Bab 7

Manajemen Memori

POKOK BAHASAN:

- ✓ Latar Belakang
- ✓ Ruang Alamat Logika dan Ruang Alamat Fisik
- ✓ Swapping
- ✓ Alokasi berurutan
- ✓ Paging
- ✓ Segmentasi
- ✓ Segmentasi dengan Paging

TUJUAN BELAJAR:

Setelah mempelajari materi dalam bab ini, mahasiswa diharapkan mampu:

- ✓ Memahami latar belakang manajemen memori
- ✓ Memahami maksud ruang alamat logika dan ruang alamat fisik
- ✓ Memahami teknik swapping pada manajemen memori
- ✓ Memahami teknik alokasi memori secara berurutan
- ✓ Memahami teknik alokasi memori tak berurutan yaitu sistem paging dan segmentasi
- ✓ Mengetahui implementasi manajemen memori

7.1 LATAR BELAKANG

Memori adalah pusat dari operasi pada sistem komputer modern. Memori adalah array besar dari *word* atau *byte*, yang disebut alamat. CPU mengambil instruksi dari memory berdasarkan nilai dari *program counter*. Instruksi ini menyebabkan penambahan muatan dari dan ke alamat memori tertentu.

Instruksi eksekusi yang umum, contohnya, pertama mengambil instruksi dari memori. Instruksi dikodekan dan mungkin mengambil *operand* dari memory. Setelah instruksi dieksekusi pada *operand*, hasilnya ada yang dikirim kembali ke memory. Sebagai catatan, unit memory hanya merupakan deretan alamat memory; tanpa tahu bagaimana membangkitkan (*instruction counter*, *indexing*, *indirection*, *literal address* dan lainnya) atau untuk apa (instruksi atau data). Oleh karena itu, kita dapat mengabaikan bagaimana alamat memori dibangkitkan oleh program, yang lebih menarik bagaimana deretan alamat memori dibangkitkan oleh program yang sedang berjalan.

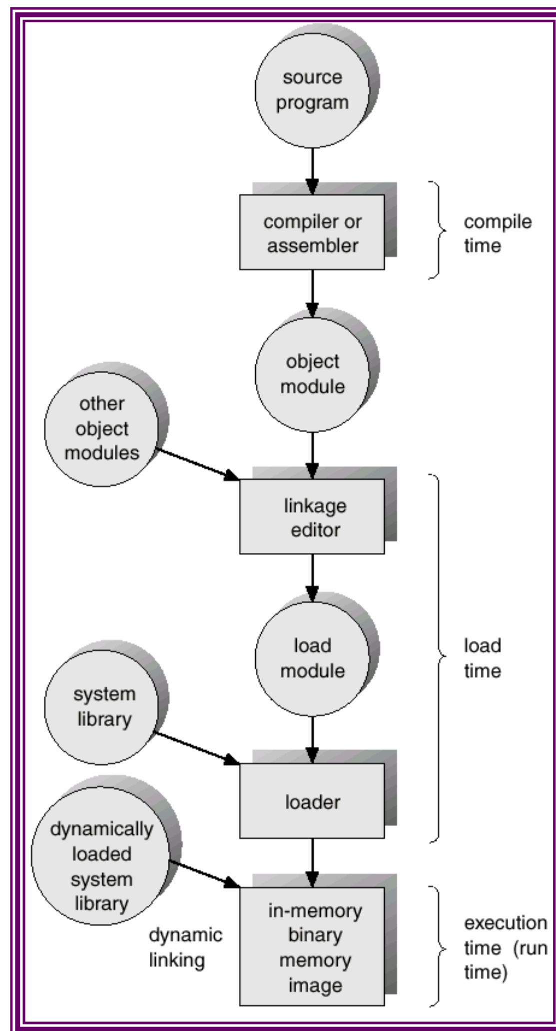
7.1.1 Pengikatan Alamat (*Address Binding*)

Pengikatan alamat adalah cara instruksi dan data (yang berada di disk sebagai file yang dapat dieksekusi) dipetakan ke alamat memori. Sebagian besar sistem memperbolehkan sebuah proses user (*user process*) untuk meletakkan di sembarang tempat dari memori fisik. Sehingga, meskipun alamat dari komputer dimulai pada 00000, alamat pertama dari proses user tidak perlu harus dimulai 00000. Pada beberapa kasus, program user akan melalui beberapa langkah sebelum dieksekusi (Gambar 7-1). Alamat pada *source program* umumnya merupakan alamat simbolik. Sebuah compiler biasanya melakukan pengikatan alamat simbolik (*symbolic address*) ke alamat relokasi dipindah (*relocatable address).* Misalnya compiler mengikatkan alamat simbolik ke alamat relokasi "14 byte from the beginning of this module". Editor Linkage mengikatkan alamat relokasi ini ke alamat absolute (*absolute addresses*) "74014".

Instruksi pengikatan instruksi dan data ke alamat memori dapat dilakukan pada saat :

- *Compile time* : Jika lokasi memori diketahui sejak awal, kode absolut dapat dibangkitkan, apabila terjadi perubahan alamat awal harus dilakukan kompilasi ulang. Misalnya : program format .com pada MS-DOS adalah kode absolut yang diikat pada saat waktu kompilasi
- *Load time* : Harus membangkitkan kode relokasi jika lokasi memori tidak diketahui pada saat waktu kompilasi.
- *Execution time* : Pengikatan ditunda sampai waktu eksekusi jika proses dapat dipindahkan selama eksekusi dari satu segmen memori ke segmen memori lain.

Memerlukan dukungan perangkat keras untuk memetakan alamat (misalnya register basis dan limit).



Gambar 7-1 : Beberapa langkah proses program user

7.1.2 Dinamic Loading

Untuk memperoleh utilitas ruang memori, dapat menggunakan *dynamic loading*. Dengan *dynamic loading*, sebuah rutin tidak disimpan di memori sampai dipanggil. Semua rutin disimpan pada disk dalam format *relocatable load*.

Mekanisme dari *dynamic loading* adalah program utama di-load dahulu dan dieksekusi. Bila suatu routine perlu memanggil routine lain, routine yang dipanggil

lebih dahulu diperiksa apakah rutin yang dipanggil sudah di-*load*. Jika tidak, *relocatable linking loader* dipanggil untuk me-*load* rutin yg diminta ke memori dan meng-ubah tabel alamat.

Keuntungan dari *dynamic loading* adalah rutin yang tidak digunakan tidak pernah di-*load*. Skema ini lebih berguna untuk kode dalam jumlah besar diperlukan untuk menangani kasus-kasus yang jarang terjadi seperti *error routine*. *Dinamic loading* tidak memerlukan dukungan khusus dari sistem operasi. Sistem operasi hanya perlu menyediakan beberapa rutin pustaka untuk implementasi *dynamic loading*.

7.1.3 Dinamic Linking

Sebagian besar sistem operasi hanya men-*support static linking*, dimana sistem *library language* diperlakukan seperti obyek modul yang lain dan dikombinasikan dengan *loader* ke dalam *binary program image*. Konsep *dynamic linking* sama dengan *dynamic loading*. Pada saat loading, linking ditunda sampai waktu eksekusi. Terdapat kode kecil yang disebut *stub* digunakan untuk meletakkan rutin library di memori dengan tepat. *Stub* diisi dengan alamat rutin dan mengeksekusi rutin. Sistem operasi perlu memeriksa apakah rutin berada di alamat memori.

Dinamic linking biasanya digunakan dengan sistem *library*, seperti *language subroutine library*. Tanpa fasilitas ini, semua program pada sistem perlu mempunyai *copy* dari *library language* di dalam *executable image*. Kebutuhan ini menghabiskan baik ruang disk maupun memori utama.

Bagaimanapun, tidak seperti *dynamic loading, dynamic linking* membutuhkan beberapa dukungan dari sistem operasi, misalnya bila proses-proses di memori utama saling diproteksi, maka sistem operasi melakukan pengecekan apakah rutin yang diminta berada diluar ruang alamat. Beberapa proses diijinkan untuk mengakses memori pada alamat yang sama. File *dynamic linking* berekstensi .dll, .sys atau .drv

7.1.4 Overlay

Sebuah proses dapat lebih besar daripada jumlah memori yang dialokasikan untuk proses, teknik *overlay* biasanya digunakan untuk kasus ini. Teknik *Overlay* biasanya digunakan untuk memungkinkan sebuah proses mempunyai jumlah yang lebih besar dari memori fisik daripada alokasi memori yang diperuntukkan. Ide dari *overlay*

adalah menyimpan di memori hanya instruksi dan data yang diperlukan pada satu waktu. Jika intruksi lain diperlukan, maka instruksi tersebut diletakkan di ruang memori menggantikan instruksi yang tidak digunakan lagi.

Sebagai contoh misalnya terdapat two-pass assembler. Selama pass 1, dibangun table symbol, dan selama pass 2 dibangkitkan kode bahasa mesin. Kita dapat membagi assembler ke dalam kode pass 1, kode pass 2, tabel symbol dan rutin umum yang digunakan baik pada pass 1 maupun pass 2. Diasumsikan ukuran komponen sebagai berikut :

Pass 1	70K
Pass 2	80K
Tabel symbol	20K
Rutin umum	30K

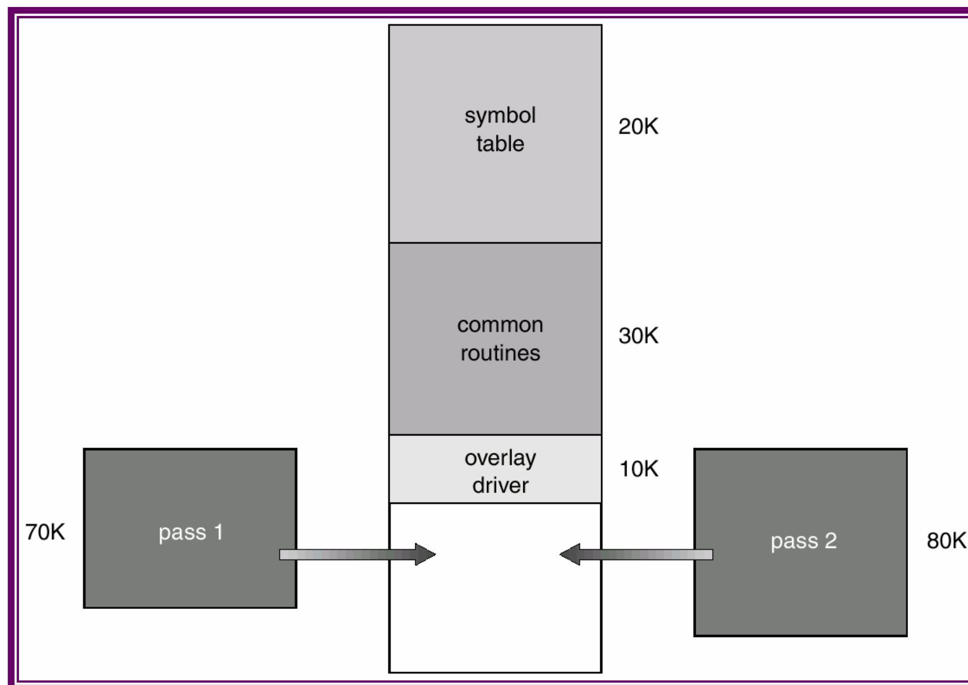
Apabila semua diletakkan di memori memerlukan 200K. Jika hanya tersedia tempat 150K, proses tidak dapat dijalankan. Pass 1 dan pass 2 tidak perlu berada di memori pada waktu yang sama. Dengan menggunakan *overlay*, rutin dibagi dalam 2 *overlay*. *Overlay A* adalah tabel symbol, rutin umum dan pass1 (membutuhkan total 120K) dan *overlay B* terdiri dari tabel symbol, rutin umum dan pass 2 (membutuhkan 130K). Ditambahkan 10K untuk driver *overlay* dan dimulai dari *overlay A*. Setelah selesai dijalankan *overlay B* dengan mengganti tempat dari *overlay A*. Gambaran *overlay* dapat dilihat pada Gambar 7-2.

Overlay tidak membutuhkan dukungan khusus dari sistem operasi. User dapat mengimplementasikannya secara lengkap menggunakan struktur file sederhana, membaca dari file ke memori dan meloncat ke memori dan mengeksekusi instruksi read yang lebih baru. Sistem operasi memberitahu hanya jika terdapat I/O yang melebihi biasanya. Penggunaan overlay terbatas untuk beberapa sistem yang mempunyai jumlah memori fisik terbatas dan kekurangan dukungan H/W untuk teknik yang lebih lanjut.

7.2 RUANG ALAMAT LOGIKA DAN RUANG ALAMAT FISIK

Alamat yang dibangkitkan oleh CPU disebut alamat logika (*logical address*) dimana alamat terlihat sebagai uni memory yang disebut alamat fisik (*physical address*).

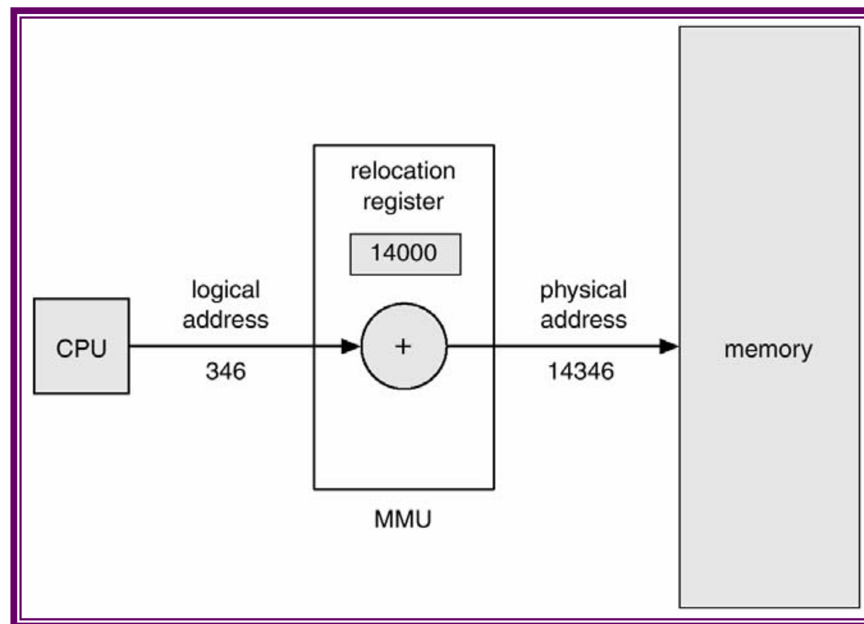
Tujuan utama manajemen memori adalah konsep meletakkan ruang alamat logika ke ruang alamat fisik.



Gambar 7-2 : Overlay

Hasil skema waktu kompilasi dan waktu pengikatan alamat pada alamat logika dan alamat memori adalah sama. Tetapi hasil skema waktu pengikatan alamat waktu eksekusi berbeda. dalam hal ini, alamat logika disebut dengan alamat maya (*virtual address*). Himpunan dari semua alamat logika yang dibangkitkan oleh program disebut dengan ruang alamat logika (*logical address space)*; himpunan dari semua alamat fisik yang berhubungan dengan alamat logika disebut dengan ruang alamat fisik (*physical address space*).

Memory Manajement Unit (MMU) adalah perangkat keras yang memetakan alamat virtual ke alamat fisik. Pada skema MMU, nilai register relokasi ditambahkan ke setiap alamat yang dibangkitkan oleh proses user pada waktu dikirim ke memori.



Gambar 7-3 : Relokasi dinamis menggunakan register relokasi

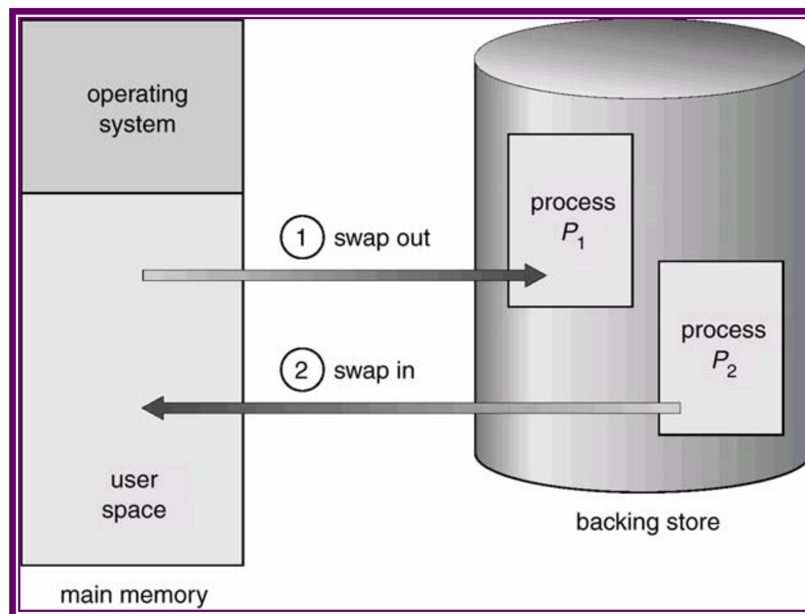
Gambar 7-3 merupakan skema yang membutuhkan dukungan perangkat keras. Register basis disebut register relokasi. Nilai dari register relokasi ditambahkan ke setiap alamat yang dibangkitkan oleh proses user pada waktu dikirim ke memori. sebagai contoh, apabila basis 14000, maka user mencoba menempatkan ke alamat lokasi 0 dan secara dinamis direlokasi ke lokasi 14000. Pengaksesan ke lokasi logika 346, maka akan dipetakan ke lokasi 14346. Sistem operasi MS-DOS yang masih keluarga intel 80X86 menggunakan empat register relokasi ketika proses *loading* dan *running*.

User program tidak pernah melihat alamat fisik secara real. Program dapat membuat sebuah penunjuk ke lokasi 346, mengirimkan ke memory, memanipulasinya, membandingkan dengan alamat lain, semua menggunakan alamat 346. Hanya ketika digunakan sebagai alamat memory akan direlokasi secara relatif ke register basis.

7.3 SWAPPING

Sebuah proses harus berada di memori untuk dieksekusi. Proses juga dapat ditukar (*swap*) sementara keluar memori ke *backing store* dan kemudian dibawa kembali ke memori untuk melanjutkan eksekusi.

Backing store berupa disk besar dengan kecepatan tinggi yang cukup untuk meletakkan copy dari semua *memory image* untuk semua user, sistem juga harus menyediakan akses langsung ke *memory image* tersebut. Contohnya, sebuah lingkungan *multiprogramming* dengan penjadwalan CPU menggunakan algoritma *round-robin*. Pada saat waktu kuantum berakhir, manajer memori akan memulai untuk menukar proses yang baru selesai keluar dan menukar proses lain ke dalam memori yang dibebaskan (Gambar 7-4). Pada waktu berjalan, penjadwal CPU (*CPU scheduler*) akan mengalokasikan sejumlah waktu untuk proses yang lain di memori. Ketika masing-masing proses menyelesaikan waktu kuantum-nya, akan ditukar dengan proses yang lain.



Gambar 7-4 : Proses swapping

Kebijakan penukaran juga dapat digunakan pada algoritma penjadwalan berbasis prioritas. Jika proses mempunyai prioritas lebih tinggi datang dan meminta layanan, memori akan *swap out* proses dengan prioritas lebih rendah sehingga proses dengan prioritas lebih tinggi dapat *di-load* dan dieksekusi.

Umumnya sebuah proses yang di-*swap out* akan menukar kembali ke ruang memori yang sama dengan sebelumnya. Jika proses pengikatan dilakukan pada saat

load-time, maka proses tidak dapat dipindah ke lokasi yang berbeda. Tetapi, jika pengikatan pada saat *execution-time* , maka kemungkinan proses ditukar ke ruang memori yang berbeda, karena alamat fisik dihitung selama waktu eksekusi.

Bila CPU *scheduler* memutuskan untuk mengeksekusi proses, OS memanggil dispatcher. Dispatcher memeriksa untuk melihat apakah proses selanjutnya pada *ready queue* ada di memori. Jika tidak dan tidak terdapat cukup memori bebas, maka dispatcher *swap out* sebuah proses yang ada di memori dan *swap in* proses tersebut. Kemudian *reload* register ke keadaan normal.

Teknik swapping yang sudah dimodifikasi ditemui pada beberapa sistem misalnya Linux, UNIX dan Windows.

7.4 ALOKASI BERURUTAN

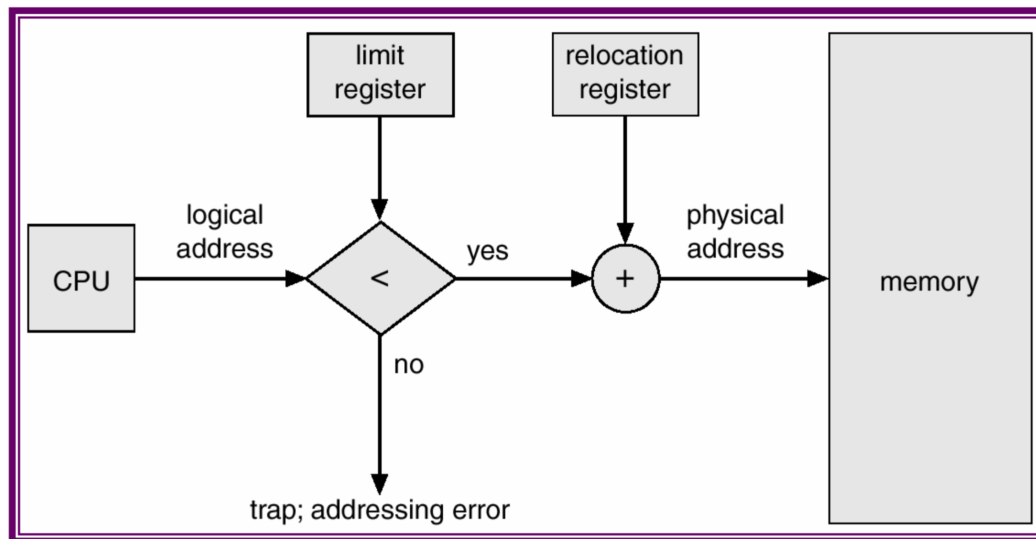
Memori utama biasanya dibagi ke dalam dua partisi yaitu untuk

- Sistem operasi biasanya diletakkan pada alamat memori rendah dengan vektor interupsi
- Proses user yang diletakkan pada alamat memori tinggi.

Alokasi proses user pada memori berupa *single partition allocation* atau *multiple partition allocation*.

7.4.1 Single Partition Allocation

Pada *single partition allocation* diasumsikan sistem operasi ditempatkan di memori rendah dan proses user dieksekusi di memori tinggi. Kode dan data sistem operasi harus diproteksi dari perubahan tak terduga oleh user proses. Proteksi dapat dilakukan dengan menggunakan register relokasi (*relocation register*) dan register limit (*limit register*). Register relokasi berisi nilai dari alamat fisik terkecil sedangkan register limit berisi jangkauan alamat logika dan alamat logika harus lebih kecil dari register limit. MMU memetakan alamat logika secara dinamis dengan menambah nilai pada register relokasi. Gambar 7-5 adalah perangkat keras yang terdiri dari register relokasi dan register limit.



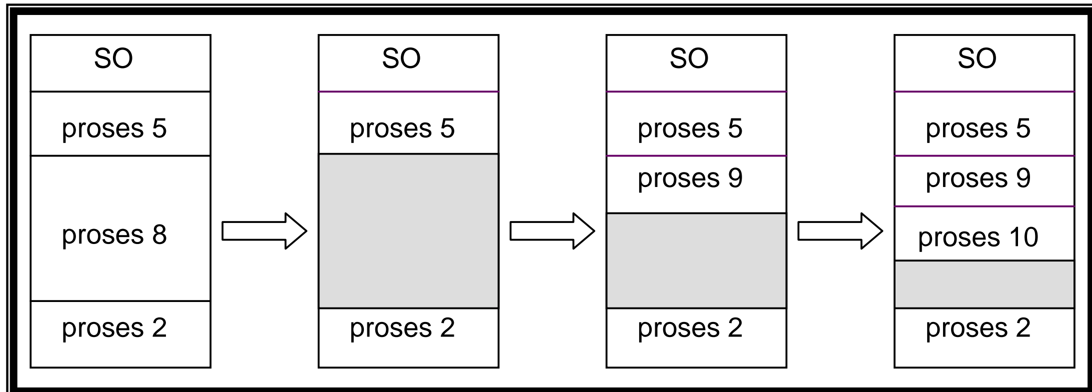
Gambar 7-5 : Perangkat keras untuk register relokasi dan limit

7.4.2 Multiple Partition Allocation

Pada *multiple partition allocation*, mengijinkan memori user dialokasikan untuk proses yang berbeda yang berada di antrian input (*input queue*) yang menunggu dibawa ke memori.

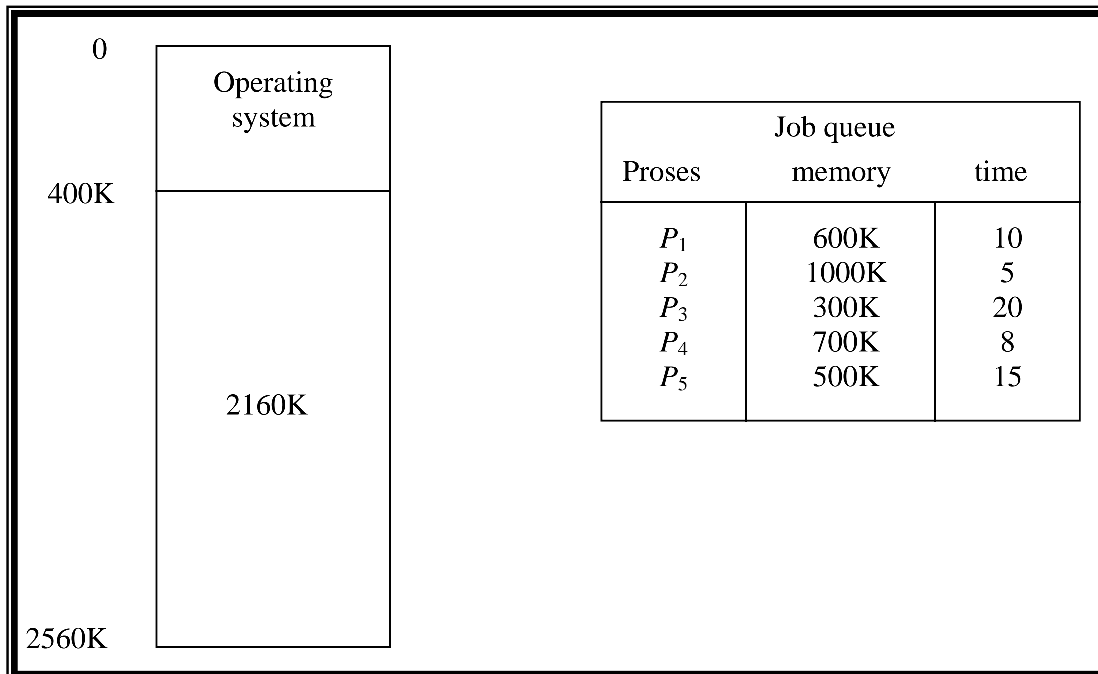
Terdapat dua skema yaitu **partisi tetap (*fixed partition*)** dimana memori dibagi dalam sejumlah partisi tetap dan setiap partisi berisi tepat satu proses. Jumlah partisi terbatas pada tingkat multiprogramming. Digunakan oleh IBM OS/360 yang disebut Multiprogramming with a Fixed number of Task (MFT). Skema yang kedua adalah **partisi dinamis (*variable partition*)** merupakan MFT yang digeneralisasi yang disebut *Multiprogramming with a Variable number of Tasks* (MVT). Skema ini digunakan terutama pada lingkungan batch.

Pada MVT, sistem operasi menyimpan tabel yang berisi bagian memori yang tersedia dan yang digunakan. Mula-mula, semua memori tersedia untuk proses user sebagai satu blok besar (*large hole*). Lubang (*hole*) adalah blok yang tersedia di memori yang mempunyai ukuran berbeda. Bila proses datang dan memerlukan memori, dicari lubang (*hole*) yang cukup untuk proses tersebut seperti Gambar 7-6. Bila ditemukan *memory manager* mengalokasikan sejumlah memori yang dibutuhkan dan menyimpan sisanya untuk permintaan berikutnya. Sistem operasi menyimpan informasi tentang partisi yang dialokasikan dan partisi yang bebas (*hole*).



Gambar 7-6 : Hole untuk proses user

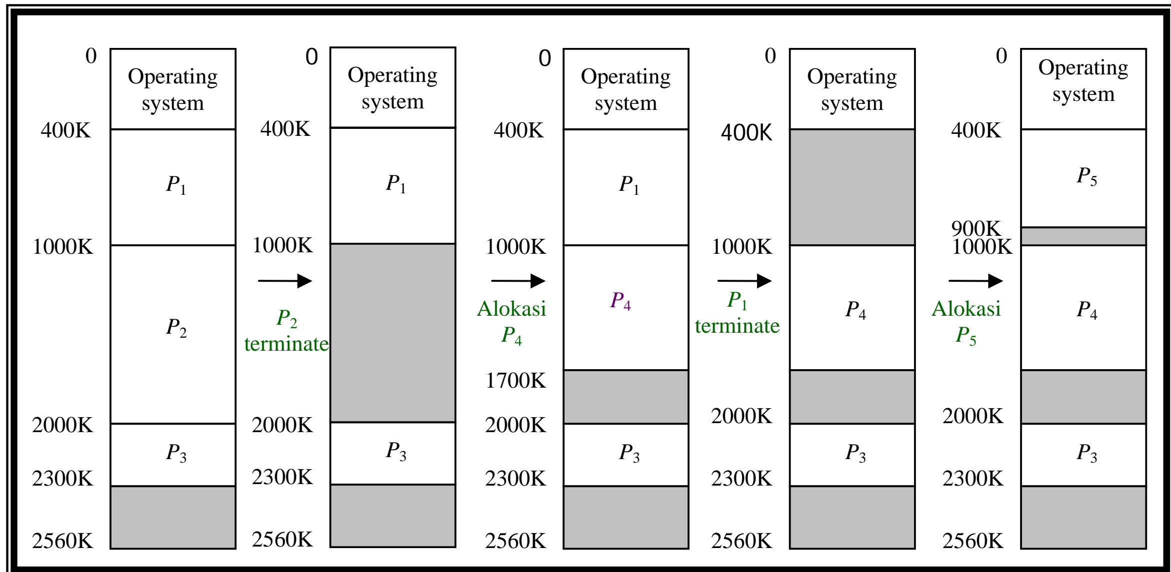
Sebagai ilustrasi, perhatikan contoh berikut pada Gambar 7-7. Diasumsikan tersedia memori 2560K dan untuk OS 400K. Sisa 2160K digunakan untuk user proses. Diasumsikan terdapat 5 job (P_1 s/d P_5) terdapat pada *input queue*. Diasumsikan penjadwalan FCFS digunakan untuk meletakkan job ke memori. Penjadwalan CPU secara *round-robin* (*quantum time* = 1) untuk penjadwalan job yang sudah terdapat di memori.



Job queue		
Proses	memory	time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

Gambar 7-7 : Contoh proses yang akan dilakukan alokasi memori

Hasil alokasi berurutan pada lubang yang cukup untuk proses dapat dilihat pada Gambar 7-8.



Gambar 7-8 : Alokasi memori pada contoh Gambar 7-7

Menggunakan MVT, terdapat beberapa lubang dengan ukuran berbeda. Bila proses datang dan memerlukan memori, dicari dari lubang yang cukup untuk proses. *Dynamic storage-allocation* dapat dilibatkan untuk memenuhi permintaan ukuran *n* dari lubang yang bebas. Strategi yang digunakan meliputi :

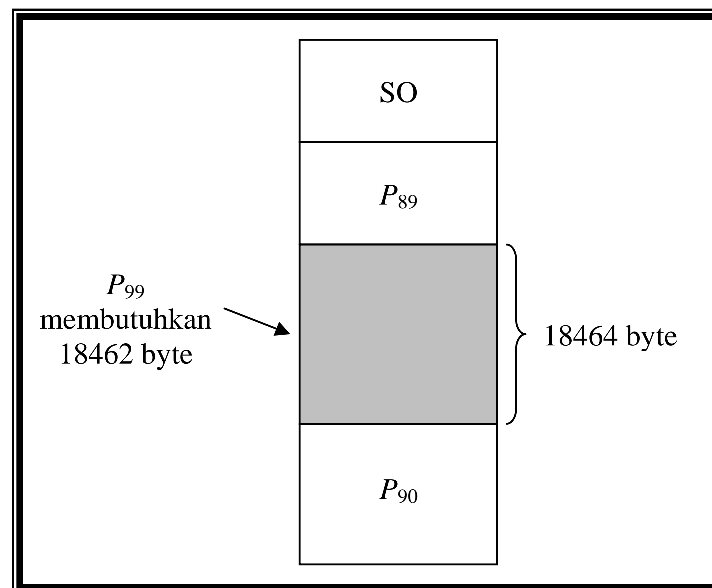
- **First-fit** : alokasi lubang pertama yang cukup untuk proses.
- **Best-fit** : alokasi lubang terkecil yang cukup untuk proses. Strategi ini memerlukan pencarian keseluruhan lubang, kecuali bila ukuran sudah terurut.
- **Worst-fit** : alokasi lubang terbesar yang cukup untuk proses. Strategi ini memerlukan pencarian keseluruhan lubang, kecuali disimpan berdasarkan urutan ukuran.

Diantara algoritma diatas, *first-fit* dan *best-fit* lebih baik dibanidngkan *worst-fit* dalam hal menurunkan waktu dan utilitas penyimpan. Tetapi *first-fit* dan *best-fit* lebih baik dalam hal utilitas penyimpanan tetapi *first-fit* lebih cepat.

7.4.3 Fragmentasi

Fragmentasi Eksternal terjadi pada situasi dimana terdapat cukup ruang memori total untuk memenuhi permintaan, tetapi tidak dapat langsung dialokasikan karena tidak berurutan. Fragmentasi eksternal dilakukan pada algoritma alokasi dinamis, terutama strategi first-fit dan best-fit.

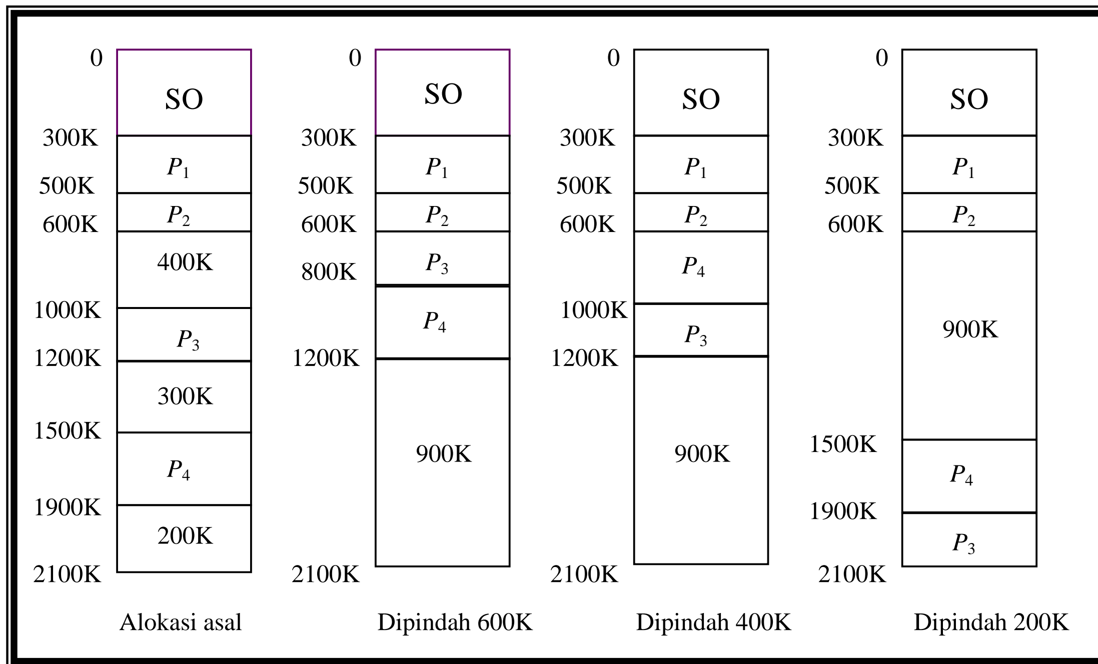
Fragmentasi Internal terjadi pada situasi dimana memori yang dialokasikan lebih besar dari pada memori yang diminta tetapi untuk satu partisi tertentu hanya berukuran kecil sehingga tidak digunakan. Pada *multiple partition,* fragmentasi internal mungkin terjadi pada situasi berikut. Misalnya terdapat lubang 18464 byte, dan proses meminta 18462 byte seperti pada Gambar 7-9. Alokasi dilakukan sesuai permintaan maka sisa lubang 2 byte. Penyimpanan lubang ini akan memerlukan memori lebih besar dari lubang itu sendiri. Pendekatannya adalah dengan mengalokasikan lubang yang sangat kecil sebagai bagian dari permintaan yang besar.



Gambar 7-9 : Fragmentasi internal

Solusi untuk masalah fragmentasi eksternal adalah dengan teknik pemadatan (*compaction*) yaitu memadatkan sejumlah lubang kosong menjadi satu lubang besar sehingga dapat digunakan untuk proses. Pemadatan tidak selalu dapat dipakai. Agar proses dapat dieksekusi pada lokasi baru, semua alamat internal harus direlokasi.

Pemadatan hanya dilakukan pada relokasi dinamis dan dikerjakan pada waktu eksekusi. Karena relokasi membutuhkan pemindahan program dan data dan kemudian mengubah register basis (atau relokasi) yang mencerminkan alamat basis baru. Terdapat beberapa cara pemadatan seperti pada Gambar 7-10.



Gambar 7-10 : Pemadatan

7.5 PAGING

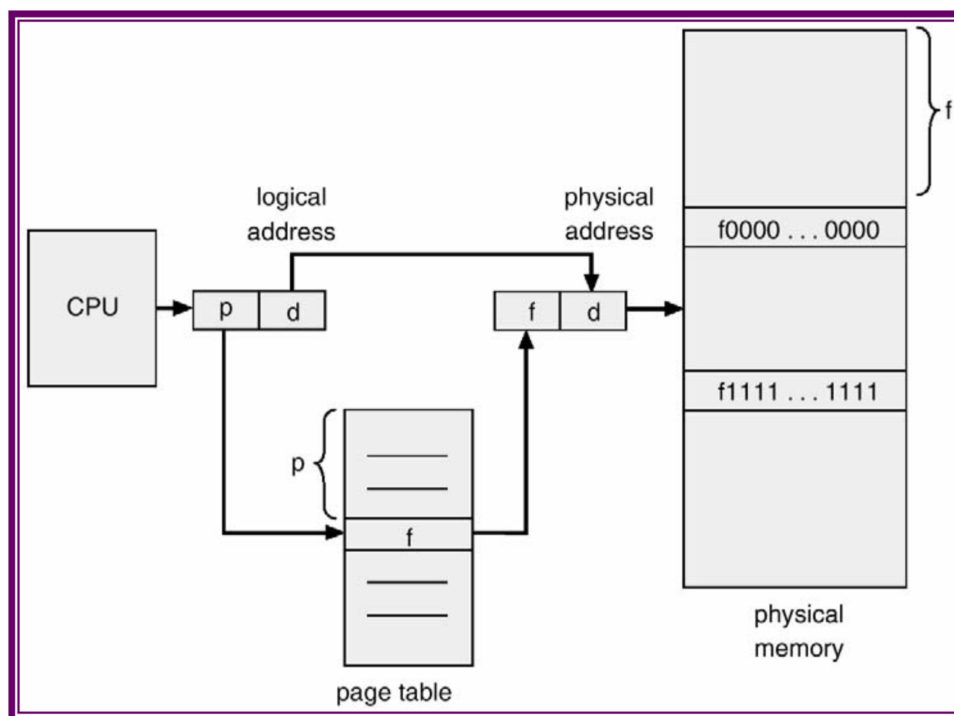
7.5.1 Konsep Dasar Paging

Paging merupakan kemungkinan solusi untuk permasalahan fragmentasi eksternal dimana ruang alamat logika tidak berurutan; mengijinkan sebuah proses dialokasikan pada memori fisik yang terakhir tersedia. Memori fisik dibagi ke dalam blok-blok ukuran tetap yang disebut *frame*. Memori logika juga dibagi ke dalam blok-blok dg ukuran yang sama yang disebut *page*. Semua daftar *frame* yang bebas disimpan. Untuk menjalankan program dengan ukuran *n page*, perlu menemukan *n frame* bebas dan meletakkan program pada *frame* tersebut. Tabel *page* (*page table*) digunakan untuk menterjemahkan alamat logika ke alamat fisik.

Setiap alamat dibangkitkan oleh CPU dengan membagi ke dalam 2 bagian yaitu :

- *Page number* (*p*) digunakan sebagai indeks ke dalam table *page* (*page table*). *Page table* berisi alamat basis dari setiap page pada memori fisik.
- *Page offset (d)* mengkombinasikan alamat basis dengan *page offset* untuk mendefinisikan alamat memori fisik yang dikirim ke unit memori.

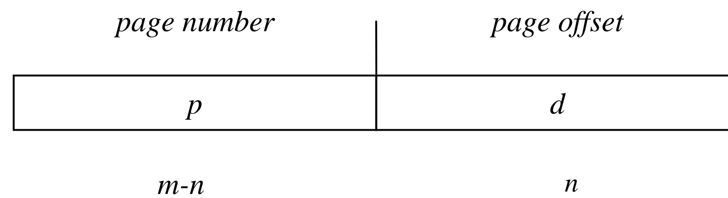
Arsitektur dari sistem paging dapat dilihat pada Gambar 7-11.

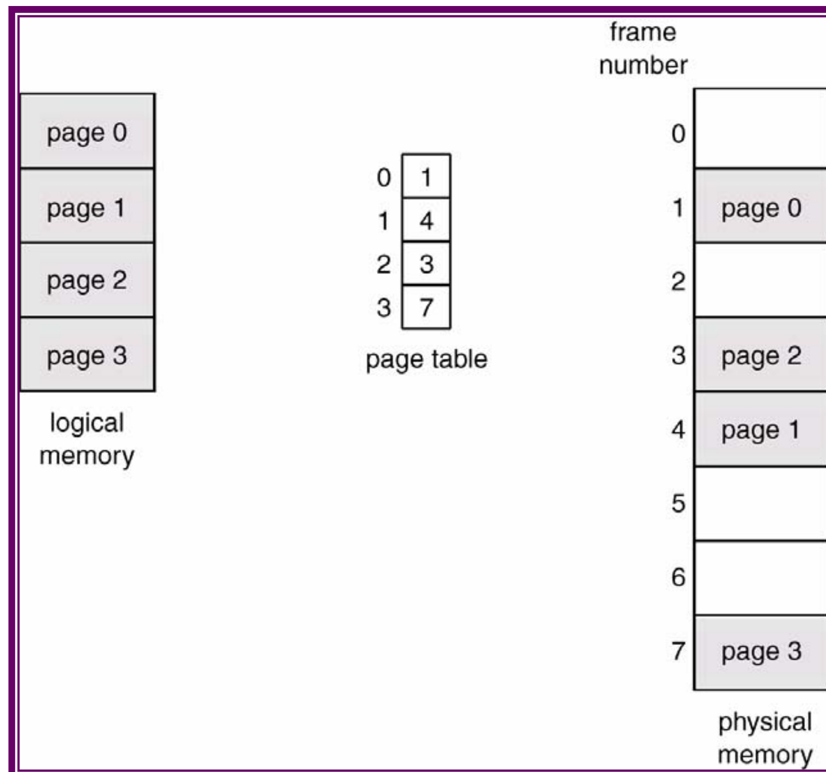


Gambar 7-11 : Arsitektur sistem paging

Model paging dapat dilihat pada Gambar 7-12.

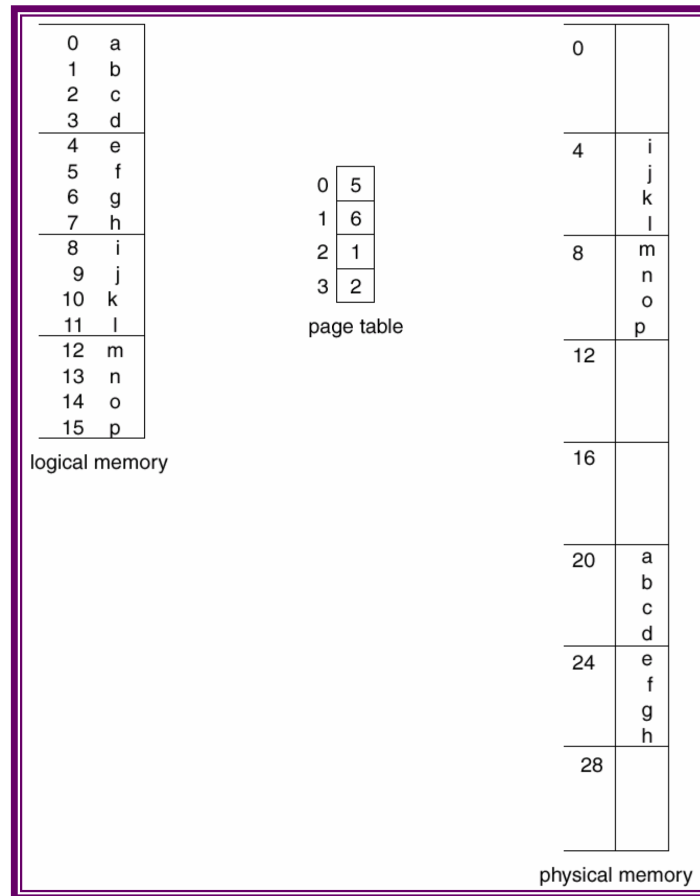
Ukuran *page* (juga *frame*) didefinisikan oleh perangkat keras. Ukuran page merupakan bilangan 2 pangkat *k*, mulai 512 (2^9) s/d 8192 (2^{13}) tergantung arsitektur computer. Bila ukuran ruang alamat logika 2^m dan ukuran satu page adalah 2^n address unit (byte/word) maka (*m-n*) bit orde tinggi dari alamat logika menunjukkan nomor *page* (*page number*) dan *n* bit orde rendah menunjukkan *page offset*.





Gambar 7-12 : Model paging

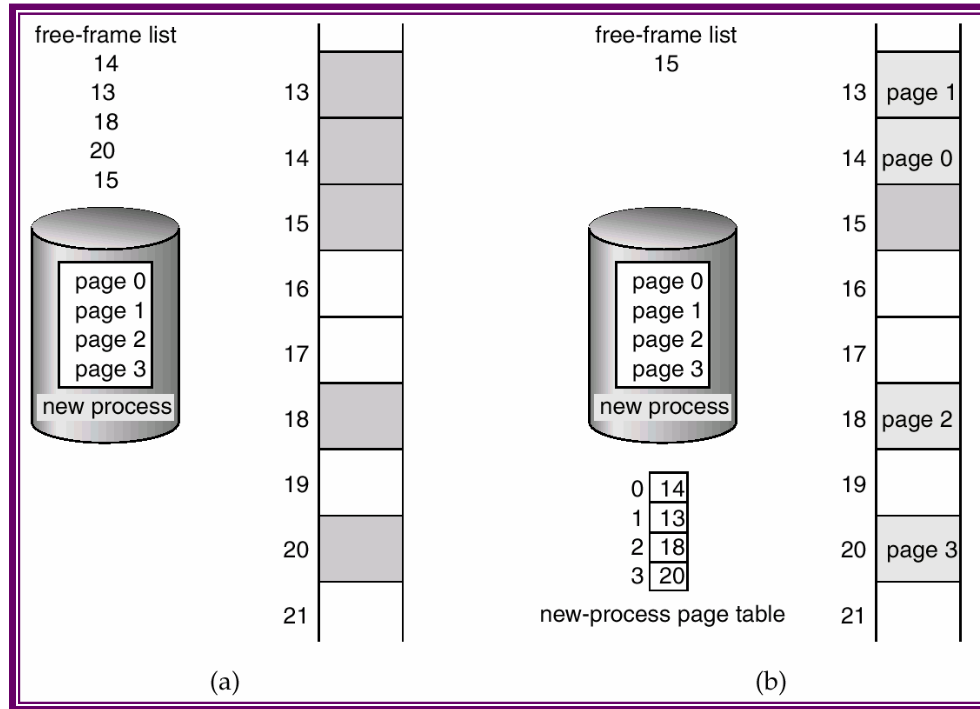
Proses pemetaan dari alamat logika ke alamat fisik yang menggunakan ukuran page 4 byte dan memori fisik 32 byte (8 page) dapat dilihat pada Gambar 7-13. Sebagai contoh alamat logika 2 berada pada *page* 0 akan dipetakan ke *frame* 5, sehingga alamat fisiknya adalah (5 X 4) + 2 = 22. Alamat logika 4 berada pada *page* 1 akan dipetakan ke *frame* 6, sehingga alamat fisiknya adalah (6 X 4) + 0 = 24. Alamat logika 9 berada pada *page* 3 akan dipetakan ke *frame* 1, sehingga alamat fisiknya adalah (1 X 4) + 1 = 5. Alamat logika 15 berada pada *page* 4 akan dipetakan ke *frame* 2, sehingga alamat fisiknya adalah (2 X 4) + 3 = 11.



Gambar 7-13 : Pemetaan alamat logika ke alamat fisik

Pada skema paging, tidak terjadi fragmentasi eksternal, karena “sembarang” *frame* dapat dialokasikan ke proses yang memerlukannya. Tetapi beberapa fragmentasi internal masih mungkin terjadi. Hal ini dikarenakan *frame* dialokasikan sebagai unit dan jika kebutuhan memori dari proses tidak menemukan *page*, maka *frame* terakhir mungkin tidak dialokasikan penuh.

Bila suatu proses datang untuk dieksekusi, maka ukurannya diekspresikan dengan *page*. Setiap *page* membutuhkan satu *frame*. Bila proses membutuhkan *n page*, maka proses tersebut juga membutuhkan *n frame*. Jika tersedia *n frame*, maka memori dialokasikan untuk proses tersebut. Alokasi *page* pada *frame* bebas dapat dilihat pada Gambar 7-14.



Gambar 7-14 : Alokasi frame: (a) Sebelum alokasi (b) Setelah alokasi

Pada *paging*, user memandang memori sebagai bagian terpisah dari memori fisik aktual. Program user memandang memori sebagai satu ruang berurutan yang hanya berisi program user tersebut. Faktanya, program user terpecah pada memori fisik, yang juga terdapat program lain. Karena sistem operasi mengatur memori fisik, perlu diwaspadai lokasi detail dari memori fisik, yaitu *frame* mana yang dialokasikan, *frame* mana yang tersedia, berapa jumlah *frame* dan lain-lain. Informasi tersebut disimpan sebagai struktur data yang disebut "*frame table*".

7.5.2 Implementasi Sistem Paging

Setiap sistem operasi mempunyai metode sendiri untuk menyimpan tabel *page*. Beberapa sistem operasi mengalokasikan sebuah tabel *page* untuk setiap proses. Pointer ke tabel *page* disimpan dengan nilai register lainnya dari PCB.

Pada dasarnya terdapat 3 metode yang berbeda untuk implementasi tabel *page* :

1. Tabel *page* diimplementasikan sebagai kumpulan dari "*dedicated*" register. Register berupa rangkaian logika berkecepatan sangat tinggi untuk efisiensi translasi

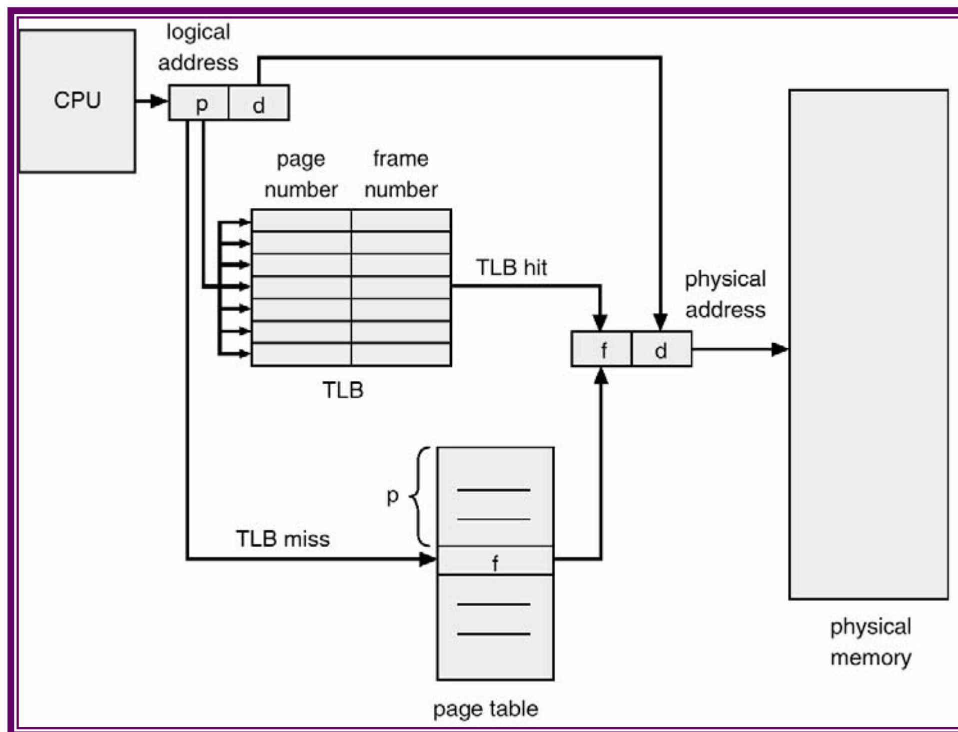
alamat paging. Contoh : DEC PDP-11. Alamat terdiri dari 16 bit dan ukuran page 8K. Sehingga tabel *page* berisi 8 entri yang disimpan pada register. Penggunaan register memenuhi jika tabel *page* kecil (tidak lebih dari 256 entry).

2. Tabel *page* disimpan pada main memori dan menggunakan *page table base registe*" (PTBR) untuk menunjuk ke tabel *page* yang disimpan di main memori. Penggunakan memori untuk mengimplementasikan tabel *page* akan memungkinkan tabel *page* sangat besar (sekitar 1 juta entry). Perubahan tabel *page* hanya mengubah PTBR dan menurunkan waktu *context-switch*. Akan tetapi penggunaan metode ini memperlambat akses memori dengan faktor 2. Hal ini dikarenakan untuk mengakses memori perlu dua langkah : pertama untuk lokasi tabel *page* dan kedua untuk lokasi alamat fisik yang diperlukan.
3. Menggunakan perangkat keras *cache* yang khusus, kecil dan cepat yang disebut *associative register* atau *translation look-aside buffers (TLBs).* Merupakan solusi standar untuk permasalahan penggunaan memori untuk implementasi tabel *page*. Sekumpulan *associative register* berupa memori kecepatan tinggi. Setiap register terdiri dari 2 bagian yaitu *key* dan *value*. Jika *associative register* memberikan item, akan dibandingkan dengan semua key secara simultan. Jika item ditemukan nilai yang berhubungan diberikan. Model ini menawarkan pencarian cepat tetapi perangkat keras masih mahal. Jumlah entry pada TLB bervariasi antara 8 s/d 2048.

Mekanisme penggunaan *associative register* (Gambar 7-15) adalah sebagai berikut :

- *Associative register* berisi hanya beberapa entry tabel *page* (sampai dengan ukuran maksimum).
- Jika memori logika dibangkitkan oleh CPU, nomor *page* berupa sekumpulan *associative register* yang berisi nomor *page* dan nomor *frame* yang berkorespondensi.
- Jika nomor *page* ditemukan pada *associative register,* nomor *frame* segera tersedia dan digunakan untuk mengakses memori.
- Sebaliknya, jika nomor *page* tidak ditemukan pada *associative register,* acuan memori ke tabel *page* harus dibuat.

- Jika nomor *frame* tersedia, maka dapat menggunakannya untuk mengakses ke memori yang tepat.
- Kemudian ditambahkan nomor *page* dan nomor *frame* ke *associative register* sehingga akan mudah ditemukan pada acuan berikutnya.
- Setiap kali tabel *page* baru dipilih, TLB harus dihapus untuk menjamin eksekusi proses berikutnya tidak menggunakan informasi translasi yang salah.



Gambar 7-15 :Perangkat keras paging dengan TLB

Persentasi waktu sebuah *page number* ditemukan pada *associative register* disebut *hit ratio*. Hit ratio 80% berarti penemuan *page number* yang tepat pada *associative register* adalah 80% dari waktu. Misalnya, untuk mencari entry di *associative register* memerlukan waktu 20 ns dan untuk mengakses memori memerlukan waktu 100 ns sehingga untuk memetakan ke memori memerlukan waktu 120 ns. Apabila tidak menemukan *page number* pada *associative register* (20 ns), maka harus lebih dahulu mengakses tabel *page* di memori (100 ns) dan kemudian akses ke lokasi memori yang tepat (100 ns). Maka *effective access time* (EAT) menjadi

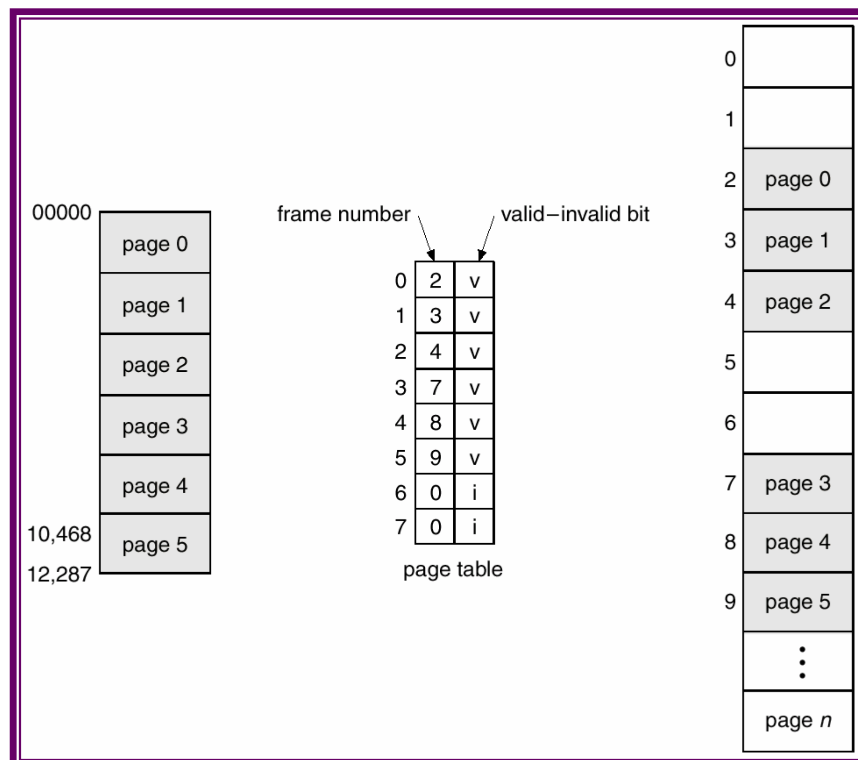
0.8 X 120 + 0.2 X 220 = 140 ns

Artinya terjadi 40% penurunan kecepatan waktu akses memori.

Hit ratio berhubungan dengan jumlah *associative register*. Apabila jumlah *associative register* antara 16 s/d 512*, maka* hit ratio yang dapat dicapai antara 80% sampai 98%. Prosessor Motorola 68030 yang digunakan pada sistem Apple Mac mempunyai TLB 22 entry. CPU Intel 80486 mempunyai 32 register dan hit ratio 98%.

7.5.3 Proteksi

Pada model *page*, proteksi memori menggunakan bit proteksi yang diasosiasikan untuk setiap *frame*. Biasanya bit proteksi disimpan pada tabel *page*. Satu bit mendifinisikan satu *page* untuk "*read and write*" atau "*read-only*". Setiap acuan ke memori melalui tabel *page* untuk menemukan nomor *frame* yang benar. Level proteksi yang lebih baik dapat dicapai dengan menambah jumlah bit yang digunakan.



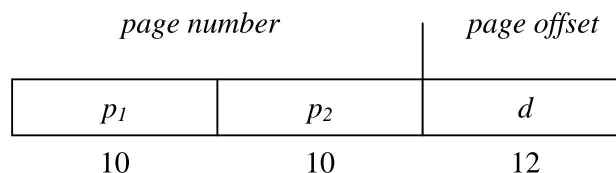
Gambar 7-16 :Valid invalid bit pada tabel page

Pada tabel *page* diberi tambahan "*valid-invalid*" bit seperti pada Gambar 7-16. Nilai "*valid*" mengindikasikan bahwa *page* berada pada ruang alamat logika yang berarti merupakan *page* yang legal (*valid*). Nilai "*invalid*" mengindikasikan bahwa *page* tidak berada pada ruang alamat logika atau page yang illegal (*invalid*). Sistem operasi mengeset bit ini untuk setiap *page* untuk mengijinkan atau tidak mengakses *page*.

7.5.4 Multilevel Paging

Model *multilevel paging* digunakan pada sistem yang mempunyai ruang alamat logika yang sangat besar yaitu antara 2^{32} s/d 2^{64}. Pada sistem ini, tabel *page* akan menjadi sangat besar. Misalnya untuk sistem dengan ruang alamat logika 32 bit dan ukuran page 4K byte, maka tabel *page* berisi 1 juta entry (2^{32} / 2^{12}). Solusinya yaitu dengan melakukan partisi tabel ke beberapa beberapa bagian yang lebih kecil.

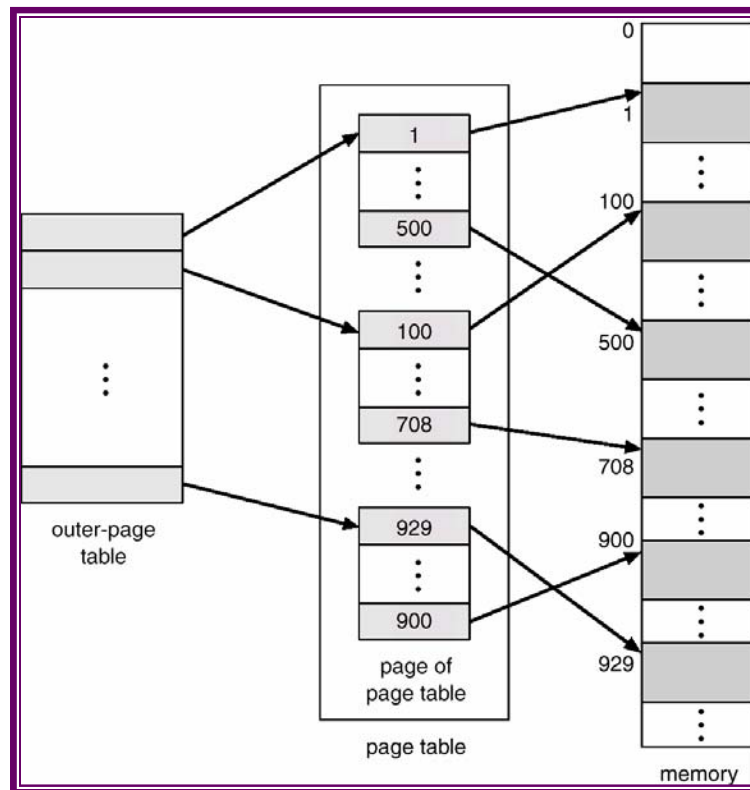
Untuk sistem dengan ruang alamat logika 32 bit dapat dipecahkan menggunakan skema *two level paging*. Pada skema ini alamat logika dibagi menjadi 20 bit untuk nomor *page* dan 12 bit untuk *page offset.* Karena tabel *page* juga merupakan page maka nomor *page* lebih jauh akan dipecah menjadi 10 bit untuk nomor *page* dan 10 bit untuk *page offset*. Maka alamat logika adalah sebagai berikut :



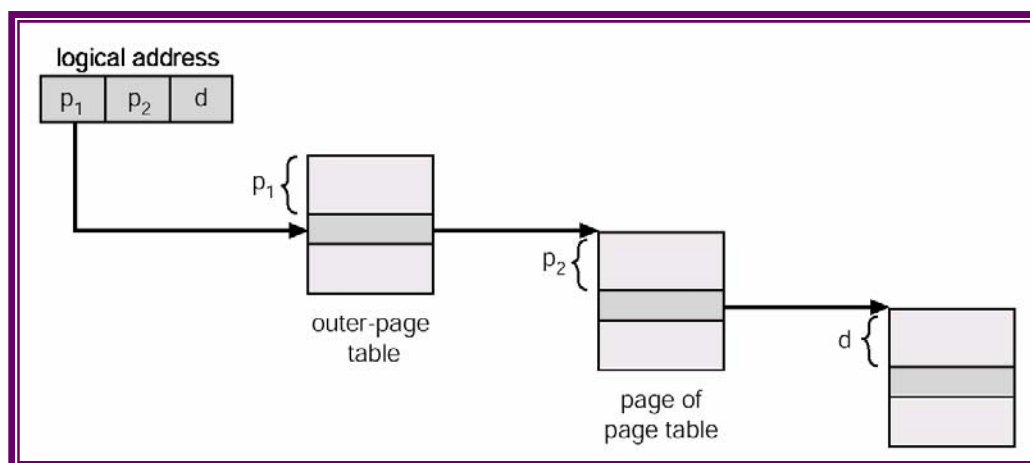
Dimana p_i adalah indeks ke table *page* luar dan p_2 adalah *displacement* dalam *page* pada table *page* luar. Skema tabel *page* pada *two level paging* dapat dilihat pada Gambar 7-17. Sedangkan arsitektur translasi alamat pada *two level paging* untuk mesin 32 bit dapat dilihat pada Gambar 7-18.

Untuk sistem dengan ruang alamat logika 64 bit tidak dapat menggunakan skema *two-level paging*. Solusi yang digunakan adalah dengan membagi tabel *page* luar ke dalam bagian yang lebih kecil : menggunakan skema *three-level* atau *four-level paging*. *Multilevel paging* dapat berakibat pada performansi sistem. Untuk skema *three-level paging*, jika kita menggunakan memori untuk menyimpan tabel, maka akan

membutuhkan 4 kali akses memori. Tetapi jika menggunakan *cache* dengan *hit ratio* 98%, *effective access time* menjadi 0.98 X 120 + 0.02 X 420.



Gambar 7-17 :Skematabel page pada two level paging

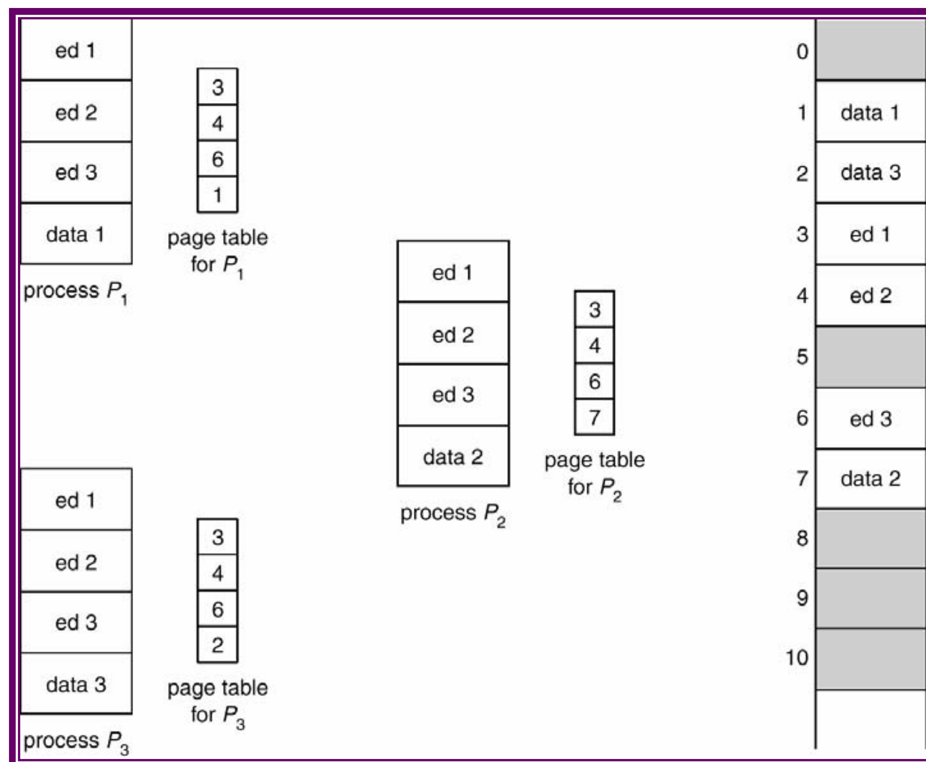


Gambar 7-18 :Skematranslasi alamat pada two level paging

7.5.5 Shared Page

Pada skema *paging*, dimungkinkan untuk sharing kode umum seperti pada Gambar 7-19. Bentuk ini penting terutama pada lingkungan *time sharing*. Satu *copy* kode read-only dibagi ke beberapa proses (misalnya editor teks, compiler dan sistem window). Kode yang dibagi harus berada pada lokasi ruang alamat logika yang sama untuk semua proses.

Kode dan data pribadi (*private*) untuk setiap proses diletakkan terpisah dari kode dan data pribadi proses lain. *Page* untuk kode dan data pribadi dapat diletakkan di sembarang tempat pada ruang alamat logika.



Gambar 7-19 :Shared page

7.6 SEGMENTASI

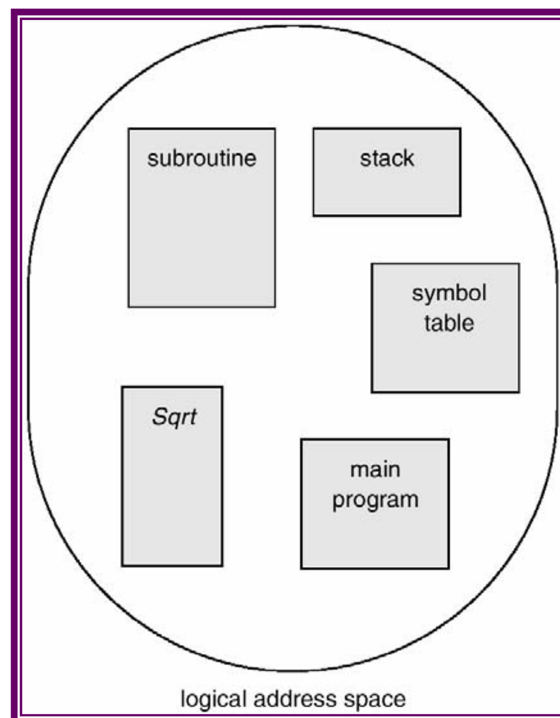
Kerugian utama dari paging adalah terdapat perbedaan antara pandangan user mengenai memori dan memori fisik aktual.

7.6.1. Konsep Dasar Segmentasi

Konsep segmentasi adalah user atau programmer tidak memikirkan sejumlah rutin program yang dipetakan ke main memori sebagai array linier dalam byte tetapi memori dilihat sebagai kumpulan segmen dengan ukuran berbeda-beda, tidak perlu berurutan diantara segment tersebut.

Segmentasi adalah skema manajemen memori yang memungkinkan user untuk melihat memori tersebut. Ruang alamat logika adalah kumpulan segmen. Setiap segmen mempunyai nama dan panjang. Spesifikasi alamat berupa nama segmen dan offset. Segment diberi nomor dan disebut dengan nomor segmen (bukan nama segmen) atau *segment number*. Segmen dibentuk secara otomatis oleh *compiler*.

Sebuah program adalah kumpulan segmen. Suatu segmen adalah unit logika seperti program utama, prosedur, fungsi, metode, obyek, variabel lokal, variabel global, blok umum, stack, tabel simbol, array dan lain-lain. Pandangan user terhadap sistem segmentasi dapat dilihat pada Gambar 7-20.



Gambar 7-20 : Pandangan user pada suatu programi

7.6.2. Arsitektur Segmentasi

Alamat logika terdiri dari dua bagian yaitu nomor segmen (s) dan offset (d) yang dituliskan dengan

<nomor segmen, offset>

Pemetaan alamat logika ke alamat fisik menggunakan tabel segmen (*segment table*), terdiri dari

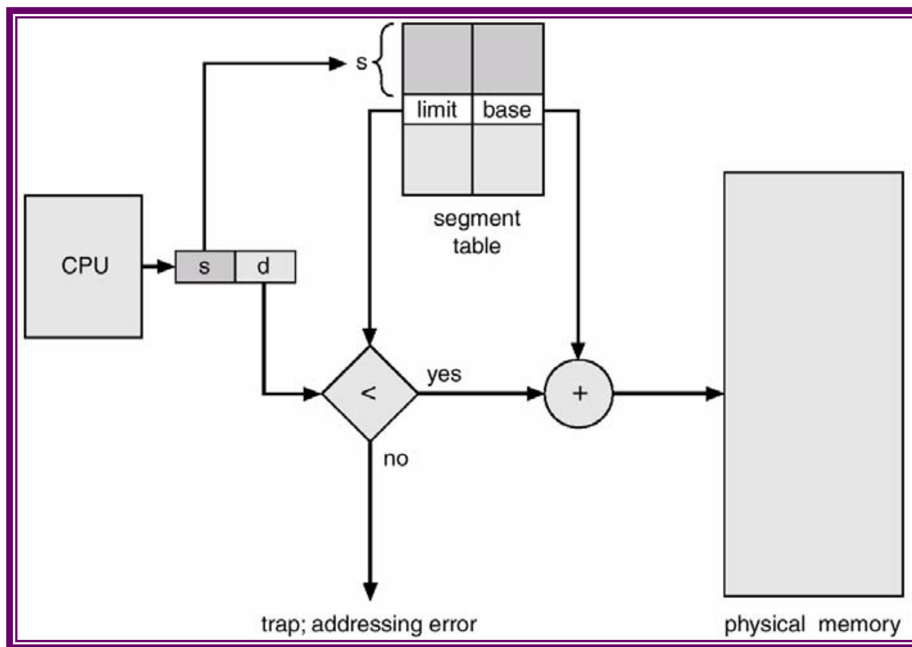
- Segmen basis (*base*) berisi alamat fisik awal
- Segmen limit merupakan panjang segmen

Seperti tabel *page*, tabel segmen dapat berupa register atau memori berkecepatan tinggi. Pada program yang berisi sejumlah segmen yang besar, maka harus menyimpan tabel *page* di memori.

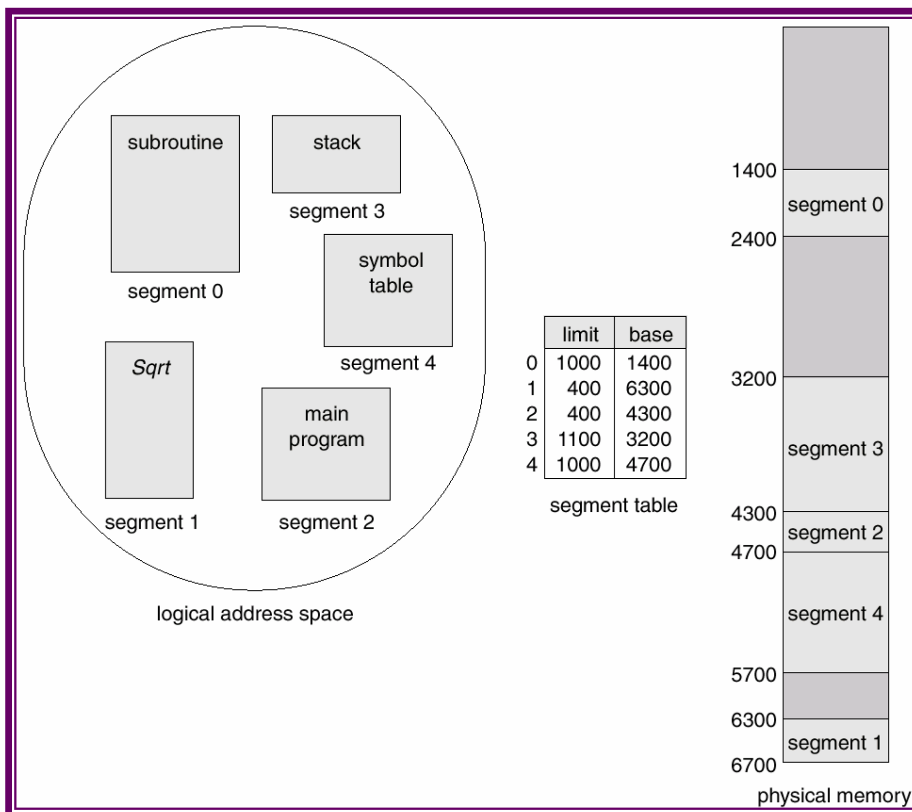
- *Segment-table base register* (STBR) digunakan untuk menyimpan alamat yang menunjuk ke *segment table.*
- *Segment-table length register* (STLR) digunakan untuk menyimpan nilai jumlah segmen yang digunakan program.
- Untuk alamat logika (s, d), pertama diperiksa apakah *segment number* s legal ($s < STLR$), kemudian tambahkan *segment number* ke STBR, alamat hasil ($STBR + s$) ke memori dari *segment table.*

Perangkat keras yang digunakan pada sistem segmentasi dapat dilihat pada Gambar 7-21.

Pemetaan dari alamat logika ke alamat fisik membutuhkan 2 acuan memori untuk setiap alamat logika. Hal ini akan menurunkan kecepatan sistem dengan faktor 2. Solusi standard yang digunakan adalah dengan *cache* (atau *associative register*) untuk menyimpan entri tabel segmen yang sering digunakan. Pemetaan alamat logika ke alamat fisik dengan menggunakan tabel segmen dapat dilihat pada Gambar 7-22. Misalnya alamat logika dengan nomor segment 0 offset 88 akan dipetakan ke alamat fisik 1400 + 88 = 1488 karena offset < limit (88 < 1000). Alamat logika dengan nomor segment 1 offset 412 akan terjadi error karena offset > limit (412 > 400). Alamat logika dengan nomor segment 2 offset 320 akan dipetakan ke alamat fisik 4300 + 320 = 4620 karena offset < limit (320 < 400).



Gambar 7-21 : Implementasi segmentasi

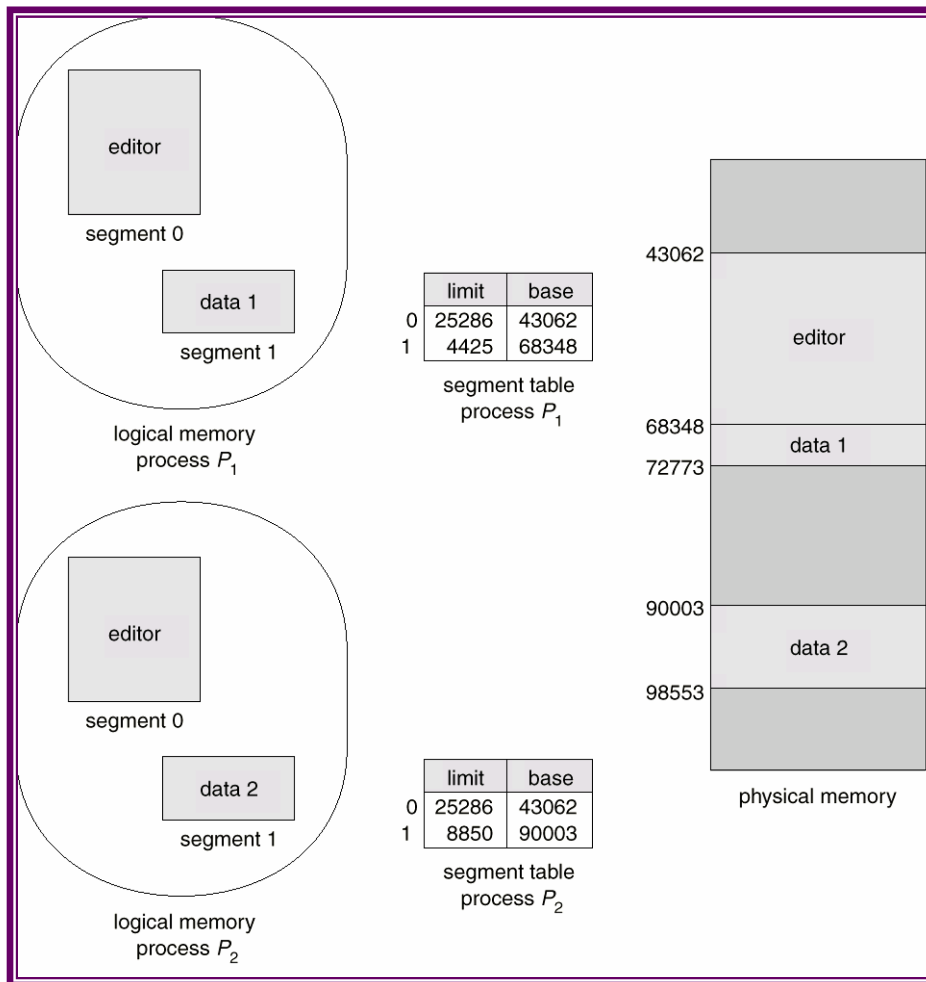


Gambar 7-22 :Contoh Segmentasi

7.6.3. Proteksi dan Sharing

Proteksi bit dapat diletakkan pada tabel segmen. Segmen instruksi dapat diproteksi sebagai segmen *read-only* atau *execute only*, segmen data dapat diproteksi sebagai segmen *read-write*. Pemetaan pada perangkat keras memory akan memeriksa bit proteksi untuk mencegah akses yang illegal.

Dengan segmentasi, juga dimungkinkan membagi kode atau data dengan proses lain. Segmen digunakan bersama-sama bila entry pada tabel segmen dari dua proses berbeda menunjuk ke lokasi fisik yang sama seperti ditunjukkan Gambar 7-23.



Gambar 7-23 :Sharing segmen

7.7 SEGMENTASI DENGAN PAGING

Pada skema ini, skema paging dan segmentasi dikombinasikan. Kombinasi diilustrasikan menggunakan 2 arsitektur berbeda : MULTICS dan Intel 386 (OS/2).

7.7.1 MULTICS

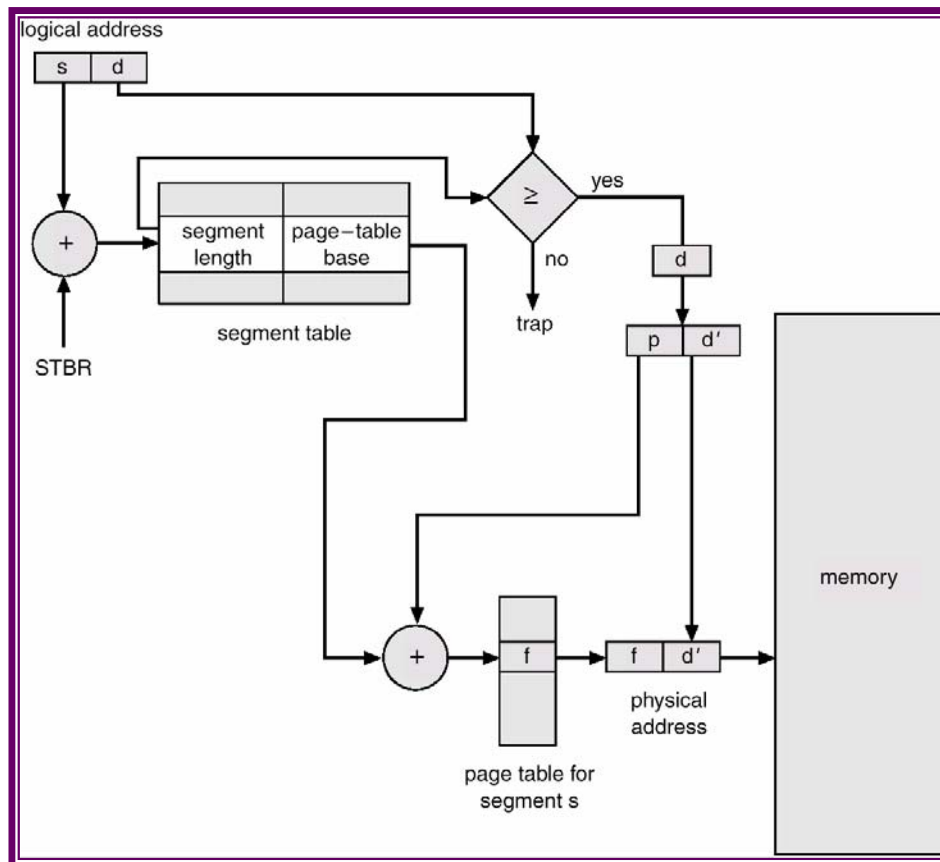
Pada sistem MULTICS, alamat logika dibentuk dari nomor segmen 18 bit dan *offset* 16 bit. Dengan ukuran segmen 64K word, setiap segmen terdiri dari 36 bit, rata-rata ukuran segmen dapat lebih besar dan fragmentasi eksternal menjadi permasalahan karena membutuhkan banyak ruang memori. Tetapi jika fragmentasi eksternal tidak menjadi permasalahan, waktu pencarian untuk mengalokasikan segmen dapat membutuhkan waktu yang lama.

Solusi yang diadopsi pada MULTICS adalah dengan melakukan *paging* pada segmen (*page the segment*). *Paging* menghilangkan fragmentasi eksternal dimana frame kosong dapat digunakan untuk *page* yang tepat. Setiap *page* terdiri dari 1K word. *Segment offset* (16 bit) dibagi ke dalam nomor *page* 6 bit dan *page offset* 10 bit. Nomor *page* mengindeks ke tabel *page* untuk memberikan nomor *framer.* Nomor *frame* dikombinasikan dengan *page offset* menunjuk ke alamat fisik. Skema translasi alamat logika ke alamat fisik pada MULTICS dapat dilihat pada Gambar 7-24.

Nomor *segmen* (18 bit) dibagi ke dalam 8 bit nomor *page* dan 10 bit *page offset,* sehinga tabel *page* terdiri dari 28 entry sehingga alamat logika pada MULTICS adalah sebagai berikut :

s_1	s_2	d_1	d_2
8	10	6	10

Dimana s_1 adalah indeks ke tabel *page* dari tabel segmen dan s_2 adalah *displacement* dalam *page* dari tabel segmen. d_1 adalah *displacement* ke tabel *page* dari segmen yang tepat dan d_2 adalah *displacement* ke alamat yang diakses.



Gambar 7-24 :Skema translasi alamat pada MULTICS

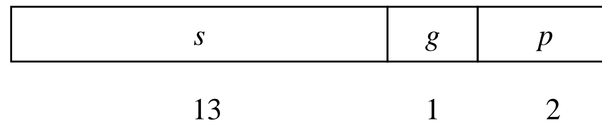
7.7.2 Intel 30386

IBM OS/2 versi 32 bit adalah sistem operasi yang menggunakan arsitektur 30386 (dan 30486). Intel 30386 menggunakan segmentasi dengan paging untuk manajemen memori. Maksimum jumlah segment per proses adalah 16K. Setiap segmen maksimal berukuran 4 gigabytes. Ukuran page adalah 4K byte.

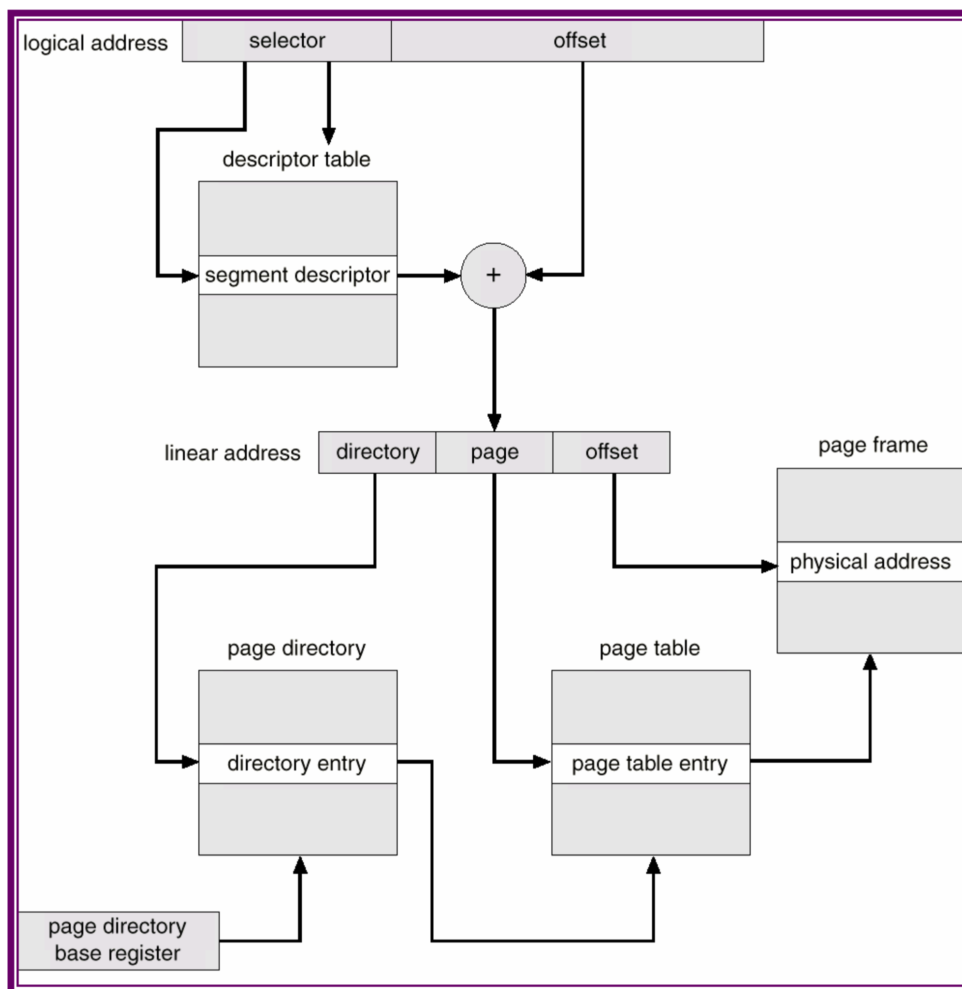
Ruang alamat logika dari suatu proses dibagi ke dalam 2 partisi :

- Partisi 1 terdiri dari 8K segmen yang pribadi (*private*) untuk proses tersebut.
- Partisi 2 terdiri dari 8K segmen yang digunakan bersama untuk semua proses

Informasi mengenai partisi pertama disimpan dalam *local descriptor table* (LDT) sedangkan informasi mengenai partisi kedua disimpan dalam *global descriptor table* (GDT). Setiap entry pada tabel LDT dan GDT terdiri dari 8 byte, dengan informasi detail tentang segmen tertentu termasuk lokasi basis dan panjang segmen. Alamat logika adalah pasangan (*selector, offset*), dimana selector sebanyak 16 bit.



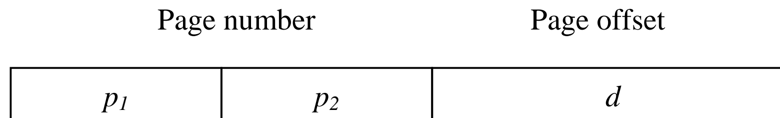
Dimana *s* menyatakan nomor segment, *g* menyatakan apakah segmen merupakan GDT atau LDT dan *p* menyatakan proteksi.



Gambar 7-25 :Skema translasi alamat pada Intel 30386

Karena setiap segmen adalah *page* dengan ukuran 4KB per *page*, sebuah *page table* terdiri dari 1 juta entri. Dan karena setiap entri terdiri dari 4 byte, setiap proses memerlukan 4MB ruang alamat fisik untuk tabel *page* saja. Solusi yang digunakan

menggunakan skema *two-level paging*. Alamat linier dibagi ke dalam nomor *page* 20 bit, dan sebuah *page offset* 12 bit. Nomor *page* dibagi ke dalam 10 bit *page directory pointer* dan 10 bit *page table pointer*. Translasi alamat logika ke alamat fisik pada Intel 30386 lebih jelasnya dapat dilihat pada Gambar 7-25.



RINGKASAN:

LATIHAN SOAL :

1. Terdapat partisi memori 100K, 500K, 200K, 300K dan 600K, bagaimana algoritma First-fit, Best-fit dan Worst-fit menempatkan proses 212K, 417K, 112K dan 426K (berurutan) ? Algoritma mana yang menggunakan memori secara efisien ?
2. Apa yang dimaksud dengan fragmentasi eksternal dan fragmentasi internal ?
3. Diketahui ruang alamat logika dengan 8 page masing-masing 1024 word dipetakan ke memori fisik 32 frame.
4. Berapa bit alamat logika ?
5. Berapa bit alamat fisik ?
6. Diketahui sistem paging dengan page table disimpan di memori
7. Jika acuan ke memori membutuhkan 200 nanosecond, berapa lama waktu melakukan paging ?
8. Jika ditambahkan associative register, dan 75 persen dari semua acuan ke page-table ditemukan dalam associative register, berapa efective access time (EAT) acuan ke memori ? (diasumsikan bahwa menemukan entri pada page table di associative register membutuhkan waktu 0, jika entri ada).