B a b 4

Memori dan Virtual Memori

Tujuan Pelajaran

Setelah mempelajari bab ini, Anda diharapkan :

- Memahami konsep dasar memori dipandang dari fisik dan logik
- Memahami pengalamatan di memori
- Memahami pertukaran data di memori
- Memahami konsep pemberian halaman pada memori
- Memahami konsep segmentasi
- Memahami konsep dasar virtual memori
- Memahami bagaimana demmand paging terjadi
- Memahami algoritma-algoritma pemindahan halaman
- Memahami pengalokasian frame
- Memahami penyebab thrasing

4.1. Latar Belakang

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

4.1.1. Pengikatan Alamat

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

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

- waktu *compile*: jika diketahui pada waktu compile, dimana proses ditempatkan di memori. Untuk kemudian kode absolutnya dapat di buat. Jika keumdian amat awalnya berubah, maka harus di compile ulang.

- waktu penempatan: Jika tidak diketahui dimana poses ditempatkan di memori, maka kompilator harus mmbuagt kode yang dapat dialokasikan. Dalam kasus pengikatan akan ditunda sampai waktu penempatan. Jika alamat awalnya berubah, kita hanya perlu menempatkan ulang kode, untuk menyesuaikan dengan perubahan.

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

4.1.2. Ruang Alamat Fisik dan Logik

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

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

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

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

4.1.3. Penempatan Dinamis

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

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

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

4.1.4. Perhubungan Dinamis dan Berbagi *Library*

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

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

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

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

4.1.5. Lapisan Atas

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

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

Untuk menempatkan semuanya sekaligus, kita akan 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 lapisan atas. Lapisan atas A untuk pass1, tabel simbol dan rutin, lapisan atas 2 untuk simbol tabel, rutin, dan pass2.

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

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

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

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

4.2. Penukaran (*Swap*)

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

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

Normalnya, sebuah proses yang ditukar keluar, akan dimasukkan kembali ke tempat memori yang sama dengan yang digunakan sebelumnya. Batasan ini dibuat oleh method pengikat alamat. Jika pengikatan dilakukan saat assemble atau load time, maka proses tidak bisa dipindahkan ke lokasi yang berbeda. Jika menggunakan pengikatan waktu eksekusi, maka akan mungkin menukar proses kedalam tempat memori yang berbeda. Karena alamat fisik dihitung selama proses eksekusi.

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

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

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

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

Ada beberapa keterbatasan *swapping*. Jika kita ingin menukar sebuah proses kita harus yakin bahwa proses sepenuhnya diam. Konsentrasi lebih jauh, jika ada penundaan I/O. Sebuah proses mungkin menunggu I/O, ketika kita ingin menukar proses itu untuk

mengosongkan memori. Jika I/O secara asinkronus, mengakses memori dari I/O buffer, maka proses tidak bisa ditukar. Misalkan I/O operation berada di antrian, karena *device* sedang sibuk. Maka bila kita menukar keluar proses P1 dan memasukkan P2, mungkin saja operasi I/O akan berusaha masuk ke memori yang sekarang milik P2.

Dua solusi utama masalah ini adalah

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

Secara umum, ruang pertukaran dialokasikan sebagai potongan disk, terpisah dari sistem berkas, sehingga bisa digunakan secepat mungkin.
Belakangan pertukaran standar pertukaran digunakan dibeberapa sistem. Ini membutuhkan terlalu banyak waktu untuk menukar dari pada untuk mengeksekusi untuk solusi managemen memori yang masuk akal. Modifikasi *swapping* digunakan dibanyak versi di UNIX. Pertukaran awalnya tidak bisa, tapi akan mulai bila banyak proses yang jalan dan menggunakan batas jumlah memori.

4.3. Alokasi Memori Yang Berdampingan

Memori biasanya dibagi menjadi dua bagian, yakni:

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

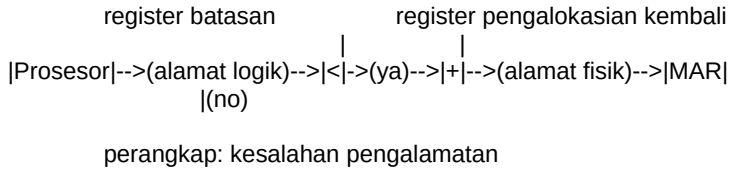
Sistem Operasi dapat dialokasikan pada memori bagian bawah (*low memory*) mau pun memori bagian atas (*high memory*). Hal ini tergantung pada letak vektor interupsi (*interrupt vector*) pada memori tersebut. Jika vektor interupsi lebih sering berada pada memori bawah, maka sistem operasi juga biasanya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

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

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

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

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



Gambar 4-1. Alokasi Kembali.

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

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

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

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

Kriteria-kriteria utama dalam metode ini antara lain:

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

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

- Alokasi memori: proses p membutuhkan k unit memori.
- Kebijakan alokasi:
    1. Sesuai yang terbaik: memilih lubang (*hole*) terkecil yang cukup besar untuk keperluan proses sehingga menghasilkan sisa lubang terkecil.

    2. Sesuai yang terburuk: memilih lubang terbesar sehingga menghasil kan sisa lubang.

    3. Sesuai yang pertama: memilih lubang pertama yang cukup besar untuk keperluan proses
        - Fragmentasi luar (*External Fragmentation*) yakni proses mengambil ruang, sebagian digunakan, sebagian tidak digunakan.

- Memori, yang tersedia untuk semua pengguna, dianggap sebagai suatu blok besar memori yang disebut dengan lubang. Pada suatu saat memori memiliki suatu daftar set lubang (*free list holes*).

- Saat suatu proses memerlukan memori, maka kita mencari suatu lubang yang cukup besar untuk kebutuhan proses tersebut.

- Jika ditemukan, kita mengalokasikan lubang tersebut ke proses tersebut sesuai dengan kebutuhan, dan sisanya disimpan untuk dapat digunakan proses lain.

4. Suatu proses yang telah dialokasikan memori akan dimasukkan ke memori dan selanjutnya dia akan bersaing dalam mendapatkan prosesor untuk pengeksekusiannya.

5. Jika suatu proses tersebut telah selesai, maka dia akan melepaskan kembali semua memori yang digunakan dan sistem operasi dapat mengalokasikannya lagi untuk proses lainnya yang sedang menunggu di antrian masukan.

6. Apabila memori sudah tidak mencukupi lagi untuk kebutuhan proses, sistem operasi akan menunggu sampai ada lubang yang cukup untuk dialokasikan ke suatu proses dalam antrian masukan.

7. ika suatu lubang terlalu besar, maka sistem operasi akan membagi lubang tersebut menjadi dua bagian, dimana satu bagian untuk dialokasikan ke proses tersebut dan satu lagi dikembalikan ke set lubang lainnya.

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

Fragmentasi luar mempunyai kriteria antara lain:

- Ruang memori yang kosong dibagi menjadi partisi kecil.

- Ada cukup ruang memori untuk memenuhi suatu permintaan, tetapi memori itu tidak lagi berhubungan antara satu bagian dengan bagian lain (*contiguous*) karena telah dibagi-bagi.

- Kasus terburuk (*Worst case*): akan ada satu blok ruang memori yang kosong yang terbuang antara setiap dua proses.

- Aturan 50 persen: dialokasikan N blok, maka akan ada 0.5N blok yang hilang akibat fragmentasi sehingga itu berarti 1/3 memori akan tidak berguna.

Perbandingan kompleksitas waktu dan ruang tiga kebijakan alokasi memori.

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

Gambar 4-2. Alokasi Kembali.

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

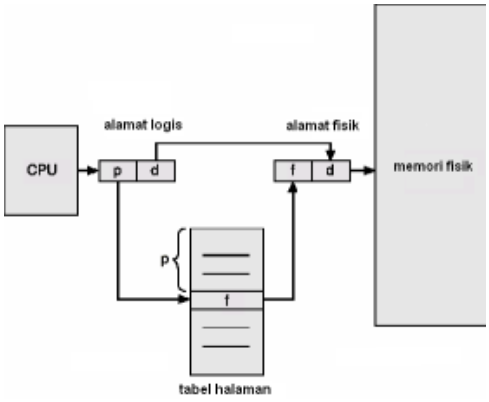
4.4. Pemberian Halaman

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

4.4.1. Metode Dasar

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

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



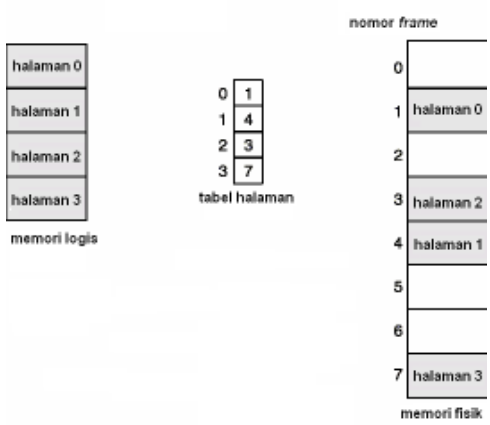
Gambar 4-3. Perangkat Keras Pemberian Halaman.

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

halaman. Jadi, alamat logisnya adalah: dimana p merupakan index ke tabel halaman dan d adalah pemindahan dalam halaman.

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

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



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

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

Ketika kita menggunakan skema pemberian halaman, kita tidak memiliki pemecah-mecahan luar: sembarang *frame* kosong dapat dialokasikan ke proses yang membutuhkan. Bagaimana pun juga kita mungkin mempunyai beberapa pemecahan di dalam. Mengingat bahwa *frame-frame* dialokasikan sebagai unit. Jika kebutuhan memori dari sebuah proses tidak menurun pada batas halaman, *frame* terakhir yang dialokasikan mungkin tidak sampai penuh. Untuk contoh, jika halamannya 2048 byte, proses 72.766 byte akan membutuhkan 35 halaman tambah 1086 byte. Alokasinya menjadi 36 frame, menghasilkan fragmentasi internal dari 2048 - 1086 = 962 byte. Pada kasus terburuknya, proses akan membutuhkan n halaman tambah satu byte. Sehingga dialokasikan n + 1 *frame*, menghasilkan fragmentasi internal dari hampir semua *frame*. Jika ukuran proses tidak bergantung dari ukuran halaman, kita mengharapkan fragmentasi internal hingga rata-rata setengah halaman per prosesnya. Pertimbangan ini memberi kesan bahwa ukuran halaman yang kecil sangat diperlukan sekali. Bagaimana pun juga, ada sedikit pemborosan dilibatkan dalam masukan tabel halaman, dan pemborosan ini dikurangi dengan ukuran halaman meningkat. Juga disk I/O lebih efisien ketika jumlah data yang dipindahkan lebih besar. Umumnya, ukuran halaman bertambah seiring bertambahnya waktu seperti halnya proses, himpunan data, dan memori utama telah menjadi besar. Hari ini, halaman umumnya berukuran 2 atau 4 kilobyte.

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

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

Karena sistem operasi mengatur memori fisik, maka harus waspada dari rincian alokasi memori fisik: frame mana yang dialokasikan, frame mana yang tersedia, berapa banyak total frame yang ada, dan masih banyak lagi. Informasi ini umumnya disimpan di struktur data yang disebut sebagai tabel frame. Tabel frame punya satu masukan untuk setiap fisik halaman frame, menandakan apakah yang terakhir teralokasi ataukah tidak, jika teralokasi maka kepada halaman mana dari proses mana.

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

4.4.2. Struktur Tabel Halaman

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

4.4.2.1. Dukungan Perangkat Keras

Implementasi perangkat keras dari tabel halaman dapat dilakukan dengan berbagai cara. Kasus sederhananya, tabel halaman diimplementasikan sebagai sebuah himpunan dari register resmi. Register ini harus yang bekecepatan tinggi agar penerjemahan alamat pemberian halaman efisien Setiap pengaksesan ke memori harus melalui peta

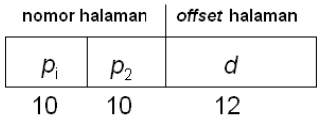
pemberian halaman, jadi ke-efisienan adalah pertimbangan utama. Pelaksana (*dispatcher*) CPU mengisi kembali register-register ini, seperti halnya ia mengisi kembali register yang lain. Instruksi untuk mengisi atau mengubah register tabel halaman adalah, tentu saja diberi hak istimewa, sehingga hanya sistem operasi yang dapat mengubah peta memori. DEC PDP-11 adalah contoh arsitektur yang demikian. Alamatnya terdiri dari 16-bit, dan ukuran halamannya 8K. Jadi tabel halaman terdiri dari 8 masukan yang disimpan di register-register cepat.

4.4.2.2. Pemeliharaan

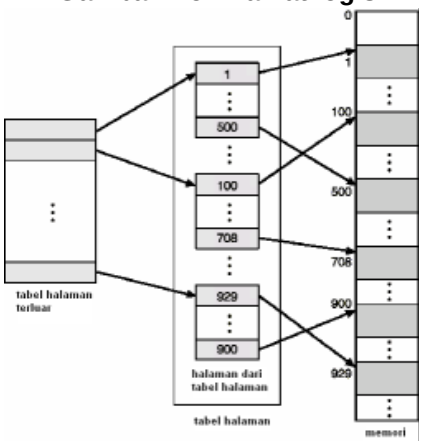
Proteksi memori dari suatu lingkungan berhalaman diselesaikan dengan bit-bit proteksi yang diasosiasikan dengan tiap-tiap *frame*. Normalnya, bit-bit ini disimpan di tabel halaman. Satu bit dapat menentukan halaman yang akan dibaca tulis atau baca saja. Setiap referensi ke memori menggunakan tabel halaman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

4.4.3. Pemberian Halaman Secara *Multilevel*

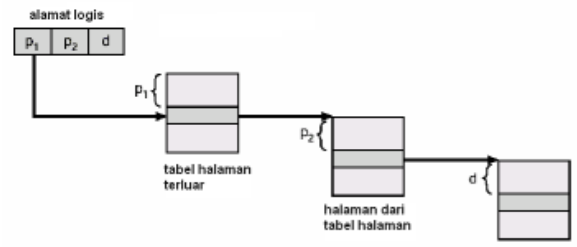
Banyak sistem komputer moderen mendukung ruang alamat logis yang sangat luas (2 pangkat 32 sampai 2 pangkat 64). Pada lingkungan seperti itu tabel halamanya sendiri menjadi sangat-sangat besat sekali. Untuk contoh, misalkan suatu sistem dengan ruang alamat logis 32-bit. Jika ukuran halaman di sistem seperti itu adalah 4K byte (2 pangkat 12), maka tabel halaman mungkin berisi sampai 1 juta masukan ((2^32)/(2^12)). Karena masing-masing masukan terdiri atas 4 byte, tiap-tiap proses mungkin perlu ruang alamat fisik sampai 4 megabyte hanya untuk tabel halamannya saja. Jelasnya, kita tidak akan mau mengalokasi tabel halaman secara berdekatan di dalam memori. Satu solusi sederhananya adalah dengan membagi tabel halaman menjadi potongan-potongan yang lebih kecil lagi. Ada beberapa cara yang berbeda untuk menyelesaikan ini.



Gambar 4-5. Alamat logis.



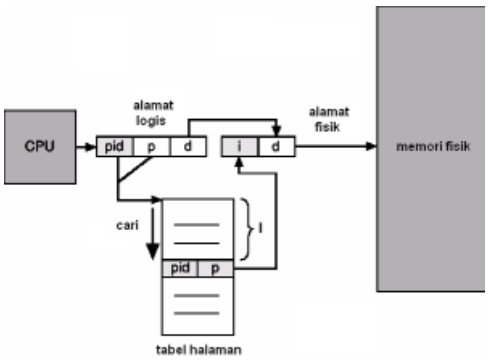
Gambar 4-6. Skema Tabel Halaman Dua Tingkat.



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

4.4.3.1. Tabel Halaman yang Dibalik

Biasanya, setiap proses mempunyai tabel halaman yang diasosiasikan dengannya. Tabel halaman hanya punya satu masukan untuk setiap halaman proses tersebut sedang gunakan (atau satu slot untuk setiap alamat maya, tanpa meperhatikan validitas terakhir). Semenjak halaman referensi proses melalui alamat maya halaman, maka representasi tabel ini adalah alami. Sistem operasi harus menterjemahkan referensi ini ke alamat memori fisik. Semenjak tabel diurutkan berdasarkan alamat maya, sistem operasi dapat menghitung dimana pada tabel yang diasosiasikan dengan masukan alamat fisik, dan untuk menggunakan nilai tersebut secara langsung. Satu kekurangan dari skema ini adalah masing-masing halaman mungkin mengandung jutaan masukan. Tabel ini mungkin memakan memori fisik dalam jumlah yang besar, yang mana dibutukan untuk tetap menjaga bagaimana memori fisik lain sedang digunakan.

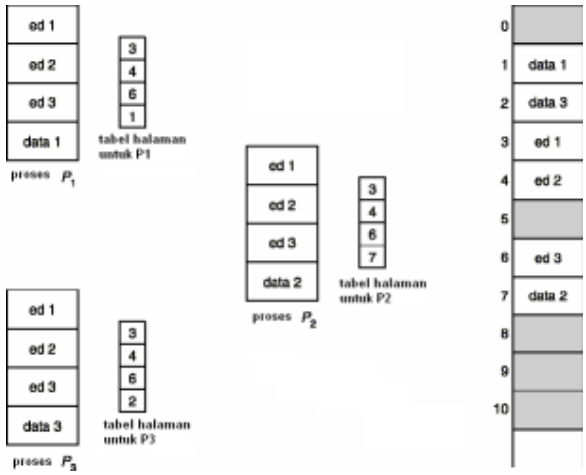


Gambar 4-8. Tabel halaman yang dibalik.

4.4.3.2. Berbagi Halaman

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

gambar) sedang dibagi-bagi diantara tiga proses. Masing-masing proses mempunyai halaman datanya sendiri.



Gambar 4-9. Berbagi kode pada lingkungan berhalaman.

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

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

Program-program lain pun juga dapat dibagi-bagi: *compiler*, *system window* database system, dan masih banyak lagi. Agar dapat dibagi-bagi, kodenya harus dimasukan kembali. System yang menggunakan tabel halaman yang dibalik mempunyai kesulitan dalam mengimplementasikan berbagi memori. Berbagi memori biasanya diimplementasikan sebagai dua alamat maya yang dipetakan ke satu alamat fisik. Metode standar ini tidak dapat digunakan, bagaimana pun juga selama di situ hanya ada satu masukan halaman maya untuk setiap halaman fisik, jadi satu alamat fisik tidak dapat mempunyai dua atau lebih alamat maya yang dibagi-bagi.

4.5. Segmentasi

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

4.5.1. Metode Dasar

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

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

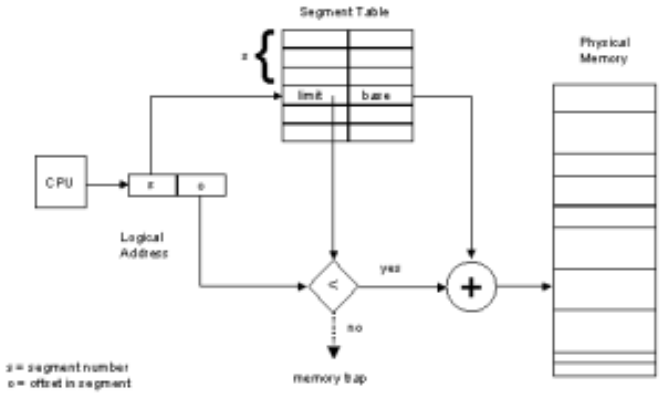
Segmentasi adalah sebuah bagian dari managemen memori yang mengatur pengalamatan dari memori yang terdiri dari segmen-segmen. *logical address space* adalah kumpulan dari segmen-segmen yang mana tiap-tiap segmen mempunyai nama dan panjang. alamat tersebut menunjukkan alamat dari segmen tersebut dan *offset*-nya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

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

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

4.5.2. Perangkat Keras

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



Gambar 4-10. Alamat Logis

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

4.5.3. Pemeliharaan dan Pembagian

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

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

4.5.4. Fragmentasi

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

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

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

4.6. Segmentasi Dengan Pemberian Halaman

4.6.1. Pengertian

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

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

4.6.2. Kelebihan Segmentasi dengan Pemberian Halaman

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

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

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

4.6.3. Perbedaan Segmentasi dan *Paging*

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

1. Segmentasi melibatkan programer (programer perlu tahu teknik yang digunakan), sedangkan dengan *paging*, programer tidak perlu tahu teknik yang digunakan.

2. Pada segmentasi kompilasi dilakukan secara terpisah sedangkan pada *paging*, kompilasinya tidak terpisah.
3. Pada segmentasi proteksinya terpisah sedangkan pada *paging* proteksinya tidak terpisah.

4. Pada segmentasi ada *shared code* sedangkan pada *paging* tidak ada *shared code*.

5. Pada segmentasi terdapat banyak ruang alamat linier sedangkan pada *paging* hanya terdapat satu ruang alamat linier.

6. Pada segmentasi prosedur dan data dapat dibedakan dan diproteksi terpisah sedangkan pada *paging* prosedur dan data tidak dapat dibedakan dan diproteksi terpisah.

7. Pada segmentasi pengubahan ukuran tabel dapat dilakukan dengan mudah sedangkan pada *Paging* pengubahan ukuran tabel tidak dapat dilakukan dengan mudah.

8. Segmentasi digunakan untuk mengizinkan program dan data dapat dipecahkan jadi ruang alamat mandiri dan juga untuk mendukung sharing dan proteksi sedangkan *paging* digunakan untuk mendapatkan ruang alamat linier yang besar tanpa perlu membeli memori fisik lebih.

4.6.4. Pengimplementasian Segmentasi dengan Pemberian Halaman Intel i386

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

4.6.4.1. *Logical Address*

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

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

2. Partisi II
   - Terdiri dari 8 K segmen yang digunakan bersama diantara proses-proses tersebut.
   - Informasi tentang partisi ini disimpan didalam *Global Descriptor Table*.

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

Logical address merupakan sepasang:

1. Selektor
   Terdiri dari angka 16 bit:
   Dimana s = jumlah segmen (13 bit)
   g = mengindikasikan apakah segmen ada di

Global Descriptor Table
atau *Local Descriptor Table*
(1 bit)
p= proteksi(2 bit)

s	g	p
13	1	2

2. *Offset*

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

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

4.6.4.2. Alamat Fisik

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

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

nomor halaman
offset halaman

p1	p2	d
10	10	12

4.7. Memori Virtual

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

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

1. Terdapat pernyataan-pernyataan atau pilihan yang hanya akan dieksekusi jika kondisi tertentu dipenuhi. Apabila kondisi tersebut tidak dipenuhi, maka pilihan tersebut tak

akan pernah dieksekusi/ diproses. Contoh dari pilihan itu adalah: pesan-pesan error yang hanya akan muncul bila terjadi kesalahan dalam eksekusi program.

2. Terdapat fungsi-fungsi yang jarang digunakan, bahkan sampai lebih dari 100x pemakaian.

3. Terdapat pealokasian memori lebih besar dari yang sebenarnya dibutuhkan. Contoh pada: *array*, *list*, dan tabel.

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

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

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

4.7.1. Pengertian

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

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

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

4.7.2. Keuntungan

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

- Berkurangnya I/O yang dibutuhkan (lalu lintas I/O menjadi rendah). Misal, untuk program butuh membaca dari disk dan memasukkan dalam memory setiap kali diakses.

- Berkurangnya memori yang dibutuhkan (*space* menjadi lebih leluasa). Contoh, untuk program 10 MB tidak seluruh bagian dimasukkan dalam memori. Pesan-pesan *error* hanya dimasukkan jika terjadi *error*.

- Meningkatnya respon, sebagai konsekuensi dari menurunnya beban I/O dan memori.

- Bertambahnya jumlah *user* yang dapat dilayani. Ruang memori yang masih tersedia luas memungkinkan komputer untuk menerima lebih banyak permintaan dari *user*.

4.7.3. Implementasi

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

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

Memori virtual dapat dilakukan melalui dua cara:

1. Permintaan pemberian halaman (*demand paging*).

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

4.8. Permintaan Pemberian Halaman (*Demand Paging*)

Merupakan implementasi yang paling umum dari memori virtual.

Prinsip permintaan pemberian halaman (*demand paging*) hampir sama dengan sistem penomoran (*paging*) dengan menggunakan *swapping*. Perbedaannya adalah *page* pada permintaan pemberian halaman tidak akan pernah di-*swap* ke memori sampai ia benar-benar diperlukan. Untuk itu diperlukan adanya pengecekan dengan bantuan perangkat keras mengenai lokasi dari *page* saat ia dibutuhkan.

4.8.1. Permasalahan pada *Page Fault*

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

1. *Page* ada dan sudah berada di memori.

2. *Page* ada tetapi belum ditaruh di memori (harus menunggu sampai dimasukkan).

3. *Page* tidak ada, baik di memori mau pun di disk (invalid reference --> abort).

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

4.8.2. Skema Bit Valid - Tidak Valid

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

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

Pengesetan bit:
Bit 1 -->
page berada di memori
Bit 0 -->
page tidak berada di memori.
(Dengan inisialisasi: semua bit
di-s*et* 0).

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

4.8.2.1. Penanganan *Page Fault*

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

1. Cek tabel internal yang dilengkapi dengan PCB untuk menentukan valid atau tidaknya bit.

2. Apabila tidak valid, program akan di-*terminate* (interupsi oleh *illegal address trap*).

3. Memilih *frame* kosong (*free-frame*), misal dari *free-frame list*. Jika tidak ditemui ada *frame* yang kosong, maka dilakukan *swap-out* dari memori. *Frame* mana yang harus di-*swap-out* akan ditentukan oleh algoritma (lihat sub bab *Page Replacement*).

4. Menjadualkan operasi disk untuk membaca *page* yang diinginkan ke *frame* yang baru dialokasikan.

5. Ketika pembacaan komplit, tabel internal akan dimodifikasi dan *page* diidentifikasi ada di memori.

6. Mengulang instruksi yang tadi telah sempat diinterupsi. Jika tadi *page fault* terjadi saat instruksi di-*fetch*, maka akan dilakukan *fecthing* lagi. Jika terjadi saat operan sedang di-*fetch*, maka harus dilakukan *fetch* ulang, *decode*, dan *fetch* operan lagi.

4.8.2.2. Permasalahan Lain yang berhubungan dengan *Demand Paging*

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

Masalah yang terjadi, antara lain mencakup:

1. Bagaimana mengulang instruksi yang memiliki beberapa lokasi yang berbeda?

2. Bagaimana pengalamatan dengan menggunakan *special-addresing mode*, termasuk *autoincrement* dan *autodecrement mode*?

3. Bagaimana jika instruksi yang dieksekusi panjang (contoh: *block move*)?

Masalah pertama dapat diatasi dengan dua cara yang berbeda.

- komputasi *microcode* dan berusaha untuk mengakses kedua ujung dari blok, agar tidak ada modifikasi*page* yang sempat terjadi.

- memanfaatkan register sementara (*temporary register*) untuk menyimpan nilai yang sempat tertimpa/ termodifikasi oleh nilai lain.

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

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

4.8.2.3. Persyaratan Perangkat Keras

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

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

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

4.9. Pemindahan Halaman

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

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

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

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

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

4.9.1. Skema Dasar

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

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

1. Cari lokasi dari halaman yang diinginkan pada *disk*

2. Cari *frame* kosong:
   a. Jika ada *frame* kosong, gunakan.
   b. Jika tidak ada *frame* kosong, gunakan algoritma pemindahan halaman untuk menyeleksi *frame* yang akan digunakan.
   c. Tulis halaman yang telah dipilih ke disk, ubah tabel halaman dan tabel *frame*.

3. Baca halaman yang diinginkan kedalam *frame* kosong yang baru, ubah tabel halaman dan tabel *frame*.

4. Ulang dari awal proses pengguna.

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

Pemindahan halaman merupakan dasar dari *demand paging*. Yang menjembatani pemisahan antara memori lojik dan memori fisik. Dengan mekanisme seperti ini, memori virtual yang sangat besar dapat disediakan untuk *programmer* dalam bentuk memori fisik yang lebih kecil. Dengan *nondemand paging*, alamat dari *user* dipetakan kedalam alamat fisik, jadi 2 set alamat dapat berbeda. Seluruh halaman dari proses masih harus berada di memori fisik. Dengan *demand paging*, ukuran dari ruang alamat logika sudah tidak dibatasi oleh memori fisik.

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

Ada beberapa algoritma pemindahan halaman yang berbeda. Kemungkinan setiap Sistem Operasi memiliki skema pemindahan yang unik. Algoritma pemindahan yang baik adalah yang memiliki tingkat kesalahan halaman terendah.
Kita mengevaluasi algoritma dengan menjalankannya dalam *string* khusus di memori acuan dan menghitung jumlah kesalahan halaman. *String* dari memori acuan disebut *string* acuan *(reference string)*.

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

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

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

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

Perlu diperhatikan bahwa selama jumlah *frame* meningkat, jumlah kesalahan halaman menurun.
Penambahan memori fisik akan meningkatkan jumlah *frame*.

4.9.2. Pemindahan Halaman Secara FIFO

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

Sebagai contoh:

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

frame halaman

Gambar 4-11. String Acuan.

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

4.9.3. Pemindahan Halaman Secara Optimal

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

Sebagai contoh:

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

Gambar 4-12. String Acuan.

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

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

4.9.4. Pemindahan Halaman Secara LRU

Jika algoritma optimal sulit untuk dilakukan, mungkin kita dapat melakukan pendekatan terhadap algoritma tersebut. Jika kita menggunakan waktu yang baru berlalu sebagai pendekatan terhadap waktu yang akan datang, kita akan memindahkan halaman yang sudah lama tidak digunakan dalam jangka waktu yang terlama. Pendekatan ini disebut algoritma LRU *(Least Recently Used)*.

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

Sebagai contoh:

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

frame halaman

Gambar 4-13. String Acuan.

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

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

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

frame halaman.

Gambar 4-14. String Acuan.

4.9.5. Pemindahan Halaman Secara Perkiraan LRU

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

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

4.9.5.1. Algoritma *Additional-Reference-Bit*

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

Sebagai contoh, seandainya register pengganti berisi 00000000, maka itu berarti halaman sudah tidak digunakan dalam periode 8 waktu terakhir, halaman yang digunakan paling tidak 1 kali akan memiliki nilai register penggati 11111111.

4.9.5.2. Algoritma *Second-Chance*

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

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

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

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

- (0,0) tidak digunakan dan tidak dimodifikasi, bit terbaik untuk dipindahkan.
- (0,1) tidak digunakan tapi dimodifikasi, tidak terlalu baik untuk dipindahkan karena halaman ini perlu ditulis sebelum dipindahkan.

- (1,0) digunakan tapi tidak dimodifikasi, terdapat kemungkinan halaman ini akan segera digunakan lagi.

- (1,1) digunakan dan dimodifikasi, halaman ini mungkin akan segera digunakan lagi dan halaman ini perlu ditulis ke *disk* sebelum dipindahkan.

Algoritma ini digunakan dalam skema manajemen memori virtual Macintosh.

4.9.6. Dasar Perhitungan Pemindahan Halaman

Banyak algoritma-algoritma lain yang dapat digunakan untuk pemindahan halaman. Sebagai contoh, kita dapat menyimpan *counter* dari nomor acuan yang sudah dibuat untuk masing-masing halaman, dan mengembangkan 2 skema dibawah ini:

ALGORITMA PEMINDAHAN HALAMAN LFU Algoritma LFU (*Least Frequently Used*) menginginkan halaman dengan nilai terkecil untuk dipindahkan. Alasannya, halaman yang digunakan secara aktif akan memiliki nilai acuan yang besar.

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

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

4.9.7. Algoritma *Page-Buffering*

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

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

4.10. Alokasi *Frame*

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

Kasus paling mudah dari memori virtual adalah sistem satu pemakai. Misalkan sebuah sistem mempunyai memori 128K dengan ukuran halaman 1K, sehingga ada 128 *frame*. Sistem operasinya menggunakan 35K sehingga ada 93 *frame* yang tersisa untuk proses tiap user. Untuk *pure demand paging*, ke-93 *frame* tersebut akan ditaruh pada daftar *frame* bebas. Ketika sebuah proses user mulai dijalankan, akan terjadi sederetan

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

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

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

4.10.1. Jumlah Frame Minimum

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

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

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

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

4.10.2. Algoritma Alokasi

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

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

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

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

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

ai = si / S x m

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

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

Satu pendekatan adalah menggunakan *proportional allocation scheme* dimana perbandingan *frame*-nya tidak tergantung pada ukuran relatif dari proses, melainkan lebih pada prioritas proses, atau tergantung kombinasi dari ukuran dan prioritas.

4.10.3. Alokasi Global lawan Local

Faktor penting lain dalam cara-cara pengalokasian *frame* ke berbagai proses adalah penggantian halaman. Dengan proses-proses yang bersaing mendapatkan *frame*, kita dapat mengklasifikasikan algoritma penggantian halaman kedalam dua kategori *broad*: Penggantian Global dan Penggantian Lokal.

Penggantian Global memperbolehkan sebuah proses untuk menyeleksi sebuah *frame* pengganti dari himpunan semua *frame*, meski pun *frame* tersebut sedang dialokasikan untuk beberapa proses lain; satu proses dapat mengambil sebuah *frame* dari proses yang lain. Penggantian Lokal mensyaratkan bahwa setiap proses boleh menyeleksi hanya dari himpunan *frame* yang telah teralokasi pada proses itu sendiri.

Untuk contoh, pertimbangkan sebuah skema alokasi dimana kita memperbolehkan proses berprioritas tinggi untuk meyeleksi *frame* dari proses berprioritas rendah untuk penggantian. Sebuah proses dapat menyeleksi sebuah pengganti dari *frame*-nya sendiri atau dari *frame-frame* proses yang berprioritas lebih rendah. Pendekatan ini memperbolehkan sebuah proses berprioritas tinggi untuk meningkatkan alokasi *frame*-nya pada expense proses berprioritas rendah.

Dengan strategi Penggantian Lokal, jumlah *frame* yang teralokasi pada sebuah proses tidak berubah. Dengan Penggantian Global, ada kemungkinan sebuah proses hanya menyeleksi *frame-frame* yang teralokasi pada proses lain, sehingga meningkatkan jumlah *frame* yang teralokasi pada proses itu sendiri (asumsi bahwa proses lain tidak memilih *frame* proses tersebut untuk penggantian).

Masalah pada algoritma Penggantian Global adalah bahwa sebuah proses tidak bisa mengontrol *page-fault*-nya sendiri. Himpunan halaman dalam memori untuk sebuah proses tergantung tidak hanya pada kelakuan paging dari proses tersebut, tetapi juga pada kelakuan *paging* dari proses lain. Karena itu, proses yang sama dapat tampil berbeda (memerlukan 0,5 detik untuk satu eksekusi dan 10,3 detik untuk eksekusi berikutnya) *due to totally external circumstances*. Dalam Penggantian Lokal, himpunan halaman dalam memori untuk sebuah proses hanya dipengaruhi kelakuan paging proses itu sendiri.

Penggantian Lokal dapat menyembunyikan sebuah proses dengan membuatnya tidak tersedia bagi proses lain, menggunakan halaman yang lebih sedikit pada memori. Jadi, secara umum Penggantian Global menghasilkan sistem throughput yang lebih bagus, maka itu artinya metode yang paling sering digunakan.

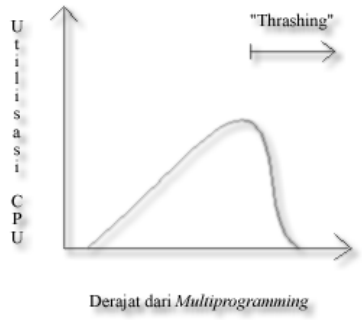
4.11. *Thrashing*

Jika suatu proses tidak memiliki *frame* yang cukup, walau pun kita memiliki kemungkinan untuk mengurangi banyaknya *frame* yang dialokasikan menjadi minimum, tetap ada halaman dalam jumlah besar yang memiliki kondisi aktif menggunakannya. Maka hal ini akan mengakibatkan kesalahan halaman. Pada kasus ini, kita harus mengganti beberapa halaman menjadi halaman yang dibutuhkan walau pun halaman yang diganti pada waktu dekat akan dibutuhkan lagi. Hal ini mengakibatkan kesalahan terus menerus.

Aktivitas yang tinggi dari *paging* disebut *thrashing*. Suatu proses dikatakan *thrashing* jika proses menghabiskan waktu lebih banyak untuk *paging* daripada eksekusi (proses sibuk untuk melakukan *swap-in swap-out*).

4.11.1. Penyebab *Thrashing*

Penyebab dari *thrashing* adalah utilisasi CPU yang rendah. Jika utilisasi CPU terlalu rendah, kita menambahkan derajat dari *multiprogramming* dengan menambahkan proses baru ke sistem. Sejalan dengan bertambahnya derajat dari *multiprogramming*, utilisasi CPU juga bertambah dengan lebih lambat sampai maksimumnya dicapai. Jika derajat dari *multiprogramming* ditambah terus menerus, utilisasi CPU akan berkurang dengan drastis dan terjadi *thrashing*. Untuk menambah utilisasi CPU dan menghentikan *thrashing*, kita harus mengurangi derajat dari *multiprogramming*.



Gambar 4-15. Derajat dari Multiprogramming.

Kita dapat membatasi efek dari *thrashing* dengan menggunakan algoritma penggantian lokal atau prioritas. Dengan penggantian lokal, jika satu proses mulai *thrashing*, proses tersebut tidak dapat mencuri *frame* dari proses yang lain dan menyebabkan proses tersebut tidak langsung mengalami *thrashing*. Jika proses *thrashing*, proses tersebut akan berada di antrian untuk melakukan *paging* yang mana hal ini memakan banyak waktu. Rata-rata waktu layanan untuk kesalahan halaman akan bertambah seiring dengan makin panjangnya rata-rata antrian untuk melakukan *paging*. Maka, waktu akses efektif akan bertambah walau pun untuk suatu proses yang tidak thrashing.

Untuk menghindari *thrashing*, kita harus menyediakan sebanyak mungkin *frame* sesuai dengan kebutuhan suatu proses. Cara untuk mengetahui berapa *frame* yang dibutuhkan salah satunya adalah dengan strategi *Working Set* yang akan dibahas pada bagian 10.5.2 yang mana strategi tersebut dimulai dengan melihat berapa banyak *frame* yang sesungguhnya digunakan oleh suatu proses. Ini merupakan model lokalitas dari suatu eksekusi proses.

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

4.11.2. Model *Working Set*

Model *Working Set* didasarkan pada asumsi lokalitas. Model ini menggunakan parameter Δ (delta) untuk mendefinisikan jendela *Working Set*. Idenya adalah untuk menentukan Δ halaman yang dituju yang paling sering muncul. Kumpulan dari halaman dengan Δ halaman yang dituju yang paling sering muncul disebut *Working Set*. *Working Set* adalah pendekatan dari program lokalitas.

Contoh 4-1. Tabel Halaman

Jika terdapat tabel halaman yang dituju dengan isinya
1 3 5 7 2 5 8 9 dengan
$\Delta = 8$,

Working Set pada waktu t1 adalah
{1, 2, 3, 5, 7, 8, 9}

Keakuratan *Working Set* tergantung pemilihan dari :
• Jika terlalu kecil, tidak akan dapat mewakilkan keseluruhan dari lokalitas.

• Jika terlalu besar, akan menyebabkan *overlap* beberapa lokalitas.
• Jika tidak terbatas, *Working Set* adalah kumpulan halaman sepanjang eksekusi proses.

Jika kita menghitung ukuran dari *Working Set*, WWSi, untuk setiap proses pada sistem, kita hitung dengan D = Δ WSSi, dimana D merupakan total *demand* untuk *frame*.

Jika total *demand* lebih dari total banyaknya *frame* yang tersedia (D > m), *thrashing* dapat terjadi karena beberapa proses akan tidak memiliki *frame* yang cukup. Jika hal tersebut terjadi, dilakukan satu pengeblokan dari proses-proses yang sedang berjalan.

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

Kesulitan dari model *Working Set* ini adalah menjaga *track* dari *Working Set*. Jendela *Working Set* adalah jendela yang bergerak. Suatu halaman berada pada *Working Set* jika halaman tersebut mengacu ke mana pun pada jendela *Working Set*. Kita dapat mendekati model *Working Set* dengan *fixed interval timer interrupt* dan *reference* bit.

Contoh: Δ = 10000 reference, *Timer interrupt* setiap 5000 reference.

Ketika kita mendapat *interrupt*, kita kopi dan hapus nilai *reference* bit dari setiap halaman.

Jika kesalahan halaman muncul, kita dapat menentukan *current reference* bit dan 2 pada bit memori untuk memutuskan apakah halaman itu digunakan dengan 10000 ke 15000 reference terakhir.

Jika digunakan, paling sedikit satu dari bit-bit ini akan aktif. Jika tidak digunakan, bit ini akan menjadi tidak aktif.

Halaman yang memiliki paling sedikit 1 bit aktif, akan berada di *working-set*.

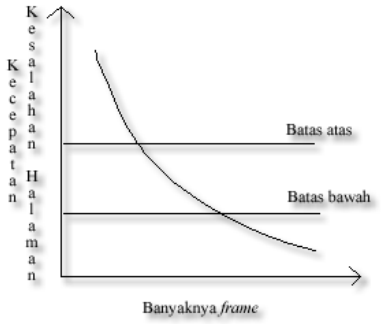
Hal ini tidaklah sepenuhnya akurat karena kita tidak dapat memberitahukan dimana pada interval 5000 tersebut, *reference* muncul. Kita dapat mengurangi ketidakpastian dengan menambahkan sejarah bit kita dan frekuensi dari *interrupt*.

Contoh: 20 bit dan interrupt setiap 1500 reference.

4.11.3. Frekuensi Kesalahan Halaman

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

Thrashing memiliki kecepatan kesalahan halaman yang tinggi. Kita ingin mengontrolnya. Ketika terlalu tinggi, kita mengetahui bahwa proses membutuhkan *frame* lebih. Sama juga, jika terlalu rendah, maka proses mungkin memiliki terlalu banyak *frame*. Kita dapat menentukan batas atas dan bawah pada kecepatan kesalahan halaman seperti terlihat pada gambar berikut ini.



Gambar 4-16. Jumlah Frame.

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

4.12. Contoh Pada Sistem Operasi

Pada bagian ini kita akan membahas beberapa contoh dalam penggunaan memori virtual.

4.12.1. Windows NT

Windows NT mengimplementasikan memori virtual dengan menggunakan demand paging melalui clustering. Clustering menanganani *page fault* dengan menambahkan tidak hanya page yang terkena fault, tetapi juga beberapa *page* yang ada dekat *page*tersebut. Saat proses pertama dibuat, dia diberikan *Working Set minimum* yaitu jumlah minimum *page* yang dijamin akan dimiliki oleh proses tersebut dalam memori. Jika memori yang cukup tersedia, proses dapat diberikan page sampai sebanyak *Working Set maximum*. Manager memori virtual akan menyimpan daftar dari *frame page* yang bebas. Terdapat juga sebuah nilai batasan yang diasosiasikan dengan daftar ini untuk mengindikasikan apakah memori yang tersedia masih mencukupi. Jika proses tersebut sudah sampai pada *Working Set maximum*-nya dan terjadi *page fault*, maka dia harus memilih page pengganti dengan menggunakan kebijakan penggantian page lokal FIFO.

Saat jumlah memori bebas jatuh di bawah nilai batasan, manager memori virtual menggunakan sebuah taktik yang dikenal sebagai *automatic working set trimming* untuk mengembalikan nilai tersebut di atas batasan. Hal ini bekerja dengan mengevaluasi jumlah page yang dialokasikan kepada proses. Jika proses telah mendapat alokasi *page* lebih besar daripada *Working Set minimum*-nya, manager memori virtual akan menggunakan algoritma FIFO untuk mengurangi jumlah *page*-nya sampai *working-set minimum*.

Jika memori bebas sudah tersedia, proses yang bekerja pada *working set minimum* dapat mendapatkan *page* tambahan.

4.12.2. Solaris 2

Dalam sistem operasi Solaris 2, jika sebuah proses menyebabkan terjadi *page fault*, kernel akan memberikan *page* kepada proses Tersebut dari daftar *page* bebas yang disimpan. Akibat dari hal ini adalah, kernel harus menyimpan sejumlah memori bebas. Terhadap daftar ini ada dua parameter yg disimpan yaitu *minfree* dan *lotsfree*, yaitu batasan minimum dan maksimum dari memori bebas yang tersedia. Empat kali dalam tiap detiknya, kernel memeriksa jumlah memori yang bebas. Jika jumlah tersebut jatuh di bawah *minfree*, maka sebuah proses *pageout* akan dilakukan, dengan pekerjaan sebagai berikut. Pertama *clock* akan memeriksa semua *page* dalam memori dan mengeset bit referensi menjadi 0.

Saat berikutnya, *clock* kedua akan memeriksa bit referensi *page* dalam memori, dan mengembalikan bit yang masih di set ke 0 ke daftar memori bebas. Hal ini dilakukan sampai jumlah memori bebas melampaui parameter *lotsfree*. Lebih lanjut, proses ini dinamis, dapat mengatur kecepatan jika memori terlalu sedikit. Jika proses ini tidak bisa membebaskan memori, maka kernel memulai pergantian proses untuk membebaskan *page* yang dialokasikan ke proses-proses tersebut.

4.12.3. Linux

Seperti pada solaris 2, linux juga menggunakan variasi dari algoritma *clock*. Thread dari kernel linux (kswapd) akan dijalankan secara periodik (atau dipanggil ketika penggunaan memori sudah berlebihan).

Jika jumlah *page* yang bebas lebih sedikit dari batas atas *page* bebas, maka thread tersebut akan berusaha untuk membebaskan tiga *page*. Jika lebih sedikit dari batas bawah *page* bebas, thread tersebut akan berusaha untuk membebaskan 6 *page* dan 'tidur' untuk beberapa saat sebelum berjalan lagi. Saat dia berjalan, akan memeriksa mem_map, daftar dari semua page yang terdapat di memori. Setiap page mempunyai byte umur yang diinisialisasikan ke 3. Setiap kali *page* ini diakses, maka umur ini akan ditambahkan (hingga maksimum 20), setiap kali kswapd memeriksa *page* ini, maka umur akan dikurangi. Jika umur dari sebuah *page* sudah mencapai 0 maka dia bisa ditukar. Ketika kswapd berusaha membebaskan *page*, dia pertama akan membebaskan *page* dari *cache*, jika gagal dia akan mengurangi *cache* sistim berkas, dan jika semua cara sudah gagal, maka dia akan menghentikan sebuah proses.

Alokasi memori pada linux menggunakan dua buah alokasi yang utama, yaitu algoritma *buddy* dan *slab*.

Untuk algoritma *buddy*, setiap rutin pelaksanaan alokasi ini dipanggil, dia memeriksa blok memori berikutnya, jika ditemukan dia dialokasikan, jika tidak maka daftar tingkat berikutnya akan diperiksa.

Jika ada blok bebas, maka akan dibagi jadi dua, yang satu dialokasikan dan yang lain dipindahkan ke daftar yang di bawahnya.

4.13. Pertimbangan Lain

Pemilihan algoritma penggantian dan aturan alokasi adalah keputusan-keputusan utama yang kita buat untuk sistem pemberian halaman. Masih banyak pertimbangan lain.

4.13.1. Sebelum Pemberian Halaman

Sebuah ciri dari sistem *demand-paging* adalah adanya *page fault* yang terjadi saat proses dimulai. Situasi ini adalah hasil dari percobaan untuk mendapatkan tempat pada awalnya. Situasi yang sama mungkin muncul di lain waktu. Saat proses *swapped-out* dimulai kembali, seluruh halaman ada di *disk* dan setiap halaman harus dibawa masuk oleh *page-fault*-nya masing-masing. Sebelum pemberian halaman mencoba untuk mencegah tingkat tinggi dari *paging* awal. Stateginya adalah untuk membawa seluruh halaman yang akan dibutuhkan pada satu waktu ke memori.

Pada sistem yang menggunakan model *working-set*, sebagai contoh, kita tetap dengan setiap proses sebuah daftar dari halaman-halaman di *working-set*-nya. Jika kita harus menunda sebuah proses (karena menunggu I/O atau kekurangan *frame* bebas), kita mengingat *working-set* untuk proses itu. Saat proses itu akan melanjutkan kembali (I/O komplit atau *frame* bebas yang cukup), kita secara otomatis membawa kembali ke memori seluruh *working-set* sebelum memulai kembali proses tersebut.

Sebelum pemberian halaman bisa unggul di beberapa kasus. Pertanyaan sederhananya adalah apakah biaya untuk menggunakan sebelum pemberian halaman itu lebih rendah daripada biaya melayani *page-fault* yang berhubungan. Itu mungkin menjadi kasus dimana banyak halaman dibawa kembali ke memori dengan sebelum pemberian halaman tidak digunakan.

4.13.2. Ukuran Halaman

Para perancang sistem operasi untuk mesin yang ada kini jarang memiliki pilihan terhadap ukuran halaman. Bagaimana pun, saat mesin-mesin baru sedang dibuat, pemilihan terhadap ukuran halaman terbaik harus dibuat. Seperti yang kau mungkin harapkan, tidak ada sebuah ukuran halaman yang terbaik.

Namun, ada himpunan faktor-faktor yang mendukung ukuran-ukuran yang bervariasi. Ukuran-ukuran halaman selalu dengan pangkat 2, secara umum berkisar dari 4.096 (2^12) ke 4.194.304 (2^22) bytes

Bagaimana kita memilih sebuah ukuran halaman? Sebuah perhatian adalah ukuran dari tabel halaman.

Untuk sebuah memori virtual dengan ukuran 4 megabytes (2^22), akan ada 4.096 halaman 1.024 bytes, tapi hanya 512 halaman 8.192 bytes. Sebab setiap proses aktif harus memiliki salinan dari tabel halamannya, sebuah halaman yang besar diinginkan.

Di sisi lain, memori lebih baik digunakan dengan halaman yang lebih kecil. Jika sebuah proses dialokasikan di memori mulai dari lokasi 00000, melanjutkan sampai memiliki sebanyak yang dibutuhkan, itu mungkin tidak akan berakhir secara tepat di batas halaman. Kemudian, sebuah bagian dari halaman terakhir harus dialokasikan (sebab halaman-halaman adalah unit-unit dari alokasi) tapi tidak digunakan (pemecahan bagian dalam). Asumsikan ketergantungan antara ukuran proses dan ukuran halaman, kita dapat mengharapkan bahwa, dalam rata-rata, satu-setengah dari halaman terakhir dari setiap proses akan dibuang. Kehilangan ini hanya 256 bytes dari sebuah halaman 512 bytes, tapi akan 4.096 bytes dari halaman 8.192 bytes. Untuk meminimalkan pemecahan bagian dalam, kita membutuhkan ukuran halaman yang kecil.

Masalah lain adalah waktu yang dibutuhkan untuk membaca atau menulis halaman. Waktu I/O terdiri dari mencari, keterlambatan dan waktu pemindahan. Waktu transfer proporsional terhadap jumlah yang dipindahkan (yaitu, ukuran tabel). Sebuah fakta bahwa yang mungkin terasa janggal untuk ukuran tabel yang kecil. Ingat kembali dari

Bab 2, bagaimana pun, keterlambatan dan waktu pencarian normalnya membuat waktu pemindahan menjadi kecil. Pada saat laju pemindahan 2 megabytes per detik, hanya menghabiskan 0.2 millidetik untuk memindahkan 512 bytes. Keterlambatan, di sisi lain, kira-kira 8 millidetik dan waktu pencarian 20 millidetik. Dari total waktu I/O (28.2 millidetik), untuk itulah, 1 persen dapat dihubungkan dengan pemindahan sebenarnya. Menggandakan ukuran halaman meningkatkan waktu I/O hingga 28.4 millidetik. Menghabiskan 28.4 millidetik untuk membaca halaman tunggal dari dari 1.024 bytes, tapi 56.4 millidetik untuk jumlah yang sama sebesar dua halaman masing-masing 512 bytes. Kemudian, keinginan untuk meminimalisir waktu I/O untuk ukuran halaman yang lebih besar

4.13.3. Tabel Halaman yang Dibalik

Kegunaan dari bentuk manajemen halaman adalah untuk mengurangi jumlah memori fisik yang dibutuhkan untuk melacak penerjemahan alamat *virtual-to-physical*. Kita menyelesaikan metode penghematan ini dengan membuat tabel yang memiliki hanya satu masukan tiap halaman memori fisik, terdaftar oleh pasangan (pengenal proses, nomor halaman).

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

Tetapi, melakukan tabel halaman luar menegasikan kegunaan tabel halaman yang dibalik? Sejak tabel-tabel ini direferensikan hanya saat *page fault* terjadi, mereka tidak perlu untuk tersedia secara cepat. Namun, mereka masing-masing diberikan atau dikeluarkan halaman dari memori sesuai kebutuhan. Sayangnya, sebuah *page fault* mungkin sekarang muncul di manager memori virtual menyebabkan halaman lain *fault* seakan-akan halaman ditabel halaman luar perlu untuk mengalokasikan *virtual page* di bantuan penyimpanan. Ini merupakan kasus spesial membutuhkan penanganan di kernel dan delay di proses melihat halaman.

4.13.4. Struktur Program

Demand paging didesain untuk menjadi transparan kepada program pemakai. Di banyak kasus, pemakai sama sekali tidak mengetahui letak halaman di memori. Di kasus lain, bagaimana pun, kinerja sistem dapat ditingkatkan jika pemakai (atau kompilator) memiliki kesadaran akan *demand paging* yang mendasar Pemilihan yang hati-hati dari struktur data dan struktur permograman dapat meningkatkan *locality* dan karenanya menurunkan laju *page fault* dan jumlah halaman di himpunan kerja. Sebuah *stack* memiliki *locality* yang baik, sejak akses selalu dibuat di atas. Sebuah *hash table*, di sisi lain, didesain untuk menyebar referensi-referensi, menghasilkan *locality* yang buruk. Tentunya, referensi akan *locality* hanyalah satu ukuran dari efisiensi penggunaan struktur data. Faktor-faktor lain yang berbobot berat termasuk kecepatan pencarian, jumlah total dari referensi dan jumlah total dari halaman yang disentuh.

4.13.5. Penyambungan Masukan dan Keluaran

Saat *demand paging* digunakan, kita terkadang harus mengizinkan beberapa halaman untuk dikunci di memori. Salah satu situasi muncul saat I/O sering diimplementasikan oleh pemroses I/O yang terpisah.

Sebagai contoh, sebuah pengendali pita magnetik pada umumnya diberikan sejumlah bytes untuk memindahkan dan sebuah alamat memoro untuk *buffer*. Saat pemindahan selesai, CPU diinterupsi.

Kita harus meyakinkan urutan dari kejadian-kejadian berikut tidak muncul: Sebuah proses mengeluarkan permintaan I/O, dan diletakkan di antrian untuk I/O tersebut. Sementara itu, CPU diberikan ke proses-proses lain. Proses-proses ini menyebabkan kesalahan penempatan halaman, dan, menggunakan algoritma penggantian global, salah satu dari mereka menggantikan halaman yang mengandung *memory buffer* untuk proses yang menunggu. Halaman itu dikeluarkan. Kadang-kadang kemudian, saat permintaan I/O bergerak maju menuju ujung dari antrian *device*, I/O terjadi ke alamat yang telah ditetapkan. Bagaimana pun, *frame* ini sekarang sedang digunakan untuk halaman berbeda milik proses lain.

4.13.6. Pemrosesan Waktu Nyata

Diskusi-diskusi di bab ini telah dikonsentrasikan dalam menyediakan penggunaan yang terbaik secara menyeluruh dari sistem komputer dengan meningkatkan penggunaan memori. Dengan menggunakan memori untuk data yang aktif, dan memindahkan data yang tidak aktif ke *disk*, kita meningkatkan *throughput*. Bagaimana pun, proses *individual* dapat menderita sebagai hasilnya, sebab mereka sekarang dapat menyebabkan *page faults* tambahan selama eksekusi mereka.

Pertimbangkan sebuah proses atau *thread* waktu-nyata. Sebuah proses mengharapkan untuk memperoleh kendali CPU, dan untuk menjalankan penyelesaian dengan delay yang minimum. Memori virtual adalah saingan yang tepat untuk perhitungan waktu-nyata, sebab dapat menyebabkan delay jangka panjang, yang tidak diharapkan pada eksekusi sebuah proses saat halaman dibawa ke memori. Untuk itulah, sistem-sistem waktu-nyata hampir tidak memiliki memori virtual.

Pada kasus Solaris 2, para pengembang di Sun Microsystems ingin mengizinkan baik *time-sharing* dan perhitungan waktu nyata pada sebuah sistem. Untuk memecahkan masalah *page-fault*, mereka memiliki Solaris 2 mengizinkan sebuah proses untuk memberitahu bagian halaman mana yang penting untuk proses itu. Sebagai tambahan untuk mengizinkan petunjuk-petunjuk akan halaman yang digunakan, sistem operasi mengizinkan pemakai-pemakai yang berhak dapat mengunci halaman yang dibutuhkan di memori. Jika, disalah-gunakan, mekanisme ini dapat mengunci semua proses lain keluar dari sistem.

Adalah perlu untuk mengizinkan proses-proses waktu-nyata untuk dapat dibatasi *low-dispatch latency*