar daripada yang diminta proses karena adanya selisih antara permintaan proses dengan alokasi *hole* yang sudah ditetapkan.

Algoritma *storage-allocation* dinamis manapun yang digunakan, tetap tidak bisa menutup kemungkinan terjadinya fragmentasi. Bahkan hal ini bisa menjadi fatal. Salah satu kondisi terburuk adalah apabila kita memiliki memori terbuang setiap 2 proses. Apabila semua memori terbuang itu digabungkan, bukan tidak mungkin akan cukup untuk menampung sebuah proses. Sebuah contoh statistik menunjukkan bahwa saat menggunakan metode *first fit* , bahkan setelah dioptimisasi, dari N blok teralokasi, sebanyak 0.5N blok lain akan terbuang karena fragmentasi. Jumlah sebanyak itu berarti kurang lebih setengah dari memori tidak dapat digunakan. Hal ini disebut dengan **aturan 50%** .

Fragmentasi ekstern dapat diatasi dengan beberapa cara, diantaranya adalah:

1. **Compactation** , yaitu mengatur kembali isi memori agar memori yang kosong diletakkan bersama di suatu bagian yang besar, sehingga proses dapat masuk ke ruang memori kosong tersebut.
2. **Paging**
3. **Segmentasi**

Fragmentasi intern hampir tidak dapat dihindarkan apabila kita menggunakan sistem partisi banyak berukuran tetap, mengingat besar *hole* yang disediakan selalu tetap.

5.3. Pemberian Halaman

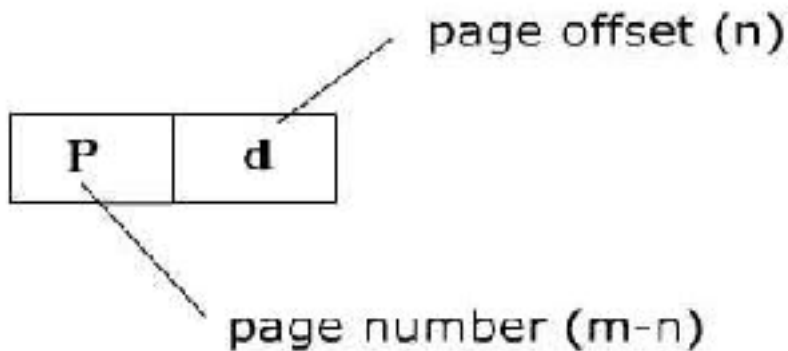
Yang dimaksud dengan pemberian halaman adalah suatu metode yang memungkinkan suatu alamat fisik memori yang tersedia dapat tidak berurutan. Pemberian halaman bisa menjadi solusi untuk pemecahan masalah luar. Untuk bisa mengimplementasikan solusi ini adalah melalui penggunaan dari skema pemberian halaman. Dengan pemberian halaman bisa mencegah masalah penting dari pengepasan besar ukuran memori yang bervariasi kedalam penyimpanan cadangan. Ketika beberapa pecahan kode dari data yang tersisa di memori utama perlu untuk ditukar keluar, harus ditemukan ruang untuk penyimpanan cadangan. Masalah pemecahan kode didiskusikan dengan kaitan bahwa pengaksesannya lebih lambat. Biasanya bagian yang menunjang untuk pemberian halaman telah ditangani oleh *hardware* . Bagaimanapun, desain yang ada baru2 ini telah mengimplementasikan dengan menggabungkan hardware dan sistem operasi, terutama pada 64 bit *microprocessor* .

5.3.1. Metode Dasar

Jadi metode dasar yang digunakan adalah dengan memecah memori fisik menjadi blok-blok berukuran tetap yang akan disebut sebagai frame. selanjutnya memori logis juga dipecah menjadi blok2 dengan ukuran yang sama disebut sebagai halaman. Selanjutnya kita membuat suatu tabel halaman yang akan menterjemahkan memori logis kita kedalam memori fisik. Jika suatu proses ingin dieksekusi maka memori logis akan melihat dimanakah dia akan ditempatkan di memori fisik dengan melihat kedalam tabel halamannya.

Untuk jelasnya bisa dilihat pada gambar 1. Kita lihat bahwa setiap alamat yang dihasilkan oleh CPU dibagi-bagi menjadi 2 bagian yaitu sebuah nomor halaman (p) dan sebuah offset halaman (d). Nomor halaman ini akan digunakan sebagai indeks untuk tabel halaman. Tabel halaman mengandung basis alamat dari tiap2 halaman di memori fisik. Basis ini dikombinasikan dengan offset halaman untuk menentukan alamat memori fisik yang dikirim ke unit memori.

Gambar 5-4. Penerjemahan Halaman



5.3.2. Keuntungan dan Kerugian Pemberian Halaman

- Jika kita membuat ukuran dari masing-masing halaman menjadi lebih besar

Keuntungan :

Akses memori akan relatif lebih cepat

Kerugian :

Kemungkinan terjadinya fragmentasi intern sangat besar

- Jika kita membuat ukuran dari masing-masing halaman menjadi lebih kecil

Keuntungan :

Kemungkinan terjadinya internal Fragmentasi akan menjadi lebih kecil

Kerugian :

Akses memori akan relatif lebih lambat

5.4. Struktur Page Table

Sebagian besar komputer modern memiliki *hardware* istimewa yaitu **unit manajemen memori (MMU)**. Unit tersebut berada diantara CPU dan unit memori. Jika CPU ingin meng *access* memori (misalnya untuk me *load* suatu instruksi atau *load* dan *store* suatu data), maka CPU mengirimkan alamat memori yang bersangkutan ke MMU, yang akan menerjemahkannya ke alamat lain sebelum melanjutkannya ke

unit memori. Alamat yang dihasilkan oleh CPU, setelah adanya *indexing* atau aritmatik *addressing-mode* lainnya disebut **alamat logis** (*virtual address*). Sedangkan alamat yang didapatkan setelah diterjemahkan oleh CPU disebut **alamat fisik** (*physical address*).

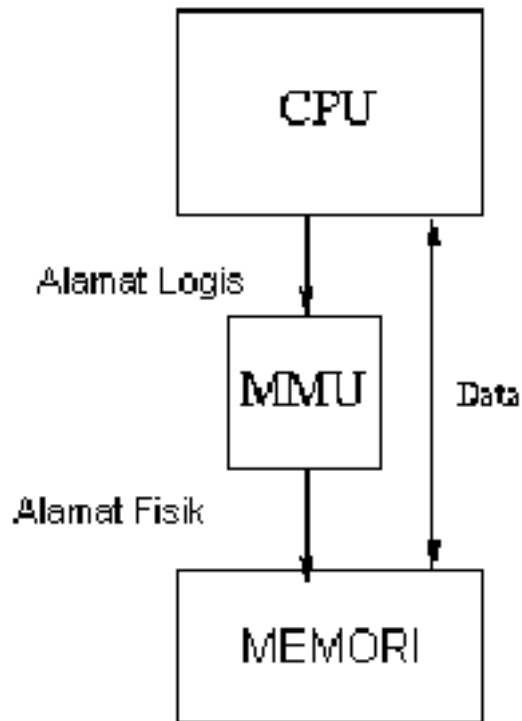
Biasanya, penterjemahan dilakukan di *granularity* dari suatu halaman. Setiap halaman mempunyai pangkat 2 bytes, diantara 1024 dan 8192 bytes. Jika alamat logis p dipetakan ke alamat fisik f (dimana p adalah kelipatan dari ukuran halaman), maka alamat $p+o$ dipetakan ke alamat fisik $f+o$ untuk setiap *offset* o kurang dari ukuran halaman. Dengan kata lain, setiap halaman dipetakan ke *contiguous region* di alamat fisik yang disebut *frame*.

MMU yang mengizinkan *contiguous region* dari alamat logis dipetakan ke *frame* yang tersebar disekitar alamat fisik membuat sistem operasi lebih mudah pekerjaannya saat mengalokasikan memori. Lebih penting lagi, MMU juga mengizinkan halaman yang tidak sering digunakan bisa disimpan di *disk*. Cara kerjanya adalah sbb: Tabel yang digunakan oleh MMU mempunyai *valid bit* untuk setiap halaman di bagian alamat logis. Jika bit tersebut di set, maka penterjemahan oleh alamat logis di halaman itu berjalan normal. Akan tetapi jika di *clear*, adanya usaha dari CPU untuk meng *access* suatu alamat di halaman tersebut menghasilkan suatu interupsi yang disebut *page fault trap*. Sistem operasi telah mempunyai *interrupt handler* untuk *page fault*, juga bisa digunakan untuk mengatasi interupsi jenis yang lain. *Handler* inilah yang akan bekerja untuk mendapatkan halaman yang diminta ke memori.

Untuk lebih jelasnya, saat *page fault* dihasilkan untuk halaman $p1$, *interrupt handler* melakukan hal-hal berikut ini:

- Mencari dimana isi dari halaman $p1$ disimpan di *disk*. Sistem operasi menyimpan informasi ini di dalam table. Ada kemungkinan bahwa halaman tersebut tidak ada dimana-mana, misalnya pada kasus saat referensi memori adalah *bug*. Pada kasus tersebut, sistem operasi mengambil beberapa langkah kerja seperti mematikan prosesnya. Dan jika diasumsikan halamannya berada dalam *disk*:
- Mencari halaman lain yaitu $p2$ yang dipetakan ke *frame* lain f dari alamat fisik yang tidak banyak dipergunakan.
- Menyalin isi dari *frame* f keluar dari *disk*.
- Menghapus *valid bit* halaman $p2$ sehingga sebagian referensi dari halaman $p2$ akan menyebabkan *page fault*.
- Menyalin data halaman $p1$ dari *disk* ke *frame* f .
- *Update* tabel MMU sehingga halaman $p1$ dipetakan ke *frame* f .
- Kembali dari interupsi dan mengizinkan CPU mengulang instruksi yang menyebabkan interupsi tersebut.

Gambar 5-5. Struktur MMU

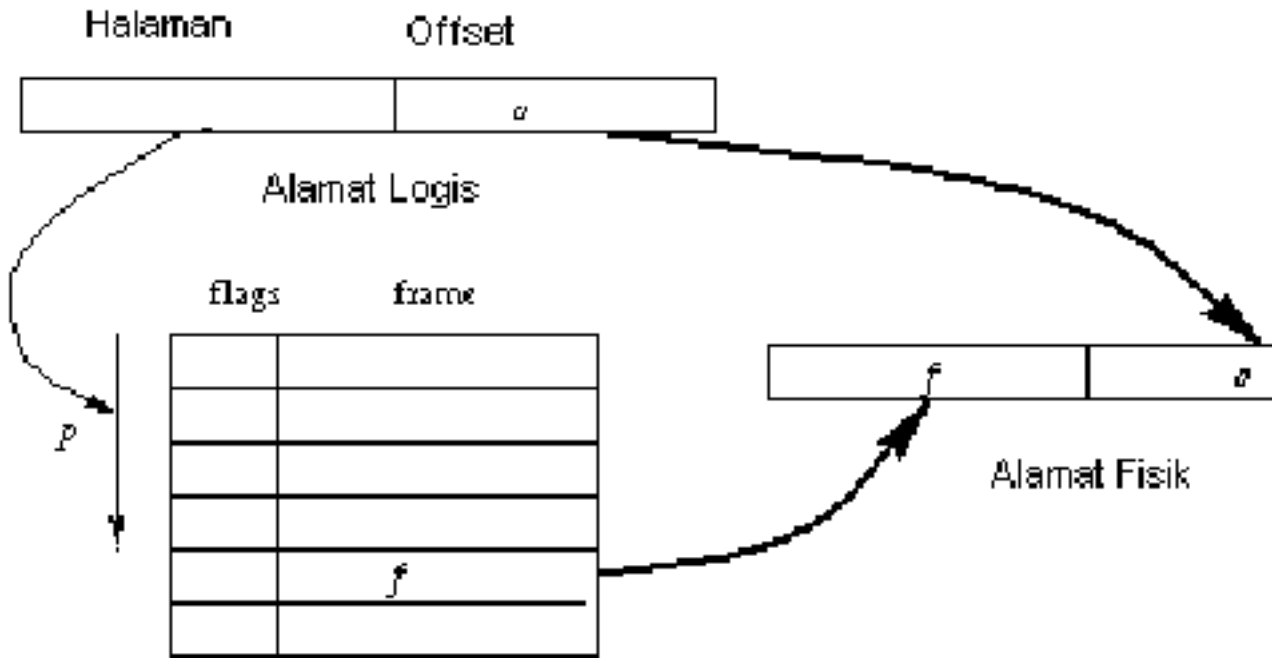


5.4.1. Page Table

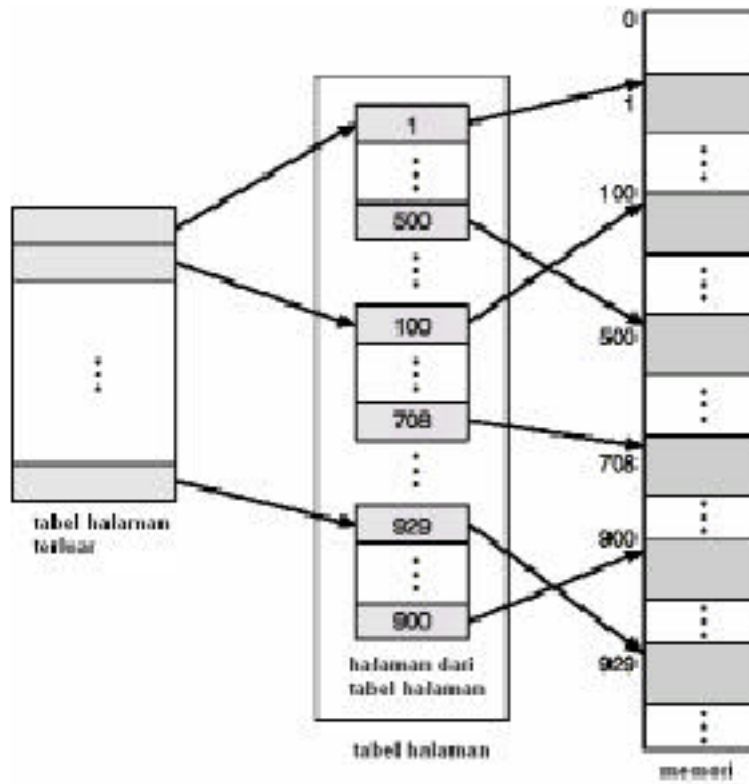
Pada dasarnya MMU terdiri dari table halaman yang merupakan sebuah rangkaian array dari masukan-masukan (*entries*) yang mempunyai indeks berupa nomor halaman (p). Setiap masukan terdiri dari *flags* (contohnya *valid bit*) dan nomor *frame*. Alamat fisik dibentuk dengan menggabungkan nomor *frame* dengan *offset*, yaitu bit paling rendah dari alamat logis.

Setiap sistem operasi mempunyai metodenya sendiri untuk menyimpan *page table*. Sebagian besar mengalokasikan *page table* untuk setiap proses. Penunjuk ke *page table* 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 *page table* perangkat keras yang benar dari tempat penyimpanan *page table* dari *user*.

Gambar 5-6. Skema Page Table



Gambar 5-7. Skema Page Table 2 tingkat



5.4.2. Pemberian Page Secara *Multilevel*

Banyak sistem komputer modern mendukung ruang alamat logis yang sangat luas (2 pangkat 32 sampai 2 pangkat 64). Pada lingkungan seperti itu *page table* -nya sendiri menjadi besar sekali. Untuk contoh, misalkan suatu sistem dengan ruang alamat logis 32-bit. Jika ukuran halaman di sistem seperti itu adalah 4K byte (2 pangkat 12), maka *page table* 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 *page table* -nya saja. Jelasnya, kita tidak akan mau mengalokasikan *page table* secara berdekatan di dalam memori. Satu solusi sederhananya adalah dengan membagi *page table* menjadi potongan-potongan yang lebih kecil lagi. Ada beberapa cara yang berbeda untuk menyelesaikan ini.

5.4.3. Page Table secara *Inverted*

Biasanya, setiap proses mempunyai *page table* yang diasosiasikan dengannya. *Page table* hanya punya satu masukan untuk setiap halaman proses tersebut sedang digunakan (atau satu slot untuk setiap alamat maya, tanpa memperhatikan 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 dibutuhkan untuk tetap menjaga bagaimana memori fisik lain sedang digunakan.

5.4.4. Berbagi Page

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

5.5. Segmentasi

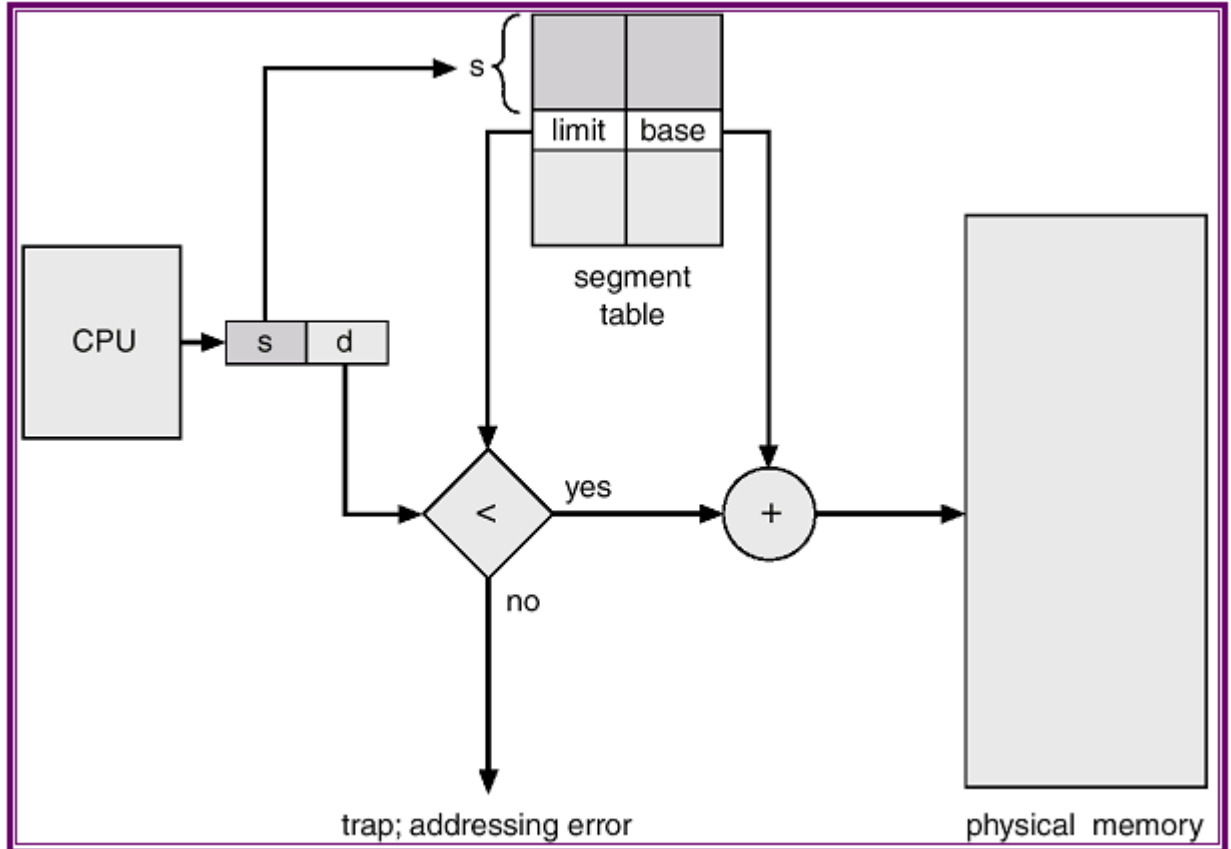
Segmentasi adalah skema manajemen memori dengan cara membagi memori menjadi segmen-segmen. Dengan demikian, sebuah program dibagi menjadi segmen-segmen. *Segmen* adalah sebuah *logical unit*, yaitu unit yang terdiri dari beberapa bagian yang berjenis yang sama. Contoh: program utama, variabel lokal, *procedure* dan sebagainya. Berbeda dengan *page*, ukuran tiap segmen tidak harus sama dan memiliki 'ciri' tertentu. Ciri tertentu itu adalah nama segmen dan panjang segmen. Nama segmen dirujuk oleh nomor segmen sedangkan panjang segmen ditentukan oleh *offset*.

5.5.1. Arsitektur Segmentasi

Ukuran tiap segmen tidak harus sama. Saat sebuah program atau proses dimasukkan ke CPU, segmen yang berbeda dapat ditempatkan dimana saja di dalam *main memory* (dapat menggunakan cara *first-fit* atau *best-fit*).

Alamat logis dari sebuah segmen adalah alamat dua dimensi, sedangkan alamat fisik memori adalah alamat satu dimensi. Oleh karena itu, agar implementasinya menjadi mudah (dari alamat logis ke alamat fisik) diperlukan Tabel Segmen yang terdiri dari *base* dan *limit*. *Base* menunjukkan alamat awal segmen (dari alamat fisik) dan *limit* menunjukkan panjang segmen.

Gambar 5-8. Arsitektur Segmentasi



alamat logis ----> s dan d s ---> nomor segmen / index di dalam tabel segmen d ---> *offset* Jika *offset* kurang dari nol dan tidak lebih besar dari besarnya *limit* maka *base* akan dijumlahkan dengan d (*offset*), yang dijumlahkan itu adalah alamat fisik dari segmen tersebut.

5.5.2. Saling Berbagi dan Proteksi

Segmen dapat terbagi jika terdapat elemen di tabel segmen yang berasal dari dua proses yang berbeda yang menunjuk pada alamat fisik yang sama. Saling berbagi ini muncul di level segmen dan pada saat ini terjadi semua informasi dapat turut terbagi. Proteksi dapat terjadi karena ada bit-proteksi yang berhubungan dengan setiap elemen dari segmen tabel. Bit-proteksi ini berguna untuk mencegah akses ilegal ke memori. Caranya: menempatkan sebuah *array* di dalam segmen itu sehingga *Memory Management Hardware* secara otomatis akan mengecek indeks *array* -nya legal atau tidak.

5.5.3. Alokasi yang Dinamis

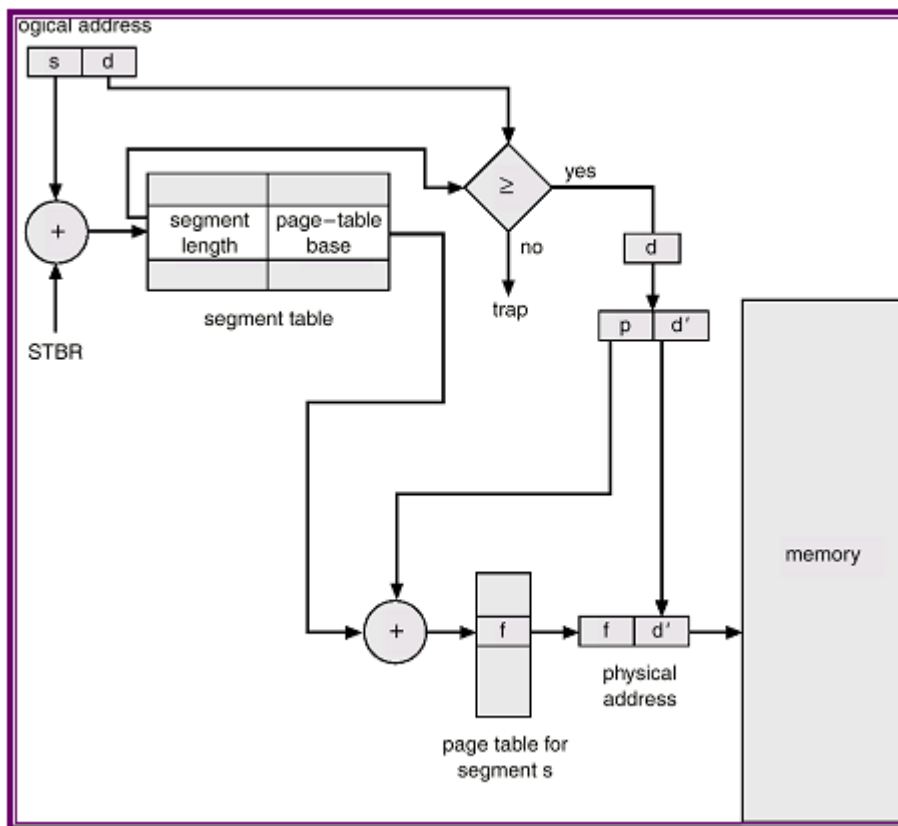
5.5.4. Masalah dalam Segmentasi

- Segmen dapat Membesar
- Muncul Fragmentasi Luar
- Bila Ada Proses yang Besar

5.5.5. Segmentasi dengan *paging*

Kelebihan *paging* : tidak ada fragmentasi luar - alokasinya cepat. Kelebihan Segmentasi: saling berbagi - proteksi.

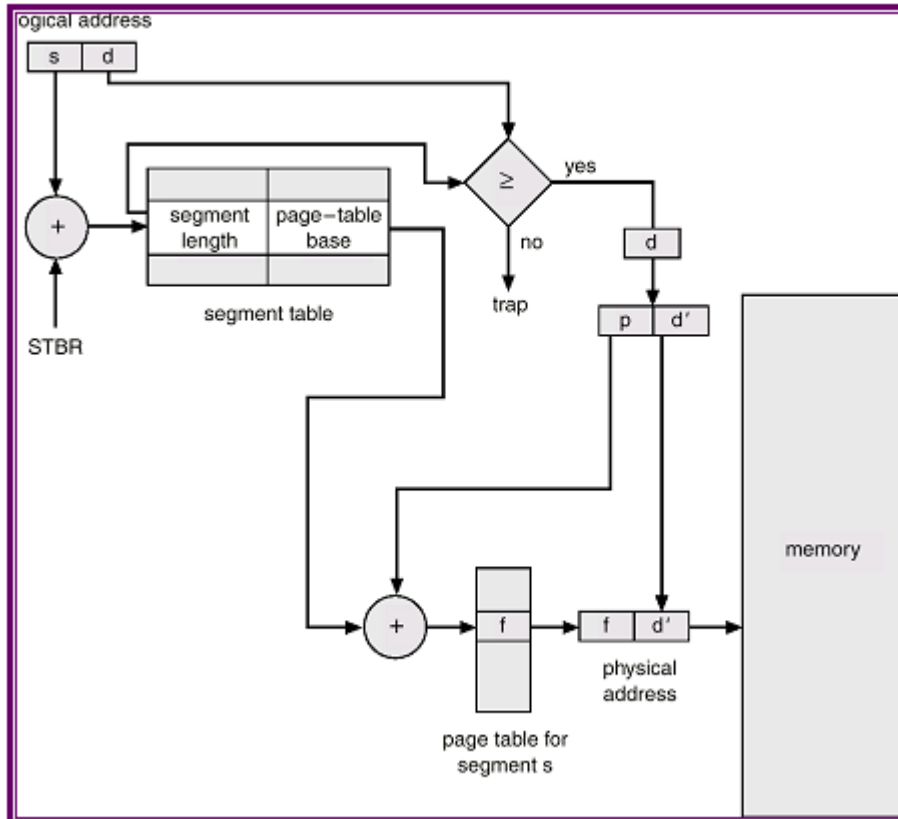
Gambar 5-9. Segmentasi dengan *paging*



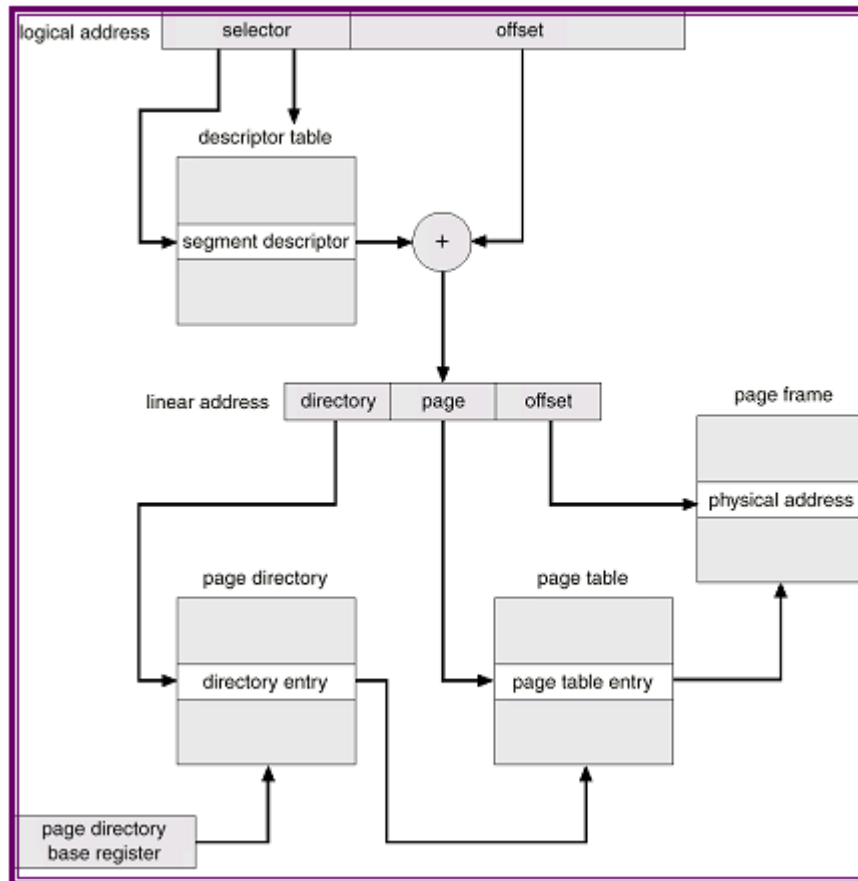
5.5.6. Penggunaan Segmentasi

MULTICS

Gambar 5-10. Penggunaan Segmentasi dengan *paging* pada MULTICS



INTEL

Gambar 5-11. Penggunaan Segmentasi dengan *paging* pada INTEL 30386

5.6. Pengantar Memori Virtual; Demand Paging

Manajemen memori pada intinya adalah menempatkan semua bagian proses yang akan dijalankan kedalam memori sebelum proses itu dijalankan. Untuk itu, semua bagian proses itu harus memiliki tempat sendiri di dalam memori fisik.

Tetapi tidak semua bagian dari proses itu akan dijalankan, misalnya:

- Pernyataan atau pilihan yang hanya akan dieksekusi pada kondisi tertentu. Contohnya adalah pesan-pesan *error* yang hanya muncul bila terjadi kesalahan saat program dijalankan.
- Fungsi-fungsi yang jarang digunakan.

- Pengalokasian memori yang lebih besar dari yang dibutuhkan. Contoh: *array*, *list* dan tabel.

Pada memori berkapasitas besar, hal-hal ini tidak akan menjadi masalah. Akan tetapi, pada memori yang sangat terbatas, hal ini akan menurunkan optimalisasi utilitas dari ruang memori fisik. Sebagai solusi dari masalah-masalah ini digunakanlah konsep memori virtual.

5.6.1. Pengertian

Memori virtual adalah suatu teknik yang memisahkan antara memori logis dan memori fisiknya. Teknik ini menyembunyikan aspek-aspek fisik memori dari user dengan menjadikan memori sebagai lokasi alamat virtual berupa *byte* yang tidak terbatas dan menaruh beberapa bagian dari memori virtual yang berada di memori logis.

Berbeda dengan keterbatasan yang dimiliki oleh memori fisik, memori virtual dapat menampung program dalam skala besar, melebihi daya tampung dari memori fisik yang tersedia.

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

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

5.6.2. Keuntungan

Sebagaimana dikatakan di atas bahwa hanya sebagian dari program yang diletakkan di memori fisik. Hal ini memberikan keuntungan:

- Berkurangnya proses I/O yang dibutuhkan (lalu lintas I/O menjadi rendah). Misalnya untuk program butuh membaca dari *disk* dan memasukkan dalam memory setiap kali diakses.
- *Space* menjadi lebih leluasa karena berkurangnya memori fisik yang digunakan. Contoh, untuk program 10 MB tidak seluruh bagian dimasukkan dalam memori fisik. Pesan-pesan error hanya dimasukkan jika terjadi error.
- Meningkatkan respon, karena 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*.

5.6.3. Implementasi

Gagasan utama dari memori virtual adalah ukuran gabungan program, data dan stack melampaui jumlah memori fisik yang tersedia. Sistem operasi menyimpan bagian-bagian proses yang sedang digunakan di memori fisik (*main memory*) dan sisanya diletakkan di disk. Begitu bagian yang berada di disk

diperlukan, maka bagian di memori yang tidak diperlukan akan dikeluarkan dari memori fisik (*swap-out*) dan diganti (*swap-in*) oleh bagian disk yang diperlukan itu.

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

Memori virtual dapat dilakukan melalui dua cara:

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

5.6.4. Demand Paging

Demand paging atau permintaan pemberian halaman adalah salah satu implementasi dari memori virtual yang paling umum digunakan. *Demand paging*) pada prinsipnya hampir sama dengan permintaan halaman (*paging*) hanya saja halaman (*page*) tidak akan dibawa ke ke dalam memori fisik sampai ia benar-benar diperlukan. Untuk itu diperlukan bantuan perangkat keras untuk mengetahui lokasi dari *page* saat ia diperlukan.

Karena demand paging merupakan implementasi dari memori virtual, maka keuntungannya sama dengan keuntungan memori virtual, yaitu:

- Sedikit I/O yang dibutuhkan.
- Sedikit memori yang dibutuhkan
- Respon yang lebih cepat
- Dapat melayani lebih banyak *user*

5.6.4.1. Permasalahan pada Demand Paging

Ada tiga kemungkinan kasus yang dapat terjadi pada saat dilakukan pengecekan pada *page* yang dibutuhkan, yaitu: *Page* ada dan sudah berada di memori. Statusnya *valid* ("1"). *Page* ada tetapi masih berada di disk belum diberada di memori (harus menunggu sampai dimasukkan). Statusnya *invalid* ("0"). *Page* tidak ada, baik di memori maupun di *disk* (*invalid reference* --> *abort*).

Saat terjadi kasus kedua dan ketiga, maka proses dinyatakan mengalami *page fault* . Perangkat keras akan menjebaknya ke dalam Sistem Operasi.

5.6.4.2. Skema Bit Valid - Tidak Valid

Dengan meminjam konsep yang sudah pernah dijelaskan dalam Pokok Bahasan 5, maka dapat ditentukan *page* mana yang ada di dalam memori dan mana yang tidak ada di dalam memori.

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

Pengasetan 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.

5.6.4.3. Penanganan *Page Fault*

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

- Memeriksa tabel internal yang dilengkapi dengan PCB untuk menentukan valid atau tidaknya bit.
- Apabila *invalid*, program akan di-*terminate* (interupsi oleh *illegal address trap*). Jika valid tapi proses belum dibawa ke *page*, maka kita *page* sekarang.
- Memilih *frame* kosong (*free-frame*), misalnya dari *free-frame list*. Jika tidak ditemui ada *frame* yang kosong, maka dilakukan *swap-out* dari memori. *Frame* mana yang harus di-*swap-out* akan ditentukan oleh algoritma (lihat sub bab *Page Replacement*).
- Menjadwalkan operasi disk untuk membaca *page* yang diinginkan ke *frame* yang baru dialokasikan.
- Ketika pembacaan komplet, ubah *validation bit* menjadi "1" yang berarti *page* sudah diidentifikasi ada di memori.
- Mengulang instruksi yang tadi telah sempat diinterupsi. Jika tadi *page fault* terjadi saat instruksi di-*fetch*, maka akan dilakukan *fetching* lagi. Jika terjadi saat operan sedang di-*fetch*, maka harus dilakukan *fetch* ulang, *decode*, dan *fetch* operan lagi.

5.6.4.4. Apa yang terjadi pada saat *page-fault* ?

Page-fault menyebabkan urutan kejadian berikut :

- Ditangkap oleh Sistem Operasi.
- menyimpan *register user* dan proses.
- Tetapkan bahwa interupsi merupakan *page-fault*.
- Periksa bahwa referensi *page* adalah legal dan tentukan lokasi *page* pada disk.
- Kembangkan pembacaan disk ke *frame* kosong.

- Selama menunggu, alokasikan CPU ke pengguna lain dengan menggunakan penjadwalan CPU.
- Terjadi interupsi dari disk bahwa I/O selesai.
- Simpan register dan status proses untuk pengguna yang lain.
- Tentukan bahwa interupsi berasal dari disk.
- Betulkan *page table* dan tabel yang lain bahwa *page* telah berada di memory.
- Tunggu CPU untuk dialokasikan ke proses tersebut
- Kembalikan register user, status proses, *page table* , dan *resume* instruksi interupsi.

Pada berbagai kasus, ada tiga komponen yang kita hadapi pada saat melayani *page-fault* :

- Melayani interupsi *page-fault*
- Membaca *page*
- Mengulang kembali proses

5.6.4.5. Kinerja Demand Paging

Menggunakan Effective Access Time (EAT), dengan rumus :

$$EAT = (1-p) \times ma + p \times \text{waktu } page \text{ fault}$$

- p : kemungkinan terjadinya page fault ($0 < p < 1$)
- $p = 0$; tidak ada page-fault
- $p = 1$; semuanya mengalami page-fault
- ma : waktu pengaksesan memory (memory access time)

Untuk menghitung EAT, kita harus tahu berapa banyak waktu dalam pengerjaan *page-fault* .

Contoh penggunaan *Effective Access Time*

Diketahui : Waktu pengaksesan memory = $ma = 100$ nanodetik Waktu page fault = 20 milidetik

Maka, $EAT = (1-p) \times 100 + p (20 \text{ milidetik}) = 100 - 100p + 20.000.000p = 100 + 19.999.900p$ (milidetik)

Pada sistem *demand paging* , sebisa mungkin kita jaga agar tingkat *page-fault* -nya rendah. Karena bila *Effective Access Time* meningkat, maka proses akan berjalan lebih lambat.

5.6.4.6. 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:

- Bagaimana mengulang instruksi yang memiliki beberapa lokasi yang berbeda?

- Bagaimana pengalamatan dengan menggunakan *special-addressing mode* , termasuk *autoincrement* dan *autodecrement mode* ?
- Bagaimana jika instruksi yang dieksekusi panjang (contoh: *block move*)?

Masalah pertama dapat diatasi dengan dua cara yang berbeda.

- komputasi *microcode* dan berusaha untuk mengakses kedua ujung dari blok, agar tidak ada modifikasi *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 men- *set* bit FPD (*first phase done*) sehingga *restart instruction* tidak akan dimulai dari awal program, melainkan dari tempat program terakhir dieksekusi.

5.6.4.7. 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 "valid-invalid bit"
 - Valid ("1") artinya page sudah berada di memori
 - *invalid* ("0") artinya page masih berada di disk.
- Memori sekunder, digunakan untuk menyimpan proses yang belum berada di memori.

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

5.7. Aspek Demand Paging : Pembuatan Proses

Sistem *demand paging* dan memori virtual memberikan banyak keuntungan selama pembuatan proses berlangsung. Pada subbab ini, akan dibahas mengenai dua teknik yang disediakan oleh memori virtual untuk meningkatkan kinerja pembuatan dan pengaksesan suatu proses.

5.7.1. Copy-On-Write

Dengan memanggil sistem pemanggilan **fork()**, sistem operasi akan membuat proses anak sebagai duplikat dari proses induk. Sistem pemanggilan *fork()* bekerja dengan membuat salinan alamat proses induk untuk proses anak, lalu membuat duplikat *page* milik proses induk tersebut. Tapi, karena setelah pembuatan proses anak selesai, proses anak langsung memanggil sistem pemanggilan *exec()* yang menyalin alamat proses induk yang kemungkinan tidak dibutuhkan.

Oleh karena itu, lebih baik kita menggunakan teknik lain dalam pembuatan proses yang disebut sistem **copy-on-write**. Teknik ini bekerja dengan memperbolehkan proses anak untuk menginisialisasi penggunaan halaman yang sama secara bersamaan. *page* yang digunakan bersamaan itu, disebut dengan *copy-on-write page*, yang berarti jika salah satu dari proses anak atau proses induk melakukan penulisan pada *page* tersebut, maka akan dibuat juga sebuah salinan dari halaman itu.

Sebagai contoh, sebuah proses anak hendak memodifikasi sebuah *page* yang berisi sebagian dari stack. Sistem operasi akan mengenali hal ini sebagai *copy-on-write*, lalu akan membuat salinan dari *page* ini memetakannya ke alamat memori dari proses anak, sehingga proses anak akan memodifikasi *page* salinan tersebut, dan bukan *page* milik proses induk. Menggunakan teknik *copy-on-write* ini, *page* yang akan disalin adalah *page* yang dimodifikasi oleh proses anak atau proses induk. *Page-page* yang tidak dimodifikasi akan bisa dibagi untuk proses anak dan proses induk.

Saat suatu halaman akan diduplikat menggunakan teknik *copy-on-write*, digunakan tehnik **zero-fill-on-demand** untuk mengalokasikan *page* kosong sebagai tempat meletakkan hasil duplikat. *Page* kosong tersebut dialokasikan saat *stack* atau *heap* suatu proses akan diperbesar atau untuk mengatur halaman *copy-on-write*. *Zero-fill-on-demand page* akan dibuat kosong sebelum dialokasikan, yaitu dengan menghapus isi awal dari halaman. Karena itu, dengan *copy-on-write*, *page* yang sedang disalin akan disalin ke sebuah *zero-fill-on page*.

Teknik *copy-on-write* digunakan oleh beberapa sistem operasi seperti Windows 2000, Linux, dan Solaris2.

5.7.2. Memory-Mapped Files

Kita dapat menganggap berkas I/O sebagai memori akses *routine* pada teknik memori virtual. Cara ini disebut dengan *memory mapping* sebuah berkas yang mengizinkan sebuah bagian dari alamat virtual dihubungkan dengan sebuah berkas. Dengan teknik *memori mapping* sebuah blok *disk* dapat dipetakan ke ke sebuah halaman pada memori.

Proses membaca dan menulis sebuah berkas ditangani oleh akses memori *routine* agar memudahkan mengakses dan menggunakan sebuah berkas yaitu dengan mengizinkan manipulasi berkas melalui memori dibandingkan memanggil dengan sistem pemanggilan *read()* dan *write()*.

Beberapa sistem operasi menyediakan *memory mapping* hanya melalui sistem pemanggilan yang khusus dan menjaga semua berkas I/O yang lain dengan menggunakan sistem pemanggilan yang biasa.

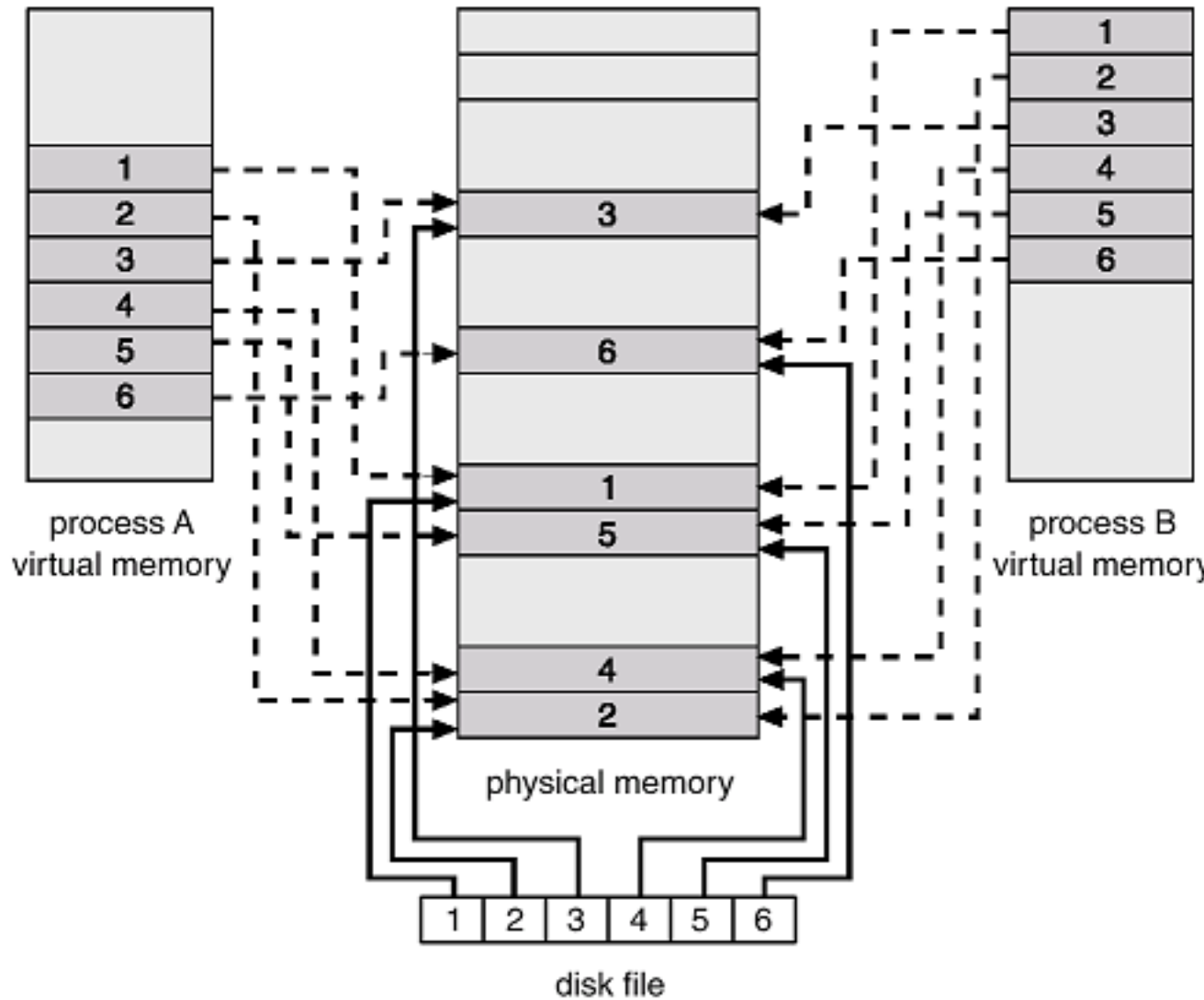
Proses yang banyak diperbolehkan untuk memetakan berkas yang sama ke dalam memori virtual dari masing-masing berkas, agar data dapat digunakan secara bersamaan. Memori virtual memetakan tiap proses ke dalam halaman yang sama pada memori virtual, yaitu halaman yang menyimpan salinan dari blok *disk*.

Dengan sistem *memory mapping* sistem pemanggilan dapat juga mendukung fungsi *copy-on-write*, mengizinkan proses untuk menggunakan sebuah berkas secara bersamaan pada keadaan *read only*, tapi

tetap memiliki salinan dari data yang diubah.

Berikut ini merupakan bagan dari proses *memory-mapped files*

Gambar 5-12. Bagan proses *memory-mapped files*

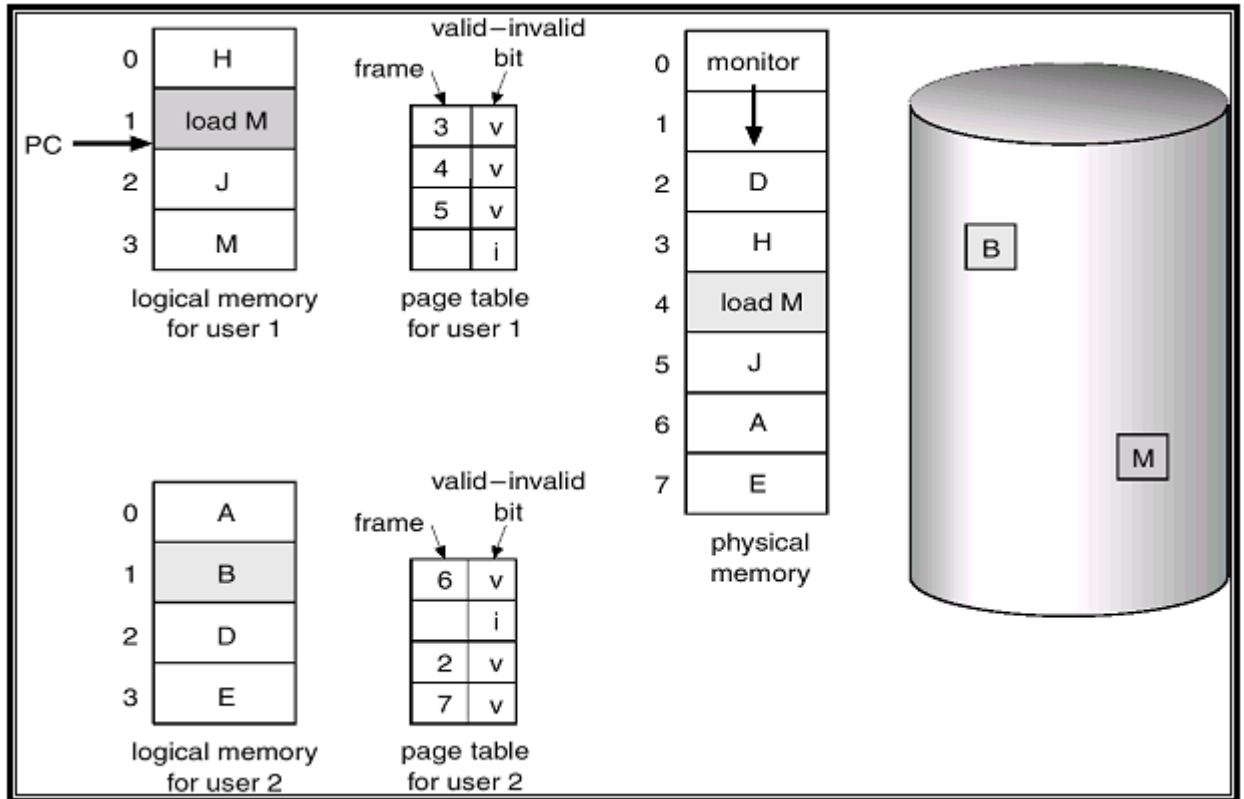


5.8. Konsep Dasar *Page Replacement*

Masalah page fault pasti akan dialami oleh setiap page minimal satu kali. Akan tetapi, sebenarnya sebuah proses yang memiliki N buah *page* hanya akan menggunakan $N/2$ diantaranya. Kemudian *demand*

paging akan menyimpan I/O yang dibutuhkan untuk mengisi N/2 page sisanya. Dengan demikian utilisasi CPU dan *throughput* dapat ditingkatkan.

Gambar 5-13. Kondisi yang memerlukan *Page Replacement*



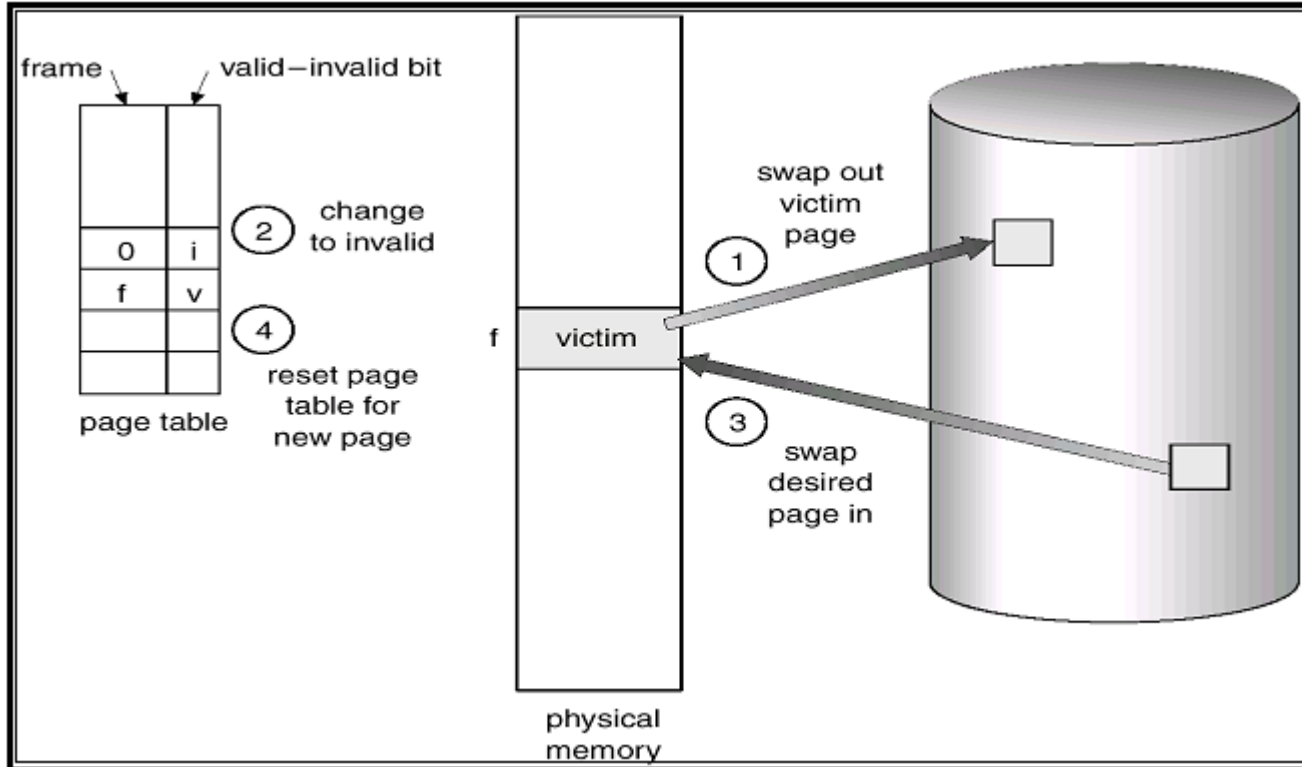
Upaya yang dilakukan oleh demand paging dalam mengatasi page fault didasari oleh *page replacement*. Sebuah konsep yang akan kita bahas lebih lanjut dalam subbab ini.

5.8.1. Konsep Dasar

Pendekatan dari *page replacement*: "Jika tidak ada frame yang kosong, cari frame yang tidak sedang digunakan, lalu kosongkan dengan memindahkan isinya ke dalam swap space dan ubah semua tabelnya sebagai indikasi bahwa page tersebut tidak akan lama berada di dalam memori."

Dalam *page replacement*, frame kosong seperti tersebut di atas, akan digunakan sebagai tempat penyimpanan dari page yang salah.

Gambar 5-14. Page Replacement



Rutinitas yang dilakukan dalam *page replacement* antara lain:

- Mencari lokasi dari page yang diinginkan pada disk.
- Mencari frame yang kosong. :
 - Jika ada, maka gunakan frame tersebut.
 - Jika tidak ada, maka tentukan frame yang tidak sedang dipakai, lalu kosongkan frame tersebut. Gunakan algoritma *page replacement* untuk menentukan frame yang akan dikosongkan.
 - Tulis page yang dipilih ke disk, ubah page-table dan frame-table.
- Membaca page yang diinginkan ke dalam frame kosong yang baru.
- Mengulangi user process dari awal.

Rutinitas di atas belum tentu berhasil. Jika kita tidak dapat menemukan frame yang kosong atau akan dikosongkan, maka sebagai jalan keluarnya kita dapat melakukan pentransferan dua page (satu masuk, satu keluar). Cara ini akan menambah waktu pelayanan page fault dan waktu akses efektif. Oleh karena itu, perlu digunakan bit tambahan untuk masing-masing page dan frame yang diasosiasikan dalam perangkat keras.

Sebagai dasar dari demand paging, *page replacement* merupakan "jembatan pemisah" antara logical memory dan physical memory. Mekanisme yang dimilikinya memungkinkan memori virtual berukuran sangat besar dapat disediakan untuk programmer dalam bentuk physical memory berukuran lebih kecil.

Dalam demand paging, jika kita memiliki banyak proses dalam memori, kita harus menentukan jumlah frame yang akan dialokasikan ke masing-masing proses. Ketika *page replacement* diperlukan, kita harus memilih frame yang akan dipindahkan(dikosongkan). Masalah ini dapat diselesaikan dengan menggunakan algoritma *page replacement*.

Ada beberapa macam algoritma *page replacement* yang dapat digunakan. Algoritma yang terbaik adalah yang memiliki tingkat page fault terendah. Selama jumlah frame meningkat, jumlah page fault akan menurun. Peningkatan jumlah frame dapat terjadi jika physical memori diperbesar.

Evaluasi algoritma *page replacement* dapat dilakukan dengan menjalankan string acuan di memori dan menghitung jumlah page fault yang terjadi. Sebagai contoh untuk memperoleh string acuan : --> jika suatu proses memiliki urutan alamat : 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105 ; per 100 bytes-nya dapat kita turunkan menjadi reference string : 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.

5.9. Algoritma *Page Replacement*

Page replacement diimplementasikan dalam algoritma *page replacement* yang bertujuan untuk menghasilkan *page-fault rate* terendah. Ada beberapa algoritma *page replacement* yang berbeda. Pemilihan algoritma yang kurang tepat dapat menyebabkan peningkatan *page-fault rate* sehingga proses berjalan lebih lambat.

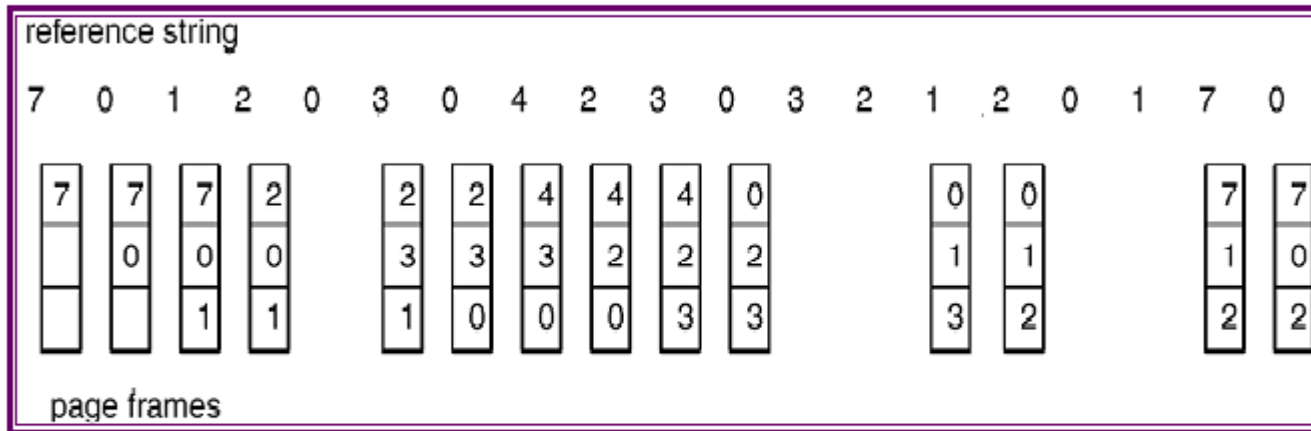
5.9.1. Algoritma FIFO (*First In First Out*)

Prinsip yang digunakan dalam algoritma FIFO yaitu *page* yang diganti adalah *page* yang paling lama berada di memori. Algoritma ini adalah algoritma *page replacement* yang paling mudah diimplementasikan.

Pengimplementasian algoritma FIFO dilakukan dengan menggunakan *queue* untuk menandakan *page* yang sedang berada di dalam memori. Setiap *page* baru yang diakses diletakkan di *tail* dari *queue*. Apabila *queue* telah penuh dan ada *page* yang baru diakses maka *page* yang berada di *head* dari *queue* akan diganti.

Kelemahan dari algoritma FIFO adalah kinerjanya yang tidak selalu baik. Hal ini disebabkan karena ada kemungkinan *page* yang baru saja keluar dari memori ternyata dibutuhkan kembali. Di samping itu dalam beberapa kasus, *page-fault rate* justru bertambah seiring dengan meningkatnya jumlah *frame*, yang dikenal dengan nama Anomali Belady.

Gambar 5-15. Contoh Algoritma FIFO



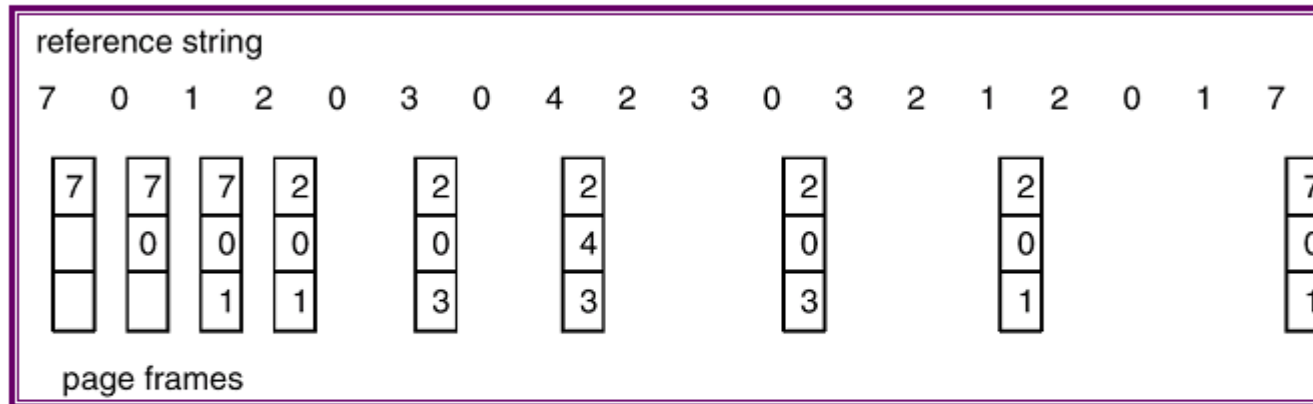
Algoritma FIFO

5.9.2. Algoritma Optimal

Algoritma optimal pada prinsipnya akan mengganti *page* yang tidak akan digunakan untuk jangka waktu yang paling lama. Kelebihannya antara lain dapat menghindari terjadinya anomali Belady dan juga memiliki *page-fault rate* yang terendah diantara algoritma-algoritma *page replacement* yang lain.

Algoritma ini cukup sulit untuk diimplementasikan karena harus dapat mengetahui *page-page* yang akan muncul.

Gambar 5-16. Contoh Algoritma Optimal



Algoritma Optimal

5.9.3. Algoritma LRU (*Least Recently Used*)

Algoritma LRU akan mengganti *page* yang telah tidak digunakan dalam jangka waktu terlama. Kelebihan dari algoritma LRU yaitu seperti halnya pada algoritma optimal, tidak akan mengalami anomali Belady.

Pengimplementasiannya dapat dilakukan dengan dua cara, yaitu dengan menggunakan:

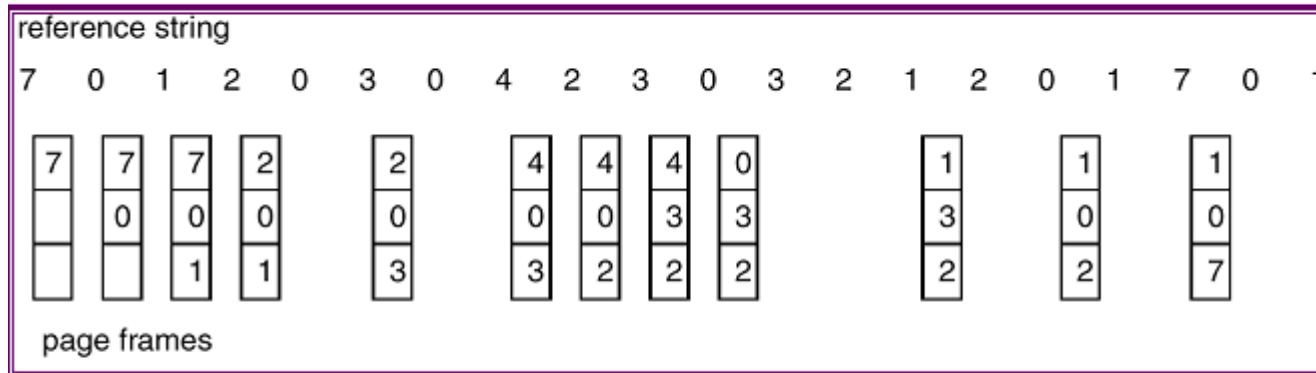
- **Counter**

Cara ini dilakukan dengan menggunakan *clock*. Setiap *page* memiliki nilai. Ketika mengakses ke suatu *page* baru, nilai pada *clock* akan ditambah 1. *Page* yang diganti adalah *page* yang memiliki nilai terkecil.

- **Stack**

Penggunaan implementasi ini dilakukan dengan menggunakan stack yang menandakan *page-page* yang berada di memori. Setiap kali suatu *page* diakses, akan diletakkan di bagian paling atas stack. Apabila ada *page* yang perlu diganti, maka *page* yang berada di bagian paling bawah stack akan diganti, sehingga setiap kali *page* baru diakses tidak perlu mencari *page* yang akan diganti. Akan tetapi cara ini lebih mahal implementasinya dibandingkan dengan menggunakan counter.

Gambar 5-17. Contoh Algoritma LRU



Algoritma LRU

5.9.4. Algoritma Perkiraan LRU

Pada dasarnya algoritma perkiraan LRU memiliki prinsip yang sama dengan algoritma LRU, yaitu *page* yang diganti adalah *page* yang tidak digunakan dalam jangka waktu terlalu lama, hanya saja dilakukan modifikasi pada algoritma ini untuk mendapatkan hasil yang lebih baik. Perbedaannya dengan algoritma LRU terletak pada penggunaan bit reference. Setiap *page* yang berbeda memiliki bit reference. Pada awalnya bit reference diinisialisasikan oleh perangkat keras dengan nilai 0. Nilainya akan berubah menjadi 1 bila dilakukan akses ke *page* tersebut.

Ada beberapa cara untuk mengimplementasikan algoritma ini, yaitu:

- **Algoritma *Additional-Reference-Bits***

Setiap *page* akan memiliki bit reference yang terdiri dari 8 bit byte sebagai penanda. Pada awalnya semua bit nilainya 0, contohnya : 00000000. Setiap selang beberapa waktu, *timer* melakukan interupsi kepada sistem operasi, kemudian sistem operasi menggeser 1 bit ke kanan. Jadi *page* yang selalu digunakan pada setiap periode akan memiliki nilai 11111111. *Page* yang diganti adalah *page* yang memiliki nilai terkecil.

- **Algoritma *Second-Chance***

Algoritma ini menggunakan *circular queue* yang berarti menggunakan juga prinsip algoritma FIFO disamping menggunakan algoritma LRU. Apabila nilai bit reference-nya 0, *page* dapat diganti. Dan apabila nilai bit reference-nya 1, *page* tidak diganti tetapi bit reference diubah menjadi 0 dan dilakukan pencarian kembali.

- **Algoritma *Enhanced Second-Chance***

Algoritma ini mempertimbangkan 2 hal sekaligus, yaitu bit reference dan bit modifikasi. Ada 4 kemungkinan yang dapat terjadi :

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

5.9.5. Algoritma *Counting*

Dilakukan dengan menyimpan counter dari nomor *reference* yang sudah dibuat untuk masing-masing *page* . Penggunaan algoritma ini memiliki kekurangan yaitu lebih mahal. Algoritma *Counting* dapat dikembangkan menjadi 2 skema dibawah ini:

- **Algoritma LFU (*Least Frequently Used*)**

Page yang diganti adalah *page* yang paling sedikit dipakai (nilai counter terkecil) dengan alasan *page* yang digunakan secara aktif akan memiliki nilai *reference* yang besar.

- **Algoritma MFU (*Most Frequently Used*)**

Page yang diganti adalah *page* yang paling sering dipakai (nilai counter terbesar) dengan alasan bahwa *page* dengan nilai terkecil mungkin baru saja dimasukkan dan baru digunakan

5.9.6. Algoritma *Page Buffering*

Sistem menyimpan *pool* dari *frame* yang kosong. Prosedur ini memungkinkan suatu proses mengulang dari awal secepat mungkin, tanpa perlu menunggu *page* yang akan dipindahkan untuk ditulis ke *disk* karena *frame* -nya telah ditambahkan ke dalam *pool frame* kosong. Teknik seperti ini digunakan dalam sistem VAX/ VMS.

5.10. Strategi Alokasi *Frame*

5.10.1. Alokasi *Frame*

Masalah yang penting dalam alokasi *frame* dengan penggunaan memori virtual adalah bagaimana membagi memori yang bebas untuk beberapa proses yang sedang dikerjakan.

Dapat kita ambil 1 contoh yang mudah pada sistem satu pemakai. Misalnya sebuah sistem mempunyai memori 128K dengan ukuran page 1K, sehingga ada 128 *frame*. Sistem operasi menggunakan 35K sehingga ada 93 *frame* yang tersisa ada 93 *frame* yang tersisa untuk proses tiap user. Untuk *pure demand paging*, ke-93 *frame* tersebut akan ditaruh pada *frame* bebas. Ketika sebuah *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 *page* 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.

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

Pada dasarnya yaitu proses pengguna diberikan *frame* bebas yang mana saja. Masalah yang muncul ketika *demand paging* dikombinasikan dengan *multiprogramming*. Hal ini terjadi karena *multiprogramming* menaruh dua (atau lebih) proses di memori pada waktu yang bersamaan.

5.10.1.1. Jumlah *Frame* Minimum

Ada berbagai batasan dalam mengalokasikan *frame*. Kita tidak dapat mengalokasikan *frame* lebih dari jumlah *frame* yang ada. Intinya adalah berapa minimum *frame* yang harus dialokasikan agar jika sebuah instruksi dijalankan, semua informasinya ada dalam memori. Jika terjadi *page fault* sebelum eksekusi selesai, instruksi tersebut harus diulang. Sehingga kita harus mempunyai jumlah *frame* yang cukup untuk menampung semua *page* yang dibutuhkan oleh sebuah instruksi.

Jumlah *frame* minimum yang bisa dialokasikan ditentukan oleh arsitektur komputer. Sebagai contoh, instruksi *move* pada PDP-11 adalah lebih dari satu kata untuk beberapa modus pengalamatan, sehingga instruksi tersebut bisa membutuhkan dua halaman. Sebagai tambahan, tiap operannya mungkin merujuk tidak langsung, sehingga total ada enam *frame*. Kasus terburuk untuk IBM 370 adalah instruksi MVC. Karena instruksi tersebut adalah instruksi perpindahan dari penyimpanan ke penyimpanan, instruksi ini butuh 6 bit dan dapat memakai dua halaman. Satu blok karakter yang akan dipindahkan dan daerah tujuan perpindahan juga dapat memakai dua halaman, sehingga situasi ini membutuhkan enam *frame*.

5.10.1.2. Algoritma Alokasi

Algoritma pertama yaitu *equal allocation*. Algoritma ini memberikan bagian yang sama, sebanyak m/n *frame* untuk tiap proses. Sebagai contoh ada 93 *frame* tersisa dan 5 proses, maka tiap proses akan

mendapatkan 18 *frame*. *Frame* yang tersisa, sebanyak 3 buah dapat digunakan sebagai *frame* bebas cadangan.

Sebuah alternatif yaitu pengertian bahwa berbagai proses akan membutuhkan jumlah memori yang berbeda. Jika ada sebuah proses sebesar 10K dan sebuah proses basis data 127K dan hanya kedua proses ini yang berjalan pada sistem, maka ketika ada 62 *frame* bebas, tidak masuk akal jika kita memberikan masing-masing proses 31 *frame*. Proses pertama hanya butuh 10 *frame*, 21 *frame* lain akan terbuang percuma. Untuk menyelesaikan masalah ini, kita menggunakan algoritma kedua yaitu *proportional allocation*. Kita mengalokasikan memori yang tersedia kepada setiap proses tergantung pada ukurannya.

Ukuran memori virtual untuk proses $p_i = s_i$, dan $S = \sum s_i$.

Lalu, jika jumlah total dari *frame* yang tersedia adalah m , kita mengalokasikan proses a_i ke proses p_i , dimana a_i mendekati :

$$a_i = s_i / S \times m$$

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

Mengingat hal itu, dengan *equal allocation* ataupun *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. Algoritma ini dinamakan alokasi prioritas.

5.10.1.3. Alokasi Global lawan Lokal

Hal penting lainnya dalam pengalokasian *frame* adalah pergantian *page*. Proses-proses bersaing mendapatkan *frame*, maka dari itu kita dapat mengklasifikasikan algoritma pergantian halaman kedalam dua kategori; Pergantian Global dan Pergantian Lokal. Pergantian global memperbolehkan sebuah proses mencari *frame* pengganti dari semua *frame* yang ada, walaupun *frame* tersebut sedang dialokasikan untuk proses yang lain. Hal ini memang efisien. tetapi ada kemungkinan proses lain tidak mendapatkan *frame*. Pergantian lokal memberi aturan bahwa setiap proses hanya boleh memilih *frame* pengganti dari *frame-frame* yang memang dialokasikan untuk proses itu sendiri.

Sebagai contoh, misalkan ada sebuah skema alokasi yang memperbolehkan proses berprioritas tinggi untuk mencari *frame* pengganti dari proses yang berprioritas rendah. Proses berprioritas tinggi ini dapat mencari *frame* pengganti dari *frame-frame* yang telah dialokasikan untuknya atau dari *frame-frame* yang dialokasikan untuk proses berprioritas lebih rendah.

Dalam pergantian lokal, jumlah *frame* yang teralokasi tidak berubah. Dengan Pergantian 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 pergantian).

Masalah pada algoritma Penggantian Global adalah bahwa sebuah proses tidak bisa mengontrol *page-fault* -nya sendiri. *page-page* 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). Dalam Penggantian Lokal, *page-page* 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.

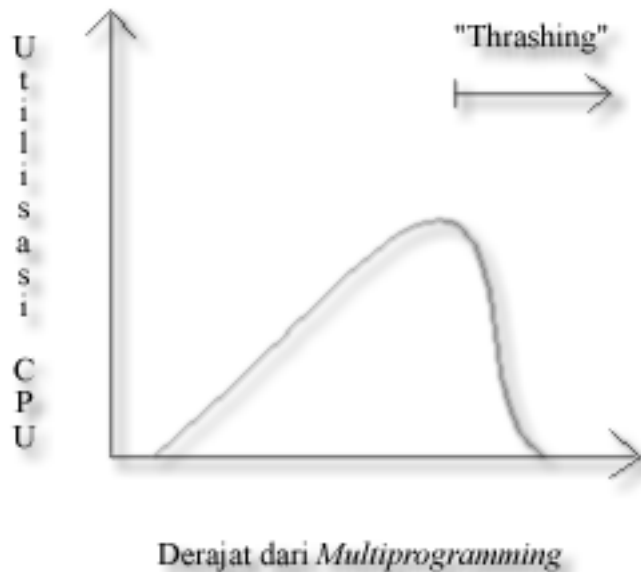
5.10.2. Thrashing

Aktifitas yang tinggi dari *paging* disebut *Thrashing* . Aktifitas *paging* yang tinggi ini maksudnya adalah sistem sibuk melakukan *swap-in* dan *swap-out* dikarenakan banyak *page-fault* yang terjadi. Jika suatu proses tidak memiliki *frame* yang cukup, walaupun bisa dikurangi sehingga alokasi *frame* nya minimum, tetap ada *page* dalam jumlah besar yang memiliki kondisi aktif menggunakannya. Maka hal ini mengakibatkan *page-fault* . Pada kasus ini, kita harus mengganti bebetapa halaman menjadi halaman yang dibutuhkan walaupun halaman yang diganti pada waktu dekat akan dibutuhkan lagi. Hal ini mengakibatkan *page-fault* yang terus-menerus

5.10.2.1. Penyebab Thrashing

Utilitas dari CPU dapat menyebabkan *Thrashing* . Jika Utilitas CPU rendah, maka sistem akan menambah derajat dari *multiprogramming* yang berarti menambah jumlah proses yang sedang berjalan. Dengan makin bertambahnya jumlah proses, maka utilitas dari CPU akan berkurang drastis, karena aktifitas *paging* yang tinggi dan menyebabkan *thrashing* .

Untuk meningkatkan utilitas dari CPU, kita menurunkan derajat dari *multiprogramming*

Gambar 5-18. Derajat *Multiprogramming*

5.10.2.2. Membatasi Efek *Thrashing*

Efek dari *thrashing* dapat dibatasi dengan algoritma pergantian lokal atau prioritas. Dengan pergantian lokal, jika satu proses mulai *thrashing*, proses tersebut dapat mencuri *frame* dari proses yang lain dan menyebabkan proses itu tidak langsung *thrashing*. Jika proses mulai *thrashing*, proses itu akan berada pada antrian untuk melakukan *paging* yang mana hal ini memakan banyak waktu. Rata-rata waktu layanan untuk *page-fault* 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*.

Selama satu proses di eksekusi, model lokalitas berpindah dari satu lokalitas satu ke lokalitas lainnya. Lokalitas adalah kumpulan *page* yang aktif digunakan bersama. Suatu program pada umumnya dibuat pada beberapa lokalitas sehingga ada kemungkinan terjadi *overlap*. *Thrashing* dapat muncul bila ukuran lokalitas lebih besar dari ukuran memori total.

5.10.2.3. Model Working Set

Strategi *Working set* dimulai dengan melihat berapa banyak *frame* yang sesungguhnya digunakan oleh suatu proses. *working set model* merupakan model lokalitas dari suatu eksekusi proses. Model ini menggunakan parameter Δ (delta) untuk mendefinisikan *working set window*. Untuk menentukan halaman yang dituju, yang paling sering muncul. Kumpulan dari *page* dengan Δ *page* yang dituju yang paling sering muncul disebut *working set*. *Working set* adalah pendekatan dari program lokalitas.

Contoh :

Keakuratan *working set* tergantung pada pemilihan Δ :

1. Jika Δ terlalu kecil, tidak akan dapat mewakili keseluruhan dari lokalitas.
2. Jika Δ terlalu besar, akan menyebabkan *overlap* beberapa lokalitas.
3. Jika Δ tidak terbatas, *working set* adalah kumpulan *page* sepanjang eksekusi program.

Jika kita menghitung ukuran dari *Working Set*, *WWSi*, untuk setiap proses pada sistem, kita hitung dengan $D = WWSi$, 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 pemblokiran dari proses-proses yang sedang berjalan.

Strategi *Working Set* menangani *thrashing* dengan tetap mempertahankan derajat dari *multiprogramming* setinggi mungkin.

Contoh : $\Delta = 1000$ reference, *Timer interrupt* setiap 5000 reference.

Ketika kita mendapat *interrupt*, kita kopi dan hapus nilai *reference bit* dari setiap *page*. Jika kesalahan *page* muncul, kita dapat menentukan *current reference bit* dan 2 pada bit memori untuk memutuskan apakah *page* 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*.

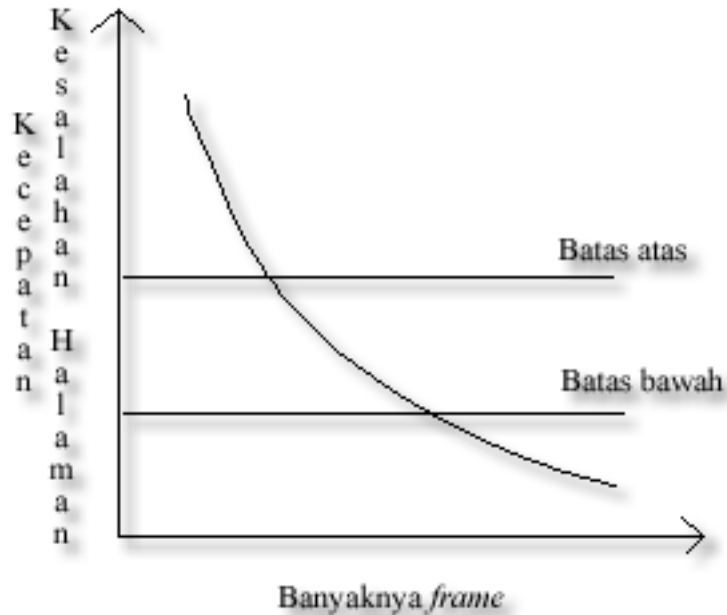
5.10.2.4. Frekuensi Page-Fault

Working-set dapat berguna untuk *prepaging*, tetapi kurang dapat mengontrol *thrashing*. Strategi menggunakan frekuensi *page-fault* mengambil pendekatan yang lebih langsung.

Thrashing memiliki kecepatan *page-fault* 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 *page-fault* seperti terlihat pada gambar berikut ini.

Gambar 5-19. Kecepatan *page-fault*



Jika kecepatan *page-fault* yang sesungguhnya melampaui batas atas, kita mengalokasikan *frame* lain ke proses tersebut, sedangkan jika kecepatan *page-fault* di bawah batas bawah, kita pindahkan *frame* dari proses tersebut. Maka kita dapat secara langsung mengukur dan mengontrol kecepatan *page-fault* untuk mencegah *thrashing* .

5.11. Pertimbangan Lain

Pemilihan algoritma penggantian dan aturan alokasi adalah keputusan- keputusan utama yang kita buat untuk sistem *paging* . Selain itu, masih banyak pertimbangan lain.

5.11.1. *Prepaging*

Sebuah ciri dari sistem *demand-paging* murni adalah banyaknya *page fault* yang terjadi saat proses

dimulai. Situasi ini merupakan hasil dari percobaan untuk mendapatkan tempat pada awalnya. Situasi yang sama mungkin muncul di lain waktu. Misalnya saat proses *swapped-out* dimulai kembali, seluruh *page* ada di *disk* dan setiap *page* harus dibawa ke dalam *memory* yang akan mengakibatkan banyaknya *page fault*. *Prepaging* mencoba untuk mencegah *paging* awal tingkat tinggi ini. Strateginya adalah untuk membawa seluruh *page* yang akan dibutuhkan pada satu waktu ke *memory*

Sebagai contoh, pada sistem yang menggunakan model *working-set*, untuk setiap proses terdapat daftar dari semua *page* yang ada pada *working set*-nya. Jika kita harus menunda sebuah proses (karena menunggu *I/O* atau kekurangan *frame* bebas), daftar *working set* untuk proses tersebut disimpan. Saat proses itu akan dilanjutkan kembali (permintaan *I/O* telah terpenuhi atau *frame* bebas yang cukup), secara otomatis seluruh *working set*-nya akan dibawa ke dalam *memory* sebelum memulai kembali proses tersebut.

Prepaging dapat berguna pada beberapa kasus. Yang harus dipertimbangkan adalah apakah biaya menggunakan *prepaging* lebih sedikit dari biaya menangani *pagefault* yang terjadi bila tidak memakai *prepaging*. Jika biaya *prepaging* lebih sedikit (karena hampir seluruh *page* yang di *prepage* digunakan) maka *prepaging* akan berguna. Sebaliknya, jika biaya *prepaging* lebih besar (karena hanya sedikit *page* dari yang di-*prepage* yang digunakan) maka *prepaging* akan merugikan.

5.11.2. Ukuran page

Para perancang sistem operasi untuk mesin yang sudah ada jarang memiliki pilihan terhadap ukuran *page*. Akan tetapi, saat merancang sebuah mesin baru, harus dipertimbangkan berapa ukuran *page* yang terbaik. Pada dasarnya tidak ada ukuran *page* yang paling baik, karena banyaknya faktor-faktor yang mempengaruhinya.

Salah satu faktor adalah ukuran *page table*. Untuk sebuah *virtual memory* dengan ukuran 4 megabytes (2^{22}), akan ada 4.096 *page* berukuran 1.024 bytes, tapi hanya 512 *page* jika ukuran *page* 8.192 bytes. Setiap proses yang aktif harus memiliki salinan dari *page table*-nya, jadi lebih masuk akal jika dipilih ukuran *page* yang besar.

Di sisi lain, pemanfaatan *memory* lebih baik dengan *page* yang lebih kecil. Jika sebuah proses dialokasikan di *memory*, mengambil semua *page* yang dibutuhkannya, mungkin proses tersebut tidak akan berakhir pada batas dari *page* terakhir. Jadi, ada bagian dari *page* terakhir yang tidak digunakan walaupun telah dialokasikan. Asumsikan rata-rata setengah dari *page* terakhir tidak digunakan, maka untuk *page* dengan ukuran 256 bytes hanya akan ada 128 bytes yang terbuang, bandingkan dengan *page* berukuran 8192 bytes, akan ada 4096 bytes yang terbuang. Untuk meminimalkan pemborosan ini, kita membutuhkan ukuran *page* yang kecil.

Masalah lain adalah waktu yang dibutuhkan untuk membaca atau menulis *page*. Waktu *I/O* terdiri dari waktu *seek*, *latency* dan transfer. Waktu transfer sebanding dengan jumlah yang dipindahkan (yaitu, ukuran *page*). Sedangkan waktu *seek* dan *latency* biasanya jauh lebih besar dari waktu transfer. Untuk laju pemindahan 2 MB/s, hanya dihabiskan 0.25 milidetik untuk memindahkan 512 bytes. Waktu *latency* mungkin sekitar 8 milidetik dan waktu *seek* 20 milidetik. Total waktu *I/O* 28.25 milidetik. Waktu transfer sebenarnya tidak sampai 1%. Sebagai perbandingan, untuk mentransfer 1024 bytes, dengan ukuran *page* 1024 bytes akan dihabiskan waktu 28.5 milidetik (waktu transfer 5 milidetik). Namun dengan *page* berukuran 512 bytes akan terjadi 2 kali transfer 512 bytes dengan masing-masing transfer menghabiskan waktu 28.25 milidetik sehingga total waktu yang dibutuhkan 56.5 milidetik. Kesimpulannya, untuk meminimalisasi waktu *I/O* dibutuhkan ukuran *page* yang lebih besar.

Masalah terakhir yang akan dibahas disini adalah mengenai *page fault* . Misalkan ukuran *page* adalah 1 *byte* . Sebuah proses sebesar 100 KB, dimana hanya setengahnya yang menggunakan *memory* , akan menghasilkan *page fault* sebanyak $51200 \text{ page} < \text{faults}$. Sedangkan bila ukuran *page* sebesar 200 KB maka hanya akan terjadi 1 *page fault* . Jadi untuk mengurangi jumlah *page fault* dibutuhkan ukuran $> \text{page}$ yang besar.

Masih ada faktor lain yang harus dipertimbangkan (misalnya hubungan antara ukuran *page* dengan ukuran sektor pada *paging device*). Tidak ada jawaban yang pasti berapa ukuran *page* yang paling baik. Sebagai acuan, pada 1990, ukuran *page* yang paling banyak dipakai adalah 4096 *bytes* . Sedangkan sistem modern saat ini menggunakan ukuran *page* yang jauh lebih besar dari itu.

5.11.3. TLBreach

Hit ratio dari *TLB* adalah persentasi alamat virtual yang diselesaikan dalam *TLB* daripada di *page table* . *Hit ratio* sendiri berhubungan dengan jumlah masukan dalam *TLB* dan cara untuk meningkatkan *hit ratio* adalah dengan menambah jumlah masukan dari *TLB* . Tetapi ini tidaklah murah karena *memory* yang dipakai untuk membuat *TLB* mahal dan haus akan tenaga.

Ada suatu ukuran lain yang mirip dengan *hit ratio* yaitu *TLBreach* . *TLB reach* adalah jumlah *memory* yang dapat diakses dari *TLB* , jumlah tersebut merupakan perkalian dari jumlah masukan dengan ukuran *page* . Idealnya, *working set* dari sebuah proses disimpan dalam *TLB* . Jika tidak, maka proses akan menghabiskan waktu yang cukup banyak mengatasi referensi *memory* di dalam *page table* daripada di *TLB* . Jika jumlah masukan dari *TLB* dilipatgandakan, maka *TLBreach* juga akan bertambah menjadi dua kali lipat. Tetapi untuk beberapa aplikasi hal ini masih belum cukup untuk menyimpan *working set* .

Cara lain untuk meningkatkan *TLBreach* adalah dengan menambah ukuran *page* . Bila ukuran *page* dijadikan empat kali lipat dari ukuran awalnya (misalnya dari 32 KB menjadi 128 KB), maka *TLB reach* juga akan menjadi empat kali lipatnya. Namun ini akan meningkatkan fragmentasi untuk aplikasi-aplikasi yang tidak membutuhkan ukuran *page* sebesar itu. Sebagai alternative, OS dapat menyediakan ukuran *page* yang bervariasi. Sebagai contoh, UltraSparc II menyediakan *page* berukuran 8 KB, 64 KB, 512 KB, dan 4 MB. Sedangkan Solaris 2 hanya menggunakan *page* ukuran 8 KB dan 4 MB.

5.11.4. page table yang Dibalik

Kegunaan dari *page table* yang dibalik adalah untuk mengurangi jumlah *memory* fisik yang dibutuhkan untuk melacak penerjemahan alamat virtual-to- physical. Metode penghematan ini dilakukan dengan membuat tabel yang memiliki hanya satu masukan tiap *page memory* fisik, terdaftar oleh pasangan (proses-id, nomor- *page*).

Karena menyimpan informasi tentang halaman *memory* virtual yang mana yang disimpan di setiap *frame* fisik, *page table* yang dibalik mengurangi jumlah fisik *memory* yang dibutuhkan untuk menyimpan informasi ini. Bagaimana pun, *page table* yang dibalik tidak lagi mengandung informasi yang lengkap tentang ruang alamat *logic* dari sebuah proses, dan informasi itu dibutuhkan jika *page* yang direferensikan tidak sedang berada di *memory* . *Demand paging* membutuhkan informasi ini untuk memproses *page faults* . Agar informasi ini tersedia, sebuah *page table* eksternal (satu tiap proses) harus tetap disimpan. Setiap tabel tampak seperti *page table* per proses tradisional, mengandung informasi dimana setiap halaman virtual berada.

Tetapi, apakah *page table* eksternal menegaskan kegunaan *page table* yang dibalik? Karena tabel-tabel ini direferensikan hanya saat *page fault* terjadi, mereka tidak perlu tersedia secara cepat. Namun, mereka dimasukkan atau dikeluarkan dari *memory* sesuai kebutuhan. Sayangnya, sekarang *page fault* mungkin terjadi pada manager *virtual memory*, menyebabkan *page fault* lain karena pada saat mem-*page in page table* eksternal, ia harus mencari virtual *page* pada *backing store*. Kasus spesial ini membutuhkan penanganan di *kernel* dan *delay* pada proses *page-lookup*.

5.11.5. Struktur Program

Pemilihan struktur data dan struktur pemrograman secara cermat dapat meningkatkan *locality* dan karenanya menurunkan tingkat *page fault* dan jumlah *page* di *working set*. Sebuah *stack* memiliki *locality* yang baik, karena akses selalu dari atas. Sebuah *hash table*, di sisi lain, didesain untuk menyebar referensi-referensi, menghasilkan *locality* yang buruk. Tentunya, referensi akan *locality* hanyalah satu ukuran dari efisiensi penggunaan struktur data. Faktor-faktor lain yang berbobot berat termasuk kecepatan pencarian, jumlah total dari referensi dan jumlah total dari *page* yang disentuh.

5.11.6. I/O Interlock

Saat *demand paging* digunakan, kita terkadang harus mengizinkan beberapa *page* untuk dikunci di *memory*. Salah satu situasi muncul saat I/O dilakukan ke atau dari user (*virtual*) *memory*. I/O sering diimplementasikan oleh prosesor I/O yang terpisah. Sebagai contoh, sebuah pengendali pita magnetik pada umumnya diberikan jumlah *bytes* yang akan dipindahkan dan alamat *memory* untuk *buffer*. Saat pemindahan selesai, CPU diinterupsi.

Harus diperhatikan agar 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 menimbulkan *page fault*, dan, menggunakan algoritma penggantian global, salah satu dari mereka menggantikan halaman yang mengandung *memory buffer* untuk proses yang menunggu tadi. *page-page* untuk proses tersebut 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 *page* berbeda milik proses lain.

Ada 2 solusi untuk masalah ini. Salah satunya adalah jangan pernah meng-*execute I/O* kepada user *memory*. Sedangkan solusi lainnya adalah dengan mengizinkan *page* untuk dikunci dalam *memory*.

5.11.7. Pemrosesan Real Time

Diskusi-diskusi di bab ini telah dikonsentrasikan dalam menyediakan penggunaan yang terbaik secara menyeluruh dari sistem komputer dengan meningkatkan penggunaan *memory*. Dengan menggunakan *memory* 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 mendapatkan *page fault* tambahan selama eksekusi.

Pertimbangkan sebuah proses atau *thread* waktu-nyata. Proses tersebut berharap untuk memperoleh kendali CPU, dan untuk menjalankan penyelesaian dengan *delay* yang minimum. *Virtual Memory* adalah kebalikan dari *real-time computing*, sebab dapat menyebabkan *delay* jangka panjang yang tidak

diharapkan pada eksekusi sebuah proses saat *page* dibawa ke *memory*. Untuk itulah, sistem-sistem *real time* hampir tidak memiliki *virtual memory*.

5.11.8. Contoh pada Sistem Operasi

Pada bagian ini akan dibahas bagaimana Windows NT, Solaris 2, dan Linux mengimplementasi *virtual memory*.

5.11.9. Windows NT

Windows NT mengimplementasikan *virtual memory* dengan menggunakan *demand paging* melalui *clustering*. *Clustering* menanganani *page fault* dengan menambahkan tidak hanya *page* yang terkena *fault*, tetapi juga *page-page* yang berada disekitarnya. Saat proses pertama dibuat, dia diberikan *working set* minimum yaitu jumlah minimum *page* yang dijamin akan dimiliki oleh proses tersebut dalam *memory*. Jika *memory* yang cukup tersedia, proses dapat diberikan *page* sampai sebanyak *working set* maximum. Manager *virtual memory* akan menyimpan daftar dari *frame page* yang bebas. Terdapat juga sebuah nilai batasan yang diasosiasikan dengan daftar ini untuk mengindikasikan apakah *memory* 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 aturan *page replacement local*.

Saat jumlah *memory* bebas jatuh di bawah nilai batasan, manager *virtual memory* 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 *virtual memory* akan mengurangi jumlah *page*-nya sampai *working-set* minimum. Jika *memory* bebas sudah tersedia, proses yang bekerja pada *working set* minimum dapat mendapatkan *page* tambahan.

5.11.10. 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 *memory* bebas. Terhadap daftar ini ada dua parameter yg disimpan yaitu *minfree* dan *lotsfree*, yaitu batasan minimum dan maksimum dari *memory* bebas yang tersedia. Empat kali dalam tiap detiknya, *kernel* memeriksa jumlah *memory* 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 *memory* dan mengeset bit referensi menjadi 0. Saat berikutnya, *clock* kedua akan memeriksa bit referensi *page* dalam *memory*, dan mengembalikan bit yang masih di set ke 0 ke daftar *memory* bebas. Hal ini dilakukan sampai jumlah *memory* bebas melampaui parameter *lotsfree*. Lebih lanjut, proses ini dinamis, dapat mengatur kecepatan jika *memory* terlalu sedikit. Jika proses ini tidak bisa membebaskan *memory*, maka *kernel* memulai pergantian proses untuk membebaskan *page* yang dialokasikan ke proses-proses tersebut.

5.11.11. 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 *memory* 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 enam *page* dan tidur untuk beberapa saat sebelum berjalan lagi. Saat dia berjalan, akan memeriksa *mem_map*, daftar dari semua *page* yang terdapat di *memory*. Setiap *page* mempunyai *byte* umur yang diinisialisasikan ke tiga. 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* sistem berkas, dan jika semua cara sudah gagal, maka dia akan menghentikan sebuah proses. Alokasi *memory* 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 *memory* 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.

Bab 6. Sistem Berkas

Bab 7. I/O

Bab 8. Studi Kasus: GNU/Linux

Lampiran A. *GNU Free Documentation License*

Version 1.1, March 2000

Copyright © 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

A.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download

anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

- K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

A.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME.

Permission is granted to copy, distribute and/ or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.