|  |  |  |
| --- | --- | --- |
| **Mata Kuliah** | **:** | **Arsitektur dan Organisasi Komputer** |
| **Bobot Sks** | **:** | **3 sks** |
| **Dosen Pengembang** | **:** | **Catur Nugroho, S.Kom., M.Kom** |
| **Tutor** | **:** |  |
| **Capaian Pembelajaran Mata Kuliah** | **:** | **Mahasiswa mampu menggunakan dan menerapkan konsep & definisi memori komputer** |
| **Kompetentsi Akhir Di Setiap Tahap (Sub-Cpmk)** | **:** | **Mahasiswa mampu menggunakan dan menerapkan konsep & definisi memori komputer** |
| **Minggu Perkuliahan Online Ke-** | **:** | **Sesi 5** |

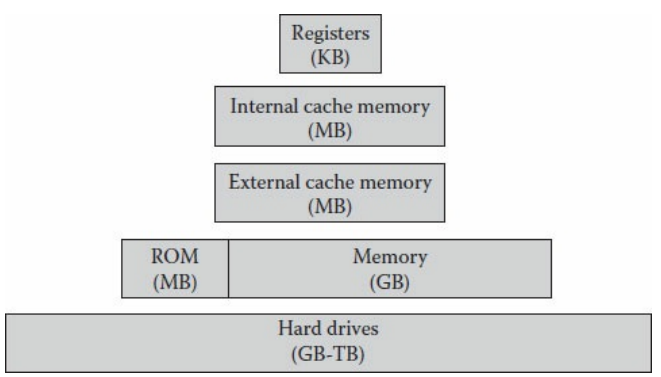
1. Cache Memori :

Modul ini berfokus pada memori cache (*Cache Memori*). Dengan menggunakan gambar arsitektur umum, kita dapat menghubungkan memori cache (*Cache Memori*). dan kontribusinya terhadap kinerja sistem (Gambar 5.1).



Gambar 5.1 Cache memory.

Seperti yang dinyatakan dalam bab sebelumnya, memori cache adalah lapisan penting dalam hierarki memori, dan kontribusi utamanya adalah dalam meningkatkan kecepatan eksekusi. Hirarki memori digambarkan sekali lagi pada Gambar 5.2, tetapi kali ini penekanannya adalah pada ukuran berbagai tingkat hierarki. Level paling lambat dan terbesar (sejauh menyangkut kapasitas) adalah disk. Saat ini, disk standar yang digunakan di komputer pribadi (PC) memiliki kapasitas mulai dari beberapa ratus gigabyte hingga beberapa terabyte. Selain itu, dengan memanfaatkan komputasi awan (*cloud computing*), di mana sumber daya sistem berada di server jarak jauh, kapasitas disk meningkat secara signifikan.



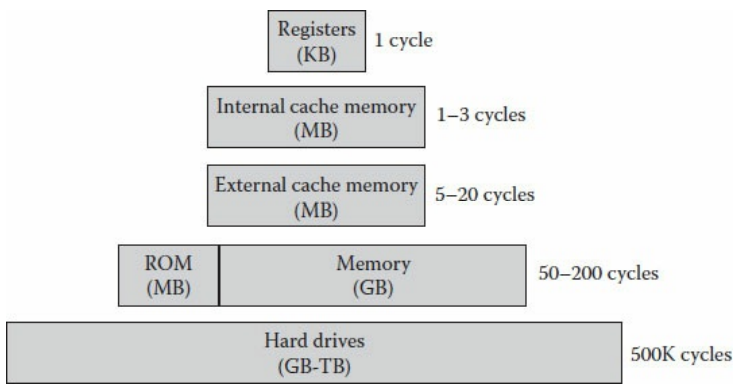
Gambar 5.2, Memory hierarchy—size

Memori utama (*Random Access Memory* [RAM]), yang mewakili level kedua, memiliki kapasitas standar mulai dari beberapa gigabyte hingga ratusan gigabyte. Memori cache, yang selanjutnya tingkat, biasanya dibagi menjadi beberapa komponen, masing-masing dengan tujuan yang berbeda dan ukuran yang berbeda, tingkat terakhir dari hirarki memori adalah register yang biasanya sangat terbatas. RAM yang dijelaskan pada bab sebelumnya digunakan untuk menyimpan program dan data. Ada komponen memori lain yang disebut read-only memory (ROM), yang digunakan oleh sistem operasi dan perangkat keras dan ditujukan untuk komponen (program dan/atau data) yang tidak sering berubah. Terlepas dari namanya, beberapa ROM yang tersedia saat ini dapat diubah; terkadang, diperlukan alat perekam khusus.

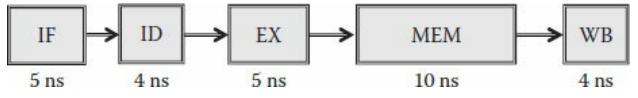
Meski begitu, penggunaan utamanya tetap untuk sistem operasi khusus atau fungsi perangkat keras. Dengan demikian, ROM tidak tersedia untuk program komputer standar. Salah satu atribut ROM adalah memori yang tidak mudah menguap, yang berarti ia mempertahankan isinya bahkan jika daya dimatikan. Untuk alasan itu, ROM digunakan oleh program boot yang untuk menghidupkan sistem.

Komponen lain yang disimpan dalam ROM adalah program/ data yang diperlukan untuk mengelola beberapa perangkat I/O. Biasanya, jenis data ini tidak akan diubah oleh perangkat. Di komputer modern, beberapa ROM digantikan oleh memori flash, yang merupakan perangkat ***non-volatil*** yang dapat ditulis ulang jika diperlukan.

Gambar 5.3 menggambarkan hierarki memori tetapi mencakup *access times* rata-rata tambahan dari setiap level. Dalam berbagai sistem, jumlah siklus dapat bervariasi; namun, itu masih memberikan dasar yang baik untuk perbandingan. Dengan mengamati gambar, perbedaan yang sangat besar antara *access times* berbagai level menjadi jelas dan realistis. Hanya dengan menyadari perbedaan ini, seseorang dapat memahami kontribusi **memori cache** terhadap kinerja sistem.



Gambar 5.3 *Memory hierarchy*—*access time*.

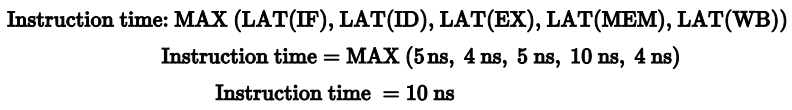


Gambar 5.4 *Pipeline execution times*.

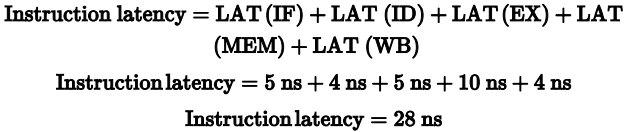
* *Instruction fetch* (IF) *requires 5 ns*
* *Instruction decode* (ID) *requires* 4 ns
* *Execute* (EX) *requires* 5 ns
* *Memory access* (MEM) *requires* 10 ns
* *Write back* (WB) *requires* 4 ns

Gambar 5.4 menggambarkan waktu eksekusi dengan skala dari berbagai tahapan. Mulailah dengan menghitung kinerja pipeline dan latensi\* dari satu instruksi. Kinerja ini penting untuk menilai efisiensi pipa. Namun, waktu yang diperlukan untuk mengeksekusi satu instruksi tidak dapat menjadi indikasi efisiensi, dan kinerja sistem dihitung dengan jumlah instruksi yang dilakukan selama satu unit waktu.

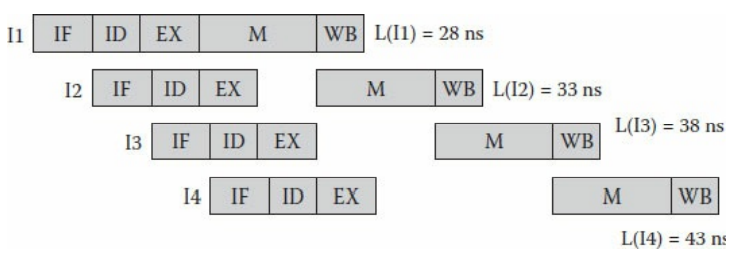
Secara teoritis, menghitung kinerjanya sederhana. Dengan asumsi pipa bekerja tanpa bahaya, kita harus mencari tahu jumlah waktu maksimum yang diperlukan untuk satu tahap. Karena mekanisme pipeline, tahapan yang lebih panjang mendominasi waktu eksekusi karena tahapan yang lebih pendek harus menunggu. Karena dalam eksekusi pipelined, prosesor menyelesaikan instruksi lain dengan setiap siklus, waktu siklus sistem menunjukkan waktu eksekusi instruksi. Waktu yang dibutuhkan untuk satu instruksi sebagai bagian dari eksekusi pipelined diberikan oleh:



Arti dari instruksi yang dieksekusi setiap 10 ns adalah bahwa sistem mampu mengeksekusi 108 instruksi per detik, dengan asumsi, tentu saja, bahwa pipa bekerja tanpa penundaan atau bahaya. Latensi instruksi dihitung dengan cara yang sama:



Perhitungan menunjukkan bahwa jumlah waktu yang dibutuhkan untuk mengeksekusi satu instruksi, terlepas dari pipa, adalah 28 ns. Sayangnya, hasil ini hanya berlaku untuk eksekusi terisolasi dari satu instruksi saja. Tidak hanya ini situasi yang tidak mungkin, karena kita tidak pernah mengeksekusi hanya satu instruksi, tetapi juga hasil yang diperoleh tidak berharga. Dalam hal ini, karena pipa yang tidak seimbang, ada bahaya yang diabaikan oleh perhitungan sebelumnya. Bahaya ini terkait dengan akses memori.Saat membangun urutan eksekusi dengan semua tahapannya, menjadi jelas bahwa bahaya sangat mempengaruhi kinerja sistem (Gambar 5.5). Meskipun instruksi pertama dieksekusi dalam 28 ns, ini tidak terjadi dengan instruksi berikut. Instruksi kedua harus menunggu selama tahap eksekusi keempat, jadi alih-alih 28 ns akan membutuhkan 33 ns. Bahaya memori berlanjut dan meningkat dengan setiap instruksi tambahan dieksekusi. Setiap instruksi tambahan tersebut meningkatkan jumlah waktu yang dibutuhkan oleh 5 ns. Ini berarti bahwa instruksi ketiga membutuhkan 38-ns, instruksi keempat membutuhkan 43 ns, dan seterusnya.



Gambar 5.5 table eksekusi.

1. **ORGANISASI MEMORY**

Secara teknis, memori adalah perangkat digital yang digunakan untuk menyimpan data. Berdasarkan definisi umum ini, hard drive, CD-ROM, kunci disk-on, dan sebagainya adalah berbagai jenis perangkat memori. Dalam bab ini, penekanannya adalah pada memori komputer (***Random Access Memory*** [RAM]), yang tidak seperti kebanyakan perangkat lain, hanya digunakan untuk penyimpanan data sementara. Setelah memutar sistem mati dan hidup lagi, atau hanya me-reboot (me-restart) komputer, semua data yang tersimpan di memori hilang. Oleh karena itu, memori disebut sebagai ***volatile*** berbeda dengan perangkat penyimpanan lain yang mempertahankan data bahkan setelah listrik dimatikan, yang disebut ***non-volatile.***

Memori komputer dapat digambarkan sebagai matriks sel satu dimensi yang digunakan untuk menyimpan dan mengambil data. Setiap sel memiliki alamat, yang menyediakan mekanisme unik untuk mengakses data yang disimpannya. Seperti halnya matriks, alamat digunakan sebagai indeks ke sel. Dalam berbagai implementasi komputer, sel memori mungkin memiliki ukuran yang berbeda. Terkadang, itu adalah satu byte (8-bit), dan lain kali mungkin kata dari berbagai ukuran. Jika ukuran sel adalah satu byte, memori digambarkan sebagai byte yang dapat dialamatkan, atau setiap alamat mengacu pada 1 byte. Ini juga merupakan informasi minimal yang harus

diakses (dibawa dari memori atau dikirim ke memori).

Sel adalah unit atom, jadi jika ukurannya 1-byte, semua akses memori akan menjadi kelipatan 1-byte. Di sisi lain, jika ukurannya adalah satu kata, semua akses memori akan menjadi kelipatan satu kata. Ketika ada kebutuhan untuk mengakses bagian byte yang lebih kecil, seluruh byte akan dibawa dari memori dan bagian yang tidak relevan akan disembunyikan.

Pertimbangkan contoh byte yang digunakan untuk menampung delapan sakelar biner yang berbeda. Ketika diperlukan untuk memeriksa, misalnya, bit nomor lima, seluruh byte akan dimuat dari memori, dan CPU akan menggunakan instruksi AND\* dengan konstanta 0000 01002. Jika hasilnya nol, berarti bit nomor lima adalah nol atau "***False***"; jika tidak, itu akan menjadi "***True***."

Komputer generasi pertama menyebut memori sebagai RAM untuk membedakannya dari perangkat serial seperti ***pita magnetik***. Nama tersebut dipilih untuk menggambarkan fitur penting dalam mengakses sel secara langsung tanpa perlu membaca semua sel sebelumnya. Akses langsung ini menyiratkan bahwa waktu akses konstan dan tidak bergantung pada lokasi sel. Meskipun semua memori menyediakan akses langsung, nama RAM masih digunakan beberapa dekade setelah diciptakan.

Gambar 4.3 memberikan representasi visual dari organisasi memori (*partial view*). Dalam sistem seperti PC, di mana memori dapat dialamatkan byte, register biasanya lebih besar dan didasarkan pada kata-kata. Perlu dicatat, bagaimanapun, bahwa istilah kata memiliki arti yang berbeda, dan berbagai sistem menggunakan ukuran yang berbeda untuk kata-kata mereka.

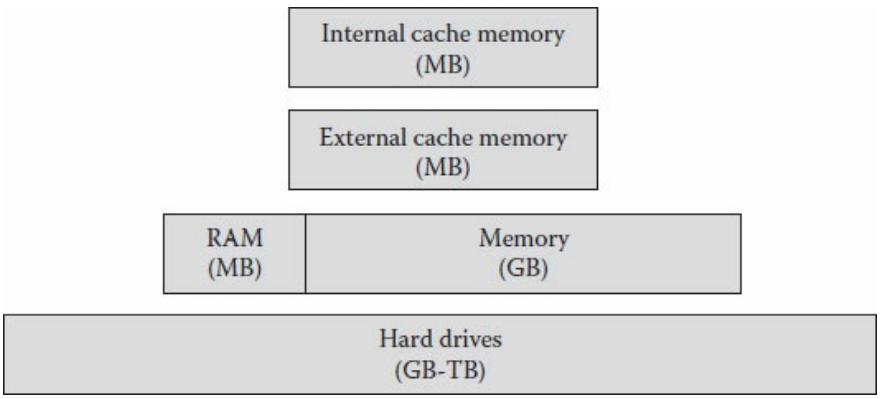


Gambar 4.3 Memory organization

Dalam beberapa sistem, mungkin mewakili 16 bit; di lainnya, seperti arsitektur berbasis x86, 32 bit; dan di PC modern, ini adalah 64 bit. Register yang lebih besar menyiratkan bahwa register mencakup beberapa byte. Namun, terlepas dari ukuran register atau jumlah byte yang dikandungnya, memori masih dapat dialamatkan byte.

Salah satu register penting untuk memahami operasi memori adalah ***Memory Address Register*** (MAR), yang menyimpan alamat byte yang akan dibaca atau ditulis ke memori. Seperti register lain dalam sistem berbasis x86, register ini adalah 32-bit register. Ini berarti bahwa alamat terbesar yang didukung adalah 2 32 – 1 (atau 4 GB memori). Selama tahun 1990-an, jelas bahwa, karena keterbatasan ukuran kata harus ditingkatkan menjadi 64-bit *words;.* Sistem lain yang menggunakan kata 64-bit; namun, untuk pasar PC dengan sejumlah besar paket perangkat lunak, perubahannya jauh lebih lambat, dan 64-bit *version* baru mulai muncul selama dekade pertama abad 21. *Representasi Data* ukuran kata yang lebih besar memberikan tingkat akurasi yang lebih tinggi saat perhitungan aritmatika. Namun demikian, kontributor utama untuk beralih ke arsitektur 64-bit adalah keterbatasan memori yang dikenakan oleh arsitektur 32-bit. Selama tahun 1990-an, menjadi jelas bahwa kecuali memori maksimum yang didukung ditingkatkan, platform PC tidak akan dapat berkembang lebih jauh.

Ada banyak jenis memori, misalnya, ***volatil*** dan ***non-volatil***, tetapi ada perbedaan penting tambahan yang didasarkan pada kecepatan. Pengembang perangkat lunak melihat memori sebagai bagian dari platform perangkat keras dan karena itu lebih memilih untuk mendapatkan memori sebanyak mungkin serta memori tercepat yang tersedia. Ini, tentu saja, mungkin, tetapi terkait dengan biaya tinggi. Untuk alasan itu, sebagian besar sistem komputer menerapkan hierarki memori. Memori yang dekat dengan prosesor lebih cepat dan lebih mahal, dan karena alasan itu, kuantitasnya lebih kecil. Karena semakin jauh dari prosesor, itu menjadi lebih lambat, lebih murah, dan lebih besar (Gambar 4.4).



Gambar 4.4 Memory hierarchy.

Selain register—yang merupakan jenis memori sementara, terbatas, dan sangat cepat yang berada di dalam prosesor—dalam banyak kasus, sistem menyertakan beberapa ***cache memory*** yang meningkatkan kecepatan akses hingga data dan instruksi yang digunakan. Di sisi lain, disk kadang-kadang dianggap sebagai jenis memori tambahan karena pada saat dijalankan, hanya bagian dari program yang dieksekusi yang berada di memori dan bagian lain yang tidak digunakan yang disimpan di disk. Dengan demikian, disk bertindak sebagai perpanjangan memori.

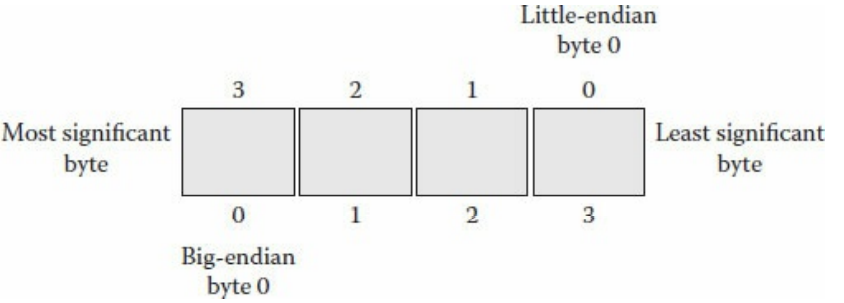
Memori utama (*Main Memory* = RAM) biasanya diimplementasikan menggunakan teknologi ***Dynamic*** RAM (DRAM), yang ditandai dengan kebutuhan untuk me-refresh data setiap beberapa milidetik. Jenis memori ini hanya membutuhkan satu transistor\* dan satu kapasitor untuk setiap bit. Fakta bahwa komponen-komponen ini sangat kecil, dan seiring dengan kemajuan teknologi, mereka menjadi lebih kecil, memberikan kemampuan merancang sistem dengan miliaran byte, di sisi lain *Cache Memory*, diimplementasikan menggunakan teknologi yang lebih cepat (static RAM [SRAM]), yang tidak harus di-refresh tetapi membutuhkan antara empat dan enam transistor untuk setiap bit. Inilah alasan mengapa *Cache Memory* juga lebih mahal dibandingkan dengan memori standar.

Skematis akses memori dalam byte-*addressable* PC menggunakan 2 register internal :

* ***Memory Address Register*** (MAR), digunakan untuk menyimpan alamat memori yang relevan untuk membaca dari atau tulis ke dalam
* ***Memory Data Register*** (MDR), digunakan untuk menyimpan data yang akan ditulis ke dalam memori atau datamembaca dari memori

Versi sederhana dari akses memori dapat dijelaskan dengan langkah-langkah berikut. Jika prosesor perlu membaca data dari alamat 1000, prosesor harus memasukkan nilai 1000 ke dalam MAR dan mengeluarkan perintah baca. Kontroler Memori akan mengambil konten dan menempatkannya di MDR, di mana akan tersedia untuk prosesor. Di sisi lain, jika prosesor perlu menyimpan data dalam memori, misalnya, karena instruksi dalam program yang sedang dieksekusi, prosesor menempatkan data di MDR, menempatkan alamat yang sesuai di MAR, dan mengeluarkan menulis perintah. Ini adalah tugas pengontrol memori untuk benar-benar melakukan operasi yang diperlukan.

Prosedur baca dan tulis memori yang dijelaskan sebelumnya pada awalnya ditujukan untuk satu byte. Memori dapat dialamatkan byte dan register berukuran byte. Ketika ukuran *word* meningkat, misalnya menjadi 16 bit (2-byte), pengontrol memori harus membaca / menulis 2 byte. Ini diterapkan dengan cara yang berbeda oleh berbagai produsen perangkat keras. PC, misalnya, menggunakan mekanisme di mana byte orde rendah disimpan di lokasi paling kiri dan byte orde tinggi disimpan di lokasi byte berikutnya. Untuk alasan itu, kata 16-bit ditulis ke dalam memori dalam urutan terbalik. Misalnya, kata 16-bit yang berisi karakter "AB" ditulis dalam memori sebagai "BA," byte pertama adalah karakter paling kiri, dan yang kedua adalah karakter paling kanan. Karena alasan kompatibilitas, hal ini juga berlaku ketika kata bertambah ukurannya, word 32-bit berisi "ABCD" ditulis ke dalam memori sebagai "DCBA."



Gambar 4.5 *Big- dan little-endian.*

Metode ini, yang disebut ***little-endian***, yang diterapkan terutama oleh berbagai PC, sementara banyak komputer lain menggunakan metode ***big-endian***, yang lebih intuitif dan dapat dimengerti

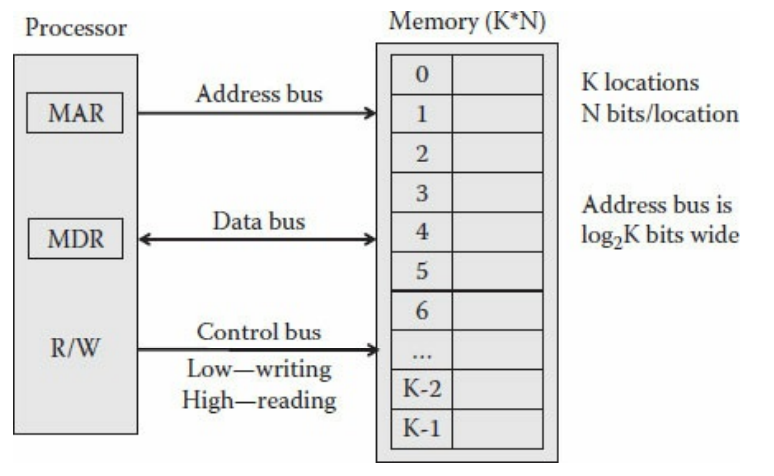
1. **Penjelasan**

Dengan cara “standar”, kata 32-bit yang direpresentasikan pada Gambar 5.6 ditulis dari kiri ke kanan (big-endian), yang berarti byte paling signifikan ada di sebelah kiri dan byte paling tidak signifikan ada di sebelah kanan. Dengan metode little-endian, byte yang paling signifikan ada di sebelah kanan dan Metode ***little-endian*** tetap ada karena kompatibilitas yang dipertahankan dalam arsitektur PC. Awalnya, ketika ukuran kata meningkat dari 1 byte menjadi 2 byte, pengontrol memori harus

mentransfer 2 byte satu persatu. penerapan metode *little-endian* lebih sederhana, dan tetap, meskipun pengontrol memori mentransfer seluruh kata pada suatu waktu.

Karena penyebaran PC yang luas dan fakta bahwa mereka menggunakan metode little-endian, beberapa vendor menerapkan metode ganda (***bi-endian***). Ini berarti bahwa sistem mendukung kedua metode (***big*** dan ***little-endian***), dan dimungkinkan untuk beralih dari satu metode ke metode lainnya saat boot. Nanti versi menyediakan mekanisme berbasis perangkat lunak untuk beralih di antara metode. Untuk keakuratan deskripsi ini, perlu dicatat bahwa ada metode lain yang mengintegrasikan kedua metode tersebut. Komputer PDP, dirancang oleh DEC, menggunakan kata 32-bit. Sebuah kata yang berisi "ABCD" ditulis dalam memori sebagai "BADC." Ini berarti bahwa kata itu dibagi menjadi dua bagian. Bagian-bagiannya ditulis secara berurutan (big-endian); namun, byte di setiap setengahnya adalah ditulis menggunakan little-endian. Oleh karena itu, metode ini disebut mixed-endian atau middle-endian.

Terlepas dari popularitas metode little-endian di antara PC, perlu dicatat bahwa sebagian besar jaringan menggunakan metode big-endian. Sistem dan insinyur perangkat lunak tidak perlu repot dengan data migrasi dari satu metode ke metode lainnya karena ini dilakukan secara otomatis oleh perangkat keras.



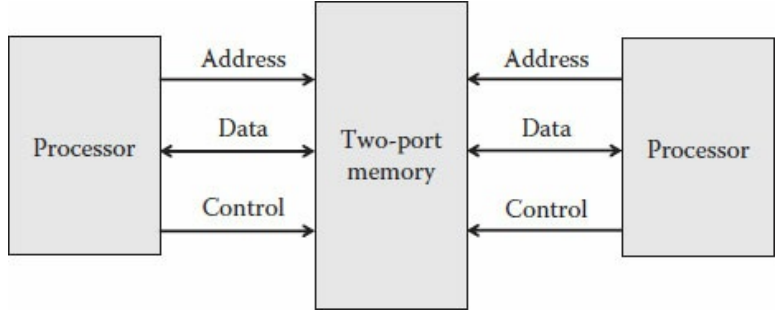
Gambar 4.6 Bus memori.

Dalam arsitektur berbasis alamat byte, memori diatur dalam byte, dan semua transfer data ke dan dari memori dilakukan dalam blok yang berisi beberapa byte. Dimungkinkan untuk mengakses beberapa byte dalam satu siklus akses; namun, tidak mungkin untuk membaca atau menulis kurang dari satu byte penuh. Seperti yang telah dijelaskan sebelumnya, bahkan jika program harus mengakses satu bit, ia harus membaca satu byte dan menutupi semua bit yang tidak relevan lainnya. Mekanisme untuk mentransfer data antara memori dan prosesor didasarkan pada bus (masalah ini akan dijelaskan dalam bab-bab selanjutnya). Logikanya, bus dibagi menjadi tiga bus yang berbeda dengan fungsi yang berbeda. Satu bertanggung jawab untuk mengirimkan alamat yang relevan yang dimasukkan ke dalam MAR. Bus kedua bertanggung jawab untuk mentransfer data.

Bus ini harus cukup lebar untuk menampung seluruh register. Bus ketiga dimaksudkan untuk memberi sinyal arah data jika dibaca dari memori atau ditulis ke memori

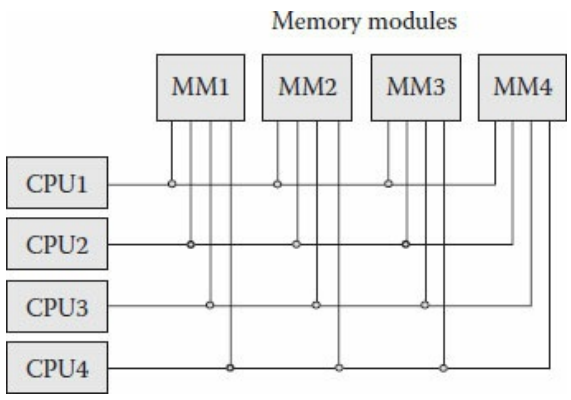
Lebar\* masing-masing bus harus sesuai untuk mentransfer data yang diperlukan. Alasannya, bus alamat harus cukup lebar untuk mewakili alamat sel terakhir dalam memori. Jika k adalah alamat fisik terakhir, lebar bus alamat harus Log2k untuk mengakomodasi ukuran sel memori. Dalam ***byte-addressable*** arsitektur, lebarnya minimal 1 byte. Namun, dalam kasus di mana register lebih lebar, misalnya 32 bit, maka bus juga harus lebih lebar. Jika tidak, untuk mentransfer data ke dalam register, bus akan membutuhkan empat siklus, yang akan memperlambat kecepatan transfer serta eksekusi. Di komputer modern, lebar bus data secara signifikan lebih besar daripada sel memori, sehingga beberapa byte ditransfer dalam satu siklus. Bus ketiga digunakan sebagai sinyal kontrol sehingga bisa sangat sempit. Register internal yang sesuai harus berukuran sama. MDR menyimpan data memiliki lebar yang sama dengan bus data MAR menyimpan alamat memiliki lebar yang sama dengan bus alamat.

Memori dan bus memori secara signifikan lebih lambat dibandingkan dengan prosesor. salah satu alasan untuk menerapkan hierarki memori serta meningkatkan lebar bus. Namun, untuk mencegah atau meminimalkan kemungkinan kemacetan memori, taktik tambahan adalah dikembangkan, Arsitektur komputer set instruksi yang dikurangi (RISC) dikembangkan dengan tujuan meminimalkan akses memori dengan menggunakan banyak register, yang menambah hierarki memori. Sistem operasi virtual memberikan kontribusi berbasis perangkat lunak tambahan untuk masalah ini. Namun demikian, dengan semua taktik yang digunakan, masih ada kasus di mana akses memori dapat secara signifikan menghambat sistem pertunjukan. Salah satu contohnya adalah sistem paralel dengan memori bersama. Dalam kasus ini, beberapa dan terkadang banyak prosesor mencoba mengakses memori, dan total bandwidth mungkin tidak memadai. Dalam kasus seperti itu tidak hanya lebar bus yang bertambah tetapi terkadang bus tambahan juga ditambahkan ke memori gambar 4.7 menggambarkan sistem dual-prosesor, yang mengimplementasikan satu set bus terpisah untuk masing-masing prosesor.



Gambar 4.7 *Dual-port memory*

Ini berarti bahwa jika satu bus sibuk mentransfer data ke satu prosesor, prosesor lainnya tidak harus menunggu, dan dapat menggunakan bus sendiri. Solusi ini disebut memori port ganda karena ada dua bus yang mampu bekerja secara paralel. Demikian pula, dimungkinkan untuk meningkatkan jumlah bus yang menghubungkan memori ke prosesor (Gambar 4.8).



Gambar 4.8 *Multiport memory*

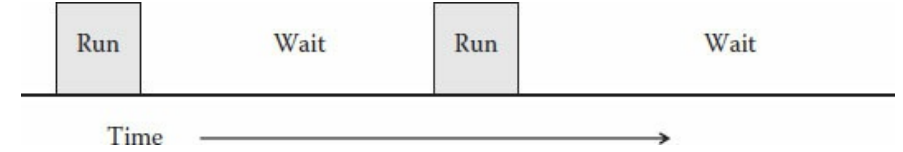
Menerapkan ***additional ports*** (atau bus) tambahan menyiratkan biaya tambahan, yang terkadang mungkin signifikan. Kontributor biaya utama tidak terkait dengan hardware (*the cable*) melainkan mekanisme sinkronisasi yang diperlukan yang harus diterapkan untuk mencegah penulisan paralel dari dua prosesor ke lokasi yang sama, atau pembacaan data sebelum diperbarui. oleh prosesor yang berbeda. Gambar 4.8 menggambarkan sistem empat prosesor dengan memori bersama dan dengan beberapa bus paralel. Untuk meningkatkan kecepatan memori, dibagi menjadi empat bagian yang berbeda, masing-masing mampu bekerja secara paralel dengan yang lain.

1. **RUNNING PROGRAMS.**

Cara program berjalan pada sistem dan penggunaan memorinya telah berubah pada tahun-tahun mengikuti perkembangan teknologi. Komputer generasi pertama dan sistem operasi yang mendukungnya hanya mampu menjalankan satu program pada satu waktu. Sistem operasi memuat program sebelum eksekusi. Memori fisik harus cukup besar untuk menampung seluruh program serta utilitas sistem yang mungkin diperlukan. Jika memori yang tersedia tidak cukup besar, program tidak dapat dimuat dan tidak dapat dijalankan. Mode operasi ini disebut *mono-programming,* karena hanya satu program yang dieksekusi pada waktu tertentu.

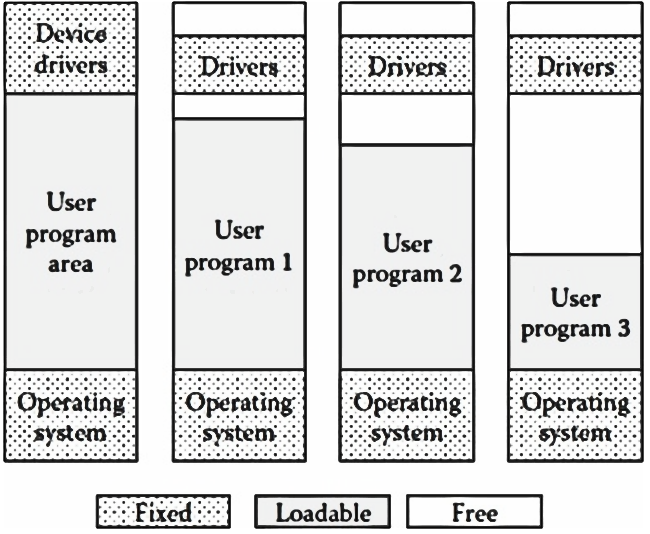
Mode operasi ini sangat tidak efisien, karena tidak menyediakan sarana untuk memanfaatkan sistem secara memadai. Namun, karena sistem pertama hanya menjalankan satu program pada satu waktu, ini secara inheren menciptakan masalah penggunaan. Ketika program sedang menunggu input atau output, dan masih secara signifikan lebih lambat dari prosesor, artinya berarti prosesor sedang menunggu idle.

Gambar 4.9 menggambarkan situasi di mana beberapa waktu prosesor mengeksekusi program, tetapi pada waktu lain hanya menunggu



Gambar 4.9 *Mono-programming*.

Gambar 4.10.menggambarkan empat situasi berbeda. Diagram paling kiri adalah sistem menunggu untuk mengeksekusi program, misalnya : setelah sistem di-boot. Sebagian dari memori digunakan oleh sistem operasi dan sebagian lagi digunakan oleh berbagai driver perangkat. Bagian yang tidak digunakan atau area program adalah untuk memuat program yang akan dijalankan. Tiga diagram lainnya mewakili tiga program yang sedang dieksekusi. Setiap program membutuhkan ukuran memori yang berbeda, jadi dalam semua kasus ini, beberapa bagian dari memori tidak digunakan.

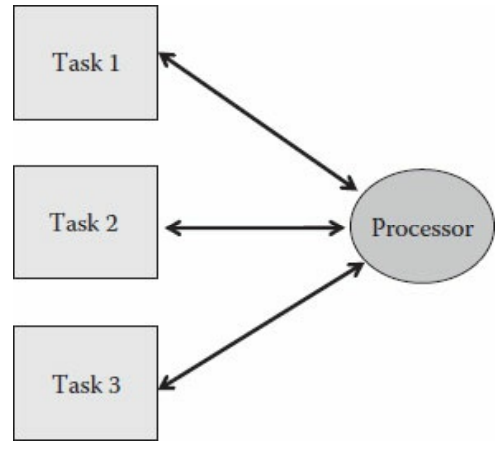


Gambar 4.10.Monoprogramming: Memory content.

Metode monoprogramming yang tidak memungkinkan pemanfaatan penuh sistem, menyerukan perubahan mendesak. Peningkatan tersebut mencakup beberapa perubahan perangkat keras serta beberapa modifikasi sistem operasi untuk memungkinkan lebih dari satu program dijalankan pada satu waktu. Selain pemanfaatan yang lebih baik dari sumber daya yang mahal, perpindahan dari monoprogramming juga memberikan manfaat tambahan:

* **Membagi program menjadi program yang lebih kecil :** Daripada menjalankan satu program yang panjang, lebih baik dan lebih efisien untuk menjalankan beberapa program satu demi satu, jadi jika masalah terjadi selama program berjalan lama, itu harus ***di-restart***.\* Perubahan ini cara mengembangkan program menjadi sangat penting dengan pengenalan program interaktif serta arsitektur modern yang menggunakan beberapa prosesor dan beberapa inti. Selanjutnya, dengan perkembangan rekayasa perangkat lunak disiplin, program yang dulu sangat besar digantikan oleh pendekatan modular untuk memudahkan proses pengembangan serta memungkinkan pemeliharaan yang lebih mudah di masa depan.
* **Manajemen sumber daya yang lebih baik :** Dengan metode ***mono-programming,*** sebuah program dapat menggunakan sumber daya komputer selama berjalan, dan tidak ada program lain yang dapat berjalan. Ketika ada beberapa program yang berjalan secara paralel, sistem operasi bertanggung jawab untuk mendefinisikan mekanisme pengaturan prioritas, sehingga sumber daya dapat dibagi di antara beberapa program prioritas yang identik.
* **Dukungan untuk program interaktif:** Dukungan untuk penggunaan interaktif dikembangkan pada tahap selanjutnya, tetapi sebelum dapat diimplementasikan, sistem harus dapat mendukung eksekusi lebih dari satu program pada satu waktu. Perlu dicatat, dan akan dijelaskan kemudian, bahwa istilah paralel tidak akurat. Karena hanya ada satu prosesor, berarti hanya satu program/tugas yang dijalankan; namun, ketika satu program sedang menunggu input atau output, program lain dapat melanjutkan. Pengguna interaktif memiliki kesan bahwa pekerjaannya adalah satu-satunya yang mengeksekusi

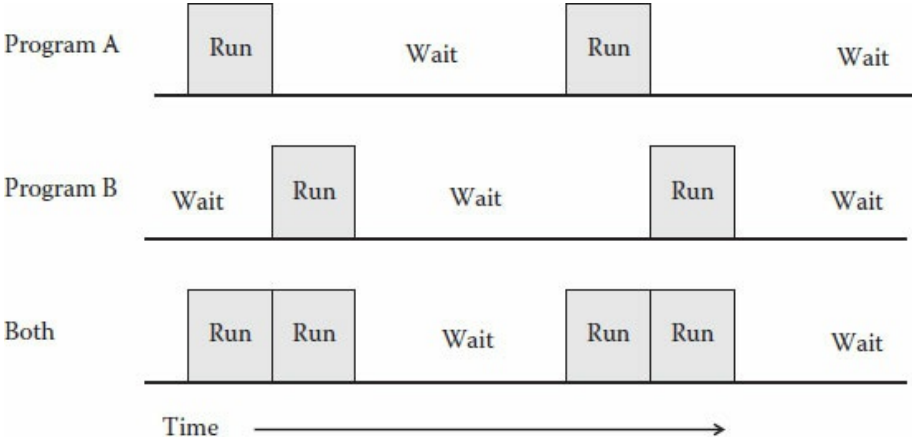
Gagasan untuk mengubah sistem agar dapat mendukung banyak program bermula dari pemahaman bahwa komputer cukup cepat untuk menjalankan beberapa tugas, sehingga setiap tugas tersebut tidak akan mengetahui keberadaan tugas lainnya dan akan “fell (merasa)” seolah-olah berjalan dengan sendirinya. Sebagai bagian dari fitur baru, istilah baru ***time sharing*** diciptakan, yang mewakili perilaku sistem selama pelaksanaan beberapa tugas. Prosesor membagi waktunya di antara semua tugas.



Gambar 4.11 *time sharing*

Prosesor bekerja pada satu tugas untuk jumlah waktu yang telah ditentukan, seperti yang ditentukan oleh sistem operasi, dan kemudian berpindah ke tugas berikutnya, yang mendapatkan waktunya, dan seterusnya. Selama bertahun-tahun, banyak algoritma penjadwalan dikembangkan untuk menangani jumlah waktu yang diperoleh setiap tugas; namun, ini adalah bagian dari sistem operasi. Penjadwalan tugas mencakup keputusan tentang tugas yang akan dieksekusi, jumlah waktu yang dijalankan, pengaturan prioritas, dan sebagainya, dan dilakukan oleh operator sistem.

Fitur perangkat keras tambahan diperlukan untuk menyediakan kemampuan yang diperlukan, seperti melindungi integritas tugas dan mencegah situasi di mana tugas mencoba mengakses data milik tugas lain atau sistem operasi. Perangkat keras harus mendukung pengalihan konteks, yang berarti menyimpan seluruh lingkungan eksekusi tugas saat menunggu giliran untuk dijalankan. Lingkungan eksekusi ini mencakup konten register perangkat keras pada saat *pre-emption*, konten stack, status file terbuka yang sedang digunakan, dan sebagainya.

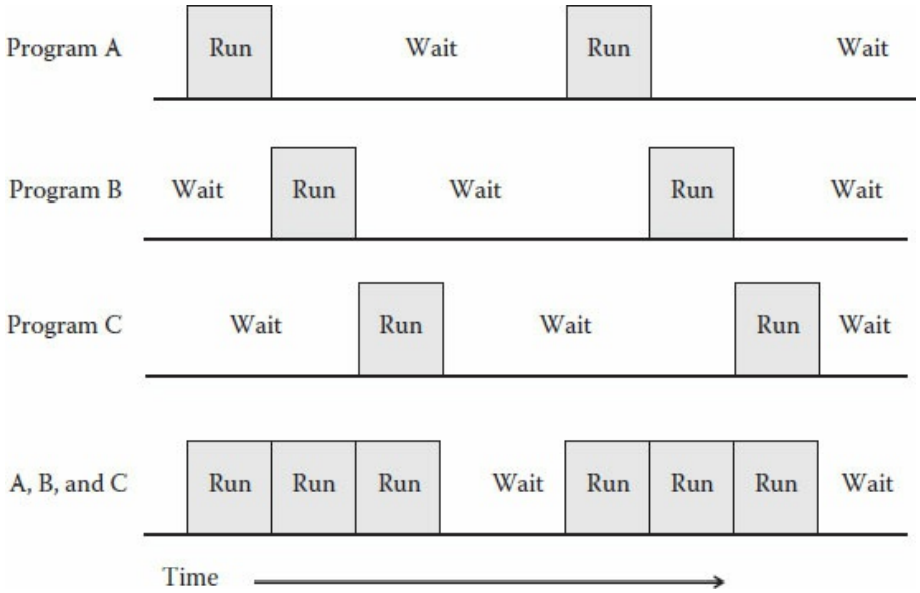


Gambar 4.12 Dual programming

Gambar 4.11 menggambarkan bahwa dalam sistem single-processor, mengacu pada program/tugas yang berjalan secara paralel, istilah yang lebih baik adalah *time sharing*, karena prosesor berbagi waktu dan tugas. Gambar 4.11 tersebut menggambarkan tiga tugas yang sedang dalam proses eksekusi, berbagi sumber daya sistem, dan menggunakan prosesor tunggal yang tersedia, untuk lebih memahami kemajuan teknologi, dari mode *mono-programming* di mana setiap program/tugas bekerja dengan sendirinya, sistem pindah ke *dual-programming*, yang menyediakan kemampuan untuk menjalankan dua tugas "secara paralel" (Gambar 4.12).

Bagian atas diagram menggambarkan situasi ketika hanya tugas A yang dieksekusi. Bagian tengah mengacu pada pelaksanaan tugas B, dan bagian bawah menggambarkan sistem ketika kedua tugas tersebut berlari. Jelas bahwa kedua tugas tidak dijalankan secara paralel. Satu tugas hanya berjalan ketika sistem dalam keadaan idle, yaitu adalah, ketika tugas lain tidak berjalan.

Masalah penting adalah waktu idle prosesor berkurang. Ini berarti bahwa pemanfaatan sumber daya sistem meningkat dan sistem komputer lebih produktif, yang meningkatkan rasio harga/kinerjanya. Perlu dicatat bahwa ada kemungkinan bahwa tugas kedua (B) tidak akan tiba tepat setelah tugas A selesai. Dalam kasus seperti itu, itu akan memiliki untuk menunggu gilirannya, yang dapat mengakibatkan beberapa peningkatan *response time,* tetapi prosesor menganggur waktu akan berkurang. Berdasarkan keberhasilan *dual programming*, pengembangan untuk meningkatkan metode sehingga mengakomodasi lebih banyak tugas yang dijalankan "secara paralel." Ini adalah metode multi-programming (Gambar 4.14, menggambarkan tiga tugas).



Gambar 4.13 Tiga tugas "secara paralel."

Seperti halnya *dual-programming*, waktu idle prosesor terus berkurang. Namun, perlu dicatat bahwa angka tersebut sangat sederhana. Dalam kehidupan nyata, tidak ada yang dapat memastikan bahwa tugas akan tiba di jendela *idle prosesor*. Namun, seperti yang dijelaskan sebelumnya (dalam contoh *dual-programming*),

bahkan jika tugas tiba pada saat yang sama, itu adalah tanggung jawab sistem operasi untuk menjalankannya, tetapi beban kerja yang meningkat akan memungkinkan pemanfaatan sistem yang lebih baik, yang akan meningkatkan rasio harga/kinerja.

Jumlah tugas yang dijalankan secara bersamaan tentu saja tidak terbatas pada tiga, dan di sebagian besar komputer modern, jumlah ini jauh lebih besar; terutama karena konsep memori virtual, yang tidak memerlukan seluruh program untuk berada di memori, hanya bagian yang berjalan. Jumlah tugas yang dijalankan secara bersamaan dapat ditingkatkan selama sumber daya yang tersedia dapat mendukungnya. Asumsi yang mendasari untuk meningkatkan jumlah tugas berasal dari beberapa pemahaman :

* Sebagian besar program tidak hanya menggunakan prosesor dan membutuhkan sumber daya tambahan. Selanjutnya, selama eksekusi, program mungkin memerlukan data yang disimpan di disk (*hard-drive*). Ketika manusia terlibat memberikan masukan, mungkin membutuhkan waktu yang sangat lama; yaitu, program sedang menunggu input, tetapi pengguna sedang berbicara di telepon atau bahkan pergi makan siang. Dalam contoh jika sistem menggunakan pendekatan mono-programming, pemanfaatan sistem akan sangat rendah.
* Memori di komputer pertama terbatas dan sangat mahal. Meskipun di modern komputer, ini bukan masalah lagi karena kapasitas memori telah meningkat secara dramatis dan harga tidak signifikan (salah satu konsekuensi dari hukum Moore), kami masih menikmati perkembangannya yang diperlukan untuk mengatasi masalah masa lalu. Ini bukan satu-satunya kasus di mana teknologi uang muka diperlukan untuk memperbaiki masalah yang tidak ada lagi.
* Prosesor, serta seluruh sistem komputer, sangat mahal, dan penekanan khusus ditempatkan pada menggunakannya dengan cara terbaik dan paling efisien. Ini juga masalah masa lalu, karena sistem komputer saat ini relatif murah. Selanjutnya, dengan revolusi PC, masalah harga/kinerja tidak penting lagi. Karena harganya yang murah, orang-orang membeli PC dan menggunakannya hanya untuk sebagian kecil dari hari. Pemanfaatan berlebihan digantikan oleh waktu respons yang lebih baik karena pengalaman pengguna yang lebih baik.
* Isu bersejarah lainnya adalah pemahaman bahwa pertukaran antar tugas kadang-kadang tidak mungkin dan, jika mungkin, itu memakan waktu. Sejak itu perangkat keras telah dimodifikasi, instruksi pengalihan konteks\* baru telah ditambahkan, dan overhead yang terkait dengan tugas switching telah diturunkan. Namun demikian, setiap pengalihan konteks seperti itu masih melibatkan beberapa tingkat overhead yang harus dipertimbangkan. Misalnya, dalam arsitektur modern yang menggunakan banyak *register*, *overhead* lebih besar karena sistem, antara lain, harus menyimpan isi *register* ini dan kemudian memuatnya kembali dengan konten baru.

1. **ESTIMASI PENGUNAAN PROCESSOS**

Dimungkinkan untuk memperkirakan penggunaan prosesor saat menjalankan beban tertentu untuk melakukannya kita harus memperkirakan persentase waktu tugas/proses menunggu input atau output. Pemanfaatan prosesor dapat berguna dalam kasus di mana *system’s engineer* harus mengetahui arsitektur yang dibutuhkan secara tepat. Di satu sisi, penurunan biaya perangkat keras memungkinkan untuk memutuskan konfigurasi yang jauh lebih besar; di sisi lain, jika sistem akan digunakan oleh pengguna dan ada ribuan sistem yang akan diinstal, pengambilan keputusan yang lebih andal diperlukan. Perkiraan diberikan oleh rumus :



*Dimana :*

*p* adalah persentase rata-rata waktu proses menunggu I/O

*n* adalah jumlah proses yang dieksekusi dalam sistem

Rumus detailnya adalah :



Dimana :

*p1* adalah persentase waktu proses pertama menunggu I/O

*p2* adalah persentase waktu proses kedua menunggu SO, dan seterusnya

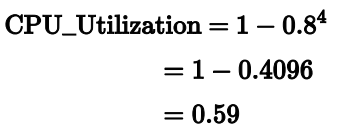
*n*  adalah jumlah proses

Misalnya, dengan asumsi

p = 0,8 (rata-rata proses menunggu I/O 80% dari waktu)

n = 4 (ada empat proses yang dieksekusi)

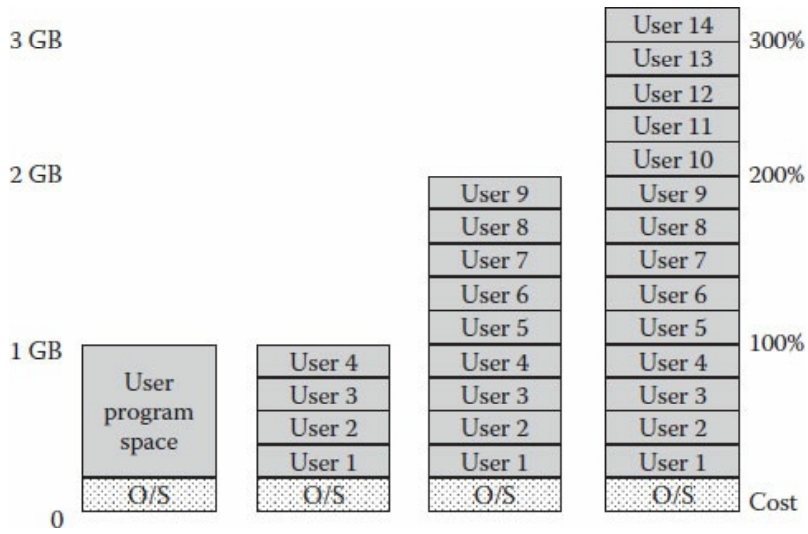
pemanfaatan prosesor akan menjadi



Dengan asumsi tidak adanya hambatan tambahan, seperti memori, pemanfaatan prosesor akan menjadi 59%.

Menggunakan rumus penggunaan CPU dapat memberikan mekanisme yang efektif dalam pengambilan keputusan mengenai peningkatan sistem, dengan asumsi ukuran memori dari sistem yang diberikan membatasi pemanfaatan CPU-nya. Ini terjadi ketika ukuran memori yang tersedia dan bebas menyediakan ruang untuk sejumlah tugas. Karena biaya upgrade memori diketahui, mudah untuk menghitung efek upgrade ini. Hal ini ditunjukkan pada Gambar 4.14. menggambarkan tiga konfigurasi:

Diagram paling kiri menggambarkan sistem tepat setelah proses boot selesai. Sistem ini terdiri dari 1 GB memori; bagian digunakan oleh sistem operasi sementara bagian lainnya kosong dan akan digunakan untuk proses pengguna.



Gambar 4.14 Contoh penghitungan penggunaan CPU

* Diagram berikutnya adalah sistem yang sama, memori menampung 4 proses pengguna. contoh asumsi bagian memori yang kosong dapat menampung empat proses dengan mempertimbangkan ukuran rata-rata setiap proses.
* Diagram berikutnya menggambarkan sistem yang ditingkatkan di mana ukuran memori digandakan. 1 GB sistem ini memiliki 2 GB. Dalam sistem ini, jumlah proses yang dieksekusi secara bersamaan juga meningkat, dan alih-alih empat proses dalam arsitektur 1 GB, sekarang ada sembilan proses pengguna.
* Diagram (paling kanan) terakhir menggambarkan sistem asli setelah ditingkatkan dua kali, dan alih-alih yang asli 1 GB, ukuran memori sekarang menjadi 3 GB. Sekali lagi, jumlah proses pengguna juga meningkat, dan alih-alih empat proses dalam arsitektur asli dan sembilan proses dalam sistem 2 GB, sistem sekarang mampu menjalankan 14 proses secara bersamaan.
* Skala paling kanan tidak berhubungan dengan biaya sistem tetapi dengan ukuran memori dan biaya yang terkait hanya dengan memori.

Jika rata-rata setiap proses tersebut menunggu I/O 80% dari waktu, maka pemanfaatan prosesor dari konfigurasi asli akan menjadi 59%. Meningkatkan sistem dengan menambahkan 1 GB memori meningkat jumlah prosesor serta pemanfaatan prosesor. Menggunakan rumus untuk n = 9 mengungkapkan bahwa pemanfaatan prosesor naik menjadi 87%. Karena biaya pemutakhiran diketahui, mudah untuk menilai apakah kinerja tambahan tambahan yang diperoleh dengan meningkatkan penggunaan CPU sebesar 28% (87% – 59% = 28%) dapat dibenarkan dengan biaya tambahan. Penilaian yang sama dapat diulang untuk arsitektur ketiga.

Ketika ukuran memori meningkat menjadi 3 GB, ada 14 proses bersamaan. Utilisasi prosesor akan meningkat menjadi 96 % dan sekali lagi, karena biaya peningkatan diketahui, manfaatnya dapat dinilai berdasarkan peningkatan kinerja sistem. Dalam kasus khusus ini, pemutakhiran pertama menghasilkan peningkatan kinerja 28%, sedangkan pemutakhiran kedua hanya menghasilkan tambahan 9%, jadi mungkin tindakan terbaik adalah menggunakan sistem 2 GB.

Isu penting mengenai pemanfaatan prosesor adalah hal nyata bahwa semua perhitungan dan penilaian dapat dilakukan tanpa peningkatan sebenarnya, sehingga mungkin bagi *system’s engineer* untuk mencapai solusi tanpa mengeluarkan uang untuk peningkatan atau menggunakan salah satu dari yang tersedia. Perhitungan ini didasarkan pada asumsi bahwa perilaku proses situs diketahui. Sebenarnya, sulit untuk memperkirakan persentase waktu suatu proses menunggu I/O.

Selain itu, rumus di atas tidak memperhitungkan sudut pandang pengguna. Jika sistem yang dijelaskan untuk tugas-tugas interaktif, mempertimbangkan bahwa penggunaan CPU sebesar 96% dapat memberikan peningkatan harga/kinerja, tetapi *response time* sangat tinggi. Dalam kasus seperti itu, pemanfaatan yang ditingkatkan berdampak negatif mengingat waktu yang terbuang oleh pengguna *system’s response*.

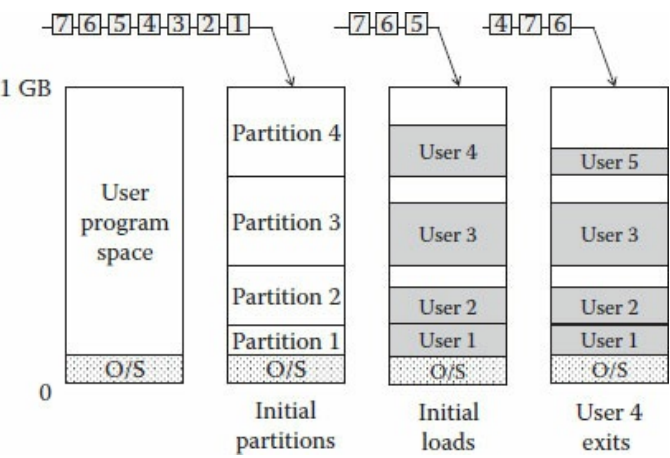
1. **PARTISI**

Untuk mengimplementasikan *multi-programming* dengan menggunakan berbagai mekanisme. Pendekatan pertama dan paling sederhana adalah dengan mendefinisikan partisi. ***Partisi*** adalah area tertentu dalam memori yang menyimpan program yang akan dieksekusi. Ini adalah jenis lingkungan yang berjalan, partisi didefinisikan selama proses boot. Setiap partisi memiliki ukuran berbeda, yang tidak dapat diubah sampai booting berikutnya. Ukuran total partisi adalah memori bebas tersedia pada sistem untuk proses pengguna. Ukuran partisi menentukan proses yang dapat dijalankan di partisi tertentu. Sebelum konsep ***virtual memori*** dikembangkan, seluruh proses dimuat, hanya beberapa partisi yang dapat diterapkan. Jumlah partisi yang ditentukan menentukan jumlah proses yang dapat dijalankan secara bersamaan.

Mengelola partisi dan memutuskan program mana yang akan dimuat ke partisi mana yang menjadi tanggung jawab sistem operasi. Ide partisi mencoba untuk meniru komputer mono-programming pertama, tujuannya adalah untuk membagi komputer menjadi beberapa lingkungan kerja yang ditentukan, masing-masing menyerupai sebuah sistem *mono-program*. Dengan demikian, implementasi partisi pertama menggunakan partisi berukuran tetap, total memori bebas yang tersedia dibagi dengan jumlah partisi. Hanya pada tahap selanjutnya para desainer memahami bahwa masalah yang terkait dengan monoprogramming ada dengan partisi berukuran tetap dan identik. Tahap selanjutnya adalah mendefinisikan partisi dengan ukuran variabel.

Dengan menggunakan ide ini, partisi terus meniru konsep *mono-programming* tetapi dengan beberapa perbaikan. Peran sistem operasi untuk menangani partisi dan untuk memutuskan, berdasarkan persyaratan memori proses, partisi mana yang paling cocok. Untuk menerapkan perubahan ini, sistem operasi harus mengetahui jumlah partisi dan ukurannya serta ukuran setiap proses yang dieksekusi. Setiap proses/program yang dieksekusi menentukan kebutuhan memorinya, dan parameter yang digunakan sistem operasi dalam alokasinya keputusan, seperti pada Gambar 4.15.

Angka ini berkaitan dengan partisi tetap, tetap dalam konteks ini tidak menentukan ukuran tetap dari partisi, melainkan jumlah partisi ditentukan saat boot. Artinya jumlah partisi tetap dan tidak dapat diubah hingga proses booting berikutnya.



Gambar 4.15. Fixed partitions.

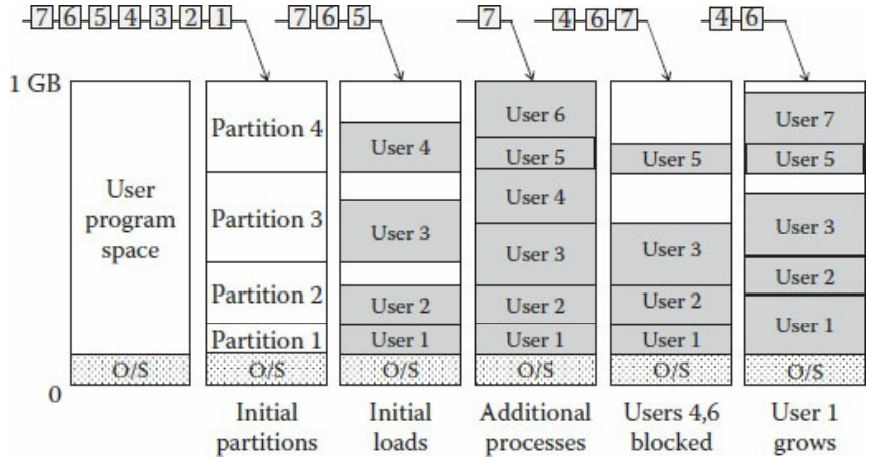
**Gambar tersebut terdiri dari empat diagram yang terhubung :**

* Diagram paling kiri mendefinisikan karakteristik memori sistem. Dalam hal ini, ini adalah sistem 1 GB, diagram menggambarkan memori bebas yang tersedia untuk proses pengguna. Untuk menyederhanakan diagram, driver perangkat, yang seharusnya dimuat di alamat memori yang lebih tinggi, tidak disertakan.
* Diagram sebelah kanan berhubungan dengan sistem setelah ruang memori bebas dibagi empat bagian (*partisi*). Partisi tetap, mengubah jumlah partisi atau mengubah ukurannya boot sistem yang baru. Di bagian atas diagram adalah daftar (*antrian*) proses prioritas yang menunggu untuk dieksekusi.
* Diagram berikutnya menggambarkan situasi setelah sistem operasi mulai mengalokasikan partisi ke proses yang tersedia. Sistem operasi mengalokasikan empat partisi pertama ke partisi yang paling cocok, meskipun dapat dilihat bahwa, dalam banyak kasus, persyaratan memori proses lebih rendah daripada ukuran partisi, yang menghasilkan memori yang tidak digunakan.
* Diagram berikutnya menggambarkan situasi beberapa waktu kemudian. Pada suatu saat, Proses 4 selesai dieksekusi dan kemudian sistem operasi harus mengalokasikan proses berikutnya dalam barisan. Sayangnya, Proses 5 membutuhkan memori yang sangat sedikit, tetapi satu-satunya partisi yang tersedia adalah yang besar. Penjadwal\* mengalokasikan partisi besar ke tugas kecil, membuang sebagian besar memori partisi. Perlu dicatat bahwa sistem operasi modern mungkin memiliki kebijaksanaan tinggi mengenai proses penjadwalan, misalnya, membuat keputusan berdasarkan beberapa parameter tambahan; namun, dalam contoh ini, keputusan penjadwal didasarkan pada satu parameter saja, proses berikutnya sejalan.

Mekanisme partisi tetap, meskipun tidak optimal, memberikan kemungkinan yang diperlukan untuk menjalankan beberapa proses secara paralel. Namun, kontribusi terpentingnya adalah membuka jalan mekanisme yang lebih canggih—partisi dinamis.

Pemahaman bahwa partisi tetap masih membuang sumber daya memori mendorong perkembangan baru. Partisi dinamis didasarkan pada partisi tetap dengan satu langkah maju. Konsep partisi dinamis didasarkan pada gagasan bahwa jumlah partisi dan ukurannya dapat diubah oleh sistem operasi selama operasi normal.

Hal ini dimaksudkan untuk memberikan manajemen memori yang lebih baik dan pemanfaatan prosesor yang lebih baik dengan meningkatkan jumlah proses yang dijalankan secara bersamaan. Sayangnya, ada juga beberapa kelemahan; karena overhead sistem operasi meningkat, perlu ada mekanisme baru untuk menangani "lubang", yang merupakan bagian dari memori yang tidak terisi yang harus dipelihara, dipindahkan, dan digabungkan, seperti yang ditunjukkan pada Gambar 4.16.



Gambar 4.16. Dynamic partitions.

Angka didasarkan pada Gambar 4.16, dan tiga diagram pertama (paling kiri) identik:

* Diagram paling kiri mendefinisikan karakteristik memori sistem. Sekali lagi, ini adalah 1 GB sistem, diagram menggambarkan memori bebas yang tersedia untuk proses pengguna.
* Diagram berikutnya (di sebelah kanan) berhubungan dengan sistem setelah ruang memori kosong dibagi menjadi empat partisi dengan ukuran berbeda. Di bagian atas diagram adalah antrian proses menunggu untuk dieksekusi.
* Diagram berikutnya menggambarkan situasi setelah sistem operasi mengalokasikan partisi untuk setiap proses. Dalam hal ini, karena hanya ada empat partisi, maka empat proses dimuat ke dalam memori.
* Setelah mengalokasikan empat proses, sistem operasi menyadari bahwa ada lubang di ujungnya dari beberapa partisi. Selanjutnya, proses selanjutnya yang sejalan (Proses 5) sangat kecil, dan jumlah memori yang dibutuhkan dapat diperoleh dengan menggabungkan lubang yang tersisa. Itu sistem operasi memindahkan dua partisi atas sedemikian rupa sehingga semua lubang digabungkan untuk membuat segmen memori yang berkesinambungan. Sistem operasi kemudian mendefinisikan partisi baru untuk digunakan untuk segmen memori baru dan mengalokasikannya ke Proses 5. Selanjutnya, tersedia memori cukup untuk memuat Proses 6 juga, sehingga sistem operasi membuat tambahan partisi dan memuat Proses 6 ke dalamnya. Diagram menggambarkan situasi setelah keenam proses telah dimuat.
* Pada suatu saat, Proses 4 dan 6 selesai dieksekusi dan memori dialokasikan untuk ini proses ditandai sebagai bebas. Mengenai lubang yang sedang dibuat, sistem operasi menggunakan berbagai pendekatan. Aturan dasarnya adalah tidak melakukan pekerjaan tambahan jika tidak diperlukan. Untuk itu alasannya, dalam banyak kasus sistem operasi tidak akan menggunakan manajemen lubang dinamis. Itu manajemen lubang dinamis menyiratkan bahwa sistem operasi akan memindahkan partisi secara berurutan untuk membuat segmen kontinu yang lebih besar. Perilaku umum adalah manajemen lubang berdasarkan permintaan, di mana hanya jika suatu proses membutuhkan potongan memori yang lebih besar, sistem operasi akan memindahkannya partisi untuk membuatnya. Dalam contoh khusus ini, manajemen lubang memori dilakukan tuntutan. Perlu dicatat bahwa manajemen lubang adalah overhead ke sistem, jadi untuk dinamis manajemen lubang, sistem operasi meningkatkan overhead sistem, tetapi siap untuk kebutuhan memori masa depan. Dengan manajemen lubang per permintaan, overhead lebih rendah, tetapi ketika permintaan datang, itu akan memakan waktu lebih lama untuk memenuhinya.
* Pada tahap selanjutnya, Proses 1 meminta memori tambahan. Sebagai bagian dari perbaikan sistem operasi, beberapa fungsi baru ditambahkan untuk membantu pengguna mengelola proses dengan lebih baik persyaratan memori. Salah satu fungsi tersebut adalah alokasi memori (MALLOC).\*Sayangnya, Proses 1 dalam partisi dan menggunakan seluruh ukurannya. Bahkan mungkin itu proses ditempatkan di partisi yang lebih besar dan sistem operasi membatalkan alokasi yang tidak perlu memori, yang ditugaskan ke partisi yang berbeda. UNTUK menyediakan memori yang diminta, sistem operasi dapat meluncurkan Proses 1 hingga partisi yang lebih besar tersedia. Ketika memori tersedia, Proses 1 akan digulirkan lagi sehingga dapat melanjutkan bekerja. Dalam kasus ini digambarkan, sistem operasi memeriksa apakah itu dapat memenuhi permintaan dan menyadari bahwa ada banyak lubang demi partisi 3. Kemudian sistem operasi memindahkan partisi 2 dan 3 ke atas dan membuat potongan yang baru diminta yang digabungkan dengan partisi 1.

1. **VIRTUAL MEMORY**

Solusi yang dijelaskan di bagian sebelumnya dimaksudkan untuk memberikan kemungkinan untuk menjalankan beberapa tugas secara bersamaan. Namun, ketika ukuran aplikasi meningkat, memori yang lebih besar hanya dapat menyimpan sejumlah tugas yang terbatas, yang menimbulkan masalah baru. Seperti yang telah disebutkan, faktor motivasi utama untuk perbaikan teknologi adalah ekonomis.

Komputer pertama sangat mahal dan untuk membenarkan pengeluaran, manajer pusat komputer harus memastikan bahwa sistem digunakan sebanyak mungkin. Vendor komputer menyediakan mekanisme untuk *multi-programming* dalam upaya memberikan rasio harga/kinerja yang lebih baik. Namun, teknologi yang ada saat itu, yang didasarkan pada pemuatan seluruh program ke dalam memori, menghambat upaya ini. Meskipun ukuran memori dapat ditingkatkan, langkah ini cukup signifikan.

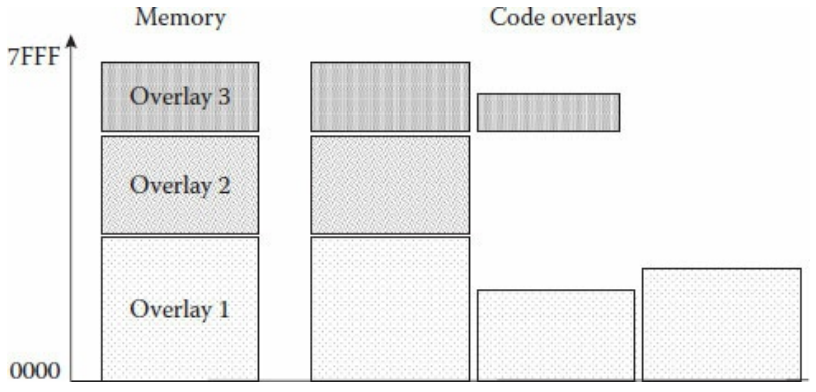
Selanjutnya, setelah menganalisis dengan cermat perilaku program pelaksana, menjadi jelas bahwa setiap program memiliki bagian besar yang jarang digunakan dan bagian lain yang tidak digunakan sama sekali. Misalnya, beberapa bagian dari program digunakan untuk penanganan kesalahan, dan mungkin saja dalam proses tertentu, tidak ada kesalahan yang harus ditangani. Perilaku ini mengarah pada pemahaman bahwa tidak perlu memuat seluruh program ke dalam memori, asalkan mekanisme lain dapat dikembangkan yang akan memuat bagian yang hilang sesuai permintaan.

Motivator penting lainnya adalah pengembangan pasar aplikasi off-the-shelf. Aplikasi tersebut harus dapat berjalan pada setiap sistem, bahkan jika ukuran memori fisiknya terbatas. Dalam kasus seperti itu, mungkin berjalan lebih lambat, tetapi harus bisa berjalan. Ada beberapa solusi yang menyediakan sarana untuk menjalankan program membutuhkan lebih banyak memori daripada memori fisik yang tersedia. Salah satu solusi sederhana adalah meminta pengembang menangani masalah ini.

Pengembang dengan program dan memahami pola pelaksanaan, sehingga solusi yang disesuaikan dapat dikembangkan. Solusi semacam itu terkadang terbatas pada sistem tertentu. Contoh dari pendekatan tersebut adalah dengan menggunakan mekanisme overlay. Asumsi yang mendasari adalah bahwa program dapat dibagi menjadi bagian-bagian yang berbeda (*overlay*) dan setiap bagian tersebut dapat dieksekusi secara independen dari bagian lain. Atau, program dapat dikembangkan sedemikian rupa sehingga menggunakan bagian-bagian berbeda yang dieksekusi satu demi satu. Saat menggunakan sistem memori terbatas, hanya satu bagian pada satu waktu yang akan dimuat ke dalam memori, dan ketika selesai, itu akan dibongkar, memberi ruang untuk bagian berikutnya untuk dimuat dan dieksekusi.

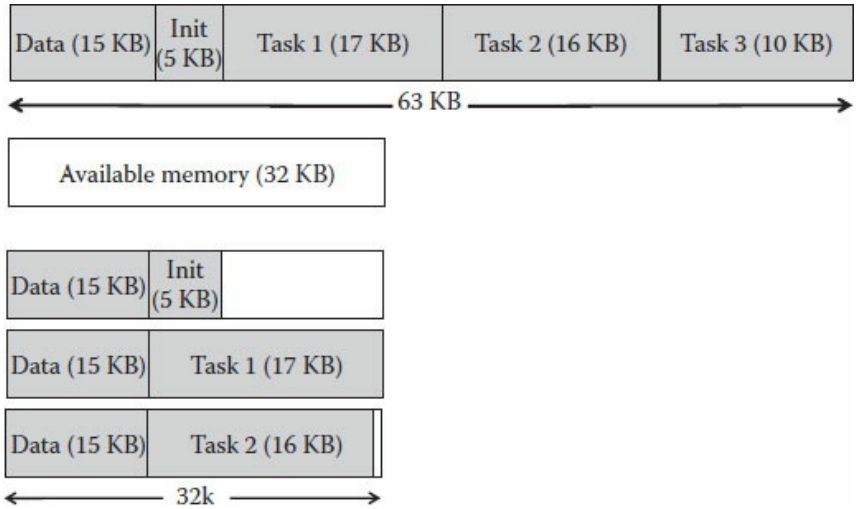
Meskipun komputer modern telah mengadopsi metode lain yang lebih efisien dan canggih, mekanisme overlay digunakan oleh beberapa peralatan serta telepon seluler. Sebagai contoh, mari kita asumsikan program tertentu dibagi menjadi enam bagian, masing-masing bertanggung jawab atas aktivitas yang berbeda. Tiga bagian pertama bekerja secara independen; yaitu, ketika bagian satu bekerja tidak akan pernah berinteraksi dengan bagian dua dan tiga, atau ketika bagian dua berjalan tidak memanggil atau berhubungan dengan data yang berada di bagian satu dan tiga.

Bagian lima dan enam bekerja independen. Bagian empat, adalah kerangka kerja umum yang berjalan di seluruh program, dan menyediakan layanan yang dibutuhkan oleh semua bagian. Solusi lama dan sederhana adalah memuat semua bagian ke dalam memori, yang merupakan langkah tidak efisien. Pendekatan yang lebih baik adalah dengan mendefinisikan tiga overlay seperti yang dijelaskan pada Gambar 4.17.



Gambar 4.17. Contoh overlay

Hamparan pertama digunakan untuk memuat bagian satu, dua, atau tiga. Lapisan kedua digunakan untuk memuat infrastruktur umum (bagian empat), dan lapisan ketiga digunakan untuk memuat bagian lima atau enam. Untuk klarifikasi lebih lanjut, kami akan menggunakan contoh nyata. Dengan asumsi bahwa untuk menjalankan aplikasi di beberapa sistem tertanam, diperlukan memori sebesar 63 KB (lihat Gambar 4.18). Karena beberapa kendala, perangkat yang disematkan ini hanya menampung memori 32 KB. Untuk menjembatani keterbatasan fisik dan persyaratan memori perangkat lunak, overlay digunakan. Sistem bekerja dengan tabel dan variabel umum yang membutuhkan 15 KB. Bagian ini tersedia untuk semua bagian lain, dan alasan ini akan dimuat sepanjang durasi. Program dibagi menjadi empat bagian independen:



Gambar 4.18. Contoh sistem tertanam berbasis overlay

1. Bagian pertama membutuhkan 5 KB untuk inisialisasi sistem dan untuk alasan itu hanya digunakan di awal lari.
2. Bagian kedua membutuhkan 17 KB.
3. Bagian ketiga membutuhkan 16 KB.
4. Bagian keempat membutuhkan 10 KB.

Bagian dua, tiga, dan empat bekerja secara independen tetapi membutuhkan akses ke bagian umum. Artinya bagian-bagian tersebut tidak saling memanggil dan tidak mengakses data yang tersimpan di bagian lain. Untuk mengurangi jumlah memori yang diperlukan, memori 32-KB yang tersedia dibagi menjadi dua bagian. Yang pertama adalah potongan statis 15-KB yang ditujukan untuk data umum. Selain itu, potongan kedua dari 17-KB akan digunakan untuk kode. Potongan kedua harus, tentu saja, cukup besar untuk mengakomodasi overlay kode terbesar yang.

Selama fase inisialisasi, bagian pertama dimuat ke dalam potongan kedua. Setelah inisialisasi adalah selesai, potongan yang sama akan digunakan untuk memuat bagian kode lainnya berdasarkan perilaku program (Gambar 4.18). Perlu dicatat bahwa mekanisme overlay hanya mungkin dalam kasus di mana program dapat dibagi menjadi bagian-bagian pelaksana yang berbeda dan ada pemisahan lengkap antara bagian-bagian ini. Terkadang, diperlukan duplikat fungsi/metode serupa untuk mencapai pemisahan yang diperlukan. Duplikasi ini, tentu saja, bertentangan dengan salah satu prinsip dasar dan penting dari praktik pemrograman yang baik. Replikasi kode yang berlebihan serta pemahaman bahwa pengoptimalan manual bukanlah cara yang tepat meyakinkan industri untuk mencari pendekatan yang berbeda.

Menjalankan program bahkan ketika memori fisik tidak mencukupi telah selesai secara otomatis dengan memanfaatkan mekanisme yang disebut memori virtual. Yang mendasari dan terbukti asumsi adalah bahwa program mengandung banyak bagian untuk menangani berbagai situasi dan kasus; namun, ketika program berjalan, ia hanya menggunakan sebagian kecil dari kode. Misalnya, jika selama pelaksanaan aplikasi perbankan yang menangani rekening giro, semua penarikan valid, aplikasi tidak harus menjalankan kode yang menangani semua kemungkinan situasi tidak valid (tanggal tidak valid, dana di rekening tidak mencukupi, dll.). Bagian khusus dari aplikasi ini tidak harus ada di memori, sehingga membebaskan ruang untuk tujuan lain yang lebih berguna. Menerapkan memori virtual menyiratkan pengembangan mekanisme yang hanya akan memuat sebagian dari menjalankan aplikasi, sementara bagian lain akan dimuat sesuai permintaan. Ini tentu saja berlaku asalkan sistem dapat memuat bagian yang hilang dengan cukup cepat. Bagian yang tidak diperlukan akan tetap pada disk (hard drive) dan untuk alasan itu, ketika berkaitan dengan hierarki memori (Gambar 5.5), disk adalah bagian dari organisasi memori. Jika kita membekukan sistem saat aplikasi sedang berjalan, kita akan dapat melihat beberapa bagian dari aplikasi itu yang berada di memori, sementara yang lain bagian masih pada disk. Mekanismenya disebut memori virtual karena organisasi memori sangat berbeda dari pemahaman biasa. Seperti yang dinyatakan sebelumnya memori adalah matriks satu dimensi dari sel-sel berurutan. Sel memori virtual yang digunakan, namun tidak berurutan. Untuk menerjemahkan alamat memori virtual menjadi nyata alamat, mekanisme perangkat keras khusus diperlukan. Program ini mewakili pandangan pengembang dari sistem di mana memori adalah susunan sel yang berkesinambungan, sementara secara fisik sel-sel ini mungkin berada di memori di lokasi yang berbeda dan mungkin tidak berkelanjutan. Terjemahan antara alamat virtual dan fisik dilakukan dengan cepat selama eksekusi untuk setiap akses memori. Untuk penjelasan yang lebih rinci, kami akan mendefinisikan alamat virtual dan fisik:

* Alamat virtual dibuat oleh program pada waktu eksekusi dengan mengakses *variabel* memanggil fungsi/metode. Setelah kompilasi, alamat tempat metode atau variabel berada menjadi bagian dari instruksi. Alamat ini dianggap sebagai alamat virtual kemungkinan konten yang diperlukan tidak dalam memori.
* Alamat fisik adalah tempat sel atau metode tertentu berada di memori. Selama eksekusi, ketika program mencoba mengakses lokasi virtual tertentu, perangkat keras harus menerjemahkan alamat virtual ke alamat fisik. Sebagai bagian dari terjemahan, perangkat keras juga memeriksa apakah alamat yang diperlukan ada di memori. Dalam hal ini, akses akan dilakukan. Jika alamatnya tidak ada memori, perangkat keras akan mengeluarkan interupsi yang akan ditangkap oleh sistem operasi. Dia tanggung jawab sistem operasi untuk memuat konten yang diperlukan. Selama proses pemuatan, program ditunda dan program lain dapat menggunakan prosesor. Hanya setelah konten di alamat telah dimuat apakah program yang terputus dapat melanjutkan eksekusinya.

Mekanisme *virtual memori* didasar konsep overlay tetapi mencakup peningkatan signifikan. Ada dua perbedaan antara kedua mekanisme tersebut. Dengan *virtual memori*, program dibagi menjadi bagian-bagian berukuran tetap, dan penanganan bagian ini dilakukan secara otomatis oleh perangkat keras dan sistem operasi. Sebagai contoh, mari kita asumsikan ada program yang membutuhkan memori 5 MB. Programnya dibagi menjadi bagian dengan panjang tetap yang disebut halaman. Untuk menggunakan halaman 64 KB berarti program ini berisi 80 halaman. Seperti yang telah disebutkan, tidak semua halaman ini harus ada di memori selama eksekusi.

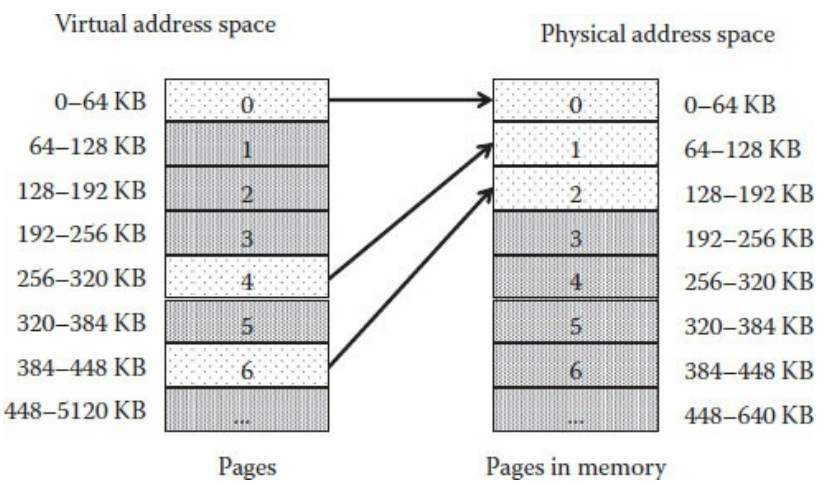
Memori fisik juga dibagi menjadi segmen 64 KB disebut frame. Setiap frame dimaksudkan untuk menyimpan satu halaman. Dalam menentukan ukuran halaman/bingkai, para desainer: menyeimbangkan antara dua tren yang bertentangan. Di satu sisi, semakin kecil ukuran halaman/bingkai, pemanfaatan memori yang lebih baik. Di sisi lain, ukuran halaman/bingkai kecil membutuhkan lebih banyak operasi I/O untuk memuat halaman. Sebagai tambahan, sistem operasi memelihara tabel yang berisi alamat halaman, dan dengan tren meningkatkan ukuran memori, tabel ini menjadi sangat besar. Sebuah tabel halaman besar meningkatkan sistem overhead dan mengurangi jumlah memori yang tersedia untuk program pengguna.

Untuk beberapa produsen sedang mempertimbangkan untuk menggandakan ukuran halaman/bingkai, yang membagi dua ukuran tabel ini. Di masa lalu, ketika komputer secara signifikan lebih mahal dan sistem digunakan untuk durasi yang lebih lama, beberapa sistem mendukung ukuran halaman yang berbeda.

menggambarkan contoh sebelumnya; sisi kiri didedikasikan untuk aplikasi dan halamannya, dan sisi kanan mewakili sistem dengan jumlah memori terbatas. Alamat aplikasi adalah alamat virtual karena, seperti dapat dilihat pada gambar, tidak semua halaman dimuat ke dalam memori fisik. Selanjutnya, halaman aplikasi yang dimuat berada di lokasi yang berbeda dalam memori fisik. Ketika program mulai berjalan, kami akan menganggapnya mengeksekusi instruksi yang ada di halaman nol. Ini berarti bahwa halaman nol akan dimuat ke dalam bingkai yang tersedia berikutnya. Dalam kasus khusus ini, bingkai pertama yang tersedia adalah bingkai nol, jadi halaman dimuat ke dalam bingkai nol. Program melanjutkan eksekusinya, dan pada titik tertentu, ia mengakses atau memanggil metode yang berada di halaman empat.

Karena halaman ini tidak ada di memori, perangkat keras tidak dapat menyelesaikan terjemahan alamat, dan itu mengeluarkan interupsi. Sistem operasi menyadari jenis interupsi dan menunda program. Kemudian ia menemukan halaman yang hilang dan memuatnya ke dalam bingkai berikutnya yang tersedia.

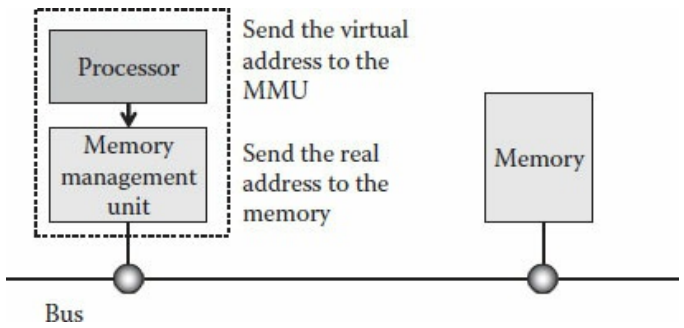
Halaman dimuat ke dalam bingkai satu. Hanya setelah halaman dimuat, program akan diizinkan untuk melanjutkan eksekusinya. Program berlanjut sesuai contoh ini, dan pada suatu saat program mengakses halaman nol sekali lagi. Halaman ada di memori, jadi perangkat keras menerjemahkan alamat virtual ke alamat sebenarnya dan program berlanjut tanpa mengetahuinya bekerja di lokasi berbeda di memori. Beberapa waktu kemudian, program mengakses halaman enam. Sekali lagi, halaman tidak ada di memori, jadi perangkat keras memberi sinyal ke sistem operasi, yang memuat halaman, dan seterusnya hingga program selesai.



Gambar 4.19. Contoh Pages dan frames

Sistem operasi yang memuat halaman yang diperlukan menangani semua bingkai dalam sistem, dan menggunakan berbagai algoritme untuk mengoptimalkan eksekusi program. Misalnya, bertanggung jawab untuk mencegah situasi di mana satu program mendominasi sistem dan menggunakan semua memori yang tersedia. Dalam kasus lain ketika frame tidak tersedia, sistem operasi mungkin mendahului frame dari program lain yang berjalan di sistem.

Algoritma yang digunakan oleh operasi sistem dalam menangani bingkai, tetapi ini berada di luar cakupan buku ini. Namun, ada masalah lain yang relevan untuk insinyur sistem. Sebagai bagian dari praktik pengembangan perangkat lunak yang baik, program yang dikembangkan harus modular dan dibagi menjadi fungsi atau kelas dan metode. fungsi atau metode harus relatif kecil, yang mengurangi kompleksitas dan memudahkan pengembangan dan pemeliharaan di masa mendatang. Memahami mekanisme memori virtual memberikan alasan tambahan untuk menjaga fungsi dan metode relatif kecil. Dengan potongan kode, ada kemungkinan lebih besar seluruh fungsi atau metode akan berada di satu halaman, yang akan menurunkan jumlah halaman yang dimuat selama eksekusi. Untuk menangani terjemahan alamat, prosesor memiliki unit khusus (***Memory Management unit*** [MMU]). tambahan dari ALU dan *Control Unit* yang disebutkan sebelumnya (Gambar 4.20).

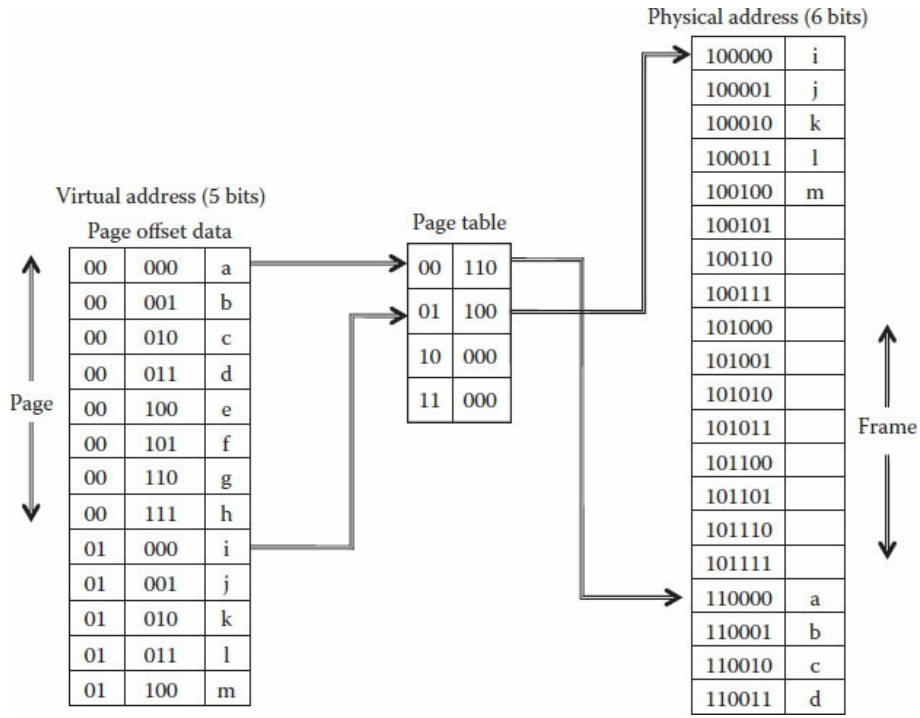


Gambar 4.20 Address translation

Gambar 4.20 menggambarkan MMU, bertanggung jawab menangani komunikasi ke dan dari memori serta menerjemahkan *virtual address* yang disimpan dalam instruksi di alamat nyata di memori. Sebuah program yangdieksekusi oleh prosesor perlu akses beberapa *virtual address*. Alamat berisi variabel yang dibutuhkan oleh instruksi pelaksana, atau instruksi berikutnya yang dieksekusi. Program berhubungan dengan ruang alamatnya sebagai *array sel* yang berdekatan menggunakan *virtual address*. ***Control Unit*** bertanggung jawab mengambil instruksi berikutnya serta mengambil operan yang diperlukan, mengirimkan *virtual address* ke MMU. Diterjemahkan untuk mengetahui lokasi sebenarnya. Sebagai bagian dari terjemahan, MMU memeriksa apakah halaman tersebut ada dalam memori. Dengan asumsi ada di memori, sel yang diperlukan akan ditransfer dari memori ke ***Control Unit***.

Untuk memahami proses penerjemahan, contoh sederhana berdasarkan sistem ***teoritis***. Dengan asumsi ini bekerja dengan ***8-byte*** halaman dan setiap program menggunakan tidak lebih dari empat halaman, bahwa kemungkinan alamat terbesar yang didukung oleh sistem adalah 31 (alamat mulai 0- 31). Ini berarti ruang alamat dalam sistem ini didasarkan pada 5 bit.\* 5 bit ini akan dibagi menjadi dua kelompok.

Grup pertama akan digunakan untuk menentukan nomor halaman, dan karena program tidak dapat menggunakan lebih dari empat halaman, bagian ini akan menggunakan 2 bit Kedua bagian akan digunakan untuk menentukan perpindahan di halaman. Karena ukuran halaman adalah 8 byte, ruang alamat akan membutuhkan 3-bit (Gambar 4.21). Contoh nyata yang sangat sederhana mungkin sistem yang digunakan untuk nomor katalog di gudang. Asumsikan bahwa setiap komponen memiliki angka lima digit unik yang dapat dilihat sebagai gabungan dari dua nilai. Satu nilai (dua digit) menentukan ruangan tempat komponen berada, dan nilai kedua (tiga digit) menentukan rak yang relevan. Dengan memeriksa nomor katalog, sangat jelas di mana komponen tertentu berada. Karena ruang alamat virtual berurutan, alamat program berurutan dan meskipun demikian, alamat dapat dibagi menjadi dua angka. Gambar 4.21 menggambarkan program teoritis yang menggunakan 13 byte.



Gambar 4.21 Translation process.

Alamat yang dibutuhkan program adalah 00000–01100. Sisi kiri gambar menjelaskan program byte. Dua kolom paling kiri mewakili alamat, sedangkan kolom kanan berisi karakter yang berbeda sehingga mudah untuk membedakan antara byte yang berbeda. Alamat virtual adalah gabungan dari keduanya, kolom paling kiri. Misalnya, karakter ***“a”*** di alamat 00000, sedangkan karakter “k” di alamat 01010. kolom kiri. Misalnya, karakter “a” di alamat 00000, sedangkan karakter “k” di alamat 01010.

Bagian berikutnya dari diagram (tengah gambar) adalah tabel halaman. Ini adalah komponen utama dari terjemahan alamat, karena menyimpan informasi pemetaan. Untuk setiap halaman, tabel berisi lokasi sebenarnya di memori, serta informasi apakah ada di memori atau tidak. Dalam kasus khusus ini, halaman nol program terletak di bingkai enam (biner 0110) di memori, sedangkan halaman satu program terletak di bingkai nomor empat (biner 0100) di memori.

Sisi kanan gambar adalah tata letak real memori dengan halaman yang dimuat serta bingkai kosong. Terlihat dari tabel halaman, bahwa halaman nol program sebenarnya berada pada frame enam (110), dan halaman pertama program berada pada frame empat. Penerjemahan memori terdiri dari beberapa langkah sederhana:

* 1. Ekstrak nomor halaman dari alamat virtual
  2. Gunakan nomor halaman sebagai indeks untuk tabel halaman
  3. Tarik keluar lokasi baru halaman
  4. Jika halaman tidak ada dalam memori, lakukan interupsi.
  5. Jika halaman tidak ada dalam memori, lakukan interupsi Gabungkan alamat halaman baru dengan perpindahan dari alamat virtual
  6. Nilai gabungan mewakili lokasi memori sebenarnya.

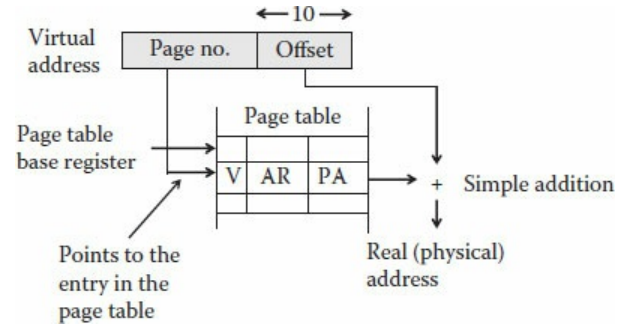
Membagi alamat menjadi dua bagian (nomor halaman dan perpindahan di dalam halaman) adalah alasan di balik persyaratan bahwa ukuran halaman adalah pangkat dua. Dalam sistem nyata di mana ruang alamat besar, misalnya, 4 GB (arsitektur 32-bit), MMU membagi alamat menjadi dua kelompok ini. Jumlah bit yang dialokasikan untuk perpindahan didorong oleh ukuran halaman. Ukuran halaman umum adalah 4 KB, tetapi angka lain seperti 8 KB juga tersedia.

Oleh karena itu, jika sistem 4 GB menggunakan halaman 1 KB (1024 byte), maka ukuran halaman akan membutuhkan 10 bit dan 22 bit sisanya digunakan untuk nomor halaman. Saat menggunakan halaman 4 KB, sistem akan memiliki 12 bit yang ditetapkan untuk perpindahan halaman dan 20 bit jumlah halaman. Setiap alamat harus diterjemahkan dan terjemahan harus mengakses tabel halaman, sebagian besar sistem operasi telah menambahkan beberapa fungsionalitas ke entri tabel halaman.

Setiap entri tersebut mencakup informasi tambahan yang digunakan untuk keamanan tambahan dan operasi yang efisien. Misalnya, untuk menambahkan mekanisme kontrol akses yang akan memberikan informasi tentang proses yang dapat mengakses halaman ini dan tentang proses lain yang tidak diizinkan. Fungsionalitas lainnya adalah *ready* bit, jika disetel, berarti halaman telah dimuat. Sebaliknya jika bit clear berarti masih dalam proses loading belum tersedia, padahal frame sudah dialokasikan ke program. Contoh lain adalah ketika halaman dimuat tetapi untuk beberapa alasan sistem operasi memutuskan untuk mendahuluinya karena ruang diperlukan oleh program lain. Dalam kasus seperti itu, meskipun halaman tersebut masih ada, halaman tidak tersedia karena ditimpa oleh konten lain.

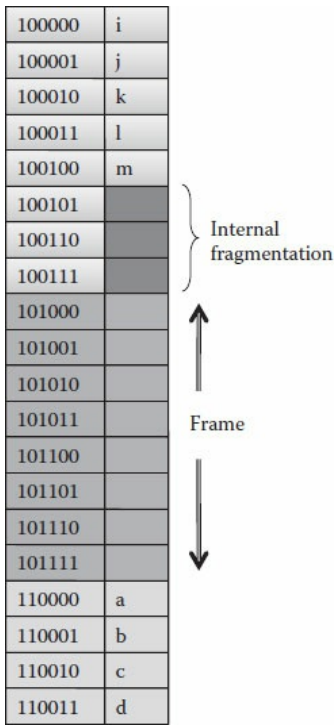
Fungsionalitas tambahannya dapat digunakan tidak hanya untuk kontrol akses tetapi juga untuk menentukan jenis akses. Untuk menetapkan sedikit memberi sinyal jika halaman ditulis oleh program, atau jika hanya dapat dibaca. Dalam kasus seperti itu, jika program pengakses mencoba menulis atau memodifikasi halaman, perangkat keras akan mengeluarkan interupsi dan sistem operasi harus memutuskan situasinya.

Gambar 4.22 menggambarkan tabel halaman, yang digunakan sebagai bagian dari terjemahan alamat. Ini berisi untuk setiap halaman virtual ***physical address*** (PA), tetapi di samping itu, setiap entri menyertakan informasi tentang hak akses serta bit validitas (V). Dalam contoh khusus ini, ukuran halaman adalah 1 KB. Setiap program, dan sebenarnya setiap proses yang berjalan dalam sistem, memiliki tabel halamannya sendiri. Itu berada di memori, dan register khusus menunjuk ke awal. Dengan menambahkan nomor halaman virtual ke isi register ini, sebuah pointer dibuat. Pointer ini digunakan untuk menemukan lokasi sebenarnya (PA) halaman.



Gambar 4.22. Page table

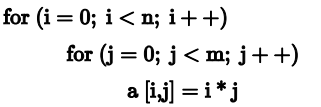
Selain banyak manfaat ***virtual memory***, penting untuk menambahkan sistem seperti tidak membuat "lubang" di memori. Ini berarti sistem ***virtual memory*** tidak memerlukan mekanisme yang mengatur lubang dengan memindah dan memadatkan (lihat bagian “Partisi” dalam bab ini). Ketika semua bingkai berukuran sama, tidak ada lagi fragmen eksternal.\* Namun, mungkin masih ada fragmen internal (Gambar 4.23).



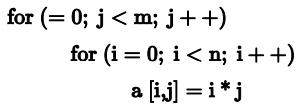
Gambar 4.23. Internal fragments

Angka menggambarkan *real memory* ketika program teoritis dimuat. Contoh, setiap halaman berisi 8 byte, tetapi halaman kedua hanya membutuhkan 5 byte. Oleh karena itu, frame berisi halaman ini (*frame nol*) akan memiliki 3 byte yang tidak digunakan (lubang). Memahami cara memori diatur sangat penting bagi pengembang perangkat lunak juga. Diasumsikan bahwa matriks dua dimensi harus diinisialisasi.

Loop bersarang (*nested loops*) mengakses semua elemen matriks. Pemrograman sederhana, tidak banyak perbedaan indeks mana yang akan berada di loop dalam dan mana yang akan berada di loop luar. Dengan kode semu berikut:



setara dengan kode berikut :



Kedua loop ini akan menghasilkan hasil yang sama. Namun, mempertimbangkan lokalitas referensi dan fakta bahwa *memori virtual* (dan *cache*) diatur dalam halaman, maka akan ada perbedaan dalam waktu eksekusi kedua kode. Tergantung pada bahasa pemrograman tertentu dan cara mengalokasikan ***array*** dalam matriks, tetapi biasanya lebih baik memiliki indeks kedua di dalam loop. Dalam kasus ini, kode pertama menghasilkan eksekusi yang lebih cepat karena akses elemen data lebih berurutan. Beberapa kompiler modern mendeteksi jika kode tidak dioptimalkan.

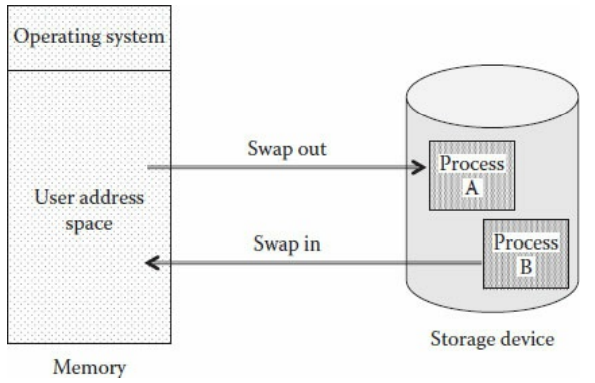
1. SWAP

Sistem operasi modern mengelola banyak aspek sistem dengan penekanan khusus untuk memastikan operasi yang tepat dan efisien. Salah satu kegiatan tersebut terkait dengan mengidentifikasi situasi meronta-ronta dan mengambil langkah-langkah yang diperlukan untuk memperbaikinya. Seperti yang telah dijelaskan sebelumnya, ***thrashing*** terjadi ketika berbagai proses bersaing untuk sejumlah kecil frame. Pada gilirannya proses memuat halaman yang diperlukan untuk eksekusinya; namun, selain terus memuat halaman, sistem melakukan pekerjaan nyata yang sangat sedikit karena sebagian besar Sumber Daya sistem terkait dengan pemuatan halaman.

Beberapa sistem operasi dapat mengidentifikasi pola-pola ini, menemukan satu atau lebih proses bermasalah, dan menangguhkan sementara satu atau lebih proses ini. Penangguhan dicapai dengan menggulirkan proses keluar dari memori dan menulis semua halaman (atau segmen) pada disk. Jika ini adalah sistem berbasis segmen, meskipun program menggunakan lebih banyak memori, biasanya peluncuran ini akan cukup sederhana karena segmen adalah bagian memori yang berkesinambungan.

Sistem berbasis halaman program menggunakan lebih sedikit memori; namun, sistem operasi harus "**mengumpulkan**" halaman yang tersebar di seluruh memori. Setelah beberapa waktu, sistem operasi akan mengembalikan program yang diluncurkan, dengan asumsi konflik pada frame yang tersedia akan minimal. Dalam banyak kasus, situasi ini disebabkan oleh campuran program yang dijalankan secara paralel. Jika sebagian program memuat banyak halaman, dapat menyebabkan situasi tersebut, sedangkan jika beberapa program tidak perlu memuat, dapat dikendalikan. Proses meluncurkan seluruh program ini disebut swap out (***roll out***), dan proses mengembalikan program ke memori disebut *swap-in* (***roll in***). Perbedaan signifikan antara eksekusi standar, misalnya, dari sistem berbasis halaman dan *swapping*.

Saat program sedang berjalan, beberapa halaman diminta, dan karena halaman ini tidak ada di memori, terjadi kesalahan halaman. Proses ditunda dan proses lain diizinkan untuk bekerja. Sistem operasi memuat halaman yang diperlukan, dan hanya setelah dimuat, proses akan dimasukkan ke dalam antrian proses yang menunggu untuk dijalankan. Dalam beberapa kasus, ketika sebuah frame harus ditetapkan, sistem operasi harus mendahului sebuah frame yang dimiliki oleh beberapa proses lain. Jika itu adalah halaman data dengan konten yang telah diubah, sistem operasi harus menulis konten kembali ke disk sebelum bingkai dibebaskan. Ini adalah situasi normal ketika proses dijalankan dalam sistem. Saat sebuah proses di-swap, semua halamannya ada di disk, dan tidak ada di antrian proses\* mana pun yang menunggu prosesor atau event. Sistem, tentu saja, tahu tentang proses-proses yang ditukar ini dan kapan situasinya akan berubah. Mungkin jumlah proses yang berjalan secara paralel akan berkurang atau campuran proses yang berjalan akan berubah, yang akan menambah jumlah frame bebas. Gambar 4.23 menggambarkan proses swapping, di mana seluruh proses ditulis ke disk (***swapped out***) atau dimuat dari disk (***swapped in***).



Gambar 4.23 *swapped out & swapped in*

**H. MEMORY PERFORMANCE**

Hukum Moore (***Moore’s law***) yang memprediksi evolusi teknologi, berkaitan dengan jumlah transistor dalam chip. Karena memori diimplementasikan menggunakan transistor, maka hukum memberikan prediksi yang baik mengenai ukuran memori di masa depan. Banyak PC saat ini didasarkan pada arsitektur 64-bit di mana 48 bit digunakan untuk menangani memori. Ini mewakili ruang alamat besar yang, bahkan dengan prediksi Moore, akan cukup untuk waktu yang lama.\*

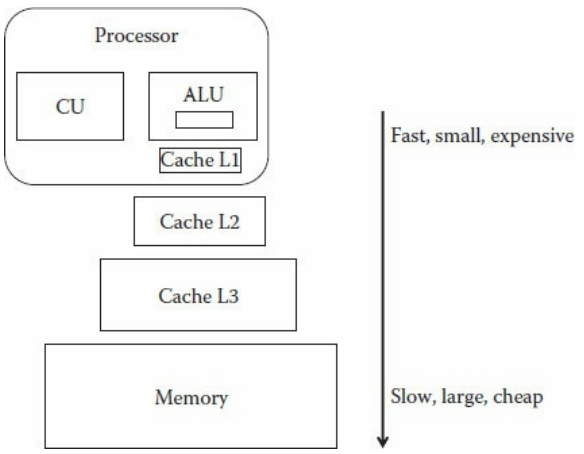
Ukuran memori bukanlah masalah yang sebenarnya dan paling utama perkembangan yang mengatasi masalah ini ketika memori sangat mahal. *Virtual Memori* adalah contoh terbaik dari solusi yang dirancang untuk meningkatkan pemanfaatan memori dan memungkinkan menjalankan aplikasi besar sambil memuat hanya bagian yang relevan. Sementara *hukum Moore* (***Moore’s law***) dapat digunakan untuk memprediksi ukuran memori, gagal memprediksi waktu akses memori, yang dulu dan masih merupakan hambatan. Solusi yang diusulkan adalah pengembangan teknologi RISC, yang mencoba meminimalkan akses memori dengan mempromosikan penggunaan register; menyediakan banyak register dan instruksi berbasis register.

Untuk permintaan terus menerus memori tambahan yang lebih besar dan lebih cepat, upaya besar dilakukan untuk memajukan teknologi. Salah satu perkembangan ini adalah pengenalan **S*ynchronous Dynamic Random Access Memory*** (SDRAM), selain standar DRAM, menyediakan mekanisme sinkronisasi dengan bus komputer.

Peningkatan kecepatan dicapai dengan mengimplementasikan pipeline, yang diimplementasikan di prosesor. Memori dapat menerima permintaan baru saat permintaan sebelumnya masih diproses. Perkembangan berikutnya dalam teknologi memori adalah **SDRAM** kecepatan data ganda (**DDR SDRAM),** yang dicirikan oleh fakta bahwa ia dapat mentransfer dua elemen data pada setiap siklus, sehingga menyediakan bandwidth hampir dua kali lipat dari SDRAM generasi sebelumnya. Perkembangan lebih lanjut dalam teknologi memori menghasilkan **DDR** generasi berikutnya, yang disebut **DDR2 SDRAM**. Karakteristik utama **DDR2** dibandingkan dengan DDR adalah bahwa ia menggandakan bandwidth sekaligus menurunkan konsumsi daya. Teknologi tidak berhenti di DDR2, sejak 2007 standar baru telah digunakan yaitu DDR3. Seperti pendahulunya, DDR3 menggandakan bandwidth yang dicapai oleh DDR2. Pada 2015, **DDR4** generasi berikutnya mulai muncul.

Perkembangan hanya berkaitan dengan peningkatan kecepatan memori utama, yang merupakan salah satu bagian dari hierarki memori yang digunakan dalam sistem komputer. Ide hierarki memori berasal dari pengembangan ***virtual memori***. Sebagian besar program yang sedang berjalan berada di disk dan hanya yang diperlukan untuk menjalankan memori. Konsep yang sama diterapkan lebih lanjut dengan menyediakan beberapa tingkat memori. Level yang dekat dengan prosesor bisa lebih kecil tetapi harus sangat cepat, sedangkan level yang jauh dari prosesor harus besar untuk menampung lebih banyak data tetapi juga bisa lebih lambat Gambar 4.24 menggambarkan komponen memori yang disusun menurut urutan dan kecepatan:

* Jenis memori pertama yang lebih kecil dan tercepat adalah register. Mereka berada di dalam prosesor itu sendiri sehingga waktu aksesnya sangat cepat (satu siklus); Namun, kapasitas mereka terbatas. Bahkan dengan arsitektur 64-bit yang mengimplementasikan file register, ada banyak lagi tujuan umum juga sebagai register tujuan khusus; jumlah puluhan register dibandingkan gigabyte memori.
* Biasanya, ada beberapa level memori cache
* Tingkat berikutnya terdiri dari memori (RAM) itu sendiri.
* Tingkat terakhir adalah disk, sangat lambat dibandingkan dengan semua tingkat lainnya tetapi di sisi lain menyediakan kapasitas yang hampir tak terbatas.



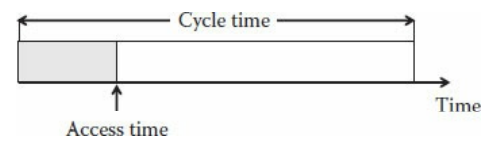
Gambar 4.24 Memory hierarchy.

Paralelisme dan pipelining adalah konsep yang diperkenalkan sebagai mekanisme untuk meningkatkan kecepatan dan bandwidth memori. Sebagian besar memori, terutama tipe DRAM, menggunakan teknologi yang perlu di-refresh, sehingga data yang disimpan harus ditulis ulang secara berkala. Ini berarti bahwa memori menggunakan dua skema waktu yang berbeda :

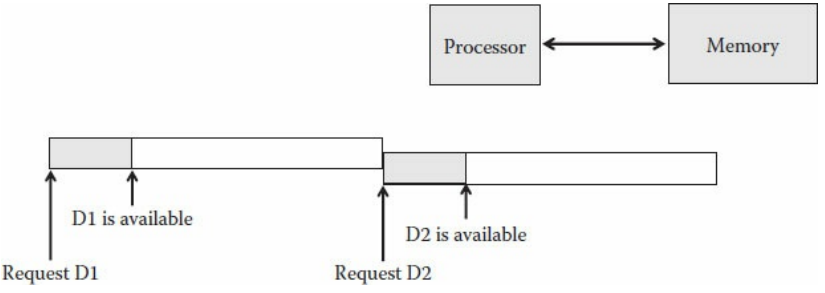
* Waktu akses (***Access time***), yang menentukan waktu yang berlalu dari saat permintaan dikirim hingga saat data diterima. Waktu ini dapat dilihat sebagai waktu respons memori. Waktu respons menentukan jumlah waktu yang berlalu antara pengiriman permintaan dan waktu hasil ditampilkan di layar. Dalam hal memori, waktu akses mengukur waktu respons memori.
* Waktu siklus (***Cycle time***), yang menentukan waktu antara dua permintaan berurutan atau jumlah waktu yang berlalu di antara dua permintaan berurutan yang dibuat ke memori. Untuk mengurangi biaya, memori (DRAM) menggunakan teknologi "destruktif" untuk memelihara data, perlu di-refresh.

Untuk lebih menjelaskan dua skema waktu, contoh asumsi bahwa gaji seseorang ditransfer ke rekening banknya. Jika orang tersebut mencoba untuk menarik jumlah gaji penuh dengan menggunakan ATM atau cek transaksi akan seketika. Ini mewakili waktu akses (mengakses uang di akun). Namun, jika orang tersebut mencoba menarik kembali jumlah tersebut, dia harus menunggu siklus pembayaran berikutnya. Ini mewakili waktu siklus, yang menentukan jumlah waktu antara dua siklus penarikan. Gambar 4.25 menggambarkan dua skema waktu. Gambar 4.26 memberikan penjelasan tambahan dan menunjukkan dua akses memori berurutan.

D1 diminta dan dikirimkan. Sebelum D2 dapat diminta, waktu siklus harus berlalu.



Gambar 4.25 Skema pengaturan waktu memory



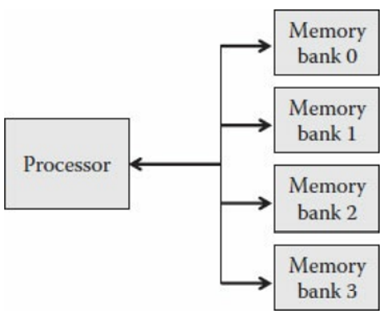
Gambar 4.26 “Standard” *access pattern.*

Jika program membaca data dari memori, asumsinya adalah bahwa setelah membaca satu sel bank akan terkunci. Program dapat membaca sel berikutnya milik bank lain. Setelah membaca bank kedua akan dikunci juga, tetapi program dapat membaca sel lain dari bank yang tidak terkunci. Gambar 4.27 menggambarkan sistem dengan empat bank memori. Menggunakan arsitektur ini dan dengan asumsi program mengakses memori secara serial, waktu akses yang dicapai dijelaskan Gambar 4.28.

Jika rasio antara waktu akses memori dan waktu siklus memori adalah 1:4, itu berarti bahwa empat bank akan diperlukan untuk memastikan pembacaan berurutan

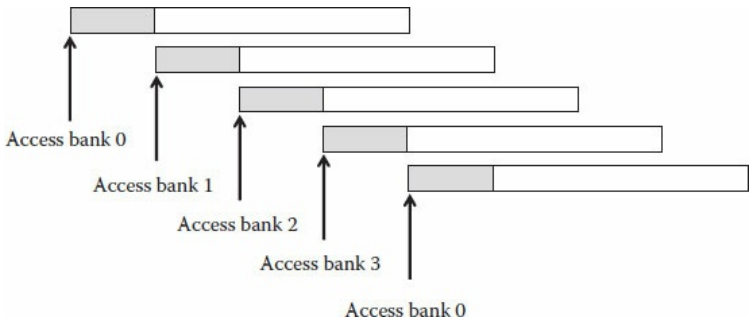
tanpa penundaan. Gambar 4.27, yang menjelaskan arsitektur dari memori multibank, Gambar 4.28 menggambarkan waktu membaca sel berurutan. Setiap persegi panjang mewakili *access time*, asumsinya program membaca alamat pertama yang dimiliki oleh bank 0. Prosesor meminta konten sel dan setelah beberapa waktu (*access time*) mendapatkan data. Ini diwakili oleh persegi panjang kecil di sebelah kiri. Setelah pengiriman data, bank terkunci untuk beberapa waktu tambahan.

Jika program, seperti dalam banyak kasus dikenyataannya, membutuhkan sel berikutnya yang berurutan, prosesor akan menangani memori, meminta konten sel berikutnya. Sekali lagi, setelah beberapa saat (waktu akses), tanggal akan tersedia; namun, bank akan dikunci untuk pembacaan tambahan. Proses ini berlanjut dengan sel ketiga dan keempat. Ketika program membutuhkan konten sel kelima, yang berada di bank yang sama dengan sel pertama, bank dibuka sekali lagi karena berhasil memulihkan data. Seperti sebelumnya, rasio waktu menentukan jumlah bank yang harus diterapkan untuk memastikan pembacaan berurutan tanpa penundaan.

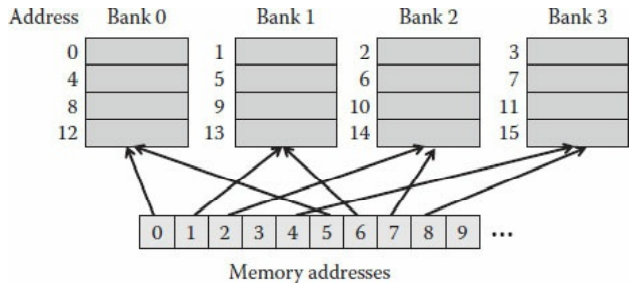


Gambar 4.27 Arsitektur memori empat bank

Siklus memori adalah empat kali waktu akses memori, berarti setelah membaca empat sel, bank pertama siap sekali lagi, dalam mengimplementasikan mekanisme bank, setiap bank harus mandiri dengan cara yang dilakukan di CPU, di mana instruksi dieksekusi oleh beberapa unit otonom independen (seperti IF, ID, EX, WB).

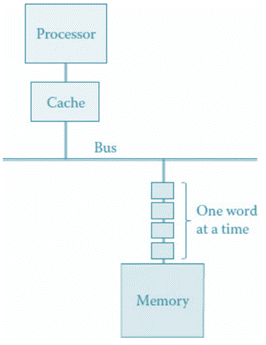


Gambar 4.28 Memory access time (4 banks).



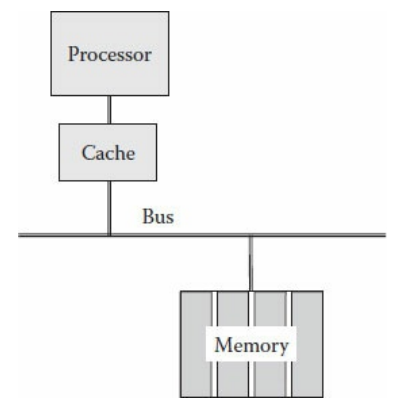
Gambar 4.29 Bank organisasi.

Arsitektur memori di mana MMU mentransfer satu kata\* pada satu waktu (Gambar 4.30). Biasanya, ini adalah pendekatan yang sangat sederhana dan murah



Gambar 4.30 One word at a time.

* Sebuah arsitektur memori yang mentransfer blok (beberapa kata) pada suatu waktu (Gambar 4.31). Jenis memori ini membutuhkan bandwidth yang lebih lebar, yang lebih mahal; namun, ia mampu mentransfer lebih banyak data.
* Memori yang dibangun menggunakan metode interleaving (Gambar 4.31), yang menyediakan bandwidth yang lebih baik karena pembagian internal.



Gambar 4.*31 Interleaving.*

**H. MEMORY PERFORMANCE**

Seperti yang telah dinyatakan dalam bab ini, memori digunakan untuk menyimpan program yang akan dieksekusi serta data yang diperlukan untuk eksekusi. Oleh karena itu, kecepatan memori secara langsung terkait dengan kinerja sistem. Meskipun dalam dekade pertama, sebagian besar upaya teknologi dihabiskan untuk meningkatkan kinerja prosesor, dalam dua dekade terakhir beberapa upaya telah ditujukan untuk meningkatkan kinerja memori juga. Saat ini, memori cache diimplementasikan menggunakan ***fast static memory*** (SRAM), dan memori utama diimplementasikan menggunakan ***dynamic*** RAM (DRAM). Setiap chip memori berisi jutaan sel dan terhubung ke bus memori, yang merupakan kabel dengan banyak kabel yang berjalan secara paralel dan digunakan untuk mentransfer beberapa byte dalam satu siklus. Lebar bus / jumlah kabel dimilikinya, menentukan jumlah bit yang akan ditransfer secara paralel. Dalam arti, menyerupai jumlah jalur di jalan raya. PC modern menggunakan bus 64-bit, yang berarti bahwa dalam setiap siklus, 8 byte ditransfer.

Selain peningkatan kapasitas memori, strukturnya juga berubah. Struktur pertama adalah susunan sel satu dimensi, ingatan modern didasarkan pada matriks dua dimensi. MAR dalam sistem modern dibagi menjadi dua register : *row address strobe* (RAS) dan *column address strobe*. Dikombinasikan bersama menyediakan alamat dua dimensi yang diperlukan. Tren miniaturisasi teknologi yang mempengaruhi seluruh industri elektronik telah mempengaruhi teknologi memori juga.

Jika, empat dekade lalu, sebuah chip memori menyertakan 1000 bit, memori yang digunakan saat ini didasarkan pada 4 GB. Faktor penting lainnya adalah kebutuhan listrik. Di dulu, memori membutuhkan 5 V, dan saat ini persyaratannya sedikit di atas 1 V. Ini lebih rendah konsumsi listrik memberikan banyak penggunaan baru untuk memori, terutama peralatan yang tidak selalu terhubung ke stopkontak listrik.

Semua operasi sistem disinkronkan oleh jam internal. Berbagai bus, yaitu dimaksudkan untuk menjembatani perangkat sistem dan mentransfer data di antara mereka, harus disinkronkan juga. Biasanya, perangkat sinkron lebih cepat daripada perangkat asinkron. Jika perangkat asinkron perlu mentransfer data, baik itu harus menunggu perangkat lain atau yang lain perangkat harus menunggu.

Penundaan ini tidak ada dalam kasus perangkat sinkron. Ini adalah alasan bahwa standar baru dikembangkan\* untuk DRAM sinkron (SDRAM). Tujuan utamanya adalah untuk meminimalkan siklus menunggu yang ada dalam transfer asinkron. Standar menggunakan 100 MHz saluran, yang berarti mentransfer blok delapan byte 100 juta kali per detik (atau transfer kecepatan 800 MB/detik).

Seperti halnya teknologi lainnya, generasi berikutnya, yang membahas masalah ini dari sudut yang berbeda, mengikuti dengan cermat. Pada akhir 1990-an, teknologi *Rambus* DRAM (RDRAM) telah diperkenalkan. Ide utama di balik teknologi ini adalah bus yang digunakan. Alih-alih itu satu bus menghubungkan semua komponen memori, RDRAM menggunakan jaringan koneksi, yang memberikan kinerja yang lebih baik. Kerugian utama adalah kenaikan harga dan fakta bahwa itu membutuhkan bus memori yang berbeda yang tidak kompatibel dengan yang sudah ada.

Keterbatasan terkait dengan implementasi Rambus menyebabkan definisi standar SDRAM baru: SDRAM DDR (***double data rate***). Versi pertama (DDR1) digunakan peningkatan yang dipinjam dari teknologi prosesor. Penggandaan kecepatan data dengan mentransfer 2 bit waktu yang sama. Versi kedua (DDR2) meningkatkan kecepatan bus dengan menggandakan waktu siklus dan menurunkan konsumsi listrik menjadi 1,8 V. Versi berikutnya (DDR3) meningkatkan bandwidth dan menurunkan konsumsi daya lebih jauh lagi. Ini diperlukan berbagai perangkat seluler baru yang, meskipun ada perkembangan dalam teknologi pembuatan baterai, masih membutuhkan waktu pengoperasian yang lebih lama. Diperkirakan bahwa selama 2016 dispersi luas DDR4 akan dimulai, yang, seperti versi lainnya, akan meningkatkan kinerja sekaligus menurunkan konsumsi daya.

Satu hal yang harus dipertimbangkan berkaitan dengan kompatibilitas. Karena kebutuhan konsumsi daya yang lebih rendah sehingga memori yang dirancang dapat mengakomodasi berbagai kebutuhan perangkat seluler, berbagai versi DDR tidak kompatibel. Ini berarti bahwa sistem yang menggunakan satu teknologi tidak akan dapat meningkatkan yang berikutnya, karena memerlukan motherboard yang berbeda. Hal ini tentu saja memperpendek umur sistem modern, yang merupakan fenomena yang dikenal masyarakat modern seperti juga dapat diamati pada peralatan lain.

**REFERENSI/DAFTAR PUSTAKA**

1. Andrew S. Tanenbaum, Structured Computer Organization Fifth Edition, Pearson Prentice Hall 2005
2. Willam Stallings, Organisasi&ArsitekturKomputerEdisikeenam, Prentice Hall 2003
3. Syahrul, Organisasi dan Arsitektur Komputer, Andi offset 2010,