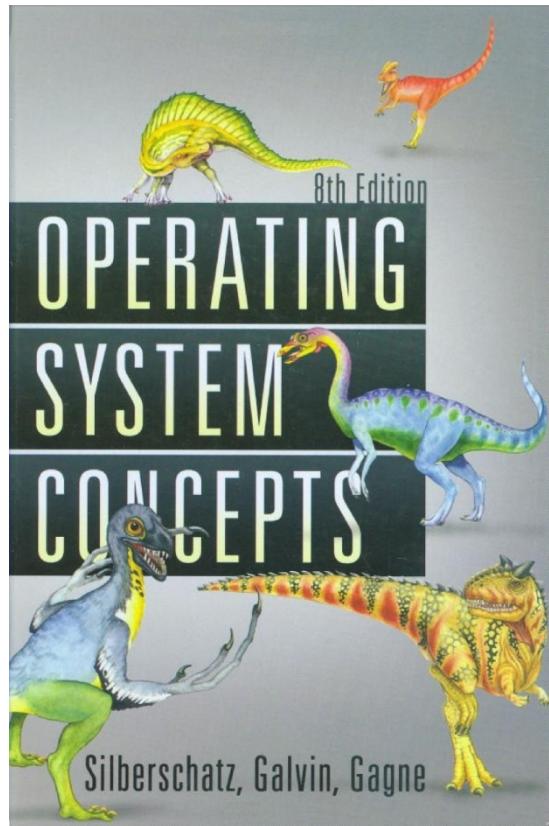


# CATATAN KULIAH SEMESTER 2



LA SURIMI

10/310845/PPA/03471

ILMU KOMPUTER FMIPA  
UNIVERSITAS GADJAH MADA  
YOGYAKARTA

2011

## CHAPTER 1. INTRODUCTION

### 1.1.What Is Operating System?

Secara Umum operating system adalah suatu program yang menjadi perantara antara user dan hardware komputer.

Mengapa harus ada perantara?karena jika seorang user dalam mengoperasikan komputer harus berhubungan langsung dengan hardwarenya maka pekerjaan user akan sangatlah rumit.Sebagai contoh untuk mencetak sebuah huruf dilayar saja dapt melibatkan ribuan perintah.Jika seperti itu pekerjaan user yang lain dapat terbengkalai.

Tujuan akhir dari adanya operating system adalah

- Mengeksekusi program yang dimiliki oleh user  
Tampa adanya operating system maka user yang ingin membuat program harus membuat program dengan bahasa yang dimengerti oleh mesin. Hal itu tentu saja sangatlah sulit,rumit dan memerlukan waktu yang tidak sedikit. Dengan adanya OS maka seorang user cukup membuat program dalam bahasa tingkat tinggi(yang lebih dapat dimengerti oleh manusia) dan diberikan ke OS. Kemudian OS lah yang akan mengeksekusi perintah dalam program sehingga dimengerti oleh mesin/hardware.
- Membuat komputer lebih mudah untuk digunakan(user friendly).
- Membantu user untuk menggunakan Sumber daya hardware dengan lebih efisien.

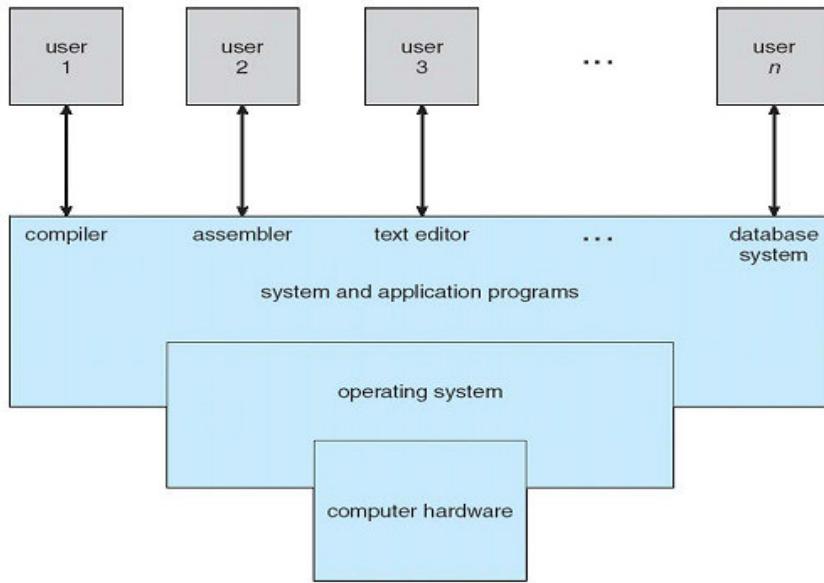
Dari paparan diatas dapat disimpulkan bahwa sebenarnya OS memiliki dua fungsi utama yaitu dapat dilihat dari sisi user dan pengalokasian sumber daya.

### 1.2.Computer System Structure

Sebelum lebih jauh memahami mengenai operating system maka ada baiknya kita mengulang sekilas mengenai structure dari sebuah komputer.Secara umum struktur komputer dibangun oleh empat komponen yaitu

- Hardware  
Penyedia sumber daya/resources bagi proses komputasi  
Contoh:memori,CPU,I/O devices
- Operating System  
Mengendalikan dan mengorganisasikan penggunaan hardware oleh aplikasi dan user.
- Program aplikasi  
Yaitu program yang dibuat oleh user untuk menyelesaikan sebuah permasalahan tertentu.Ada perbedaan mendasar antara program aplikasi dan SO. Analoginya:SO seperti pemerintah yang mengatur masyarakat(program aplikasi).Operating System mengatur dan mengorganisir penggunaan resources oleh aplikasi.Program aplikasi berjalan dengan menggunakan layanan-layanan yang disediakan oleh OS. Pada system operasi modern program aplikasi selalu berjalan diatas OS.
- User:pengguna komputer dapat berupa orang/manusia,mesin,ataupun komputer lain.





Gambar 1.2.1.Empat komponen penyusun struktur komputer

Definisi Operating System sebelumnya belumlah cukup baik. Bedasarkan fungsinya OS dapat didefinisikan

- System Operasi adalah pengalokasi resources  
Yaitu memanage semua resource(CPU, memory dan peripheral yang lain). OS juga harus dapat meresolve konflik. Jika terjadi konflik diantara beberapa aplikasi OS harus dapat memutuskan aplikasi mana yang akan diberikan resources terlebih dahulu sehingga menjamin efisiensi dan keadilan dalam penggunaan resources.
- SO sebagai pengendali program  
Yaitu agar tidak terjadi error dan menghindari penggunaan komputer yang tidak perlu.

Jadi dapat kita lihat pada fungsi yang pertama resource yang diatur sedangkan pada fungsi yang kedua program nya yang dikendalikan. Namun demikian definisi SO sangatlah beragam.

Pada saat kita membeli komputer, biasanya kita dinstalakan sebuah operating system oleh vendor. Tidak jarang beberapa orang menganggap SO adalah semua program yang dapat dilihat olehnya pada komputer tersebut atau pula semua yang diberikan oleh vendor ketika membeli sebuah komputer (biasanya dalam bentuk CD ataupun DVD). Namun sebenarnya tidaklah demikian, tidak semua program yang diinstall/diberikan ke komputer kita adalah SO. Pada saat vendor atau user menginstall OS maka sebenarnya juga ikut terinstall beberapa program aplikasi, misalnya ketika kita menginstall windows biasanya didalamnya sudah termasuk beberapa program aplikasi ataupun system program seperti Windows explorer, notepad dll.

Dalam kuliah ini SO adalah sebuah program yang berjalan dari awal kita menjalankan komputer hingga kita mematikan komputer. Program ini biasanya disebut *kernel*. Kernel diload pada saat komputer boot/startup, oleh sebuah program yang disebut program bootstrap. Program bootstraps biasanya



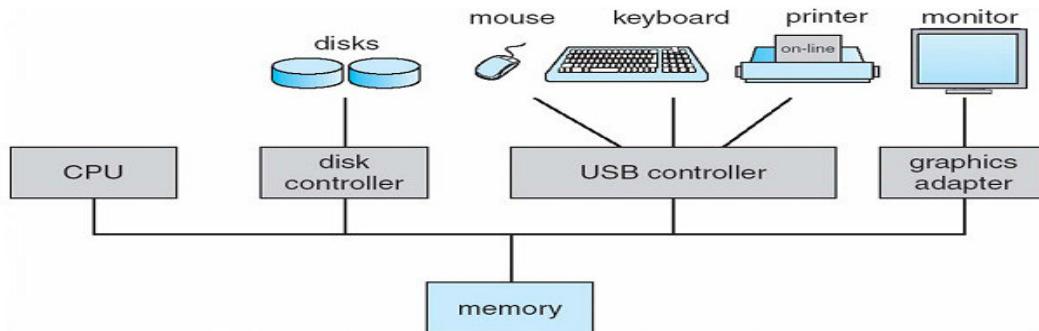
disimpan dalam ROM atau EPROM. Setelah kernel dari system operasi di load maka system operasi akan bersifat event trigger, menunggu melayani user.

### 1.3.Organisasi System Komputer

System Operasi yang akan lebih detail kita bahas kemudian, lingkungannya akan berupa organisasi komputer yang akan dibahas dibawah ini.

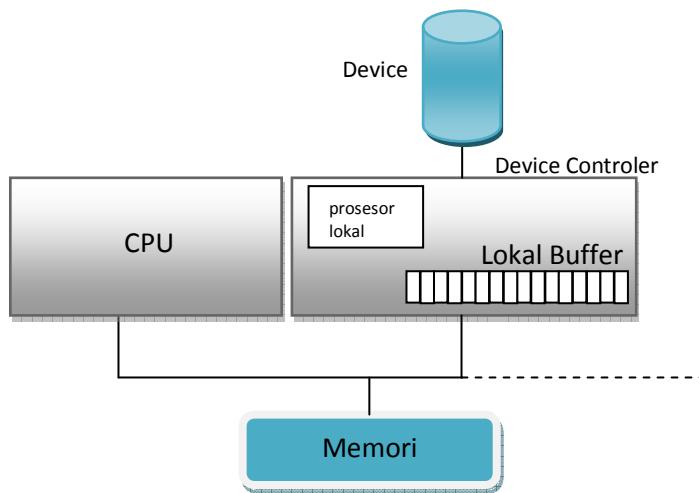
#### Operasi dari system komputer

System dari komputer modern general-purposes terdiri dari satu atau lebih CPU dan beberapa controller device yang terhubung satu sama lain melalui bus ke sebuah shared memori.



Gambar 1.3.1 System Komputer Modern

System operasi yang akan dibangun nantinya mengikuti organisasi seperti gambar diatas. Artinya kita tidak akan menemui sebuah prosesor yang memiliki jalur tersendiri ke memory. CPU dan device controller lainnya terhubung ke memori melalui satu jalur. Jadi bisa saja terjadi kompetisi penggunaan jalur bus. Namun demikian jika operasi CPU dan masing-masing device diatas sedang memproses data di prosesor lokal masing-masing maka semua device tersebut dapat berjalan concurrent. Setiap device memiliki device driver yaitu penghubung antara CPU dan Device controller. Device driver menyediakan cara bagaimana CPU dapat mengakses device controller.



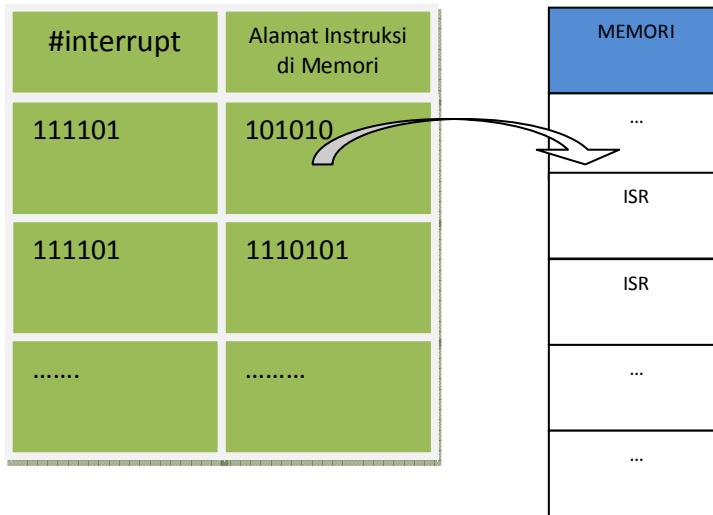
Gambar 1.3.2 Device Controller dengan prosesor dan buffer lokal



Setiap device control bertanggung jawab pada device dengan tipe tertentu. Misalnya kita mempunyai 4 buah hard disk maka hard disk tersebut semuanya dikontrol oleh satu device controller. Setiap device controller memiliki buffer dan prosesor kecil untuk mengakses buffer tersebut. Jadi device-device controller tersebut seperti komputer kecil yang dapat berjalan secara concurrent.

Misalkan CPU memerlukan sebuah data dari disk, maka cerita singkatnya, prosesor kecil di device controller akan menerima perintah dari CPU dan kemudian mengartikannya sebagai cara untuk mengakses disk misalnya putar disk nya 30 derajat dan majukan head 2 cm, kemudian baca data yang terdapat pada baris tersebut lalu pindahkan ke lokal buffer. Perlu diingat bahwa CPU tidak pernah mengambil data langsung dari device tapi selalu dari memory. Setelah data diletakan oleh prosesor kecil device ke buffer maka ia akan menginterrupt CPU bahwa data telah ada dibuffer.

Jika terjadi sebuah interrupt CPU akan menghentikan pekerjaan yang dilakukannya, dan interrupt tersebut menyebabkan control dipindahkan ke interrupt service routine (ISR) melalui interrupt vector yang berisi semua alamat memori dari service rutin yang harus dilakukan oleh CPU jika terjadi interrupt tertentu.



Gambar 1.3.3 Interrupt vektor

Pada saat CPU menghentikan proses yang sedang dilakukannya maka CPU harus mengingat apa yang sedang dilakukannya ketika interrupt itu terjadi, karena setelah ISR selesai, CPU harus kembali lagi melanjutkan pekerjaannya yang tadi di interupsi. Ketika CPU melakukan ISR maka interrupt yang lain yang ingin menginterrupt akan ditolak/didisable. Namun biasanya pada hardware modern terdapat buffer interrupt, jika CPU telah selesai melakukan ISR maka interupsi yang ada di buffer hardware akan dilakukan. Selain dibangkitkan oleh hardware interrupt juga dapat dibangkitkan oleh software. Interrupt yang dibangkitkan oleh software disebut dengan trap, biasanya penyebabnya adalah error (misalnya pembagian dengan nol).

Dari penjelasan diatas dapat dilihat bahwa OS berjalan berdasarkan interrupt atau biasa disebut interrupt driven.

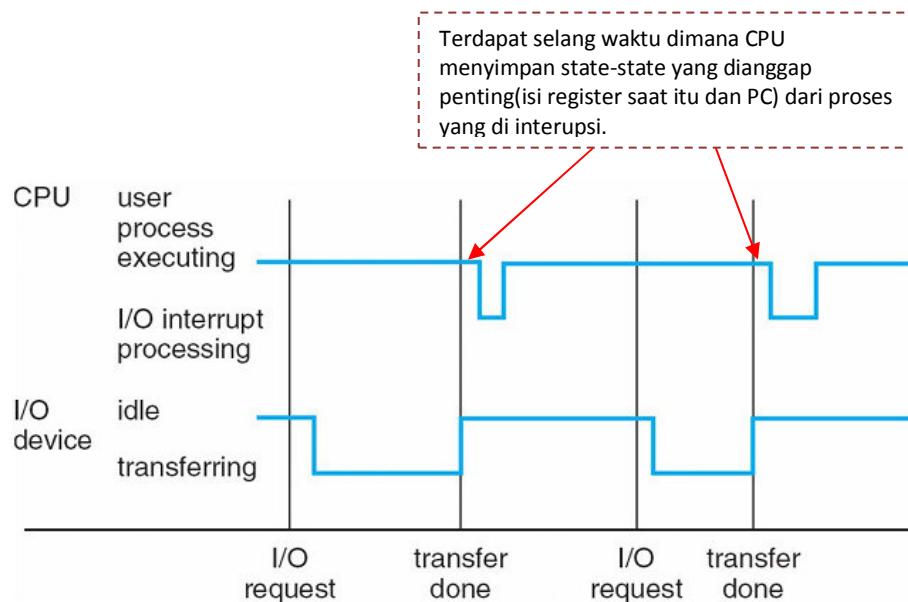
### Handling Interrupt

Pada saat terjadi interrupt maka CPU menghentikan apa yang sedang dikerjakannya, dan menyimpan informasi apa yang sedang dilakukannya. Kemudian berdasarkan nomor interrupturnya CPU akan melihat

interrupt vektor dan kemudian melihat ISR apa yang bersesuaian dengan nomor interrupt tersebut. Yang disimpan oleh CPU pada saat interrupt adalah state-state berupa isi register dan program counter. Program counter adalah berisi perintah apa yang harus dilakukan selanjutnya. State-state tersebut disimpan didaerah tertentu di memori.

Bagaimana CPU mengetahui jenis dari interrupt?

- Cara pertama adalah dengan interrupt vektor, setiap sebuah device baru diinstall, maka device baru tersebut oleh OS diberi nomor interrupt baru dan dipasangkan dengan ISR nya.
- Cara kedua dengan polling. CPU akan bertanya ke setiap device apakah ada yang harus dikerjakannya, secara terus menerus. Biasanya digunakan pada OS lama.



Gambar 1.3.4. Interrupt time line for a single process doing output.

### I/O Structure

Pada sistem operasi lama, setelah I/O mulai berjalan, kontrol akan dikembalikan ke program user hanya jika I/O telah selesai mengerjakan/komplit operasinya. Hal itu berarti selama itu CPU akan idle hingga nanti I/O menginterupt bahwa ia telah selesai mengerjakan tugasnya. Tidak ada I/O yang bekerja parallel di satu waktu.

Pada sistem operasi baru, setelah I/O mulai berjalan, kontrol akan langsung dikembalikan ke user program tanpa menunggu I/O selesai dengan pekerjaannya. Salah satu mekanismenya adalah dengan system call (akan dipelajari nanti). SO memiliki Device status table, yaitu berisikan device-device beserta status, type dan alamat masing-masing dari device-device tersebut. Device status table dapat diubah-ubah OS untuk keperluan interupsi.

### Direct Memory Acces(DMA)

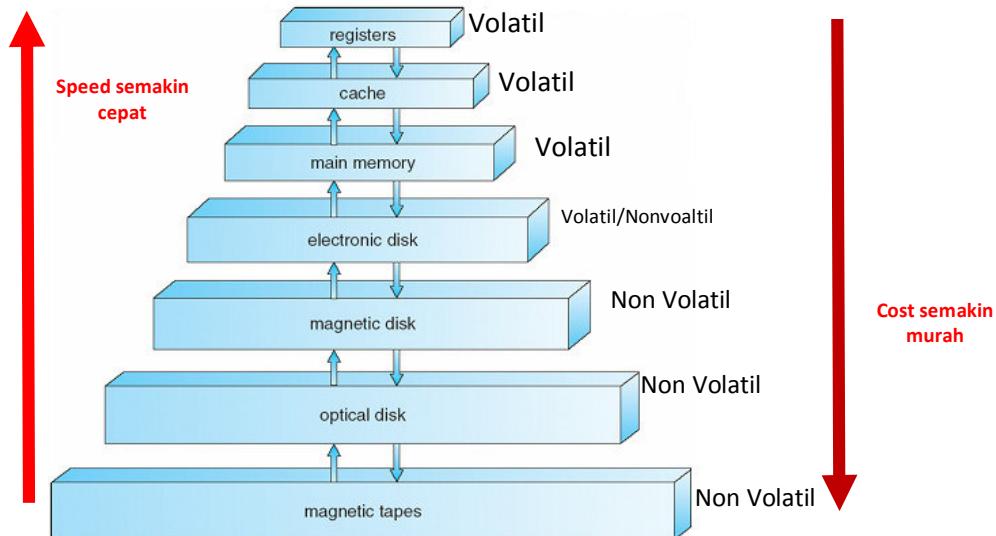
Pada saat di keyboard kita menekan satu huruf maka Device Controller dari keyboard akan memindahkan byte(keyboard byte oriented) huruf tersebut ke buffer kemudian menginterrupt CPU bahwa ada data di buffer. CPU akan merespon interup tersebut seperti yang telah dijelaskan diatas. Biasanya buffer di device controller dibuat panjang agar dapat menampung banyak huruf sekaligus.

DMA digunakan untuk device I/O berkecepatan tinggi (misalnya hardisk , disk dll).DMA bersifat blok oriented bukan byte oriented.Byte oriented maksudnya setiap memindahkan byte ke buffer maka device controller akan menginterrupt CPU.Hal seperti itu akan memerlukan waktu yang cukup lama. Dengan DMA device controller akan memindahkan data dari buffernya ke memori dalam bentuk blok langsung tanpa harus menginterrupt CPU, begitu juga sebaliknya dari memori ke buffer device controller. Jadi dibutuhkan hanya sekali interrupt ke CPU yaitu ketika perpindahan data tersebut selesai.

### Storage Structure

Main memory adalah satu-satunya media penyimpanan yang kapasitasnya besar yang dapat langsung diakses oleh CPU.Biasanya bersifat volatil dan metode aksesnya secara random.Secondary storage yaitu perluasan dari main memory,biasanya bersifat nonvolatil dan berkapasitas besar.Contoh secondary storage adalah hardisk, disk dan flashdisk. Sebenarnya flash disk bersifat volatil,namun didalam sebuah flash disk biasanya terdapat power suply kecil(batrei) yang menjaga data agar tetap tersimpan. Jika sebuah falsh disk tidak pernah dipakai dalam waktu yang lama maka data dalam FD tersebut dapat hilang.

Storage memiliki hirarki yang tersusun berdasarkan cost,volatily dan speed.



Gambar 1.3.5.Hirarki dari storage-device.

## 1.4.Computer Arsitektur

### Single Processor and Multyprocessor

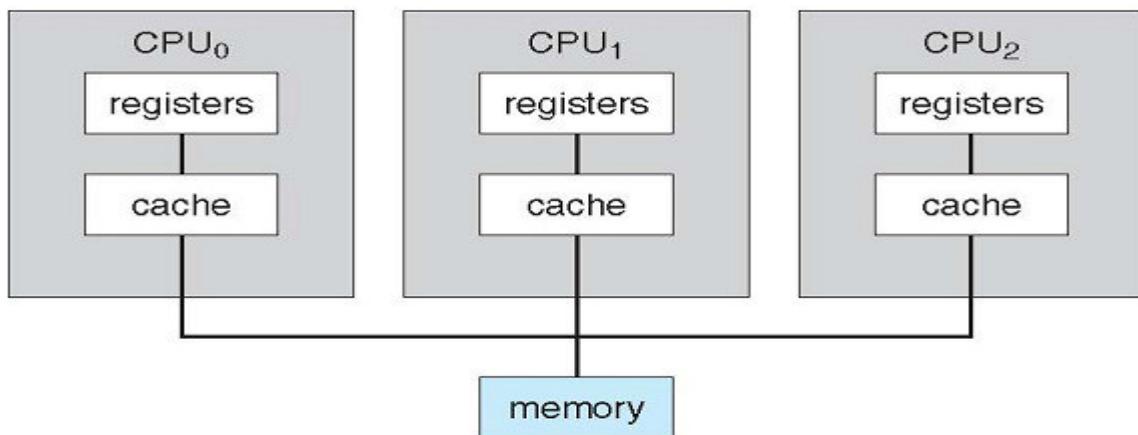
Beberapa System komputer pada masa sekarang masih menggunakan *singel prosesor general purposes* namun sudah semakin banyak system komputer yang menggunakan multiprosesor. System yang menggunakan multiprocessor biasa juga disebut sebagai parallel system.

Keuntungan yang diperoleh menggunakan multiprocessor adalah

- Meningkatkan jumlah throughput. Throughput adalah jumlah pekerjaan yang dapat dilakukan dalam satu waktu
- Economy of scale.  
Jika membandingkan antara system multyprocessor dan gabungan beberapa system single processor dari segi cost maka akan lebih hemat yang menggunakan system multiprocessor. Karena pada system multiprocessor, processor-processor dapat menggunakan peripheral,storage,powersuplay secara bersamaan(dishare). Sedangkan pada gabungan beberapa system single processor, maka untuk masing-masing procesornya harus menggunakan peripheral, storage, powersuplay tersendiri.
- Increased reliability  
Jika satu processor bermasalah maka processor yang lain dapat menangani pekerjaan yang harus dilakukan prosesor yang bermasalah.

Ada 2 type multyprocessor yaitu

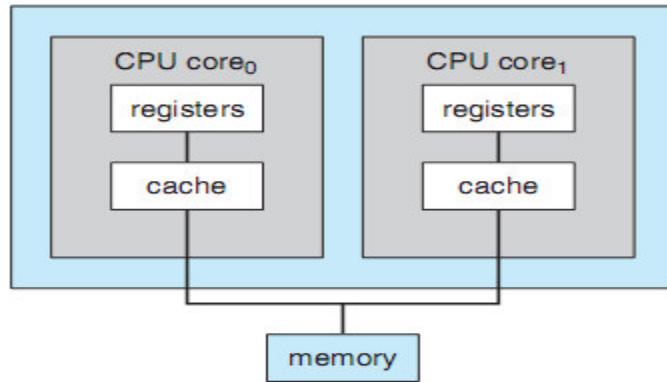
- Asymmetric  
Setiap processor memiliki tugas spesifik. Ada yang berfungsi sebagai pengontrol processor yang lain(master) dan ada yang berfungsi sebagai slave. Slave akan tergantung pada intruksi yang diberikan oleh processor master.
- Symmetric  
Masing-masing processor memiliki kedudukan yang sama. Tidak ada yang bertindak sebagai master(pengontrol)ataupun slave. Yang menentukan pembagian tugas adalah system operasi.



Gambar 1.4.1.Arsitektur Symmetric Multyprocessor .



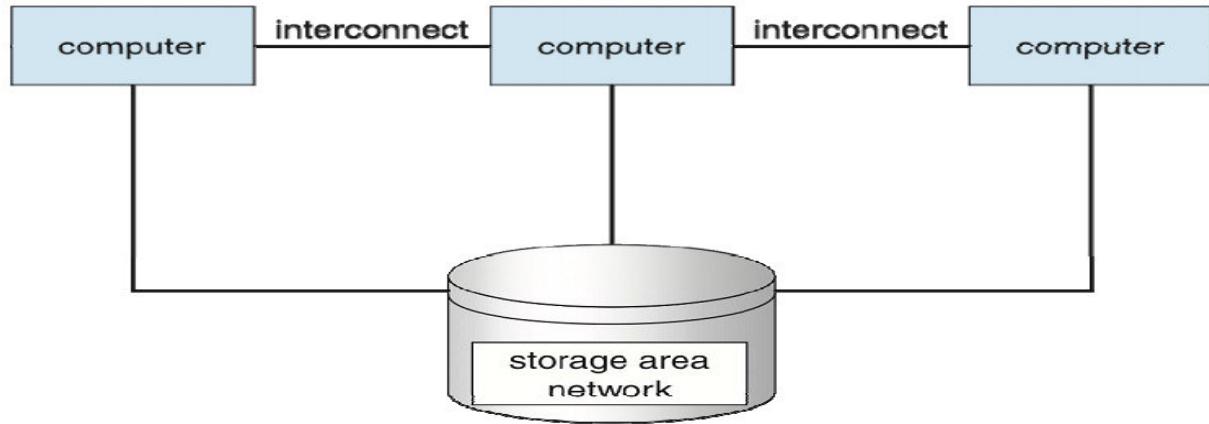
Trend komputer sekarang menggunakan multiprocessor(multicore) dalam satu chip. Dengan meletakannya dalam satu chip maka akan mempercepat komunikasi antara keduanya.



Gambar 1.4.2 Arsitektur Dual core. Dua buah core diletakan dalam satu chip.

### Clustered System

Cluster system berbeda dengan multyprocessor ,multyprocessor memiliki beberapa processor yang terdapat dalam satu komputer, sedangkan cluster system juga memiliki beberapa processor tapi letaknya di komputer yang berbeda-beda. Multyprocessor mengakses memory yang sama sedangkan cluster system mengakses memori yang berbeda-beda. Walupun komputernya terpisah-pisah komputer-komputer ini saling bekerjasama. Pada cluster system sharing storage terjadi melalui LAN atau storage-area network(SAN). System ini menyediakan layanan yang sifatnya terus menerus.



Gambar 1.4.3 Struktur Umum dari cluster system

Seperti halnya multyprocessor, cluster system juga memiliki dua type yaitu

- Asymmetric  
Salah satu komputer bersifat sebagai pengontrol ,yang selalu stand by.
- Symmetric  
Semua mesin kedudukannya setara,saling memonitor satu sama lain

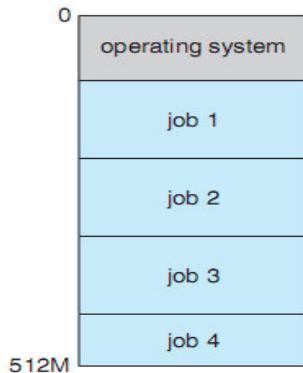
Pengembangan kecepatan processor saat ini sudah hampir mentok, maka beberapa cara dikembangkan untuk mensiasati hal tersebut . Diantaranya adalah dengan multyprocessor dan cluster system.

### 1.5.Struktur System Operasi

Salah satu aspek penting pada OS modern adalah kemampuannya dalam melakukan multiprogramming. Dengan kemampuan multiprogramming nya SO dapat mengefisiensikan penggunaan CPU atau device I/O. Satu orang user atau satu program aplikasi tidak bisa membuat CPU atau device I/O digunakan(disibukan) terus-menerus. Artinya ada saat diamanan CPU atau device I/O idle,dengan kata lain mubazir. Multiprogramming dapat mengorganisasikan pekerjaan(code atau data) sehingga CPU pada suatu waktu selalu memiliki instruksi yang harus dilakukannya.

Ide dasarnya adalah:

OS meload several job dimemori secara bersamaan(gambar 1.5.1).Karena pada umumnya kapasitas memory terbatas untuk menampung semua job maka biasanya sebagian disimpan didalam disk dan disebut job pool. Job pool berisikan proses yang sedang menunggu alokasi tempat dimemori. Pemilihan job-job mana saja yang akan di load ke memori dilakukan oleh **job scheduling**.



Gambar 1.5.1 Layout memory untuk multiprogramming

Satu set job pada memori bisa saja merupakan subset dari job yang ada di job pool. OS kemudian mulai mengambil dan memproses satu job pada memory.Terkadang sebuah job dalam keadaan menunggu kompletnya sebuah task misalnya operasi I/O. Pada saat itu, pada system yang nonmultiprogramming CPU akan idle.Namun pada system multiprogramming, SO akan langsung menswicth ke job lain. Ketika job tersebut juga harus menunggu sesuatu seperti pada job sebelumnya maka system akan menswicth ke job yang lainnya lagi dan seterusnya.Jika proses menunggu salah satu job selesai maka job tersebut akan kembali ditangani oleh CPU.Sehingga selama ada satu job yang harus dieksekusi maka CPU tidak akan idle.



Proses pemilihan job apa yang akan dikerjakan terlebih dahulu dan proses perpindahan CPU dari job yang satu ke job yang lain yang ada dalam memory diatur oleh **CPU scheduling (ingat perbedaanya dengan job scheduling)**. Proses perpindahan tersebut tidak terasa oleh manusia karena proses perpindahannya sangat cepat,ingat bahwa mata manusia merefresh atau berkedip dalam satuan detik(kurang lebih satu kedipan perdetik).

Multiprogramming terjadi pada level bawah,kosekuensi dari multiprogramming pada level yang lebih tinggi(yang dilihat oleh manusia) adalah **timesharing atau multitasking**.Misalnya pada suatu waktu dikomputer kita sedang berjalan PowerPoint, sebenarnya CPU pada saat itu tidak hanya menangani Powerpoint namun secara terus menerus berpindah-pindah dari satu job ke job yang lain tapi dengan sangat cepat (**ini disebut sebagai multi programing**).Namun Seolah olah kita melihat bahwa CPU menangani job-job tersebut bersamaan(**ini yang disebut multitasking atau timesharing**).Jadi timesharing atau adalah ekstensi logis dari CPU ketika berpindah dari job yang satu ke yang lain secara cepat sehingga user dapat berinteraksi dengan job-job tersebut ketika job-job tersebut sedang berjalan(komputasi interaktif).

Sesuatu bisa interaktif jika respon timenya kurang dari satu detik.Misalkan kita sedang bermain game,namun dibelakang program game tersebut kita juga sedang menjalankan 5 program berbeda.Pada saat tertentu CPU akan meninggalkan program game kita (program game adalah program yang sedang dihadapan kita) dan berpindah ke program lain dibelakang program game kita, agar interaktifitas tetap terjaga maka CPU harus kembali ke program game kita dalam waktu tidak boleh kurang dari satu detik.

Setiap user memiliki satu program(data dan code) yang sedang dieksekusi di memori dan disebut **proses**.Jika suatu waktu memori tidak muat untuk menampung job maka harus dilakukan **swapping**. Swapping yaitu memindahkan dulu job yang tidak sedang dieksekusi ke disk kemudian menggantikannya dengan job yang harus segera dieksekusi. Metode yang lebih umum adalah dengan menggunakan **virtual memory**(nanti dibahas).

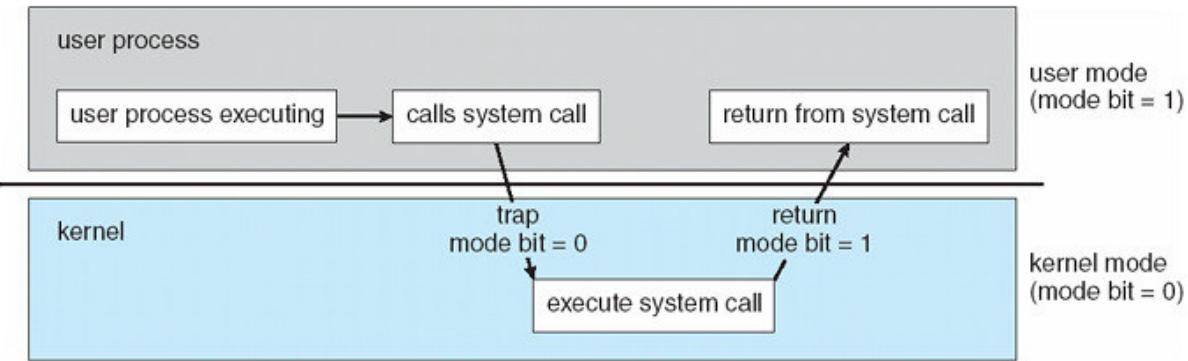
## 1.6.Operasi System Operasi

Seperti yang disebut sebelumnya SO bekerja berdasarkan interrupt driven. Hardware maupun software dapat menginterupt.Interrupt oleh software dapat terjadi karena terjadinya error,misalnya pembagian dengan nol,kesalahan alamat memory atau adanya infinite loop. Karena System Operasi dan User menshare resource yang sama dari system komputer maka harus ada cara untuk mencegah error yang terjadi tersebut agar tidak mempengaruhi program yang lain atau bahkan mempengaruhi SO nya(bisa hang). Jadi error tersebut hanya berpengaruh pada program yang menghasilkan error tersebut.

Caranya adalah dengan memisahkan antara code yang dieksekusi oleh OS dan kode yang dieksekusi oleh user.Cara seperti ini disebut dual-mode operation.Mode pada dual mode yaitu user mode dan kernel mode. Kedua mode tersebut dapat dibedakan oleh bit mode yaitu user(1) atau kernel(0),bitmode di set oleh hardware. Kita berada pada user mode ketika ketika system komputer mengeksekusi intruksi dari user application.Sebaliknya jika user meminta service dari system operasi melalui system call(akan dibahas lebih lanjut nanti)maka user harus neralih dari user mode ke kernel mode. User mode menggunakan proteksi yang lebih longgar,misalnya mengalikan dua buah matriks,jika terjadi kesalahan maka tidak terlalu ada masalah.Kernel mode menerapkan proteksi yang sangat ketat. Misalnya untuk mengakses I/O atau hardware kalau proses ini tidak diproteksi dengan baik dan mengeksekusi intruksi yang salah maka akan berakibat fatal pada keseluruhan system.Instruksi-instruksi yang dapat meng-harm system disebut **privileged instruction**. Hardware akan mengizinkan privileged instruction



dieksekusi hanya pada kernel mode. Sebaliknya jika privileged instruksi di eksekusi di user mode hardware akan menolaknya dan akan mengembalikannya ke SO sebagai error(trap).



Gambar 1.6.1 Transisi dari user mode kek kernel mode

Pada saat system boot, maka hardware akan memulai dari kernel mode. User atau system mengeksekusi user application dalam user mode dengan mode bit pada hardware adalah 1. Setiap user application melakukan interupsi(trap) memanggil system call (misalnya ingin mengakses hardisk untuk menyimpan hasil dari operasi user application), maka hardware akan beralih dari user mode ke kernel mode (bit mode diubah ke 0). Setelah system call dieksekusi maka system akan kembali ke user mode tapi sebelumnya mode bit hardware diubah ke 1.

Permasalahan infinite loop pada user application biasanya diatasi dengan timer. Jika infinite loop dibiarkan maka kontrol CPU akan selalu berada di user, tidak pernah kembali ke operating system. Sebelum operating system menyerahkan kontrol ke user maka terlebih dahulu SO akan megeset period tertentu di timer agar setelah period tersebut akan langsung terjadi interupsi oleh user dan kendali dikembalikan ke SO.

## 1.7. Computer Environment

### Komputer tradisional

Perbedaan antara environment komputer yang satu dengan yang lain tidak terlihat dengan jelas. Dahulu ada yang namanya office environment, dimana di environment ini biasanya komputer-komputer di lingkungan tersebut sudah terhubung melalui network dan biasanya dilengkapi oleh file server dan layanan print. Di rumah biasanya seorang single user memiliki sebuah komputer, dengan modem untuk mendapat layanan internet. Pada masa ini untuk membedakan kedua environment diatas sangat susah karena setiap komputer sudah memiliki perlengkapan yang sama yang dapat dipakai di environment manapun.

### Client-Server Computing

Pada masa sekarang banyak design dari lingkungan komputer menerapkan system Client –Server, dimana satu atau lebih komputer bersifat sebagai server dan melayani permintaan dari komputer-komputer client. System ini termasuk dalam kategori system terdistribusi.

Server system sendiri dapat dikategorikan menjadi dua yaitu

- Compute-server System yaitu server yang menyediakan layanan interface berupa service bagi permintaan client(seperti membaca data dari database).
- File Server menyediakan layanan antarmuka untuk client dalam menyimpan ataupun mengambil file.

### **Peer-to-peer Computing**

Salah satu jenis system terdistribusi adalah peer-to-peer computing. Bebrbeda halnya dengan client-server computing, dalam **Peer-to-peer Computing** tidak ada komputer yang bersifat sebagai server dan yang lainya sebagai client . Namun dalam **Peer-to-peer Computing** sebuah komputer suatu waktu dapat bersifat sebagai server yaitu memberikan suatu layanan dan diwaktu yang lain bersifat sebagai client(menerima layanan/atau meminta layanan ke komputer lain dalam system tersebut).

### **Web based computing**

Pada saat ini web menjamur dimana-mana. Sehingga kemungkinan suatu hari nanti seseorang hanya membutuhkan web browser di komputernya. Applikasi disediakan oleh web contohnya saat ini ada googledocs.

### **1.8.Open Source Operating System**

Membuat sebuah OS adalah pekerjaan besar, sehingga biasanya pembuat OS adalah perusahaan-perusahaan besar namun beberapa tahun belakang ada yang namanya **Open Source Operating System** operating system ini dibangun bersama-sama oleh ahli komputer yang tidak terikat pada perusahaan tertentu. Oleh karena itu source code dari OS tersebut terbuka untuk publik,contohnya adalah linux.



## CHAPTER 2. STRUKTUR SYSTEM OPERASI

### 2.1.Operating System Service

System operasi menyediakan lingkungan untuk mengeksekusi program dan layanan kepada program dan user. System Operasi memberikan beberapa fungsi layanan yang sangat membantu user diantaranya adalah:

- User Interface  
Ada beberapa jenis user interface diantaranya command interpreter, GUI(Graphic User Interface), Batch(biasa digunakan diserver).
- Menyediakan layanan untuk mengeksekusi program.  
System operasi harus mampu meload program ke memori kemudian merun nya dan mampu menghentikan eksekusinya baik itu menghasilkan error maupun tidak menghasilkan error(normal).
- I/O operation  
Program yang berjalan akan membutuhkan akses ke I/O. System operasi harus mampu membantu program untuk mengakses I/O yang dibutuhkan. Kenapa harus dibantu? sebenarnya program dapat dibuat untuk mengakses I/O sendiri tanpa melalui SO, namun akan sangat rumit dan kompleks, SO akan menyediakan interface yang lebih mudah untuk melayani akses I/O oleh program.
- File-System Manipulation  
Beberapa program harus membaca, menulis, membuat, menghapus, mencari, melist file dan direktori, juga manajemen *permission* dari file dan direktori tersebut dll. Contoh, di Windows menyediakan Windows Explorer, windows explorer bukanlah System Operasi, namun merupakan system program(dibahas nanti).
- Komunikasi  
Proses-proses yang dibuat oleh user dapat saling berkomunikasi, mempertukarkan informasi dengan proses yang lain, baik itu dalam satu sistem komputer ataupun dengan proses yang berada pada sistem komputer yang lain. Secara umum komunikasi pada komputer modern dilakukan dengan dua cara:
  1. Shared memory, terjadi pada satu sistem komputer yang sama karena tentu saja, mereka menggunakan memory yang sama.
  2. Message Passing.  
Biasanya terjadi pada jaringan, paket dikirim dari satu komputer ke komputer lain.
- Error detection  
SO harus selalu tanggap dengan terjadinya error, bukan hanya untuk kelangsungan proses dari sistem namun juga user akan rugi jika SO menutupi terjadinya error. Error dapat terjadi di CPU dan hardware(seperti memori error, power failure), di I/O device( parity error di tape, koneksi yang gagal pada jaringan, kekurangan kertas pada printer), di user program(overflow pada proses aritmatika, kelebihan pemakaian waktu CPU). Untuk masing-masing error SO harus mengambil tindakan yang tepat guna kelangsungan dan berjalan dengan benarnya komputasi (termasuk menghentikan program yang error tsb). Penyediaan fasilitas debugging juga membantu user dan kemampuan programer untuk menggunakan sistem secara effisien. Debugger mengizinkan user ataupun programer untuk melihat isi dari memory.



Selain layanan-layanan diatas masih adalagi layanan dari SO,namun jika yang telah disebutkan diatas lebih dititik beratkan pada layanan untuk membantu dan mempermudah user,layanan-layanan berikut lebih kearah bagaimana SO menjamin efisiensi dari operasi pada system itu sendiri.

- Resource allocation

Resource pada komputer cukup banyak tetapi proses yang berjalan juga banyak. Ketika multi user ataupun multi job sedang berjalan secara concurrent,maka SO harus membagi resource secara adil kepada mereka. Jadi tidak ada satupun job atau user yang menguasai secara penuh resource. Resource dapat berupa CPU cycle ,memory,file storage, I/O device. Masing-masing resource biasanya memiliki kode alokasi(CPU,memory dan file storage),request dan release kode(I/O device) yang dapat digunakan oleh job atau user jika mereka membutuhkan resources tersebut. Release kode digunakan jika user atau job telah selesai menggunakan I/O device.

- Accounting

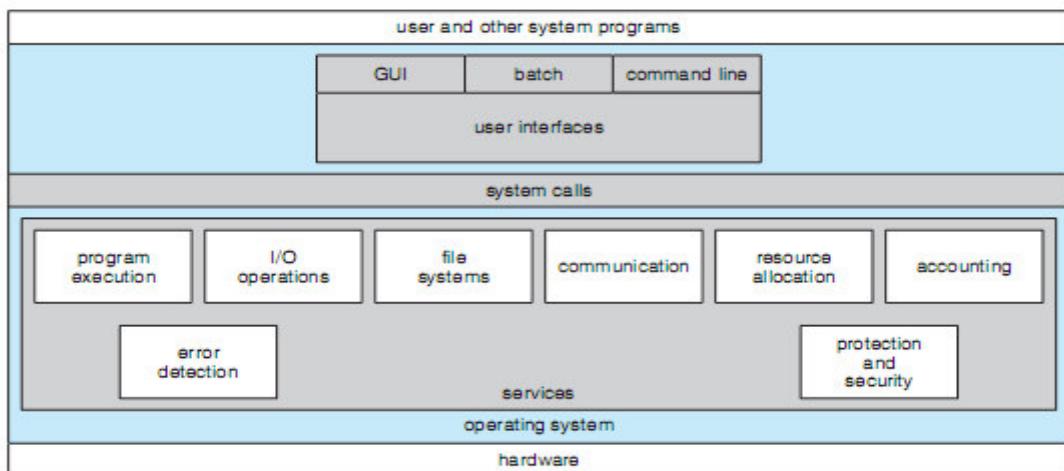
Untuk mengetahui user mana yang menggunakan resource,berapa banyak yang digunakanya,dan jenis resource apa yang digunakanya.Pengetahuan ini dapat digunakan misalnya untuk tagihan pembayaran (untuk komputer khusus yang menyediakan resource berbayar ),juga dapat digunakan hanya untuk statistic semata(komputer pribadi).

- Protection dan security

Pemilik informasi pada system yang terhubung jaringan dan memungkinkan multiuser biasanya ingin mengontrol penggunaan informasi tersebut,proses yang berjalan secara concurrent tidak boleh mempengaruhi process lain,atau bahkan mempengaruhi system itu sendiri.

- Protection,meyakinkan semua akses ke resource dapat terkendali.Tidak ada penggunaan resource secara sembarangan.
- Security,menjaga system dari pengguna luar yang tidak berkompeten,biasanya dengan user authentication.

Protection dan security berbeda,protection maksudnya seperti, seseorang diizinkan menggunakan resources hanya dengan cara-cara tertentu.Sedangkan security berusaha agar user yang dapat menggunakan resource adalah hanya user yang berhak(telah mendapatkan izin).Jika kita ingin system kita terproteksi dan secure maka kita harus melakukan tidak pencegahan(precautions) dimana saja kemungkinan dapat terjadinya serangan. Satu bagian saja yang lemah maka semua bagian akan terancam.



Gambar 2.1.Layanan Sistem Operasi

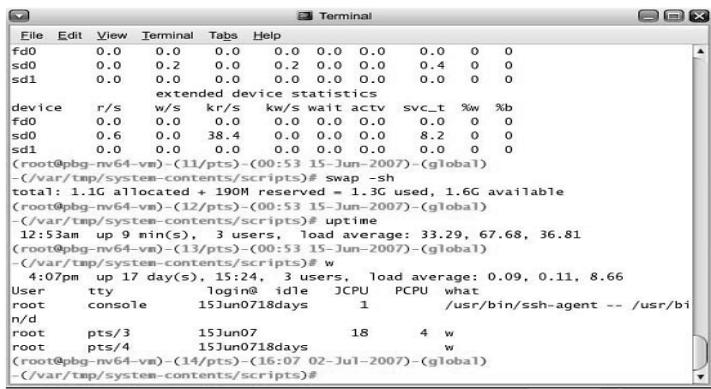


Seperti yang terlihat pada gambar diatas bagian paling atas adalah user dan system program lainnya. User dan system program lainnya dapat memanfaatkan layanan System Operasi hanya melalui user interface. User interface lah yang akan memanggil layanan-layanan System Operasi melalui system call, oleh karena itu system call adalah point terpenting pada System Operasi. Definisi tentang System Operasi sangatlah beragam dari System Operasi yang satu ke System Operasi yang lain, seperti dilihat di atas, User Interface dimasukan ke bagian System Operasi namun pada beberapa System Operasi yang lain, user interface bukanlah bagian dari System Operasi. Bagian paling bawah adalah Hardware yang melakukan pekerjaan dari layanan-layanan yang disediakan oleh System Operasi.

## 2.2 User Operating System Interface

- Command Line/ command interpreter

Beberapa sistem operasi menyediakan Command Interpreter pada kernelnya(pada system operasi yang lama). Windows dan unix memperlakukan command interpreter sebagai program khusus(bisanya sebagai system program). Pada system dengan multi command interpreter yang dapat dipilih,interpreternya disebut shell(Unix dan Linux). Shell dalam bahasa indonesia berarti cangkang,yang melindungi system dari akses langsung oleh user. Itulah mengapa dinamai shell. Di keluarga Linux terdapat beberapa macam shell seperti Bourne shell,C shell,Bourne Again shell,korn shell dan lain-lain.



The screenshot shows a terminal window titled "Terminal". The output of the command "swap -sh" is displayed, showing swap space usage. Then, the "uptime" command is run, showing the system has been up for 12 days, 9 hours, and 30 minutes, with three users logged in and a load average of 33.29, 67.68, and 36.81 respectively. Finally, the "w" command is run, showing there are no users currently active.

```
fd0      0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
sd0      0.0   0.2   0.0   0.2   0.0   0.0   0.4   0.0   0.0
sd1      0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
                                                 extended device statistics
device   r/s    w/s   kr/s   kw/s   wait   activ   svc_t   %w   %b
fd0      0.0     0.0    0.0     0.0    0.0     0.0     0.0     0.0   0
sd0      0.6     0.0    38.4    0.0    0.0     0.0     8.2     0.0   0
sd1      0.0     0.0    0.0     0.0    0.0     0.0     0.0     0.0   0
(root@pbg-nv64-vm) (11/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm) (12/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts# uptime
12:53am up 9 min(s), 3 users, load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm) (13/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts# w
4:07pm up 17 day(s), 15:24, 3 users, load average: 0.09, 0.11, 8.66
User     tty        login@  idle   JCPU   PCPU   what
root    console          15Jun0718days      1   /usr/bin/ssh-agent -- /usr/bi
n/d
root    pts/3           15Jun07          18      4   w
root    pts/4           15Jun0718days      1   w
(root@pbg-nv64-vm) (14/pts)-(16:07 02-Jul-2007)-(global)
-/var/tmp/system-contents/scripts#
```

Gambar 2.2.1 Command Interpreter Bourne shell pada solaris 10

Intruksi user biasanya diketikan di command interpreter. Pada SO modern shellnya sangatlah tipis, fungsi utamanya adalah hanya mengambil intruksi yang diketikan user dan mengeksekusinya. Ada dua pendekatan untuk melakukan ini. Pendekatan pertama, command interpreter itu sendiri, di dalamnya telah terdapat kode untuk mengeksekusi perintah user(command built-in). Pendekatan ini digunakan oleh Windows (MS-DOS), perintah seperti *cd* (*change direktori*) sudah built-in dalam Command Interpreternya. Namun pada MS-DOS, ada beberapa perintah seperti *cp*(*copy*) sudah menggunakan pendekatan kedua berikut ini. Pendekatan kedua, command interpreter tidak mengerti dengan perintah yang diketikan user, command interpreter akan meload file yang bersesuaian dengan perintah dari hardisk ke dalam memory dan kemudian file yang berisi fungsi yang bersesuaian dengan perintah user tersebut dieksekusi.

rm file.txt



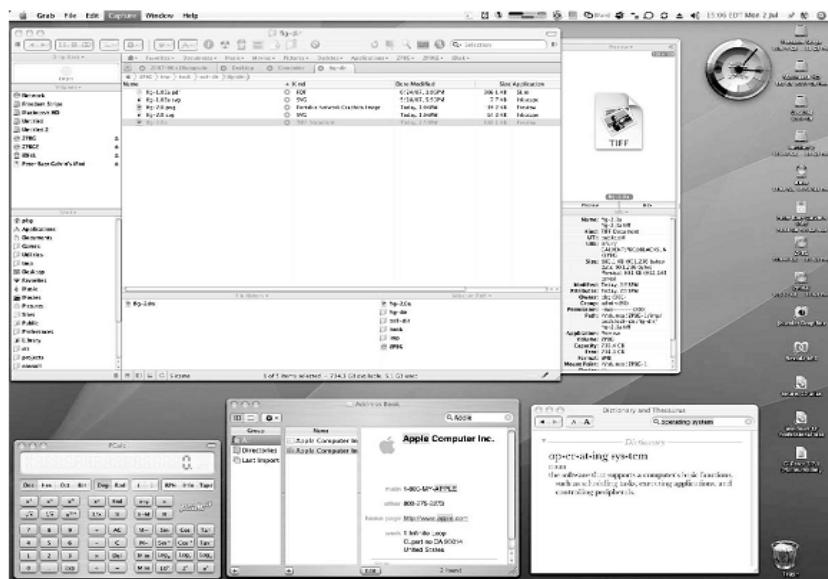
Perintah diatas akan menyebabkan command interpreter akan mencari file dengan nama rm lalu meloadnya ke memori dan mengeksekusinya bersama parameter file.txt. Fungsi rm pada perintah diatas telah didefinisikan oleh kode/script code dalam file bernama rm tadi.Pendekatan kedua ini dilakukan pada keluarga Unix.

Kelemahan dari command built-in adalah banyaknya instruksi dibatasi oleh ukuran dari command interpreternya,dan jika kita ingin menambah atau mengubah perintah maka harus merubah juga Command Interpreternya. Kelebihanya, eksekusi dari perintahnya akan lebih cepat dibanding pendekatan kedua.Karena dengan pendekatan command built-in, berarti perintah atau instruksi sudah berada didalam memory sebelumnya, sehingga pada saat sebuah instruksi ingin dijalankan oleh user maka Command interpreter tinggal mengeksekusi perintah yang sudah ada di memory,tidak perlu meload lagi suatu file dari hard disk lalu di simpan ke memory dan dieksekusi.Namun perlu diingat bahwa yang dimaksud dengan pendekatan kedua lebih lambat,sebenarnya oleh user dianggap cepat.Pendekatan kedua memiliki kelebihan pada kemampuan untuk menambah ataupun memodifikasi perintah. User dapat menambah perintah baru hanya dengan membuat sebuah file baru berisikan code fungsi yang dinginkan dan menyesuaikan nama filenya dengan nama fungsi yang dinginkan.

- Graphical User Interface

Strategi kedua agar user dapat berinteraksi dengan operating system adalah dengan menggunakan Graphical user interface(GUI). GUI sebenarnya adalah metaphor/pengandaian dari sebuah desktop(atas meja kerja). Biasanya diatas meja kerja kita meletakan file-file pekerjaan,biasanya juga file-file tersebut dikelompokan dalam sebuah tempat(map,bundel),alat-alat tulis,dan hal-hal lain yang membantu kita dalam menyelesaikan pekerjaan.

Biasanya Element GUI adalah mouse,keyboard dan monitor. Elemen terpenting pada GUI adalah mouse,dengan mouse user dapat berinteraksi dengan mudah,hanya dengan menggeser pointer mouse ke posisi icon dilayar yang merepresentasikan file,folder,program dan system function, lalu mengkliknya untuk membuka atau menjalankannya.



Gambar 2.2.2. GUI pada Mac OS X



GUI pertama kali muncul di Pusat penelitian Xerox PARC pada tahun 1970 an dan pertamakali diimplementasikan pada komputer Xerox Alto tahun 1973. Pada tahun kemunculan GUI, Xerox mengundang beberapa inventor dalam dunia komputer untuk melihat penemuan mereka,salah satunya adalah Steve Job(founder apple). Tahun 1980 an GUI berkembang dengan cepat sejak Apple Machintosh mulai juga mengembangkan GUI pada komputer-komputer keluaran mereka. Kemudian disusul oleh perusahaan komputer lain seperti Microsoft.

System Operasi modern biasanya menyediakan GUI dan juga CLI(command line interpreter). Walaupun GUI lebih menarik untuk digunakan namun banyak hal yang dapat dilakukan di CLI, sulit untuk dilakukan di GUI bahkan ada yang memang tidak dapat dilakukan melalui GUI. Misalnya jika kita ingin menjalankan sebuah program sebanyak 500 kali,maka akan sulit melakukannya di GUI namun cukup mudah jika ingin dilakukan di CLI.Kita tinggal membuat file batch di CLI, dan membuat perintah untuk membuat intruksi loop untuk menjalankan sebuah program 500 kali.

### 2.3.System Call

Perhatikan gambar 2.1 mengenai layanan yang disediakan oleh system operasi. User dan system program mengakses layanan-layanan system operasi dengan menggunakan User interface. User interface sendiri tidak langsung mengakses layanan System Operasi namun hanya mengakses System call, System call lah yang akan mengakses layanan System Operasi sesuai keinginan user.

System call menyediakan interface ke layanan yang disediakan oleh System Operasi.Interface tersebut berupa bagaimana cara mengakses layanan, apa saja data yang harus dilewatkhan, output apa yang diharapkan dan lain-lain. Dengan adanya System call maka implementasi layanan dapat dirubah-rubah.User tidak akan merasakan perubahan selama system call nya sama,yang dirasakan user kemungkinan hanya sebatas lebih lambat atau lebih cepat,namun secara fungsionalitas nya oleh user dirasakan tidak berubah sama sekali.

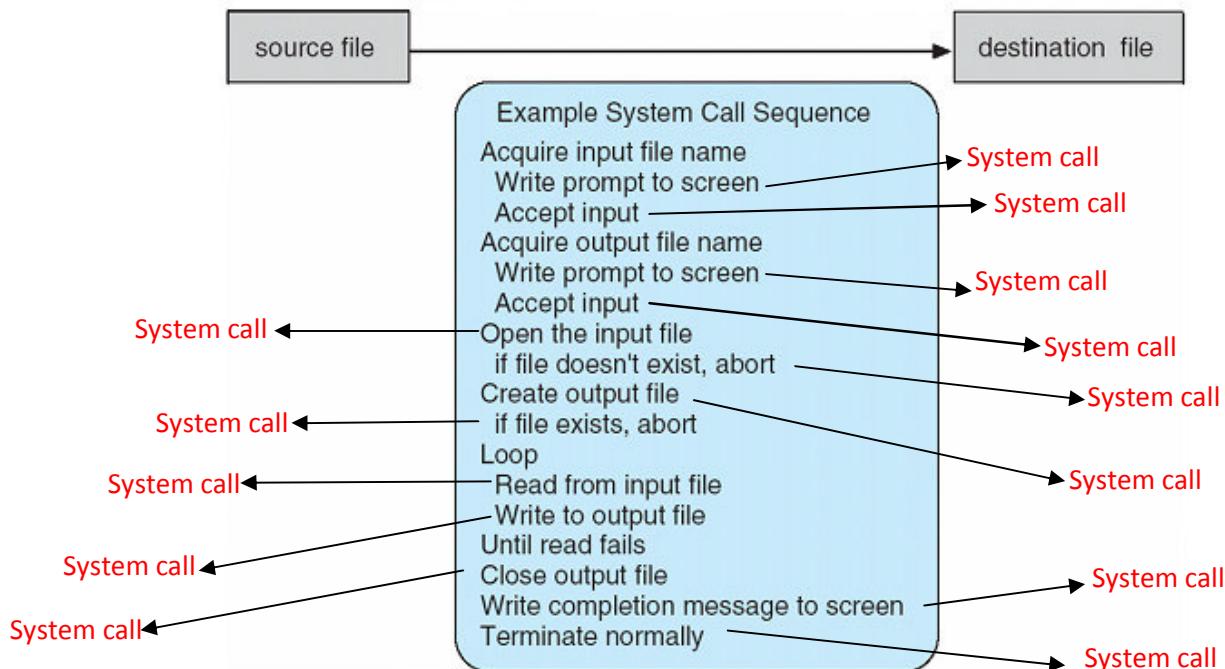
System call dibuat dengan bahasa pemrograman tingkat tinggi C dan C++,tapi untuk tugas level bawah(misalnya untuk mengakses langsung hardware) maka biasanya ditulis dalam bahasa assembly. Pada operating system modern System call dibungkus lagi oleh application programming interface (API) Pengembang aplikasi(pembuat program) menulis program berdasarkan API. API sendiri disediakan oleh pembuat system operasi agar seorang programer dapat membuat aplikasi yang berjalan diatas System Operasi. Bahasa pemrograman biasanya juga menyediakan API namun tentu saja API ini akan berjalan diatas API dari System Operasi. Ada beberapa API yang umum digunakan sekarang, Win32 API untuk Windows,POSIX API untuk system yang berbasis POSIX(Linux, mac os ,semua versi unix) dan JAVA API untuk Java Virtual Machine (JVM).

Mengapa System Call harus dibungkus lagi oleh API?

1. Portability:Programer aplikasi membuat program berdasarkan API berharap program aplikasinya dapat berjalan juga di lain system yang menggunakan API yang sama(meskipun realitasnya perbedaan arsitektur akan membuat hal ini sulit terjadi).
2. System call rumit dan memiliki banyak error. Untuk mengakses satu file saja banyak yang harus kita antisipasi dan harus siap-siap menerima ribuan tipe error. API menyediakan interface yang lebih mudah,beberapa error yang tidak terlalu penting disembunyikan dari user. Sebenarnya tidak disembunyikan namun API akan melakukan retryer beberapa kali.

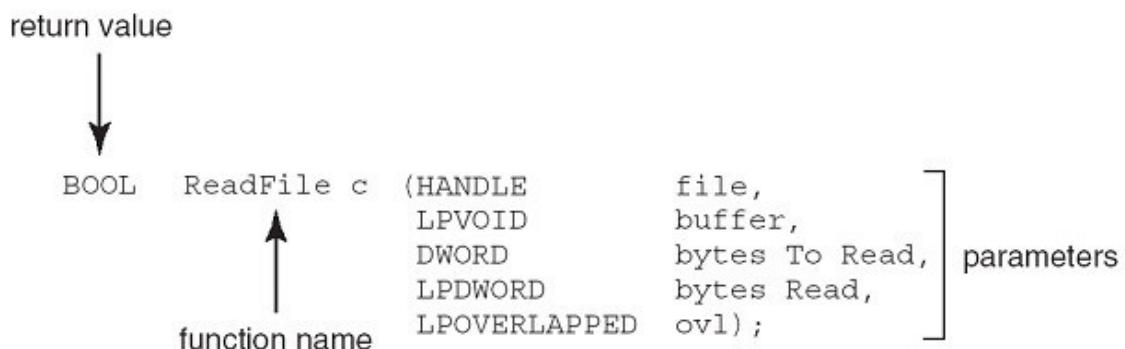


Ketika kita menjalankan program maka sebenarnya kita melakukan pemanggilan system call secara terus menerus. Terkecuali jika kita tidak mengakses I/O. Berikut ini contoh deretan pemanggilan system call pada program sederhana membaca data suatu file dan menkopinya ke file lain.



Gambar 2.3.1 Contoh penggunaan system call

Berikut ini Contoh sebuah Fungsi Standar API yaitu fungsi `ReadFile()`, fungsi ini terdapat pada API Win32 dan ditulis dalam bahasa C. Fungsi `ReadFile()` merupakan fungsi pembaca data dari file.



Deskripsi parameter-parameterannya:

`HANDLE file` : Pointer ke file yang ingin dibaca

`LPVOID buffer` : alamat memory, data akan dipindahkan/ditulis dari file ke alamat memory ini, kemudian dibaca.



DWORD bytesToRead : berapa banyak data dari file yang ingin dipindahkan ke buffer

LPDWORD bytesRead : Banyaknya data yang dipindahkan ke memori sebelumnya. Pemindahan data dari file ke buffer biasanya akan berulang-ulang karena dibatasi oleh ukuran jalur data ataupun besarnya buffer yang disiapkan. Sehingga perlu diketahui banyak data yang sebelumnya telah dipindahkan.

LPOVERLAPPED ovlp : flag, mengindikasikan terjadinya I/O overlap.

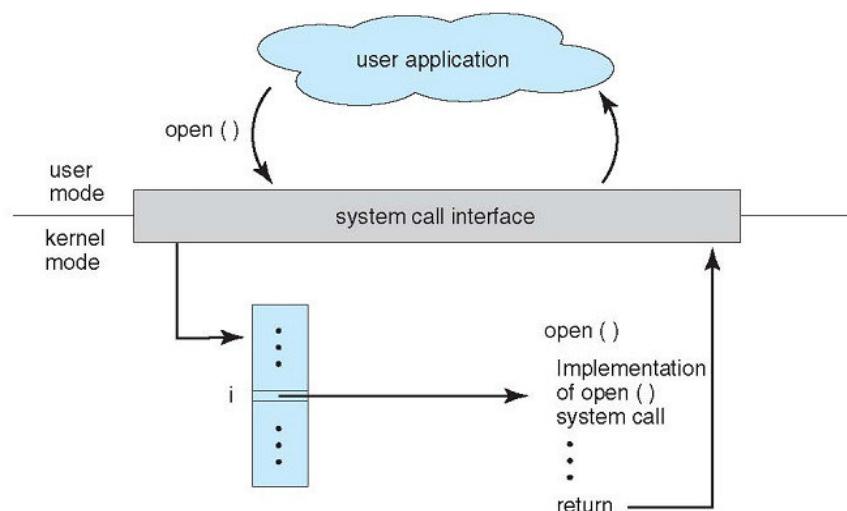
Return value mengindikasikan pembacaan file berhasil atau tidak.

Perhatikan untuk sebuah fungsi saja sudah banyak parameter yang dibutuhkan. Satu fungsi tersebut tentu saja membukus banyak system call dibelakangnya.

### Implementasi System Call

System call diimplementasikan di kernel. Sebagian besar bahasa pemrograman menyediakan System Call interface. System Call interface menyediakan link ke system call yang disediakan oleh Operating system. System call interface mengintersep panggilan fungsi yang terdapat di API dan kemudian meng invoke system call yang dibutuhkan, mengembalikan status dari system call dan return value lainnya. Biasanya sebuah bilangan/nomor diasosiasikan dengan masing-masing system call. System call interface memiliki tabel berupa daftar bilangan/nomor tersebut dan memasangkan nya dengan alamat memory dimana masing-masing system call berada.

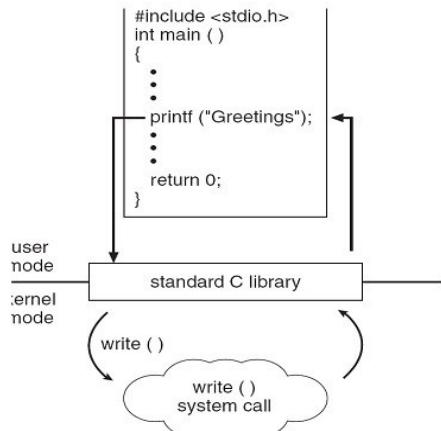
Caller(user,programer,program dll) tidak perlu mengetahui bagaimana sebuah system call diimplementasikan atau apa yang dilakukan system call saat dieksekusi. Yang perlu diketahui oleh caller adalah mematuhi API dan mengerti apa yang akan dilakukan SO sebagai tanggapan dari hasil eksekusi system call. Sebagian besar dari detail SO interface(seperti implementasi system call) disembunyikan dari programer oleh API terkecuali di linux. API dimanaged oleh run-time support library,yaitu himpunan dari fungsi-fungsi yang dimasukan kedalam library pada saat kita menginstall compiler. Namun yang ada di library tersebut hanya sebuah interface, implementasinya terdapat dalam kernel.



Gambar 2.3.2. Penanganan dari aplikasi user men-invoke system call open()

Applikasi user akan memanggil fungsi `open()` (fungsi ini telah dispesifikasi dalam API). Fungsi yang dipanggil tersebut di intercept oleh System Call Interface. System Call Interface disediakan oleh run-time support library yang juga mengatur spesifikasi API. Sehingga system call interface tahu system call mana yang dipanggil jika user men invoke fungsi `open()`. System call interface kemudian mencocokan nomor dari system call yang ingin dipanggil dengan alamat memory dimana system call tersebut berada dalam hal ini kebetulan system callnya juga `open()`. Setelah system call dieksekusi, nilai/status akan dikembalikan ke pemanggil system call tersebut.

Library Standar bahasa C menyediakan sebagian System Call interface bagi sebagian besar versi UNIX dan Linux. Sebagai contoh misalnya sebuah program dengan bahasa C men-invoke statemen `printf()` (fungsi ini telah dispesifikasi dalam API). C library yang disini berfungsi sebagai System Call interface mengintercept panggilan tersebut. C library mengetahui bahwa system call yang bersesuaian dengan fungsi `printf()` adalah system call `write()`. Kemudian C library memasangkan nomor system call `write()` dengan alamat dimana system call `write()` berada di memori, system call tersebut kemudian dieksekusi dan return value nya dikembalikan ke user.



Gambar 2.3.3. Library standar dari bahasa C dalam menangani System call write()

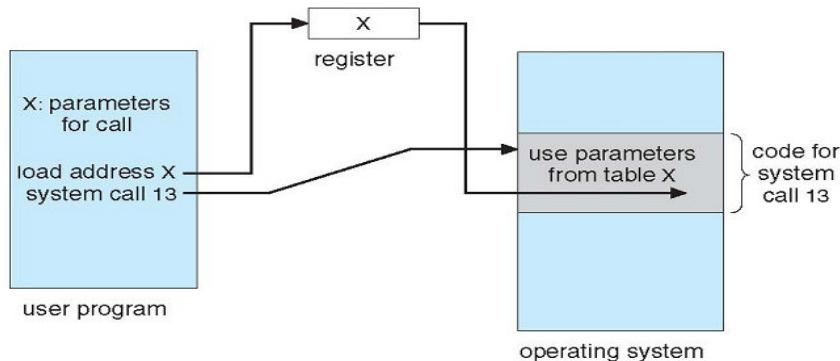
### System Call Parameter Passing

Dalam memanggil System call, sering kali dibutuhkan informasi yang lebih, tidak sekedar menyebutkan nama dari system call yang ingin kita panggil. Type dan jumlah informasi yang dibutuhkan oleh system call biasanya berbeda-beda tergantung OS dan system call itu sendiri. Ada 3 metode yang lazim digunakan untuk melewatkannya informasi/parameter yang dibutuhkan tersebut:

- Yang paling sederhana, yaitu melewatkannya parameter-parameter system call tersebut ke dalam register. Pada saat CPU akan mengeksekusi System call tersebut maka oleh CPU parameter yang dibutuhkan oleh system call tersebut diambil dari register. Cara ini memiliki kekurangan, karena pada beberapa kasus system callnya memiliki lebih banyak parameter dibanding banyaknya register.
- Metode kedua yaitu dengan menyimpan parameter-parameter dalam sebuah blok atau matriks atau tabel di memory, dan yang disimpan di register hanya alamat dari blok-blok tersebut. Metode ini digunakan di linux dan solaris.
- Metode ketiga adalah pada sistem operasi modern biasanya sebuah program dilengkapi stack. Jika sebuah program melakukan pemanggilan system call maka parameter yang dibutuhkan

system call tersebut di masukan/push kedalam stack dari program tersebut. Pada saat system operasi memasuki kernel mode maka System operasi akan mem pop parameter tersebut dari stack.

Cara block dan stack biasanya lebih banyak digunakan karena kedua metode ini tidak membatasi panjang/banyaknya parameter yang dapat dilewatkan.



Gambar 2.3.4 melewatkkan parameter dengan tabel

#### 2.4.Type System Call

System call dapat dikategorikan dalam 6 kategori utama yaitu control process, file manipulation, device manipulation, information maintenance, communication dan protection.

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory
- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes



- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes

- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

Berikut ini contoh system call pada windows dan linux

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()



## 2.5.System Program

System program menyediakan lingkungan yang menyenangkan untuk pengembangan program dan eksekusi. Sebagian user melihat system operasi sebatas system program. Beberapa system program merupakan system call interface sederhana yang membukus system call,namun sebagian dari system program juga cukup kompleks.

System program dapat dikategorikan dalam enam kategori berikut

- File management  
Program ini men-create,menghapus,copy,rename,print, dump,list dan memanipulasi file juga direktori.
- Status Information  
Beberapa program menyediakan layanan informasi, seperti informasi space memory dan disk, jumlah user dll.
- File modification  
Beberapa teks editor mampu men-create ataupun memodifikasi file yang tersimpan di disk atau tempat penyimpanan lainnya(misalnya notepad di Windows).
- Programing language Support.  
Beberapa SO menyediakan compiler,assembler,debugger untuk mendukung bahasa pemrograman yang umum.
- Program loading and execution  
Setelah sebuah program dikompilir maka harus diload ke memory untuk dieksekusi. SO biasanya telah menyediakan program absolute loader,relocatable loader, dll untuk melakukan hal tersebut.
- Communication  
Program ini menyediakan layanan untuk membuat koneksi virtual antara proses,user ataupun antara computer system.

System program atupun program kita sebenarnya berisi deretan system call.Sebagai tambahan untuk system program biasanya SO menyediakan program aplikasi, khusus untuk menyelesaikan masalah-masalah umum atau untuk melakukan operasi khusus. Contoh program aplikasi seperti pengolah kata,web browser,spreadsheets,game, dll.

## 2.6.Design dan Implementasi System Operasi

Perlu diingat bahwa dalam kuliah ini,kita tidak menjelaskan suatu system operasi tertentu. Dalam kuliah ini kita hanya mencoba mendesign dan bagaimana mengimplementasi system operasi. Dalam sub bab ini kita akan membahasa masalah-masalah yang dihadapi ketika kita mendesign dan melakukan implementasi System Operasi. Tentu saja tidak ada solusi yang komplet namun terdapat beberapa pendekatan yang telah terbukti berhasil di System Operasi sekarang.

Internal struktur antara SO bisa saja berbeda-beda satu sama lain. Membangun suatu system operasi biasanya dimulai dengan menentukan goals dan spesifikasi. Membangun system operasi dengan goal untuk user ,goal untuk tablet ,untuk server ,dekstop,berbeda pendekatannya. Pada level yang lebih tinggi design dipengaruhi juga oleh pemilihan hardware dan type system(batch,single user,multiuser,real time ,general purposes).Selain itu juga dipengaruhi oleh tujuan dari sisi user dan tujuan dari sisi system.



Dari sisi user(user goal) berkeinginan system yang akan dibangun haruslah menyenangkan untuk digunakan,mudah dipelajari,reliable,safe dan cepat.Namun dari sisi system(system goal) berkeinginan agar system yang dibuat mudah didesign,mudah diimplementasikan,mudah dimaintain,fleksibel,error-free dan efficient. Kedua goal ini saling berlawanan,misalkan untuk membuat sebuah system mudah dimaintain kita membuatnya dalam modul-modul namun hal ini berimbang system yang kita bangun akan lebih lambat karena butuh waktu untuk modul-modul tersebut saling berkomunikasi.

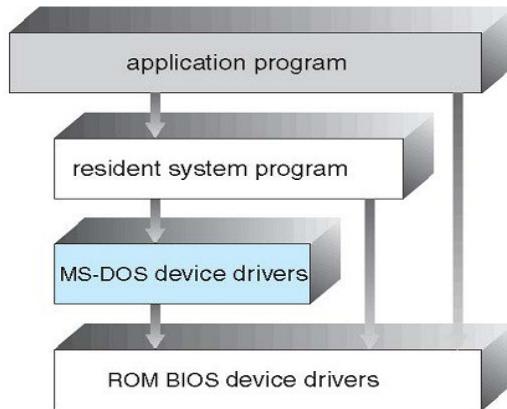
Salah satu prinsip mendesign system operasi adalah memisahkan antara policy dan mekanisme. Policy menentukan apa yang akan system kita bisa lakukan(spesifikasi system) sedangkan mekanisme menentukan bagaimana kita kan melakukannya. Jika kita ingin merancang sebuah system operasi jangan mencapur adukan keduanya,kita harus menspesifikasi dulu apa yang akan kita buat,misalnya kalau bisa system kita touchscreen,tanpa keyboard dan sebagainya,lalu bagaimana melakukannya belakangan.

Dipisahkanya policy dan mekanisme bertujuan untuk fleksibilitas. Policy dapat berubah-ubah sesuai tempat dan berjalannya waktu. Pada kasus terburuk perubahan policy dapat menyebabkan terjadinya perubahan pada mekanisme dasar. Sebuah mekanisme umum yang tidak mudah berubah terhadap perubahan policy lebih diinginkan. Sehingga diharapkan nantinya perubahan terhadap policy hanya mengubah parameter tertentu saja pada system, tidak berpengaruh ke semua parameter system.Sebagai contoh mekanisme untuk menentukan prioritas sebuah program dibanding program lain,jika policy nya terpisah dengan mekanismenya maka mekanisme tersebut dapat digunakan untuk policy lain misalnya menentukan prioritas I/O-intensive dibanding CPU-intensive,atau policy lain yang sejenis atau bahakan policy yang berlawanan.

## 2.7.Operating System Structure

### Simple struktur

Struktur paling sederhana Operating system dapat dilihat pada MS-DOS. MS-DOS dibuat dengan tujuan utama menyediakan layanan fungsionalitas yang banyak namun dispace yang terbatas. Atas pertimbangan space tersebut MS-DOS tidak dibagi-bagi dalam module semuanya dalam satu kesatuan. Walupun MS-DOS memiliki structure namun sebenarnya interface dan level fungsionalitas nya susah untuk dipisahkan.

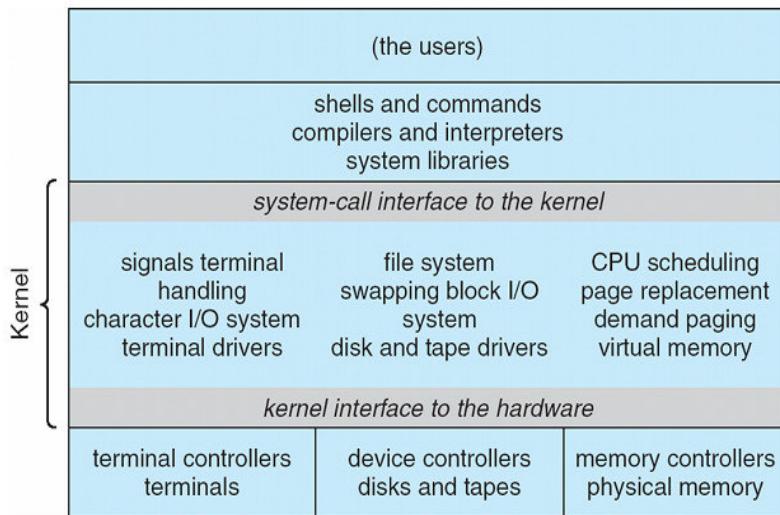


Gambar 2.7.1 Struktur Layer MS-DOS

Layer paling bawah dari MS-DOS adalah ROM BIOS device drivers merupakan device driver untuk semua driver. Pada saat itu device masih terbatas,biasanya driver tersebut adalah resident system program driver untuk I/O device saja. Seperti terlihat pada gambar diatas application program dapat langsung mengakses hardware. Di system OS modern application program mengakses hardware melalui system call. Di MS-DOS kita dapat membuat program yang berjalan-terus menerus dibelakang application program yang disebut resident system program. Struktur seperti ini cukup menyulitkan jika kita ingin mengubah sesuatu maka kita harus melihat kode nya dari ujung ke ujung. Untuk itu ada pendekatan kedua yaitu membuat struktur dari SO berlapis-lapis(layered)

### Layered approach

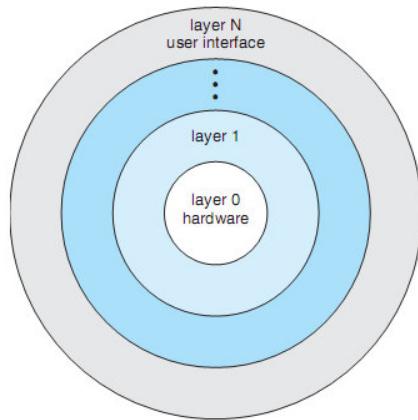
System operasi dibagi-bagi dalam sejumlah layer,layer yang satu dibangun diatas layer yang lain. Layer paling bawah(layer 0) adalah hardware dan layer paling atas (layer N) adalah interface. Karena berlayer-layer maka diantara setiap layer terdapat interface untuk menentukan bagaimana sebuah layer berkomunikasi dengan layer diatas atau dibawahnya. Keuntungan dari pendekatan ini adalah kemudahan dalam maintainnya,kita dapat merubah implementasi pada suatu layer dan tidak mempengaruhi layer yang lain.



Gambar 2.7.2.Struktur System Unix Tradisional

Pada awalnya Unix memiliki struktur yang terbatas.Operating system unix memiliki 2 bagian terpisah yaitu:

- System Program
  - Kernel
- Terdiri dari semua hal yang berada diantara system call interface dan physical hardware. Menyediakan layanan sistem file ,CPU scheduling,manajemen memori, fungsi-fungsi sistem operasi lainnya.



Gambar 2.7.3. Operating system dengan pendekatan layer

Kominikasi antara layer pada pendekatan ini adalah sangat terbatas karena setiap layer hanya dapat berkomunikasi dengan satu layer dibawahnya dan satu layer diatasnya. Sebuah layer tidak bisa mengakses langsung layer yang tidak beradjacent dengan dirinya. Pendekatan ini kurang effisien. Sebagai contoh jika user program mengeksekusi operasi I/O, maka user program akan mengeksekusi system call yang mentrapnya di layer I/O,kemudian memanggil layer manajemen memory,lalu memanggil layer CPU scheduling,baru kemudian sampai ke layer hardware(I/O device). Di setiap layer parameter akan diubah dilewatkan ke layer berikutnya dan seterusnya, semua itu memerlukan waktu.Sehingga dikatakan layered approach kurang effisien dibandingkan dengan yang non layered.

### Mikrokernel

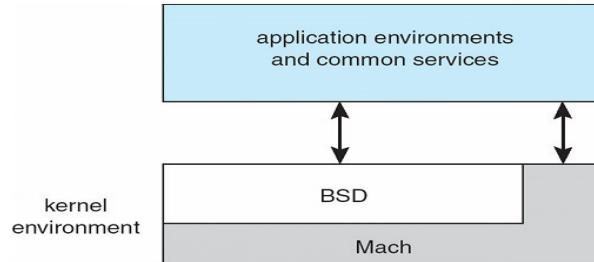
Ide utama dari mikrokernel adalah bagaimana kernel dari system operasi dibuat sekecil mungkin. Fungsionalitas yang tidak terlalu penting sebaiknya diletakan diluar kernel(user space). Namun tentu saja system call ke hardware,manajemen memori,CPU scheduling tetap berada dikernel. Fungsionalitas seperti manajemen file bisa dikeluarkan dari kernel dan ditempatkan di user space(windows dan mac) begitu juga dengan komunikasi.

Keuntungan dari pendekatan ini adalah

- Mudah untuk dikembangkan kerena kernelnya kecil
- Mudah memindahkan System operasi ke arsitektur yang baru
- Lenih handal(lebih sedikit kode yang berjalan di kernel)
- Lebih secure

Kelemahanya ada pada perfomance nya, karena banyak sekali dibutuhkan komunikasi antara user space dan kernel space.

Salah satu system operasi yang menggunakan mikrokernel adalah mac os.

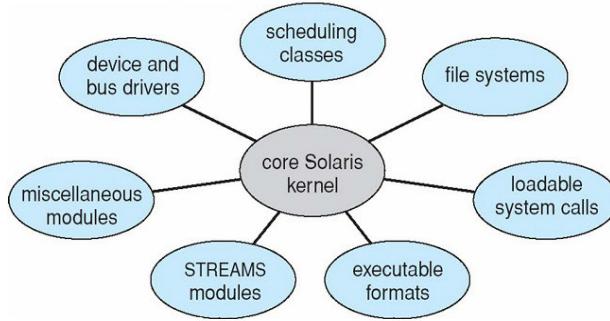


Gambar 2.7.4. Struktur Mac Os

Kernel environment mac os terdiri dari Mach mikrokernel dan BSD. BSD menyediakan layanan seperti BSD command line interface, yang mendukung sistem file dan networking, layanan implementasi POSIX APIs termasuk Pthread. Mach mikrokernel menyediakan layanan memory management, mendukung remote procedure call (RPCs) dan menyediakan layan IPC.

### Modul

Pendekatan mikrokernel biasanya digabung dengan modul. Kebanyakan System operasi sekarang menggunakan kernel modul. Biasanya menggunakan pendekatan objek oriented. Setiap komponen core dipisahkan. Maksudnya fungsionalitasnya dimasukan dalam modul-modul. Misalnya modul untuk CPU, modul untuk manajemen memori dll. Masing-masing module dapat diload sesuai kebutuhan. Secara keseluruhan mirip dengan layer tapi lebih fleksibel, karena sebuah module dapat berkomunikasi dengan module lain tanpa harus bertetangga/berdekatan.



Gambar 2.7.5 Pendekatan module pada solaris

Pada gambar terlihat bahwa fungsi kernel terlihat hanya sebatas pengatur komunikasi antara modul. Beberapa fungsionalitas yang sebelumnya berada di dalam kernel dikeluarkan dari kernel dan dibentuk dalam modul-modul tersendiri.

Kelemahan dari pendekatan ini adalah pada saat modul-modul tersebut berkomunikasi maka akan terjadi bottleneck.



### Virtual Machines(VM)

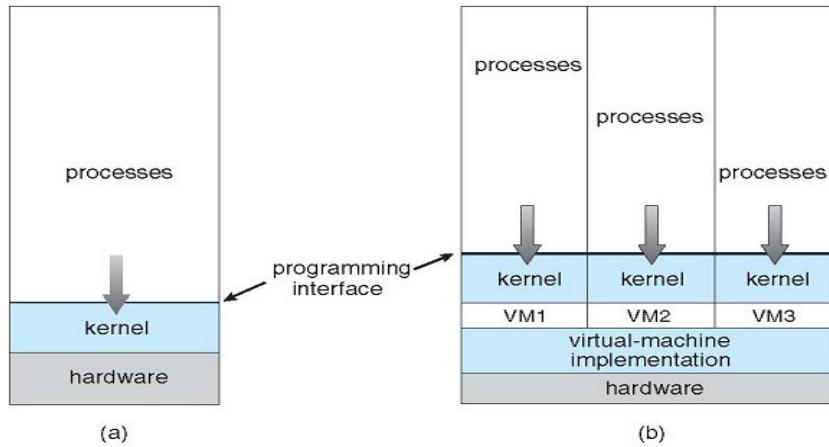
Sebelumnya kita telah melihat bagaimana cara menstruktur sebuah system operasi, yaitu dengan cara layering, mikrokernel dan modular. Ekstensi logis dari struktur tersebut dinamakan virtual machine. Virtual machine memperlakukan SO dan hardware seakan-akan adalah semuanya hardware. Jadi membuat satu layer lagi diatas SO. Virtual machine menyediakan layanan yang identik user seakan-akan kita mengakses hardware secara langsung. Jadi program yang berada diatas virtual machine merasa seolah-olah memiliki processor dan memory sendiri.

Operating system yang dijadikan tempat menginstall virtual machine disebut host dan system operasi yang diinstall diatas virtual machine disebut guest.

Virtual machine muncul pertama kali pada komputer mainframe IBM pada tahun 1972. Komputer ini terbagi dalam beberapa virtual machine dan masing-masing menjalankan system operasinya sendiri, namun juga secara bersama-sama menggunakan hardware yang sama. Setiap SO pada VM terproteksi dari yang lain namun masih dapat menshare file dan berkomunikasi lewat network.

Salah satu keuntungan menggunakan VM adalah pada proses development dan testing program (termasuk testing dan development SO). Dengan menggunakan VM kita dapat mengetest program yang kita buat ke SO yang berbeda tanpa harus berpindah-pindah komputer.

Pada komputer modern keuntungan paling besar dari adanya VM adalah **Consolidation**. Yaitu menggabung beberapa system berbeda yang memiliki resource sedikit ke satu virtual machine pada suatu sistem yang lain, sehingga dari gabungan sistem-sistem tersebut dapat dihasilkan sistem dengan resource yang lebih baik.



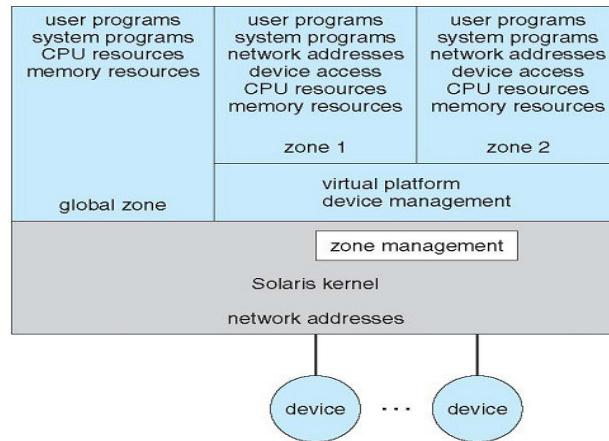
Gambar 2.7.6. Model System(a)Non Virtual machine,(b) Virtual machine

### Paravirtualization

Jika pada VM sistemnya benar-benar berusaha memimik hardware dari komputer. Misalnya untuk instruksi CPU bagi Guest yang berjalan pada VM adalah sama dengan instruksi CPU jika SO ingin mengakses CPU tanpa VM. Pada paravirtualization tidak seperti itu. Guest diberikan sistem yang tidak memimik langsung hardware. Jika ingin agar guest juga mendapatkan sistem yang memimik hardware, maka guest harus dimodifikasi. Hal ini sebenarnya memberikan keuntungan pada effisiensi penggunaan resource dan layer virtualisasi yang lebih kecil.



Pada paravirtualized guest dapat berupa SO(sama seperti di vm) tapi juga dapat berupa program aplikasi, seperti yang terdapat pada OS solaris. Pada OS solaris terdapat **Container** yang menciptakan layer virtual diantara SO dan aplikasi. Pada paravirtualization hanya terdapat satu kernel yaitu kernel pada hostnya. Dibanding VM yang menvirtualisasi hardware dan SO menjadi hardware,paravirtualized menyediakan proses di dalam container seolah-olah proses/aplikasi yang berjalan pada container tersebut hanya satu-satunya pada SO.CPU resourceny dibagi-bagi antara container namun setiap container merasa memiliki alamat network sendiri,port sendiri,user account sendiri dll.

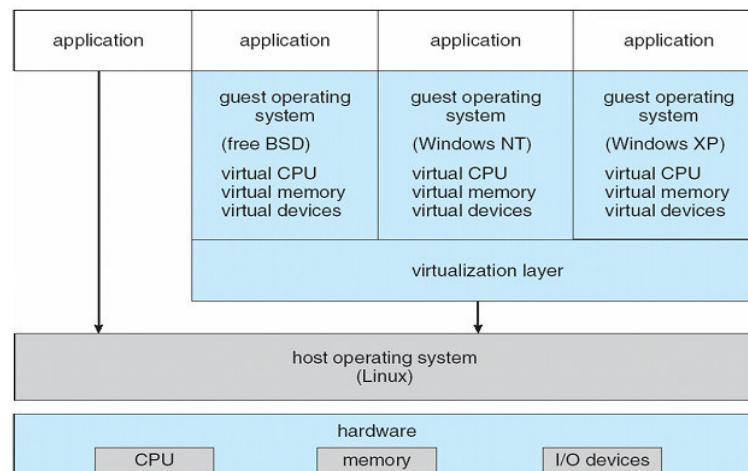


Gambar 2.7.7.Solaris dengan dua container

Contoh produk yang mengimplementasikan Virtual machine VMWare Workstation dan java Virtual Machine.

- VMWare

VMWare paling banyak digunakan untuk membangun Virtual machine. Setiap OS pada virtual machine dapat saling berkomunikasi, mempertukarkan file, manjalankan file audio di suatu OS dan dapat didengarkan di OS lainnya.



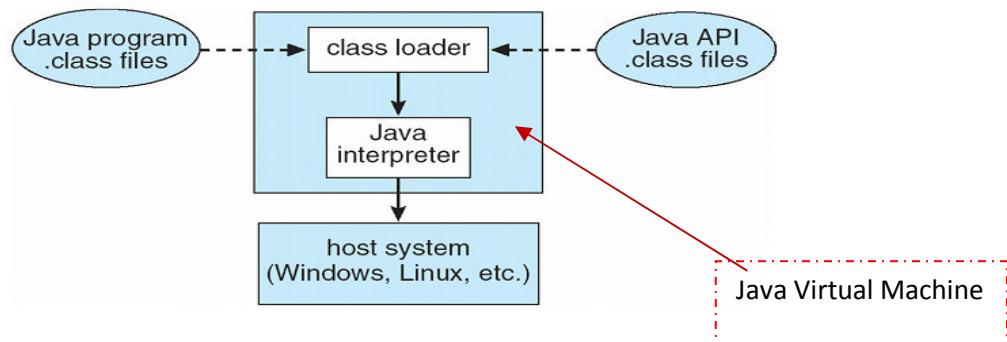
Gambar 2.7.8.Arsitektur Vmware



- Java Virtual machine

Para designer java menginginkan program java crossplatform ,dapat berjalan di SO yang berbeda. Untuk itu harus ada abstraksi yang sama terhadap mesin dibawahnya. Mac berbeda dengan windows begitu pula dengan SO lainnya.

Di java, program yang kita buat kita kompile menjadi class file,class file ini adalah file text biasa yang dapat dibaca di sembarang SO. Pada sisi yang lain terdapat Java API yang juga berbentuk class file(text file juga).Kemudian kedua class file tersebut diload oleh class loader dan kemudian dinterpretasikan oleh interpreter menjadi bahasa yang dimengerti oleh mesin dibawahnya.Jadi virtual machine nya adalah interpreter dan class loader nya.



Gambar 2.7.9.Java Virtual Machine

## Chapter 3 PROCESS

### 3.1.Process Concept

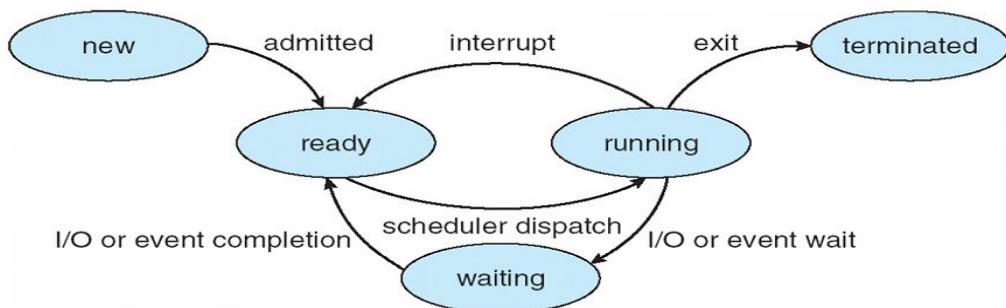
Pertanyaan yang muncul ketika membicarakan operating system adalah disebut apa semua aktivitas yang dilakukan oleh CPU. System operasi mengeksekusi banyak program .Program yang sedang dieksekusi oleh CPU disebut process. Pengertian dari sebuah proses lebih dari kode program yang terkadang disebut *section text*. Process termasuk aktivitas CPU pada suatu saat yang direpresentasikan oleh nilai dalam program counter dan isi dari register CPU, juga termasuk proses stack yang mengandung data temporari(seperti parameter,return address dan variabel lokal) dan data section yang mengandung variebal global. Sebuah proses juga termasuk heap,yaitu lokasi memori yang secara dinamis dialokasikan selama sebuah process berjalan.

Proses adalah entiti yang aktif sedangkan program adalah entiti yang pasif. Program dapat menjadi process ketika diload ke memori. Satu program mungkin saja memiliki banyak process.

### 3.2.State process

Sebuah proses sifatnya dinamis. Karena dinamis maka proses memiliki state yang menggambarkan aktivitasnya pada suatu waktu. State-state tersebut adalah :

- **New:** Proses baru saja dibuat
- **Running:** Proses sedang dieksekusi
- **Waiting:** Proses sedang menunggu suatu event(misalnya,I/O interrupt)
- **Ready:** Proses sedang menunggu untuk dieksekusi oleh CPU. Berbeda dengan waiting,ready menunggu karena mungkin disaat tersebut CPU masih mengeksekusi process lain. Biasanya ada yang namanya ready queue.
- **Terminated:** process telah selesai dieksekusi oleh CPU



Gambar 3.2.1 Diagram state dari Process

Perhatikan gambar 3.2.1;

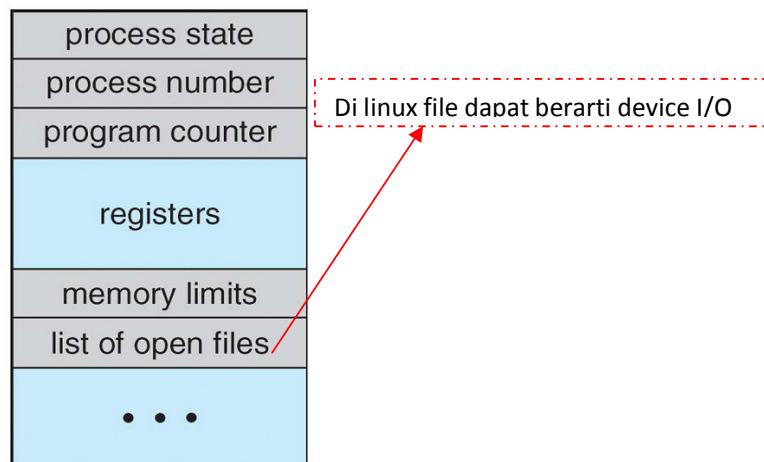
- Proses dari waiting tidak pernah langsung pindah ke running harus selalu melalui state ready dahulu
- Semua state tersebut dilakukan oleh CPU, jangan berpikir bahwa CPU hanya memiliki campur tangan ketika proses di state running.



### 3.3.Process Control Block(PCB)

Setiap proses direpresentasikan dalam Operating system oleh process control blok. PCB disimpan di memory dan mengandung informasi untuk suatu proses seperti:

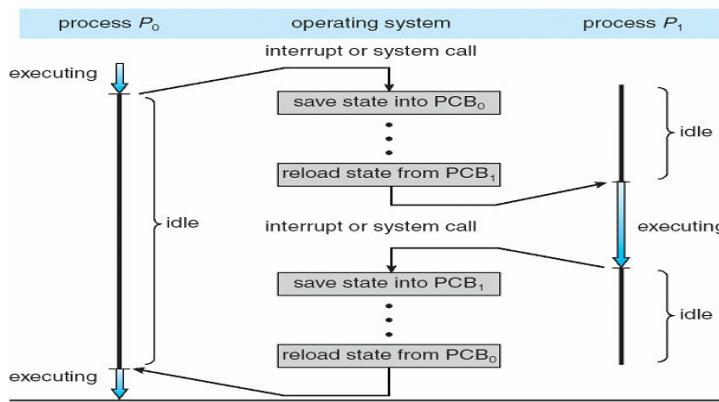
- Process State:  
Yaitu new,ready,running, waiting,terminated dan seterusnya.
- Program Counter:  
Mengidentikasikan alamat intruksi berikutnya yang harus dieksekusi oleh proses memiliki PCB ini.
- CPU register:  
CPU register berbeda-beda(jumlah dan tipenya) untuk komputer arsitektur yang berbeda.  
Didalam register biasanya terdapat accumulator,index register,stack pointer,general proposes register dan informasi kode kondisi lainnya. CPU register dan Program counter haruslah disimpan pada saat intrupsi, sehingga CPU dapat kembali lagi melanjutkan eksekusinya ke proses ini jika interupsi selesai.
- CPU –scheduling information:  
Yaitu informasi mengenai proses prioritas, pointer ke scheduling queue dan parameter scheduling lainnya.
- Memori management information:  
Yaitu Informasi memori limit , informasi mengenai base dll, tergantung dari system memori yang digunakan oleh System Operasi tersebut.
- Accounting Information  
Yaitu informasi mengenai jumlah proses, jumlah CPU, jumlah account dll.
- I/O status information  
Informasi list device I/O yang dialokasikan ke process ini,lokasi pointer(misalnya pada proses membaca file)



Gambar 3.3.1 Process Control Blok



Kapan PCB di update?



Gambar 3.3.2. CPU berpindah dari satu proses ke proses lain

PCB di update pada saat terjadi context switching, context switching terjadi ketika adanya interrupt, proses yang sedang dieksekusi memanggil system call, atau seperti pada system yang multi proses terdapat yang namanya time slice. Pada saat idle proses dapat berada di state ready atau waiting (terserah event nya). Time untuk menyimpan dan meload PCB harus secepat mungkin karena time yang digunakan CPU pada proses itu manfaatnya tidak secara langsung dirasakan oleh user.

### 3.4. Proses Scheduling

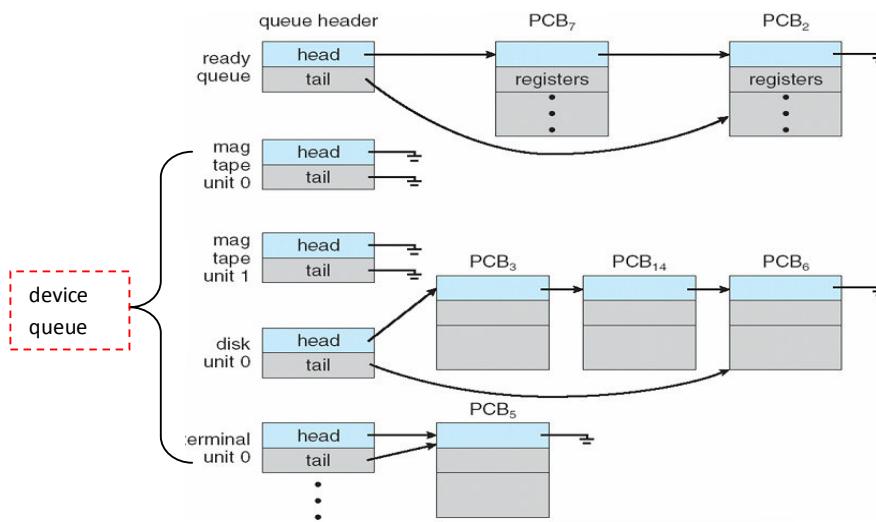
Konsep multi programing yang ada pada komputer modern sekarang ini adalah bertujuan untuk memaksimalkan penggunaan CPU. Oleh karena itu switching proses harus dilakukan secepat mungkin. Proses scheduler bertugas untuk menentukan proses mana yang berikutnya akan diproses oleh CPU. Biasanya pada system operasi modern terdapat scheduling queue diantaranya:

- **Job queue:** yaitu himpunan semua proses yang ada didalam system, baik itu proses yang sedang dieksekusi maupun yang belum dieksekusi yang masih berada di memory.
- **Ready queue** yaitu himpunan proses-proses yang ada didalam memory yang sudah siap untuk dieksekusi oleh CPU.
- **Device queue**

Setiap device memiliki device queue. Device queue berisi list semua proses yang sedang menunggu layanan dari device yang bersangkutan.

Proses selalu berpindah-pindah dari suatu queue ke queue lain. Queue biasanya diaplikasikan dengan link list. Header queue memiliki pointer ke PCB paling awal dan PCB paling akhir dari proses-proses. Pemilihan PCB mana yang akan diberi layanan atau dieksekusi terlebih dahulu dilakukan dengan beberapa cara (nanti dipelajari). Jadi tidak harus PCB yang paling dekat dengan header terlebih dahulu (misalnya PCB 7 pada gambar 3.3.3). Suatu proses tidak dapat berada dalam dua queue berbeda dalam suatu waktu. CPU hanya mengizinkan permintaan proses ke satu device pada suatu waktu tertentu.

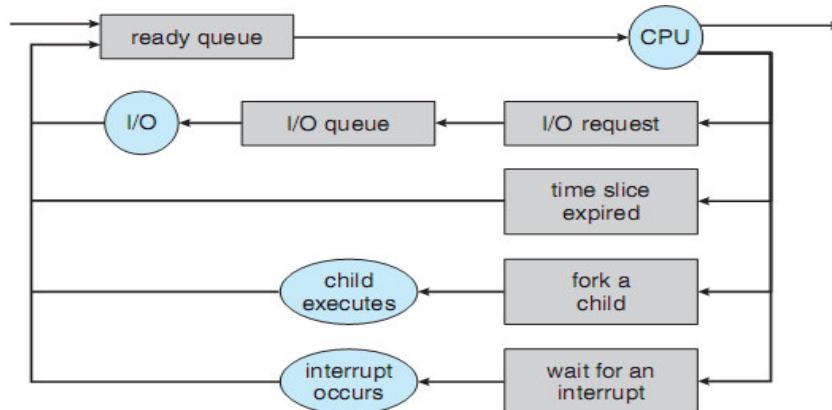




Gambar 3.3.3. I/O device queue

Proses yang baru dibuat pertama-tama dimasukan ke ready queue. Kemudian dia akan menunggu hingga dia dipilih untuk dieksekusi atau **dispatched**. Ketika dia diberikan CPU maka akan ada beberapa kemungkinan terjadi:

- Proses akan melakukan permintaan I/O sehingga dia lalu dipindahkan ke I/O queue, menunggu proses I/O. Setelah proses I/O selesai proses dikembalikan ke ready queue.
- Proses akan menghasilkan subprocess sehingga ia harus menunggu subprocess yang dihasilkannya diproses terlebih dahulu. Kemudian proses tersebut dikembalikan ke ready queue.
- Proses akan kehabisan time slice (pada multiprogramming), ia akan dikirim ke ready queue.
- Proses menunggu suatu interrupt, misalnya pada jaringan yang menunggu paket, jika paketnya tiba maka ia akan meginterupt CPU, lalu dia dimasukan ke ready queue.
- Proses yang telah dieksekusi akan diterminated, kemudian dia akan diremove dari semua queue dan resource akan didelokasi.



Gambar 3.3.4. Diagram queue, representasi dari proses scheduling

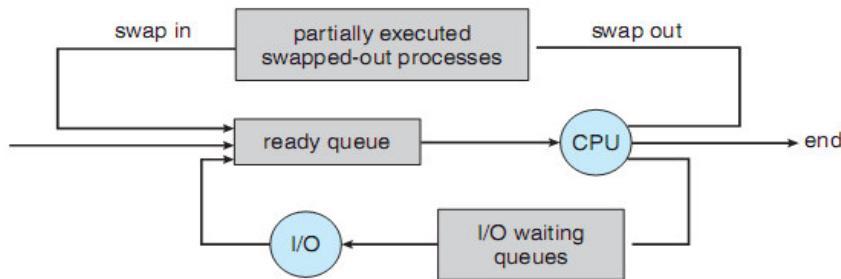


#### 4.4.Scheduler

Proses yang ada dalam queue akan dipilih oleh **scheduler** untuk dieksekusi(diberikan CPU). Dalam OS terdapat minimal dua scheduler yaitu

- **Long time scheduler** (job scheduler) bertugas untuk memindahkan proses dari pool nya ke memori yaitu tepatnya pada ready queue,
- **Short time scheduler** (CPU scheduler) bertugas untuk memindahkan proses dari ready queue untuk dieksekusi oleh CPU.

Beberapa operating system menambahkan medium time scheduler yang bertugas memindahkan/men-swap proses dari memory ke hardisk atau CPU ke hardisk (swap out) dan sebaliknya dari hardisk ke ready queue lagi(swap in). Hal ini terjadi biasanya karena kapasitas memory yang terbatas atau ada proses yang harus didahulukan eksekusinya.



Gambar 4.4.1 Penambahan medium time scheduler pada queue diagram

Short time scheduler biasanya dipanggil sangat sering sehingga proses pengambilan keputusanya harus cepat(algoritmanya harus efisien)sedangkan long time scheduler dipanggil tidak sering sehingga proses pengambilan keputusanya bisa sangat lambat(pertimbanganya sangat banyak). Long time scheduler mengontrol derajat multiprogramming,karena long time scheduler memilih proses-proses yang akan berada dalam ready queue, CPU akan berpindah-pindah dari satu proses ke proses lain dalam ready queue,oleh karena itulah mengapa dikatakan long time scheduler menentukan derajat multiprogramming. Semakin banyak proses-proses dalam ready queue maka derajat multiprogrammingnya semakin tinggi.

Terdapat dua karakteristik dari proses yaitu:

- Process I/O bound,menghabiskan waktu lebih banyak untuk melakukan proses I/O dibanding komputasi(many short CPU burst).Contohnya browser internet.
- Process CPU bound, menghabiskan waktu lebih banyak untuk komputasi(few long CPU burst),misalnya proses perkalian matriks,mungkin hanya diawalnya berhubungan dengan I/O untuk mengambil data matriksnya dari keyboard setelah itu melakukan proses komputasi.

Long time scheduler harus pintar-pintar memilih proses yang harus berada di ready queue, tidak akan efisien jika semua process yang ada di ready queue bersifat I/O bound karena akan ada proses menunggu I/O yang bisa saja sangat lama jadi harus seimbang antara proses yang I/O bound dan CPU bound.



#### 4.5.Context switching

Ketika CPU berpindah dari suatu proses ke proses yang lain maka system harus menyimpan state proses lama kemudian meload state proses baru, lalu berpindah ke proses baru tersebut melalui **context switch**. Context dari sebuah proses direpresentasikan oleh PCB. Waktu yang diperlukan oleh Context-switch overhead yaitu CPU tidak melakukan kegiatan penting saat switching. Hardware ikut menentukan waktu yang diperlukan oleh context switching karena ada beberapa hardware yang dapat memindahkan data register ke PCB dalam satu intruksi.

#### 4.6.Operasi Pada Proses

##### 4.6.1.Create Process

Proses dapat dibuat oleh proses lain melalui system call create-process. Parent proses membuat child proses(misalnya Windows Explorer memiliki child Power point di windows), jika child proses tersebut membuat proses lagi maka akan terbentuk tree of process. Setiap proses memiliki PID atau identifier, dengan PID sebuah proses di identifikasi dan dimanaged.

Dalam masalah resource(CPU time,memory,file,I/O) parent dan child dapat memilih pilihan berikut:

- Parent dan child menggunakan bersama resource, misalnya alokasi memory digunakan bersama.
- Child hanya dapat menggunakan sebagian resource parent nya.
- Parent dan child tidak menggunakan resource bersama(Windows Explorer dan Power point tidak saling menshare resource).

Dalam masalah Eksekusi:

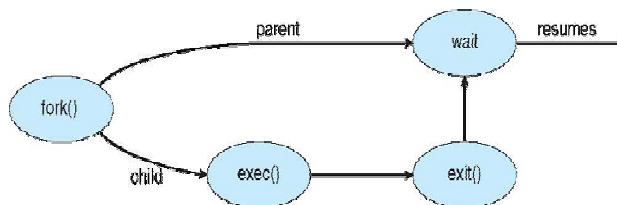
- Parent dan child dieksekusi secara concurrent
- Parent menunggu childnya dieksekusi terlebih dahulu baru dirinya dieksekusi.

Dalam masalah bentuk address space(bentuk memory)(figure 3.1 di buku):

- Child duplikasi format memori dari orang tuanya jadi child memiliki bentuk memory yang sama seperti orangtuanya(artinya memiliki data dan program yang sama).
- Child memiliki bentuk memory sendiri yang diload ke dirinya(program dan data nya berbeda, contohnya Windows Explorer dan powerpoint).

Contoh pada UNIX

- System call fork membuat proses baru
- System call exec digunakan setelah system call fork untuk mereplace bentuk memory dari proses dengan program yang baru.



Gambar 4.5.1 Proses Creation



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* fork a child process */
pid = fork();

if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
}
else if (pid == 0) { /* child process */
    execvp("/bin/ls", "ls", NULL);
}
else { /* parent process */
    /* parent will wait for the child to complete */
    wait(NULL);
    printf("Child Complete");
}

return 0;
}
```

Gambar 4.5.2 Membuat proses baru dengan system call fork pada UNIX

#### 4.6.2. Process termination

Sebuah proses diterminate karena bisa dengan salah satu dari dua kondisi berikut

1. Proses tersebut telah mengeksekusi statement terakhir dalam code programnya dan kemudian meminta System operasi untuk mendeletnya dengan memanggil system call exit(). Pada saat tersebut pula proses tersebut akan mengembalikan nilai ke parentnya melalui system call wait(). Semua resource yang dimiliki dan digunakan oleh proses tersebut seperti phisical maupun virtual memori, file yang dibukanya,buffer I/O dan lain-lain akan dealokasi oleh System Operasi.
2. Parent dapat menghentikan eksekusi child nya (system call abort()). Beberapa alasan mengapa parent men abort childnya yaitu:
  - Child tersebut telah menggunakan terlalu banyak resource
  - Child tersebut fungsinya tidak diperlukan lagi
  - Jika parent nya exit maka system operasi tidak mengizinkan childnya tetap berjalan. Semua childnya akan diterminate,kejadian ini sering disebut sebagai cascading termination

#### 4.7. Interproses Communication

Proses-proses dalam sebuah system dapat bersifat cooperating atau independent. Proses dikatakan cooperating jika proses tersebut dapat dipengaruhi atau mempengaruhi proses lain, dan sebaliknya untuk yang independent. Proses-proses yang melibatkan sharing data adalah cooperating proses.



Ada beberapa alasan proses cooperating yaitu:

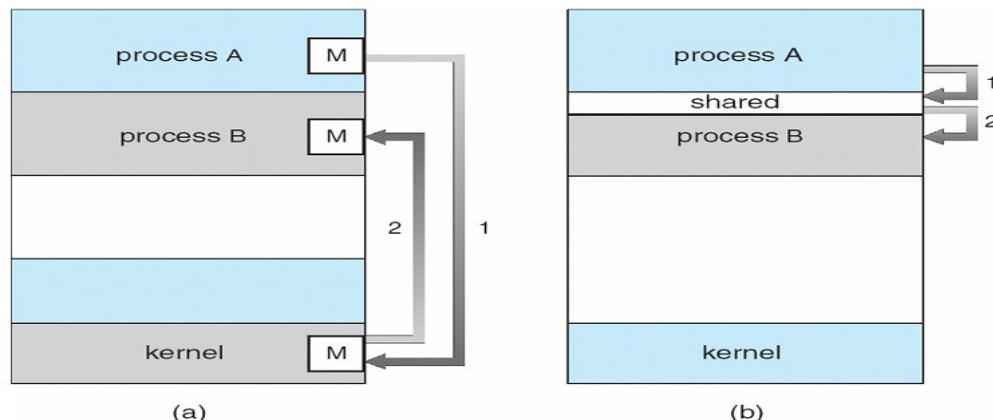
- Untuk sharing informasi, beberapa user bisa saja tertarik dengan data user lain.
- Computation speed up, jika kita ingin mempercepat suatu task, maka kita dapat membaginya dalam sub-sub task yang dapat kita kerjakan secara parallel. Sub-sub task tersebut pastinya saling mempengaruhi. Namun ingat bahwa suatu task dapat dikerjakan parallel dengan task lainnya jika sistem memiliki element proses lebih dari satu (misalnya CPU dan I/O channel).
- Modularity
- Convenience (kenyamanan), terkadang user mengerjakan beberapa task bersamaan, seperti editing, printing dan compiling.

Cooperating process membutuhkan mekanisme **Interprocess communication (IPC)** yang mengizinkan mereka untuk mempertukarkan data dan informasi.

Ada dua model dari IPC yaitu :

#### 1. Shared Memory

Pada mode ini suatu lokasi memory ditetapkan untuk dipakai bersama oleh beberapa proses yang saling cooperate. Proses-proses ini dapat saling mempertukarkan informasi dengan writing dan read dari lokasi memory yang telah ditetapkan. Normalnya sistem operasi tidak akan mengizinkan proses lain mengakses lokasi memory proses lain, oleh karena itu kedua proses harus saling setuju untuk menghapus retraksi ini. Bentuk data dan lokasinya ditentukan oleh kedua proses tidak dibawah kontrol sistem operasi. Proses-proses tersebut juga harus bertanggung jawab untuk tidak melakukan writing pada waktu bersamaan.



Gambar 4.7.1. Model Komunikasi (a) message passing dan (b) shared memory

Ilustrasi dari shared memory adalah masalah producer dan consumer. Proses Producer memproduksi informasi dan proses konsumen akan menggunakan informasi tersebut. Untuk melakukan hal tersebut maka kita memerlukan buffer untuk diisi oleh produser dan dikosongkan oleh konsumen.



Ada dua macam buffer yaitu:

- Unbounded buffer,memiliki ukuran tempat yang tidak terbatas
- Bounded buffer,memiliki kapasitas yang terbatas.

**Contoh implementasi dalam program untuk yang bounded buffer**

```
#define BUFFER_SIZE 10 //ukuran buffer

typedef struct{

    . . .

} item;

item buffer[BUFFER_SIZE];

intin = 0; //buffer mula-mula kosong

intout = 0; //buffer mula-mula kosong

//////////////////producer////////////////////

while (true) {//infinite loop
    /* Produce an item */
    while (((in = (in + 1) % BUFFER_SIZE) == out) //buffer full
        //di mod karena dalam array cycle
        ; /* do nothing --no free buffers */

    buffer[in] = item;//item yang diproduksi dimasukan ke buffer posisi ke i
    in = (in + 1) % BUFFER_SIZE;//posisi in di increment
}

//////////////////consumer////////////////////

while (true) {//infinite loop
    while (in == out)//buffer kosong
        ; // do nothing --nothing to consume

    // remove an item from the buffer

    item = buffer[out];//item diambil dari buffer posisi ke out
    out = (out + 1) % BUFFER_SIZE;//posisi out di increment
    //di mod karena dalam array cycle
}
```

Busy waiting,  
menunggu dan  
mengecek

Busy waiting,  
menunggu dan  
mengecek



## 2. Message Passing

Adalah mekanisme dari proses-proses untuk berkomunikasi dan mensikronkan action-action mereka tanpa harus share address space yang sama. Metode ini akan sangat powerful untuk proses-proses yang berada tidak dalam komputer yang sama namun terhubung oleh network.

Message passing menyediakan dua operasi yaitu:

- send(message)
- receive(message)

Jika P dan Q ingin berkomunikasi maka yang mereka lakukan:

- Men establish jalur komunikasi diantara mereka
- Saling mempertukarkan message melalui send dan receive

Komunikasi link yang harus dibentuk oleh P dan Q dapat diimplementasikan dengan

- secara fisik(misalnya shared memory(jika dalam satu komputer,seperti gambar 4.7.1(a)), hardware bus).
- secara logis:Direct Communication atau Indirect communication,automatic(dijelaskan nanti)

Dalam mengimplementasikan komunikasi link ,seorang programer harus mempertimbangkan:

- Bagaimana caranya link diestablish
  - Apakah bisa sebuah link digunakan oleh lebih dari dua process
  - Berapa banyak link diantara sepasang process yang saling berkomunikasi
  - Berapa kapasitas link
  - Apakah ukuran dari message yang dapat diakomodasi oleh link fixed size atau variabel
  - Apakah link tersebut bersifat uni directional atau bi directional.
- a. Direct Communication
- Process harus saling memberi nama secara explisit,ini penting karena di direct communication harus jelas pesan mau dikirim kemana atau diterima dari siapa.
- Send(P,message)---mengirmkan pesan ke proses P
  - Receive (Q,message)---menerima pesan dari proses Q

Properti yang harus dipenuhi oleh link komunikasi pada mode direct communication:

- Link harus diciptakan secara otomatis oleh system operasi
- Sebuah link harus diasosiasikan hanya kepada sepasang proses,jadi satu link tidak boleh untuk lebih dari dua proses.
- Diantara sepasang proses hanya ada satu buah link
- Link nya dapat bersifat uni directional atau bi directional



b. Indirect communication

Pada indirect communication terdapat perantara mailbox atau sering disebut port diantara proses-proses yang saling berkomunikasi. Setiap mailbox memiliki id yang unik. Proses hanya dapat berkomunikasi dengan proses lain hanya jika menshare mailbox.

Properti yang harus dipenuhi oleh sebuah link dalam indirect communication adalah:

- Link akan diestablish hanya jika ada lebih dari satu proses yang meshare mailbox
- Sebuah link dapat digunakan oleh banyak proses
- Masing-masing pasangan proses yang saling berkomunikasi dapat menshare beberapa communication link yang berbeda.
- Link dapat unidirectional atau bidirectional

Selain operasi mengirim dan menerima informasi dari mailbox, pada indirect terdapat operasi untuk menciptakan mailbox baru dan menghapus mailbox jika tidak diperlukan lagi.

Perintah mengirim dan menerima dapat didefinisikan secara primitive:

- Send(A,message):--mengirim message ke mailbox A
- Receive(A,message):---menerima message dari mailbox A

Namun sebenarnya terdapat masalah jika mailbox di share, perhatikan ilustrasi berikut

- P1,P2 dan P3 men share mailbox A
- P1 mengirim message ke mailbox A, P2 dan P3 mengeksekusi perintah receive()
- Proses mana yang akan menerima pesan yang dikirim oleh P1?

Jawabanya tergantung dari cara mana yang dipilih dari beberapa cara dibawah ini:

- Satu buah link hanya diperbolehkan tidak lebih dari dua proses(mirip dengan direct communication)
- Mengizinkan hanya satu proses yang mengeksekusi perintah receive() dalam satu waktu
- Mengizinkan sistem yang memilih secara acak proses mana yang akan menerima pesan tersebut. Kemudian sender diberitahu proses mana yang menerima pesannya.
- Atau mengizinkan semua proses menerima pesan tersebut(broadcasting)

### Synchronisasi

Pada message passing terdapat dua macam design perintah primitive send() dan receiver():

- **Blocking atau synchronous**

**Blocking send** → Proses sending akan diblok hingga message diterima oleh proses yang menerima atau mailbox penerima.

**Blocking receiver** → Penerima akan blok hingga dia menerima sebuah pesan



- **Non Blocking atau asynchronous**

**NonBlocking send** → Proses sending dapat dialakukan lagi walaupun tidak menunggu ack bahwa pesan sudah diterima

**Nonblocking receiver** → Penerima tidak akan blok walaupun dia belum menerima pesan, receiver dapat melakukan hal lain atau mengulang melakukan receiver()

Pemilihan blok atau non blok kembali lagi ke aplikasi yang akan kita buat, dengan memilih non blok maka aplikasi kita akan lebih fleksibel karena tidak harus menunggu namun kita harus melakukan pemrograman paralel yang cukup rumit. Dengan blocking akan lebih sederhana namun tidak fleksibel.

### **Buffering**

Baik komunikasi langsung maupun tidak langsung melakukan komunikasi data dengan diperantara oleh buffer atau queue. Queue diimplementasikan dengan 3 cara:

1. Zero capacity(tidak ada buffer)  
Jadi jika sender ingin mengirim maka ia harus menunggu receiver.
2. Bounded capacity, panjang dari buffer terbatas sehingga jika buffer full maka sender harus menunggu hingga ada tempat di buffer
3. Unbounded capacity,panjang buffer tak terbatas.  
Pengirim tidak pernah menunggu.Metode seperti ini diaplikasikan di internet dimana sender dapat terus mengirim paket tanpa harus menunggu ack dari receiver.

Kebanyakan dari system operasi mengimplementasikan ke dua jenis IPC diatas sekaligus yaitu shared memory dan message passing,. Message passing sangat useful untuk data dalam jumlah kecil(karena tidak banyak konflik yang akan ditemui),juga sangat powerfull dan mudah diimplementasikan pada komunikasi inter komputer dibanding shared memory. Sebaliknya shared memory dapat memberikan kecepatan proses komunikasi yang lebih dibandingkan message passing. Hal ini disebabkan karena message passing membutuhkan system call dalam operasinya,pemanggilan system call membutuhkan extra waktu,sedangkan shared memory hanya perlu men-estabilis link lalu komunikasi bisa berjalan.



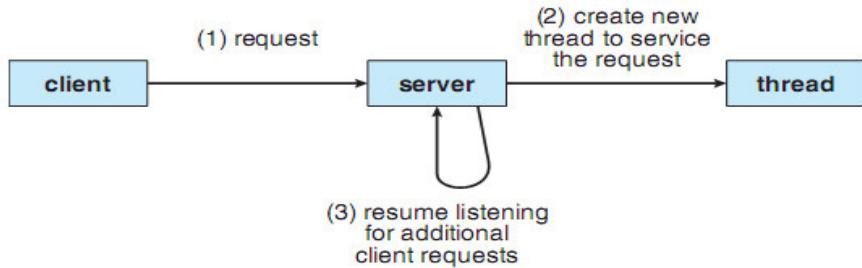
## CHAPTER 4. Threads

### 4.1.Motivation

Banyak dari software yang berjalan di System Operasi modern menggunakan multithread. Sebuah web browser mungkin saja memiliki thread untuk menampilkan gambar atau teks,dan thread yang lainya bertugas untuk mengirim dan menerima data dari jaringan. Word prosesor memiliki thread yang bertugas untuk menampilkan graphic ,thread lainya bertugas untuk merespond huruf yang ditekan di keyboard oleh user, thread yang lainya lagi melakukan checking spelling dan grammar dibelakang layar.

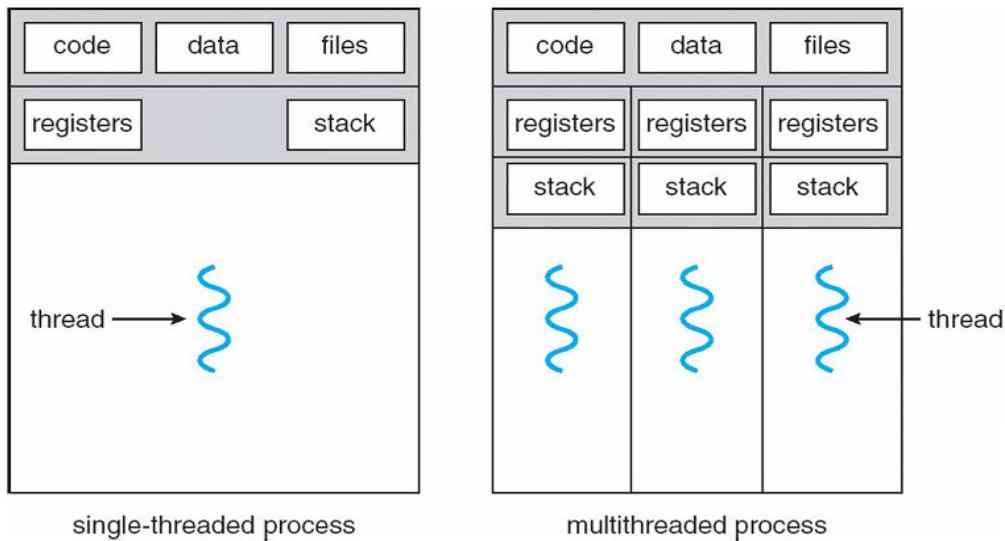
Pada suatu waktu tertentu sebuah aplikasi pasti memiliki beberapa tugas yang sejenis. Sebagai contoh sebuah webserver dapat menerima permintaan halaman web, image,sound dll dari banyak sekali client secara bersamaan. Jika webserver tersebut hanya mengimplementasikan single proses maka hanya satu client yang bisa dilayani dalam satu waktu. Hal ini menyebabkan client lain akan menunggu sangat lama.

Solusinya adalah setiap kali server menerima request maka ia harus menciptakan satu proses terpisah untuk melayani request tersebut. Meskipun demikian men-create proses membutuhkan waktu dan resource yang tidak sedikit. Pertanyaannya, mengapa proses baru harus diciptakan jika hanya melakukan pekerjaan yang sama seperti pekerjaan proses sebelumnya yang sudah ada dan itu pun menyebabkan overhead? Yang sebenarnya akan lebih efisien jika membuat sebuah proses dengan multi thread untuk menangani task-task tersebut. Web server hanya perlu menciptakan satu proses untuk memberikan layanan halaman web,image,sound dan lain-lain namun didalam nya terdapat thread-thread yang dapat melayani beberapa request client.



Gambar 1.1.1.Arsitektur server multithread

Kebanyakan dari system operasi sekarang memiliki kernel yang sifatnya multi thread. Beberapa thread bekerja pada suatu kernel, misalnya untuk managing device dan yang lainya meng handling interrupt.



Gambar 4.1.2 Proses Single dan Multithread

#### 4.2.Keuntungan Multi thread

Sebenarnya tidak ada larangan untuk menggunakan multi proses namun dengan menggunakan multi thread kita mendapatkan beberapa keuntungan diantaranya

- **Responsive**  
Ini dikarenakan thread lebih ringan daripada proses pada saat context switching. Pada saat context switching kita harus menyimpan PCB yang cukup berat dari sebuah proses sehingga memerlukan waktu yang cukup lama,dengan menggunakan thread yang perlu disimpan hanya register dan stack nya saja lalu berpindah ke thread yang lain.Oleh karena itu juga multithread lebih interaktif. Beberapa aplikasi multithread mengizinkan programnya terus berlangsung meskipun ada bagian dari program yang berhenti atau harus melakukan operasi yang panjang. Contohnya web browser mengizinkan user berinteraksi dengan layanan yang ditangani oleh thread lain dalam web browser, sementara thread lainnya meload image.
- **Resource sharing**  
Proses dapat melakukan resource sharing dengan menggunakan shared meory maupun message passing. Kedua IPC tersebut memerlukan campur tangan kernel,ikut campurnya kernel memrlukan extra waktu. Dengan thread tidak perlu campur tangan kernel karena mereka menshare resource dari proses secara default dimana mereka berada.Jadi tidak perlu ada bantuan kernel resourcenyasudah otomatis tersharing.
- **Economy**  
Mengalokasikan memory dan resource untuk membuat proses tentu saja tidak murah. Beda halnya dengan hanya membuat thread karena resourcenyadiambil dari proses dimana dia berada,begitu pula pada saat context switch yang disimpan cukup kecil(gister dan stack saja)rsehingga memakan resource yang sedikit. Oleh karena itu lebih murah.



- Scalability

Kita dapat menciptakan banyak thread untuk satu proses karena tidak akan memakan banyak resource ketimbang harus menciptakan banyak proses.

### Multicore programing

Yaitu komputer terdiri dari lebih dari satu prosesor dan berada dalam satu chip atau sering disebut multicore processing. Membuat program di multi core system sangatlah rumit, sehingga biasanya programer hanya sebatas membuat program yang multi thread, sistem operasi lah yang akan mengatur multicore nya. Misalnya SO menemukan dua buah thread yang tidak saling berhubungan maka secara otomatis SO akan menempatkan masing-masing mereka di dua core berbeda sehingga dapat berjalan secara paralel.

Jika programer yang mengambil alih pemrograman multi core nya maka ada beberapa masalah yang harus dipecahkan oleh programer tersebut:

- Dividing activity

Bagaimana caranya menbagi thread program kita, ke core yang mana yang harus kita berikan.

- Balancing

Bagaimana membuat beban pekerjaan core-core yang ada seimbang.

- Data splitting

Bagaimana membagi data

- Data dependency

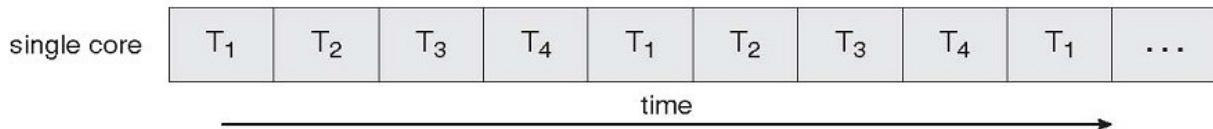
Setelah mebagi data bagaimana menjaga dependency data-data tersebut, jangan sampai tidak konsisten.

- Bagaimana melakukan testing dan debugging

Ketika program berjalan di berbagai core pastinya memiliki bagian eksekusi yang berlainan sehingga sangat sulit untuk melakukan debugging maupun testing.

### Perbedaan antara eksekusi thread di single core dan multi core

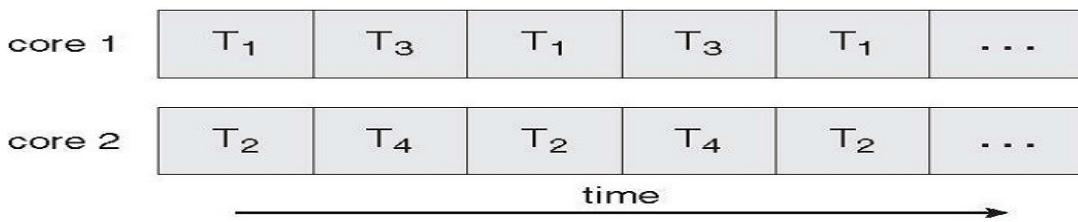
- single core



Gambar 4.2.1 Eksekusi yang Concurrent di single core

Prosesor berpindah-pindah dari thread yang satu ke thread yang lain dengan sangat cepat, lebih cepat dari respon mata manusia, sehingga user menganggap thread-thread tersebut berjalan bersamaan namun sebenarnya tidak (paralel semu). Proses eksekusi yang seperti ini disebut eksekusi yang concurrent.

- Multicore



Gambar 4.2.2. Eksekusi yang paralel di Multi core

Pada multi core dua buah thread yang tidak saling mempengaruhi dapat dipisah ke masing-masing core dan kemudian dieksekusi secara paralel(paralel yang real). Namun sebenarnya jika dilihat dari satu core, maka eksekusi nya juga menerapakan eksekusi yang concurrent.

#### 4.3. Multithread Model

Terdapat dua model multithread yaitu

- User thread  
Thread yang dimanajemen oleh library threads pada level user tanpa campur tangan kernel.  
Ada 3 thread library yang utama yaitu:
  - Posix **Pthread**
  - Win32 threads
  - Java threads
- Kernel thread  
Thread yang di support oleh kernel/operating system, semua operating system modern mensupport kernel threads.

User thread berada pada level user sebenarnya tidak berjalan, yang berjalan adalah kernel thread. Lalu bagaimana hubungan keduanya? Biasanya user kernel dipasangkan dengan suatu kernel thread. Jadi jika user menginvoke user thread maka sebenarnya yang akan berjalan adalah kernel thread pasangannya dari user kernel tersebut. Bagaimana memasangkannya?

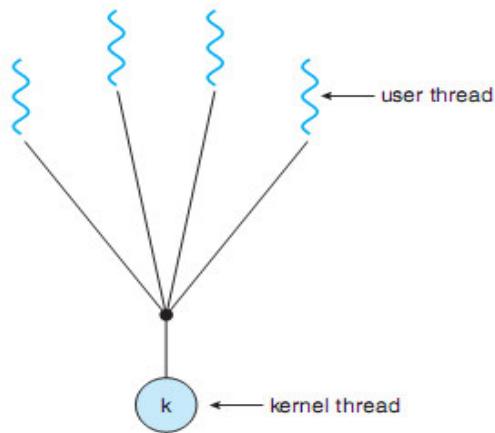
Ada beberapa model pemasangan user thread dan kernel thread

- Many to one  
Beberapa user thread dipetakan ke satu kernel thread.  
Contohnya:
  - Solaris green threads
  - GNU portabel threads

Management thread dilakukan oleh library thread di level user, setiap user kernel akan diberikan waktu oleh library thread untuk menggunakan kernel thread. Namun semua proses akan blok/berhenti jika satu user kernel mengalami blocking system call(misalnya salah satu thread memanggil I/O proses dan harus menunggu layanan dari I/O tersebut untuk waktu yang lama).



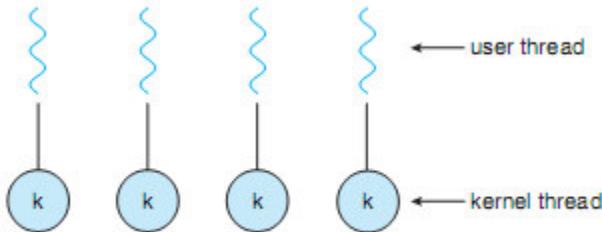
Juga karena hanya satu thread yang dapat mengakses kernel thread pada satu waktu maka thread pada model ini tidak mungkin dijalankan secara paralel pada system multi core.



Gambar 4.3.1 model many-to-one

- One-to-One

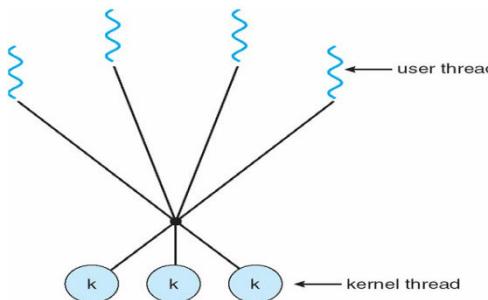
Satu user thread dipetakan ke satu kernel thread. **Keuntungannya** kita tidak perlu memikirkan masalah jika satu user kernel mengalami block, dan thread pada mode ini dapat dijalankan secara paralel pada system multi core. **Kerugiannya**: setiap kita menciptakan user thread maka harus diciptakan juga satu kernel threads. Hal ini akan menyebabkan overhead dan merusak perfomance dari aplikasi.



Gambar 4.3.2.Model one-to-one

- Many-to-many

Memetakan banyak user threads ke banyak kernel thread. Keuntungannya, bila satu thread blok maka thread yang lain masih bisa berjalan. Kemudian thread dalam mode ini dapat berjalan secara paralel pada system multicore.



Gambar 4.3.3.Model many-to-many



Selain 3 model diatas ada satu model baru yaitu two level model, model ini sama dengan many to many namun model ini juga mengizinkan suatu user threads dapat diikatkan ke suatu kernel thread.

#### 4.4.Thread Library

Thread library menyediakan programer API untuk men-create dan memanajemen thread. Ada dua pendekatan untuk mengimplementasikan thread library :

- Library nya terdapat di user space,termasuk semua code dan datanya terdapat pada user space. Ini artinya memanggil fungsi yang ada dalam library ini menghasilkan panggilan fungsi secara lokal(pengaruhnya di user space saja) dan tidak mempengaruhi system call.
- Pendekatan kedua adalah dengan library level kernel yang disupport langsung oleh Operating system. Pada kasus ini code dan data nya berada dikernel space. Melakukan invoking fungsi melalui API untuk library ini akan menghasilkan pemanggilan system call ke kernel.

#### Pthread

Phtread dapat digunakan untuk level user maupun level kernel. Phtread merupakan standar API pada POSIX(IEEE 1003.1c) untuk pembuatan dan syncronisasi threads. API dari Pthread menspesifikasi kelakuan dari library thread bukan implementasinya, untuk implementasinya diserahkan kepada desiner system operasi,bagaimana desiner tersebut menginginkan pengembangan library nya. Pthread banyak digunkan di keluarga system operasi UNIX.

#### Java Thread

Java thread dikelola oleh JVM. Biasanya diimplementasikan dengan menggunakan model thread yang diatur oleh OS. Di java thread kita tidak dapat mengakses kernel thread. Java Thread dapat dihasilkan dengan cara

- Extending thread class,menciptakan class baru yang diturunkan dari thread class.
- Mengimplementasikan Runnable Interface.

#### 4.5.Thread Issue

Dalam membangun program multithread terdapat beberapa macam issue yang harus dilihat oleh seorang programer yaitu:

- **Semantik dari system call fork() dan exec()**  
Pada bab sebelumnya telah dijelaskan mengenai semantik dari system call fork() dan exec() namun di ranah mulithread programing semantik tersebut berubah. Jika suatu thread di program memanggil fork() apakah proses yang baru terbentuk menduplicate semua thread dari proses dimana threadnya yang memanggil fork() tersebut berada?atau hanya menghasilkan proses dengan single thread?.Jika suatu thread meng-invoke exec() maka program yang ada pada parameter exec() akan menggantikan semua proses termasuk semua thread nya. Fork mana yang akan dipilih terserah dari aplikasi yang akan dibuat.



- **Thread cancellation**

Jika sebuah thread telah selesai dengan tugasnya maka thread tersebut akan mematikan dirinya sendiri. Namun bagaimana caranya menghentikan sebuah thread ketika thread tersebut masih bekerja/belum selesai?.

Ada dua cara yaitu:

1. Asynchronous cancellation

Yaitu menterminate thread tersebut sesegera mungkin

2. Deffered cancellation

Thread target harus selalu mengecek apakah ada signal dari system operasi ke dirinya untuk segera menterminate dirinya sendiri. Karena pada kasus ini tidak ada cara untuk menterminate thread target dari luar, haruslah thread tersebut yang dapat menghentikan dirinya sendiri. Ssystem seperti yang paling banyak diimplementasikan di system Operasi atau di java. Misalnya sebuah thread lagi membuka file bagaimana caranya menterminate nya jika SO tidak memiliki CPU, CPU masih berada pada thread tersebut.

- **Signal Handling**

Signal handling digunakan di linux untuk memberitahu sebuah proses bahwa suatu kejadian/event telah terjadi. Sebuah signal handler digunakan untuk meproses signal. Secara umum signal memiliki pola dasar:

- Signal digenerate oleh suatu event
- Signal kemudian di deliver ke suatu proses
- Signal di handle/ditangani

Menangani signal pada proses dengan single thread adalah sederhana, cukup me deliver ke proses tersebut, namun jika prosesnya memiliki multi thread maka ada beberapa pilihan:

- Deliver signalnya ke thread dimana signal tersebut muncul/applies
- Deliver signal ke setiap thread dalam proses tersebut(akan menginterupsi/mengganggu semua thread)
- Deliver signalnya ke suatu/certain thread di proses
- Menugaskan sebuah thread untuk menerima semua signal yang masuk ke proses dimana ia berada lalu memforwardnya ke thread yang sesuai(pada saat menciptakan thread, programer harus memberikan kode ke setiap thread tentang signal yang bersesuaian dengannya)

- **Thread pool**

Pada contoh sebelumnya kita menyebutkan contoh mengenai web server yang multi thread. Setiap kali server menerima request maka server akan menciptakan thread untuk menangani request tersebut. Jumlah waktu yang dibutuhkan untuk menciptakan setiap thread akan sangat mempengaruhi pelayanan. Selain itu jika setiap adanya request server harus membuat sebuah thread maka jumlah thread yang dibuat akan tidak terbatas sehingga menghabiskan resource seperti CPU time dan memori. Solusinya adalah dengan membuat **thread pool**. Pada saat system start up/booting maka system membuat sejumlah thread dan meletakanya dipool. Thread-thread tersebut ditidurkan menunggu pekerjaan dari SO. Seperti ketika server menerima request baru maka sebuah pool dibangunkan untuk melayani request tersebut, jika thread tersebut telah melakukan tugasnya maka ia kembali lagi ke pool. Jika suatu saat tidak ada lagi thread di pool maka SO harus menunggu sebuah pool selesai mengerjakan tugasnya.



Keuntungan thread pool:

- Biasanya menangani request lebih cepat dengan menggunakan thread pool dibanding harus membuat thread baru setiap adanya request.
- Kita dapat mengontrol banyaknya thread.

- **Thread specific data**

Suatu thread biasnya menshare data yang ada pada proses dimana ia berada. Namun terkadang suatu thread membutuhkan datanya sendiri tanpa diganggu thread lain. Contoh pada sistem proses transaksi, setiap transaksi dilayani oleh suatu thread, sehingga setiap transaksi memiliki identifier yang unik. Oleh karena itu untuk menyesuaikan dengan transaksi, thread harus menggunakan thread specific data. Hal ini juga berguna disaat kita tidak dapat mengontrol banyaknya thread yang dcreate (seperti jika kita tidak menggunakan thread pool).

- **Scheduller activation**

Dua model M:M dan model two level membutuhkan komunikasi untuk menentukan jumlah yang sesuai dari kernel thread yang dapat dialokasikan ke aplikasi. Scheduler activation menyediakan up calls yang memberikan mekanisme pemanggilan dari kernel ke thread library di user. Komunikasi ini memberikan kemampuan sebuah aplikasi menentukan jumlah kernel thread yang sesuai dengan kebutuhannya.

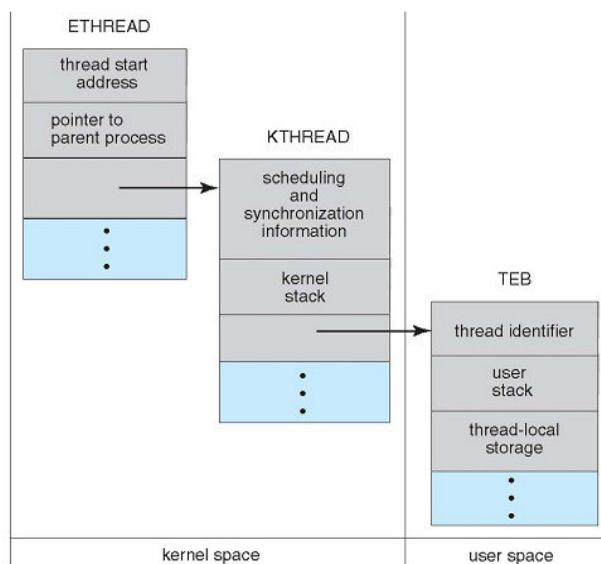
#### 4.6. Contoh Implementasi Thread di Windows dan linux

- Windows

Windows mengimplementasikan pemetaan model one-to-one. Masing-masing thread memiliki:

- Thread id
- Register set
- Stack yang terpisah antara user dan kernel
- Area penyimpanan data privat

Contxt thread



Gambar 4.6.1 Struktur data thread pada windows XP



Windows memiliki 3 jenis thread yaitu

- Ethread(executive thread block):Thread ini berada di level kernel.Thread ini hanya menyimpan alamat memori dimana thread mulai berjalan dan pointer ke parent process. Kernel ini tidak aktif(pasif).
  - Kthread(kernel thread block):Thread inilah yang aktif di level kernel.
  - TEB(thread environment block):Thread ini berada di user space
- Linux
- Jika kita membaca manual linux maka sebenarnya linux lebih menggunakan istilah task ketimbang thread.Linux melayani pembuatan proses dengan system call fork(). Untuk membuat thread linux menyediakan system call clone(). System call clone() memiliki parameter(flag) yang berguna untuk menentukan hubungan system sharing antara parent dan child:

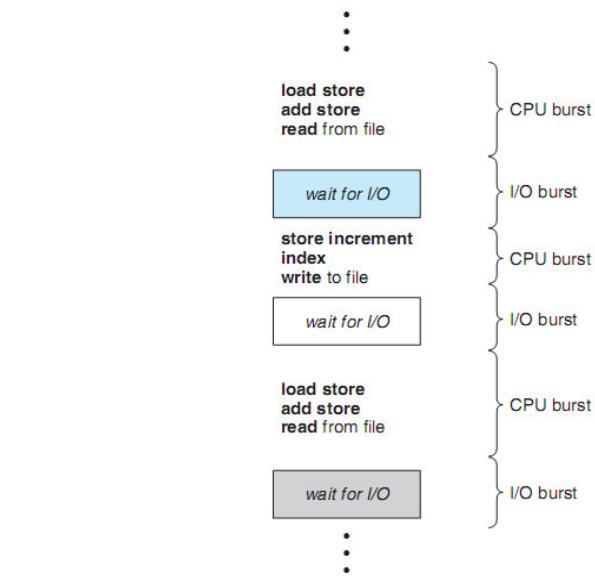
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.



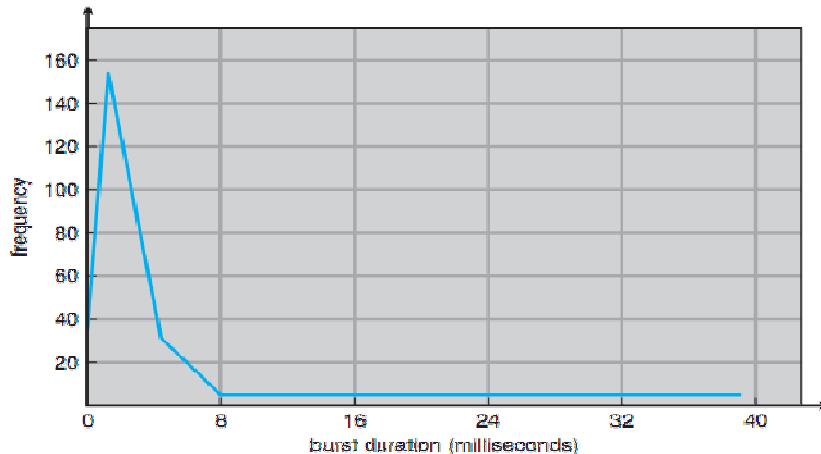
## CHAPTER 5. CPU Scheduling

### 5.1. Basic Concept

Pada sistem single prosesor hanya satu proses yang dapat berjalan pada suatu waktu, proses lain harus menunggu CPU free dan kemudian dischedule untuknya. Objektif dari multiprogramming adalah bagaimana caranya beberapa proses dapat dieksekusi sepanjang waktu guna memaksimalkan penggunaan CPU. Ide nya sederhana, sebuah proses dieksekusi hingga ia menunggu proses I/O. Pada sistem computer sederhana, pada saat itu CPU akan idle (membuang-buang waktu). Dengan multiprogramming kita berusaha menggunakan waktu tersebut sehingga lebih produktif. Beberapa proses tetapi berada di memory, jika sebuah proses lagi menunggu I/O maka System Operasi akan mengambil CPU dari proses tersebut dan memberikannya ke proses lain. Pola ini diulang terus menerus. Menentukan/menschedul proses yang akan diberi CPU adalah fungsi terpenting dari System Operasi. Keberhasilan scheduling tergantung dari dua property proses yang dieksekusi yaitu terdiri dari dua point, yaitu CPU cycle (CPU burst) dan I/O wait (I/O burst).



Gambar 5.1.1 Pergantian urutan antara CPU burst dan I/O burst



Gambar 5.1.2 Histogram dari CPU burst



Berdasarkan hasil penelitian frekuensi penggunaan CPU burst yang panjang sangatlah jarang, sedangkan CPU burst yang pendek frekuensinya sangat tinggi. Seperti yang diperlihatkan pada grafik diatas. Frekuensi dari CPU burst dari 0-8 milisecond meningkat hingga 150 namun perlakuan-lahan menurun pada CPU burst dengan durasi 6-8 milisecond dan selanjutnya durasi diatas 8 milisecond frekuensinya hampir semua dibawah 10. Distribusi ini sangat penting dalam penentuan algoritma scheduling yang baik.

### 5.2.CPU Scheduler

CPU scheduler bertugas untuk memilih proses dari ready queue dan mengalokasikan CPU ke proses tersebut. Records yang ada di queue biasanya adalah PCB dari proses. CPU scheduler akan dipanggil (CPU scheduler juga berbentuk program) ketika:

1. Pergantian state dari state running ke wait, misalnya karena request ke I/O, maka CPU scheduler harus memilih proses lain dari ready queue untuk menggantikan proses yang wait tadi, untuk diberikan CPU.
2. Ketika pergantian dari state running ke state ready, misalnya karena time slice habis atau interrupsi lain.
3. Ketika berpindahnya proses dari waiting ke state ready, misalnya karena operasi I/O yang dibutuhkan suatu proses telah selesai.
4. Ketika suatu proses di terminate.

Scheduler yang mengimplementasikan 1 dan 4 disebut **nonpreemptive** (non preemptive=tidak dapat bekerja sama) sedangkan untuk 2 dan 3 disebut **preemptive**. Pada Nonpreemptive scheduler, CPU hanya akan bisa diambil oleh System Operasi jika proses tersebut melepas kannya sendiri karena terminate atau proses tersebut menunggu proses I/O, jadi proses tidak dapat diinterupsi. Kebanyakan SO modern menerapkan yang preemptive scheduler. Namun dengan preemptive scheduler ada beberapa yang harus diawasi yaitu:

- Misalnya jika dua buah proses mengshare data, jika sebuah proses sedang running misalnya sedang menulis sebuah data lalu diinterupsi kemudian proses lain yang running berikutnya ingin membaca data tersebut akan ada inkonsistensi data.
- Pertimbangkan preemptive (interupsi) ketika di kernel mode
- Perimbangkan juga jika preemptive terjadi ketika sebuah proses sedang menjalankan aktivitas OS yang krusial.

### 5.3. Dispatcher

Komponen lain yang termasuk dalam fungsi CPU scheduling adalah dispatcher. Dispatcher adalah modul yang bertugas untuk memberikan kontrol CPU ke proses yang dipilih oleh scheduler CPU. Fungsi tersebut termasuk:

- Switching Context
- Switching ke mode user
- Melompat ke lokasi yang tepat pada program user untuk merestart program tersebut.

Karena dispatcher dipanggil di setiap context switching dan dispatcher memerlukan waktu untuk menstop sebuah proses kemudian men start proses lain (**dispatcher latency**) maka dispatcher harus secepat mungkin. Dispatcher latency mengakibatkan overhead.



#### 5.4 Kriteria Scheduling

Kriteria untuk pemilihan algoritma CPU scheduling yaitu:

- **CPU utilization**

Usahakan CPU dapat sibuk setiap saat.

- **Throughput**

Yaitu banyaknya proses yang eksekusinya komplate dalam satu satuan waktu.

- **Turnaround time**

Yaitu banyaknya waktu yang dibutuhkan oleh sebuah proses dari dia masuk ke ready queue sampai dia berhasil di eksekusi(mengeluarkan output)

- **Waiting time**

Yaitu waktu yang diperlukan oleh proses menunggu giliran di ready queue untuk dieksekusi.(waktu yang dihabiskanya di ready queue)

- **Response Time**

Pada system yang interaktif kriteria turnaround time tidak begitu bagus digunakan.Sering kali sebuah proses sudah mengeluarkan output dan melanjutkan lagi komputasinya untuk output baru walaupun output lamanya baru saja dikeluarkan ke user(bayangkan user yang mengetik di microsoft word).Jadi waktu yang diperlukan dari saat request diminta hingga response pertama diproduksi disebut response time. Terkadang response time adalah waktu yang diperlukan sebuah proses setelah output pertamanya keluar hingga mendapat giliran lagi untuk memperoleh CPU untuk memproduksi output berikutnya. Response time biasanya dibatasi oleh kecepatan output device.

Jadi kita menginginkan

- **Penggunaan CPU yang maksimal**
- **Throughput yang maksimal**
- **Turnaround time yang minimal**
- **Waiting time yang minimal**
- **Response time yang minimal(kita menginginkan saat mengetik di microsoft word munculnya huruf yang kita ketik dilayat tidak lama)**

#### 5.5 Scheduling algoritma

- **First Come First Served (FCFS)**

Misalkan terdapat 3 proses seperti dibawah ini dengan masing-masing proses memiliki next burst time(jika burst timenya selesai berarti dua kemungkinan, Proses dapat selesai atau mungkin menunggu I/O).

Process	Burst Time
P1	24
P2	3
P3	3

Asumsikan proses-proses tersebut datang dengan urutan P1,P2,P3(diready queue susunan sperti ini) maka dengan FCFS Grant Chart adalah



**Throughput:**

3 buah pekerjaan diselesaikan dalam 30 satuan waktu atau 1/10 atau 1 pekerjaan dalam 10 satuan waktu

**Turnaroundtime:**

P1=24 , P2=27 dan P3=30

$$\text{Rata-rata turnaroundtime} = (24+27+30)/3 = 27$$

**Waiting time:**

P1=0 , P2=24, dan P3=27

$$\text{Rata-rata waiting time} = (0+24+27)/3 = 17$$

**Response time=turnaround time (nanti di round robin baru kelihatan response time)**

Bagaimana jika proses-proses tersebut datang dengan urutan P2,P3 dan P1 maka Gant chart nya adalah



**Throughput:(sama dengan sebelumnya)**

3 buah pekerjaan diselesaikan dalam 30 satuan waktu atau 1/10 atau 1 pekerjaan dalam 10 satuan waktu

**Turnaroundtime:**

P1=30 , P2=3 dan P3=6

$$\text{Rata-rata turnaroundtime} = (30+3+6)/3 = 13 \text{ (tadi 27)}$$

**Waiting time:**

P1=6 , P2=0, dan P3=3

$$\text{Rata-rata waiting time} = (6+0+3)/3 = 3 \text{ (tadi 17)}$$

**Response time=turnaround time (nanti di round robin baru kelihatan response time)**

Sehingga dapat disimpulkan pula bahwa FCFS sangat terpengaruh oleh urutan kedatangan dari proses(**convoy effect**)

• **Shortest Job First (SJF) scheduling**

Salah satu yang diobservasi FCFS diatas adalah jika proses-proses yang memiliki burst time yang pendek dieksekusi lebih dahulu maka kita mendapatkan waiting dan turnaround time yang lebih kecil.Pada SJF mengasosiasikan urutan pengeksekusian dengan panjang next burst time dari proses,yang lebih pendek lebih dahulu. Pada FCFS sebenarnya kita tidak perlu tahu burst time dari proses yang perlu diketahui hanya urutan datang nya. Dengan menggunakan SJF maka kita akan mendapatkan algoritma yang optimal dari segi waiting dan

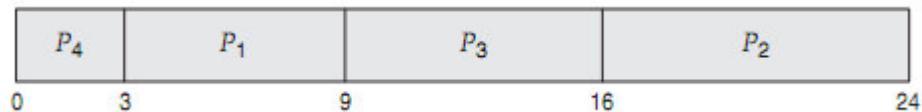


turnround time. Namun masalahnya adalah bagaimana mengetahui next burst time dari sebuah proses? sebenarnya CPU bisa diperintahkan untuk melihat kode dari proses dan menghitung lama burst nya, namun dengan melakukan itu berarti kita membuat CPU nya lebih pintar, dan artinya menambah beban kerja dan waktu extra bagi CPU. Bagaimana lagi kalau misalnya ada looping, kalau kode lompat ke program lain?. Ada cara lain yang lebih baik dari pada cara tsb, yaitu walupun hanya dengan memprediksi namun cara ini terbukti lebih baik. Dijelaskan setelah contoh berikut:

Asumsikan terlebih dahulu, kita dapat menebak next bursts time dari proses (ingat: andaikan sekarang sudah ada proses yang berjalan di CPU proses-proses dibawah ini belum berjalan masih di ready queue lalu kita akan menschedule proses mana yang akan berjalan kemudian), misalkan kita mempunyai 4 proses berikut:

Process	Burst Time
P1	6
P2	8
P3	7
P4	3

Maka grant chart dari proses-proses tersebut adalah



Dengan **waiting time**:

$$P1=3, P2=16, P3=9, P4=0$$

$$\text{Average waiting time} = (3+16+9+0)/4 = 7$$

Kita boleh mencoba melakukan permutasi (4!) untuk mencoba beberapa urutan eksekusi namun kita tidak akan mendapatkan average waiting time yang lebih pendek lagi.

#### Determining Next CPU bursts time

Kita hanya dapat memprediksi panjang dari bursts time, asumsi kita burst time yang akan datang itu mirip dengan sebelumnya. Kita akan menggunakan eksponensial averaging

#### EXPONENTIAL AVERAGING

$t_n$  = panjang burst time saat ini yang sebenarnya (tentu saja kalau sudah dieksekusi sebuah proses kita dapat mengetahui burst time nya)

$\tau_{n+1}$  = Panjang CPU burst yang akan diprediksi (kita pakai 'thou' karena hanya prediksi)

$$0 \leq \alpha \leq 1$$

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$



$\alpha$  adalah sebuah pemberat, nilainya antara 0 dan 1

Jika  $\alpha = 0$ , berarti kita lebih percaya dengan sejarah burst time sebelumnya,  $\tau_{n+1} = \tau_n$

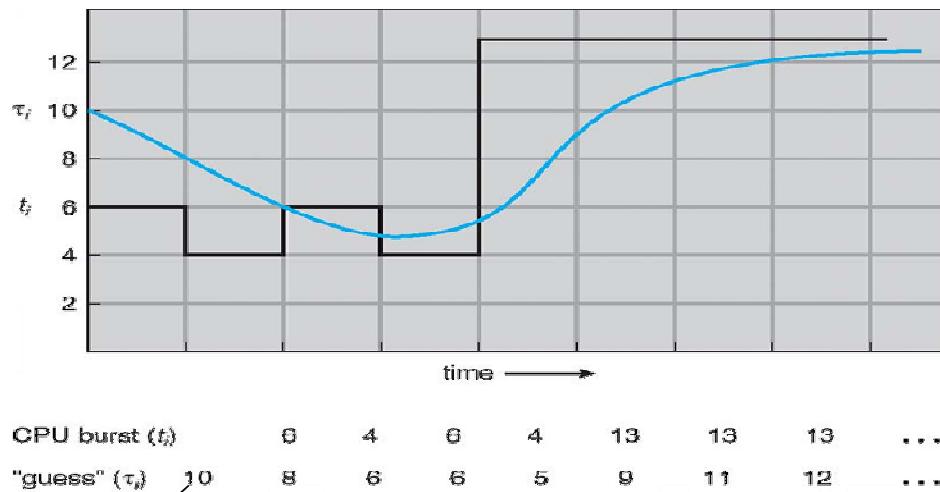
Jika  $\alpha = 1$ , berarti kita hanya percaya dengan burst time sebenarnya yang baru selesai dieksekusi  $\tau_{n+1} = t_n$

Biasanya  $\alpha$  diberi nilai 1/2

Untuk memahami behaviour dari exponential algoritma maka kita dapat mengexpand persamaan  $\tau_{n+1}$  dan mensubstitusi  $\tau_n$  kedalamnya

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0.$$

Karena  $\alpha$  dan  $(1 - \alpha)$  kurang dari atau sama dengan 1 maka setiap term yang berhasil dihitung, pemberatnya selalu lebih kecil dari yang sebelumnya. Jika kita melihat grafik dibawah ini terlihat bahwa burst time prediksi yang kita buat mengikuti trend dari burst time sebenarnya



Sebagai contoh

Process	Arrival time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Maka grant chart nya adalah



Proses satu yang duluan dimulai karena pada saat itu hanya dia yang ada di ready queue. Lalu P2 tiba pada time 1 maka sisa burst time P1 adalah 7 dan lebih besar dari pada burst time P2. Sehingga P1 akan dihentikan/preempted dan digantikan oleh P2. Begitu seterusnya untuk proses-proses yang datang berikutnya. Rata-rata waiting time nya adalah

$$[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)] / 4 = 26 / 4 = 6.5 \text{ milliseconds}$$

Jika dibandingkan dengan yang SJF non preemptive, yang average waiting timenya adalah 7,75 millisecond maka yang preemptive lebih baik.

- **Priority Scheduling**

SJF adalah kasus sepesial dari algoritma Priority scheduling. Setiap proses diberikan nomor(integer) prioritas. CPU akan dialokasikan untuk proses dengan priority yang lebih tinggi(integer paling kecil=prioritas lebih tinggi). Priority scheduling dapat bersifat preemptive juga dapat bersifat non preemptive. Namun masalah yang ditemui adalah akan adanya proses yang mengalami starvation, sebuah proses tidak pernah dapat dieksekusi oleh CPU karena proses-proses yang bersamanya dalam ready queue selalu priority lebih tinggi dari pada dia.

Solusinya adalah dengan aging(penuaan), semakin lama sebuah proses di ready queue maka priority nya akan semakin meningkat.

**Contoh:**

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Maka priority scheduling nya adalah



Average waiting time =  $(6+0+16+18+1)/5=8.2$

Turnaroundtime: P1=16, P2=1, P3=18, P4=19

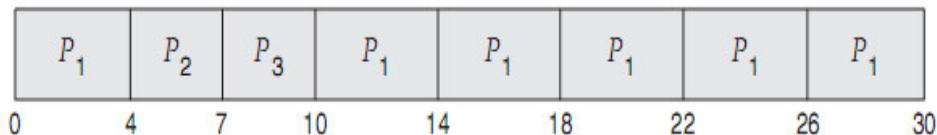
- **Round Robin**

Setiap proses mendapatkan CPU time unit yang kecil (time quantum **q**), biasanya berkisar antara 10-100 millisecond. Jika time ini habis maka proses akan di preempted dan dipindahkan ke urutan ready queue paling ujung. Jika terdapat n buah proses di ready queue dan time quantum adalah q maka masing-masing proses memperoleh  $1/n$  dari CPU time (CPU time/n). Tidak ada proses yang waiting time nya lebih dari  $(n-1)q$  time unit (**response time**). Untuk masalah perfomance ada satu hal yang harus diperhatikan, yaitu pemilihan besar kecilnya time quantum q. Jika terlalu besar maka algoritma Round Robin akan sama dengan algoritma FCFS, yang pada worstcase nya, waiting timenya sangat besar. Namun jika time quantum terlalu kecil akan banyak sekali context switching yang harus dilakukan. Context switching yang banyak menyebabkan overhead sangat tinggi. Jadi harus pintar-pintar milih.

Sebagai contoh: dengan time quantum q=4 millisecond

Process	Burst Time
P1	24
P2	3
P3	3

Maka grant Chart nya adalah

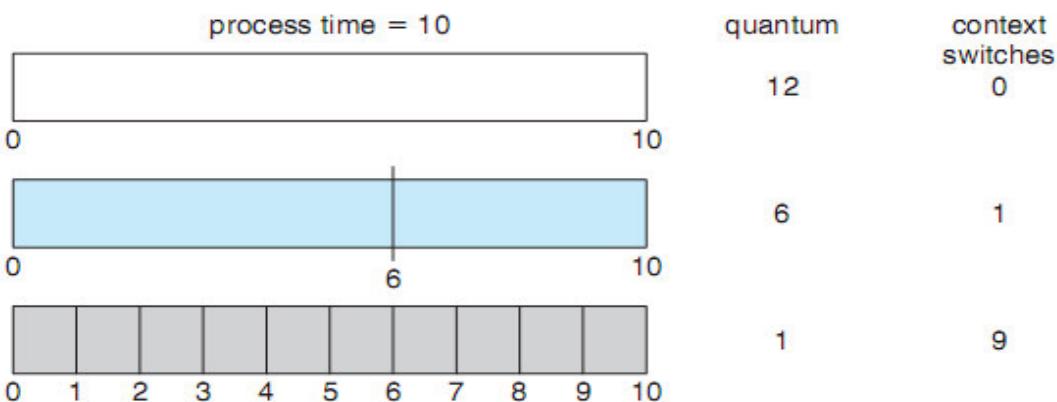


Waiting time: P1=10-4=6, P2=4, P3=7

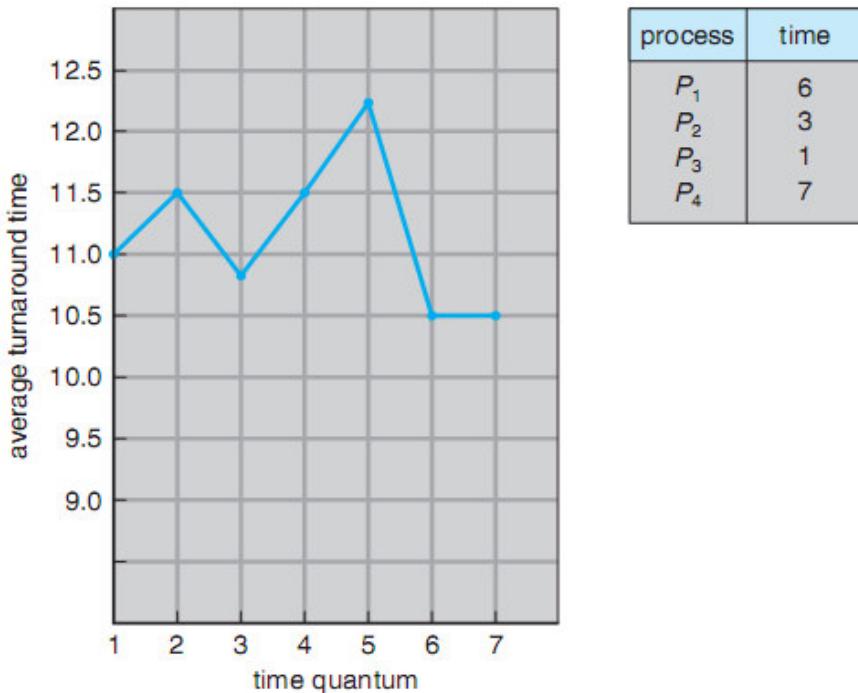
Average waiting time =  $(6+4+7)/3=5.66$  millisecond

Turnaroundtime: P1=30, P2=7, P3=10

Hubungan dari time quantum dan context switching



Hubungan turnaround time dan besarnya time quantum:



Dapat dilihat digrafik bahwa turnaround time fluktuatif terhadap besarnya time quantum

- **Multilevel Queue**

Ke empat algoritma diatas adalah algoritma dasar pada SO,namun terkadang SO menggabung-gabungkan ke 4 algoritma diatas.Pada Multilevel queue ready queue dibagi kedalam 2 level yaitu:

1. Level foreground(untuk proses yang mendukung aplikasi interaktif)
2. Level background (untuk proses yang mendukung aplikasi batch)

Setiap level queue menerapkan algoritma tersendiri

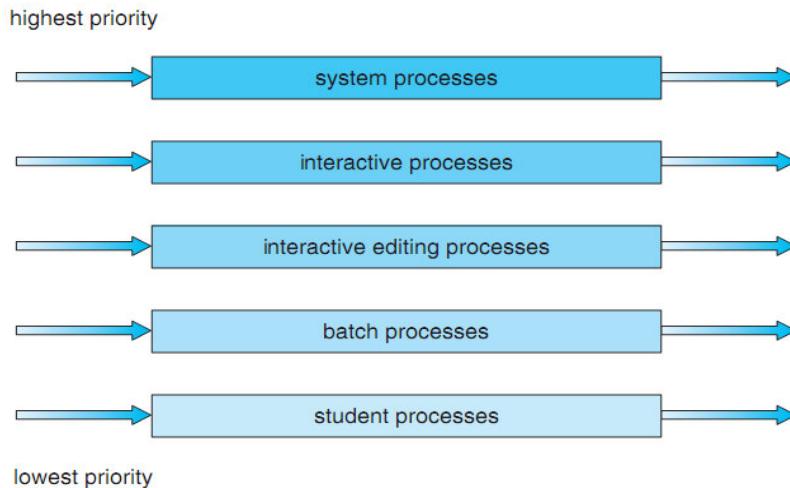
1. Foreground menggunakan algoritma RR(agar responsif)
2. Background menggunakan algoritma FCFS

Karena queuenya berlevel maka diperlukan pula algoritma untuk memilih proses yang diready queue mana yang akan dijalankan terlebih dahulu.Ada beberapa pendekatan :

- Priority scheduling yang tetap(mis:foreground akan dilayani terlebih dahulu baru kemudian yang background). Namun cara seperti ini memungkinkan startvation
- Dengan time slice,masing-masing queue mendapatkan sejumlah CPU time yang bisa digunakanya untuk melakukan scheduling proses-proses didalamnya(mis: foreground 80% dan background 20%)



Pembagian level queue nya tidak harus hanya dua level namun dapat dibagi lagi lebih dari dua level seperti gambar dibawah ini. Masing-masing level queue memiliki priority.



Gambar 5.5.1 Multilevel queue

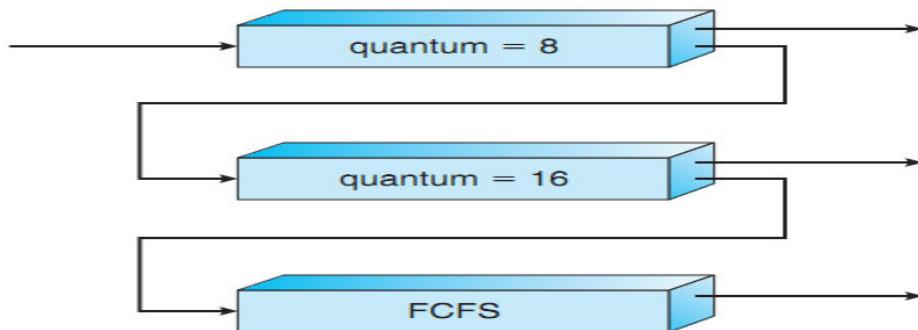
- **Multilevel Feedback queue**

Pada multilevel Feedback Queue sebuah proses dapat berpindah-pindah dari suatu queue ke queue lain. Pada algoritma ini aging dapat diimplementasikan untuk mengatasi starvation.

Sebagai contoh:

Terdapat 3 queues:

- Q0 menggunakan RR dengan time quantum 8 millisecond
- Q1 menggunakan RR dengan time quantum 16 millisecond
- Q2 menggunakan FCFS



Gambar 5.5.2. Multilevel Feedback queue

Setiap proses baru akan memasuki Q0, untuk menentukan proses mana yang akan dieksekusi dahulu dengan time quantum 8 milisecond(RR), dilakukan dengan FCFS. Jika dengan 8 milisecond proses tersebut tidak selesai maka proses tersebut dipindahkan ke Q1, proses di Q1 hanya akan diproses jika queue di Q0 kosong. Jika Q0 kosong maka seperti pada Q0 proses secara FCFS diberikan CPU time selama 16 milisecond(RR). Jika dengan 16 milisecond proses tersebut belum selesai, maka proses tersebut dipindahkan ke



Q2,proses di Q2 hanya akan diproses jika Q0 dan Q1 kosong. Jika Q0 dan Q1 kosong maka proses di Q2 akan diselesaikan dengan FCFS.

Andaikan pada saat proses di Q1 dikerjakan,terdapat proses yang masuk ke Q0 maka eksekusi proses di Q1 akan di hentikan, untuk mengeksekusi proses di Q0 yang baru masuk,begitu proses yang masuk di Q1 akan menghentikan proses di Q2. Jika sebuah proses terlalu lama di Q2 maka akan langsung dinaikkan ke Q0(aging).

Multilevel Feedback queue ditentukan oleh parameter-parameter berikut:

- Jumlah queue
- Algoritma scheduling yang digunakan di masing-masing queue
- Methode yang digunakan ketika akan menaikan prioritas sebuah proses
- Methode yang digunakan ketika akan menurunkan prioritas sebuah proses
- Methode yang digunakan untuk menentukan queue mana yang akan dimasuki oleh sebuah proses jika proses tersebut membutuhkan service.

### 5.6.Thread Scheduling

Perbedaan mendasar antara user thread dan kernel thread adalah pada bagaimana mereka dischedulkan.Pada system yang mengimplementasikan many to one dan many to many models,thread library melakukan scheduling untuk user thread agar dapat berjalan pada LWP yang ada,skema ini disebut proses –contetion scope(PCS) karena persainganya terjadi dalam proses. Jika dikatakan bahwa thread library melakukan scheduling untuk user thread agar bisa berjalan pada LWP bukan berarti bahwa user thread berjalan pada fisik CPU.Namun diperlukan Operating system untuk menschedule kernel thread untuk berjalan pada fisik CPU.Untuk memutuskan kernel thread mana yang akan berjalan di CPU,kernel menggunakan system contetion scope(SCS). Kompetisi untuk menggunakan CPU terjadi pada keseluruhan thread di system. System yang menggunakan one to one biasanya hanya menggunakan SCS.Biasanya level priority pada PCS ditentukan oleh programer.

#### - Pthread Scheduling

Pthread adalah thread pada POSIX, API pthread menyediakan pilihan untuk memilih PCS atau SCS pada saat menciptakan thread:

- Perintah PTHREAD\_SCOPE\_PROCESS akan menschedul thread dengan PCS
- Perintah PTHREAD\_SCOPE\_SYSTEM akan menschedul thread dengan SCS

### 5.7.Multiple Processor scheduling

CPU scheduling akan lebih rumit jika terdapat multiple CPU.Muti prosesor yang dibicarakan disini adalah prosesor yang homogeneous.Seperti yang telah dibahas sebelumnya multiprosesor terbagi menjadi dua yaitu:

- Asymmetric Multiprosesor yaitu hanya satu prosesor yang bersifat sebagai master,master lah yang mengatur scheduling CPU lainnya.Pada system seperti ini schedulingnya hampir sama dengan yang uni CPU.
- Symetric Multiprosesor yaitu masing-masing prosesor harus menschedul dirinya sendiri, ready queue nya bisa merupakan queue yang umum maksudnya diakses oleh semua CPU atau masing-masing CPU memiliki private ready queue.

System komputer modern biasanya menggunakan cache memory. Bila suatu proses sedang berjalan di suatu CPU maka datanya biasanya diambil dicache memory,bagaiman jika tiba-tiba proses tersebut

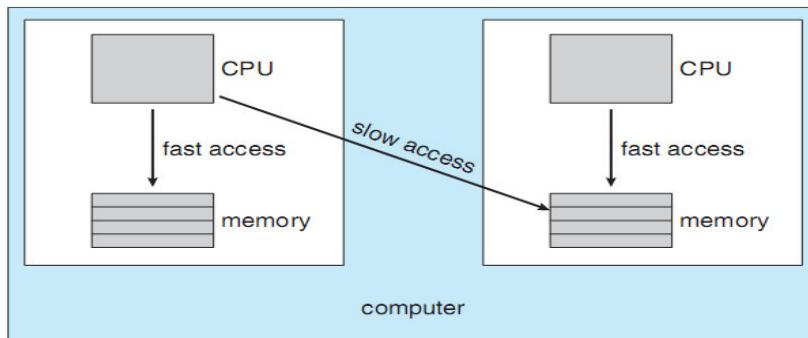


berpindah CPU artinya cache akan dirombak ulang begitu pula cache pada CPU tempat proses tersebut berpindah. Hal ini akan meng overhead pekerjaan CPU. Untuk itu dikenal prosesor affinity. Suatu proses memiliki affinity(keterkaitan) terhadap prosesor dimana ia dieksekusi pertamakali. Prosesor affinity dibagi dua jenis:

- **Soft affinity**; suatu proses tidak akan dipindah ke prosesor lain, namun tidak ada jaminan bahwa hal tersebut akan berlangsung terus. Bisa saja suatu saat proses tersebut dipindahkan namun diusahakan tidak.
- **Hard affinity** maksudnya dijamin suatu proses tidak akan berpindah prosesor.

Prosesor set pada solaris membatasi proses-proses mana saja yang dapat dijalankan pada suatu prosesor.

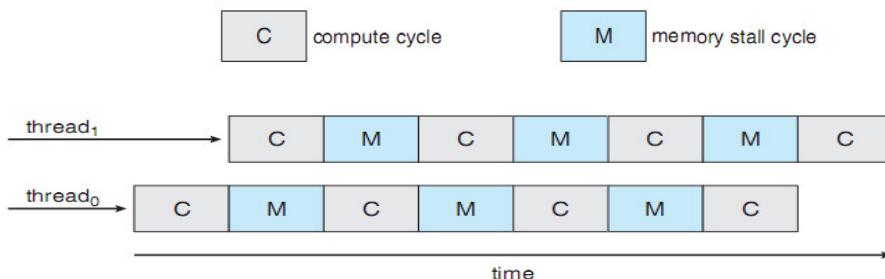
Main memory arsitektur berpengaruh pada prosesor affinity. Perhatikan gambar berikut. Prosesor A akan sebaiknya mengakses proses yang berada dalam Memory yang letaknya berdekatan dengannya (aksesnya lebih cepat) ketimbang mengakses proses yang berada di memory yang berjauhan dengan nya.



Gambar 5.7.1. Main memory arsitektur dan Prosesor affinity

### 5.8. Multi Core Processor.

Trend sekarang multiple processor ditempatkan dalam satu chip. Hal ini mengakibatkan CPU lebih cepat dan mengkonsumsi power lebih sedikit. Multiple thread per core juga semakin dikembangkan, hal ini terjadi atas pertimbangan bahwa bila suatu thread lagi dieksekusi di suatu core ada kalanya thread tersebut mengalami **memory stall**, yaitu thread tersebut menunggu data dari memory. Memory stall dapat terjadi karena Cache miss. Dengan adanya multiple thread dalam satu core maka thread yang menunggu tadi dapat diganti oleh thread lain untuk dieksekusi, karena bisa saja thread yang satu ini data nya terdapat di cache.



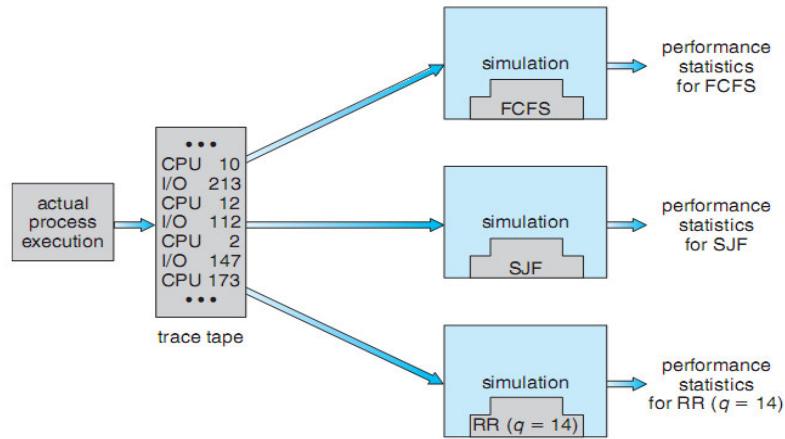
Gambar 5.8.1. Multithread dan memory stall



### 5.9. Evaluasi Algoritma

Langkah pertama untuk memilih algoritma mana yang sesuai dengan SO yang akan kita desain adalah menentukan kriteria (seperti, CPU utilization, waiting time, turnaround time, response time, throughput) lalu kemudian melakukan evaluasi. Ada beberapa teknik evaluasi namun yang umum dipakai adalah

- Dengan menggunakan deterministic modeling  
Caranya dengan mengambil sebuah pekerjaan (workload) kemudian menentukan performance tiap algoritma untuk workload tersebut
- Simulasi



Gambar 5.9.1. Evaluasi beberapa algoritma menggunakan teknik simulasi

## CHAPTER 6. Syncronisasi

### 6.1.Background

Akses Concurrent ke sebuah data shared akan menyebabkan data tersebut rawan akan inkonsistensi. Untuk menjaga agar data shared tersebut tetap konsisten maka akan dibutuhkan mekanisme untuk menjaga agar eksekusi terhadap proses-proses yang saling bekerjasama dapat teratur. Mekanisme itu disebut synkronisasi. Proses yang menggunakan shared data dapat kita ilustrasikan dengan masalah producer konsumen. Misalkan kita ingin memberikan solusi terhadap masalah produser konsumen. Kita dapat melakukannya dengan membuat sebuah variabel integer dengan nama count yang dapat mengidentifikasi jika buffer penuh. Mula-mula count di set nol. Count diincrement oleh produser setelah memproduksi sebuah item ke buffer dan akan didecrement oleh konsumen setelah mengkonsumsi sebuah item dari buffer.

#### Sisi Produser

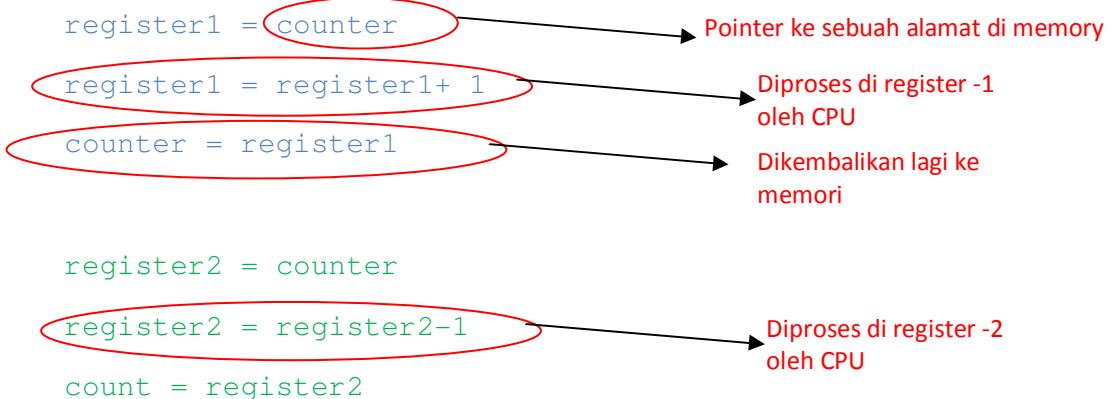
```
while (true) {  
  
    /* produce an item and put in nextProduced*/  
  
    while (counter == BUFFER_SIZE) //berarti buffer full  
  
        ; // do nothing  
        Blok   
        buffer [in] = nextProduced;  
  
        in = (in + 1) % BUFFER_SIZE;  
  
        counter++;  
  
    }  
  
    Kapan berubah? Akan dirubah oleh konsumen
```

#### Sisi konsumen

```
while (true) {  
  
    while (counter == 0)  
  
        ; // do nothing  
  
    nextConsumed= buffer[out];  
  
    out = (out + 1) % BUFFER_SIZE;  
  
    counter--;  
  
    /* consume the item in nextConsumed  
  
    }  
  
    Kapan berubah? Akan dirubah oleh produser
```



Jika kedua proses diatas dieksekusi secara concurrent maka akan ada yang salah. Perhatikan bahwa jika counter ++ dan Counter -- diimplementasikan dibahasa paling bawah/bahasa mesin



Kita tidak dapat melakukan proses penambahan di memory, semua data yang ada di memory harus dibawah dulu ke register lalu hasilnya nanti dikembalikan lagi ke memori.

Perhatikan proses interleaving dari produser dan konsumen berikut jika mula-mula count = 5

*Sebenarnya proses interleavingnya bukan hanya ini masih banyak lagi kemungkinan(\*\*) namun kita mengambil satu contoh dimana terjadi inkonsistensi data.*

S0: producer execute register1 = counter {register1 = 5}

S1: producer execute register1 = register1+ 1 {register1 = 6}

**Terjadi interupsi, sehingga terjadi context switching**

S2: consumer execute register2 = counter {register2 = 5}

S3: consumer execute register2 = register2-1 {register2 = 4}

**Terjadi interupsi, sehingga terjadi context switching**

S4: producer execute counter = register1 {count = 6 }

**Terjadi interupsi, sehingga terjadi context switching**

S5: consumer execute counter = register2 {count = 4}

Mengapa inkonsistensi data diatas dapat terjadi? bagaimana solusinya? salah satu solusinya adalah proses count++ maupun proses – harusnya jangan disela, harus diproses secara atomic.

Permasalahan ini disebut **critical session problem**.



## 6.2. Critical session problem

Misalkan terdapat n buah proses (p1,p2,...pn). Masing-masing proses memiliki critical session code yaitu ketika sebuah proses mengubah sebuah variabel, menulis ke file, updating table dan lain-lain. Setiap sebuah proses berada pada critical session maka tidak boleh ada proses lain yang berada di critical session. Ketika sebuah proses berada pada critical session maka tidak boleh disela. Hal ini menjadi tantangan tersendiri bagi kernel preemptive. Solusi untuk critical session problem ada beberapa namun harus memenuhi ke tiga syarat berikut:

1. Mutual Exclusion  
Jika proses Pi sedang berada di critical session maka tidak boleh ada proses lain yang masuk ke critical session
2. Progres  
Jika tidak ada satupun proses yang berada di critical session dan pada saat tersebut ada beberapa proses yang ingin masuk ke critical session maka pemilihan proses mana yang akan dipilih untuk masuk ke critical session tidak boleh ditunda.
3. Bounded waiting  
Harus terdapat batas atas dari jumlah sebuah proses meminta untuk memasuki critical session.

Setiap proses diasumsikan dieksekusi dalam nonzero speed dan tidak ada pandangan relative speed pada n buah proses(biar ada yang lambat dan ada yang cepat tetapi harus ada bounded waiting)

Salah satu solusi dari critical session adalah algoritma peterson

## 6.3. Peterson Solution

Peterson solution dikhususkan hanya untuk dua proses. Asumsinya adalah intruksi LOAD(load dari memori ke register) dan STORE(dari register kembali ke memori) adalah atomic. Dua proses tersebut menshare dua variabel yaitu:

- Int turn(mengidentifikasi giliran siapa yang harus memasuki critical session)
- Boolean flag[2] (mengidentifikasi bahwa proses siap memasuki critical session, flag[i] mengidentifikasi bahwa proses i siap memasuki critical session)

Script dibawah ini dikerjakan oleh kedua proses secara terpisah, namun syaratnya hanya pada saat critical system hanya ada satu proses yang boleh masuk.

```
do {  
    flag[i] = TRUE; → Request untuk masuk ke critical session  
    turn = j; → Tapi ia mempersilahkan lawannya dulu  
    while (flag[j] && turn == j);  
        yang memasuki critical session
```

### critical section

```
    flag[i] = FALSE;
```

### remainder section

```
} while (TRUE);
```



- 1) Solusi ini hanya mengizinkan sebuah proses masuk ke Critical Sesion jika Flag[lawanya]=false dan turn=dirinya sendiri

Flag[lawanya] hanya akan false hanaya jika lawanya tersebut keluar dari critical sesion(aman).

Kedua proses tidak akan pernah mengeksekusi dengan sukses while nya(Flag[dirinya]=true dan turn=dirinya sendiri) secara bersamaan karena Walupun Flag[lawanya] dan Flag[dirinya sendiri] bisa saja secara bersamaan true ,namun karena variable turn itu dishare dan setiap proses akan selalu mengeset turn ke proses lawanya maka tidak akan pernah diwaktu yang sama turn=dirinya sendiri dan turn =lawanya terjadi.Selalu ada satu yang blok di while dan ada satu yang dapat mengesekusi critical system.**Mutual exlusion terpenuhi**

- 2) Misalkan proses ke i baru keluar dari cirtical sesion nya maka proses i akan mengeset flag[i] =false namun setelah itu ia tidak melepaskan CPU nya namun langsung mengerjakan sesion remindernya ,sementara proses j sedang menunggu di while(blok di while, proses j tidak akan bisa masuk ke critical sesion karena CPU masih di proses i),setelah remiddernya selesai maka proses I kembali keatas dan mengeset flag[i]=true dan turn=j lalu dia ke while memeriksa apakah flag[i]=true(ya) dan turn=j(ya karena baru saja diubahnya) maka ia akan blok juga di while. Nah syarat dari progres agar terpenuhi adalah jika tidak ada satupun proses yang ada di kritikal sesion dan ada beberapa proses yang siap untuk masuk ke critical sesion maka pemilihan proses yang masuk ke critical sesion tidak boleh di postpone,sehingga SO akan mengambil CPU dari proses I dan akan memberikan nya ke proses j dimana status whilennya sekarang adalah flag[j]=true(ya) dan turn=j(ya karena baru saja diubah oleh proses i) sehingga proses j akan masuk kecritical sesion. **Progress terpenuhi**
- 3) Satu proses dapat masuk ke critical sesionnya maksimal setelah satu kali lawanya memasuki critical sesion(satu putaran),yaitu ketika lawanya keluar dari critical sesion maka ia akan mengeset flagnya jadi flase sehingga prosess yang sedang menunggu di while akan bisa masuk ke critical sesion.Jadi mereka bergantian masuk.(**bounded waiting terpenuhi**)

*Peterson solution sangat terbatas karena hanya dua proses.*



#### 6.4 syncronisasi hardware

Yang dibahas sebelumnya adalah solusi dari sisi software,pada pembahasan tersebut kita mengasumsikan LOAD dan STORE harus atomic(hardware).Banyak system yang menyediakan hardware yang suport untuk kode critical sesion.Pada system yang uniprosesor mudah untuk menghadapi masalah critical sesion yaitu dengan mendisable interupsi namun hal ini akan menjadi tidak effisien di system yang memiliki multiprosesor.Kita harus memberitahu semua prosesor bahwa interupsi tidak boleh terjadi jika suatu proses di critical sesion, hal ini membutuhkan ekstra waktu, dan juga akan mengurangi response time. Solusinya beberapa mesin menyediakan intruksi yang atomic diantaranya adalah

- test and set
- swap

Kedua solusi diatas didasarkan pada problem using lock.Problem using lock sebenarnya adalah mekanisme umum dari mutual exclusion, diamana jika sebuah proses ingin memasuki suatu critical sesion maka dia harus mendapatkan lock terlebih dahulu kemudian setelah ia keluar dari critical sesion maka lock tersebut dilepasnya.

```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (TRUE);
```

Hal ini menjamin hanya ada satu proses yang ada di critical sesion suatu saat.Namun kiat tidak dapat mengukur progress dan bounded waiting jadi problem using lock ini akan digabung dengan dua solusi diatas.

- **Test and set**

Kita memiliki satu set intruksi atomic test&set:

```
booleanTestAndSet(boolean*target)  
{  
    booleanrv= *target;  
    *target = TRUE;  
    return rv;  
}
```



Kemudian kita memiliki variabel **lock** yang akan dishare oleh semua proses dan akan di inisialisasi dengan FALSE , dan semua proses akan mengeksekusi code berikut

```
do {  
    while ( TestAndSet(&lock) )  
        ; // do nothing  
    // critical section  
    lock = FALSE;  
    // remainder section  
} while (TRUE);
```

Ketika sebuah proses berhasil duluan mengeksekusi while maka return value dari fungsi test and set adalah false dan variabel lock diset menjadi true oleh \*target.Karena return value nya FALSE maka proses tersebut akan memasuki critical sesion nya. Sementara proses lain tidak dpt masuk karena pada saat memanggil fungsi test and set nya nilai return value dari funsi tersebut akan TRUE karena variabel lock masih dalam keadaan TRUE. Variabel lock akan bernilai false ketika proses yang telah memasuki critical sesion keluar dan mengeset variabel Lock=FALSE.

Jika dilihat solusi ini selain sudah dapat melayani banyak proses juga memenuhi mutual exclusion dan progress(selalu ada yang bisa masuk),namun bouded waiting tidak bisa dipenuhi. Misalkan seperti tadi proses yang pertama menang dan dapat memasuki critical sesion nya karena return value nya FALSE(LOCK=false)sedangkan proses lain akan menunggu karena lock telah berubah menjadi true dan return vaeue dari fungsi adalah TRUE,selepas selesai dengan critical sesion nya maka proses tersebut akan merubah lock menjadi FALSE sehingga ada satu proses lain yang akan menang dan mengeksekusi while dan masuk ke critical sesion. Perlu dicatat bahwa persaingan proses lain tidak dalam bentuk antrian mereka akan terus bersaing untuk mengeksekusi while ,nah proses yang telah masuk ke critical sesion akan kembali lagi untuk ikut persaingan,sehingga memungkinkan akan ada satu proses yang tidak akan pernah bisa masuk ke critical sesion karena akan kalah terus.Jadi tidak ada batas atas request pada solusi ini.

- **Swap**

Kita memiliki satu set intruksi atomic swap

```
void Swap (boolean*a, boolean*b)  
{  
    boolean temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Sebenarnya kita hanya  
mempertukarkan nilai a  
ke b dan nilai b ke a



Kemudian kita memiliki varabel yang dishare lock dan insialisasi dengan FALSE dan satu buah variabel lokal dimasing-masing proses yaitu key.

```
do {  
    key = TRUE;  
    while ( key == TRUE )  
        Swap (&lock, &key );  
    //      critical section  
    lock = FALSE;  
    //      remainder section  
} while (TRUE);
```

Mula-mula sebuah proses mengeset key nya=TRUE,lalu jika proses tersebut menang maka ia akan memeriksa while nya ,karena while nya bernilai benar maka proses tersebut akan memanggil fungsi swap dan mepertukarkan

Key=TRUE  $\longleftrightarrow$  Lock=FALSE

Menjadi

Key=FALSE      Lock=TRUE

Ketika proses ini kembali memeriksa while nya(berulang karena while nya masih true) maka ia akan mendapatkan key tidak lagi TRUE sehingga ia akan keluar dari while dan mengeksekusi critical sesion.Pada saat ini proses lain tidak akan bisa masuk ke critical sesion karena Lock bernilai TRUE, jika di swap ke key ,key akan selalu true sehingga proses lain tidak bisa keluar dari while hingga proses yang tadi yang masuk, keluar dan mengeset lock =FALSE. Solusi ini mirip dengan solusi test and set sehingga masalahnya juga sama tidak bisa memenuhi bounded waiting.

Lalu bagaimana caranya agar memenuhi bounded waiting?caranya adalah dengan menambahkan antian kedalam fungsi diatas.

```
do {  
    waiting[i] = TRUE;  
    key = TRUE;  
    while (waiting[i] && key)  
        key = TestAndSet(&lock);  
    waiting[i] = FALSE;
```



```
// critical section  
  
j = (i + 1) % n;  
  
while ((j != i) && !waiting[j])  
  
j = (j + 1) % n;  
  
if (j == i)  
  
lock = FALSE;  
  
else  
  
waiting[j] = FALSE;  
  
lock = FALSE;  
  
// remainder section  
  
} while (TRUE);
```

Kita memiliki variabel lokal key dan variabel shared lock dan waiting[i].Lock diinisialisasi dengan FALSE. Mula-mula sebuah proses akan mengeset Waiting nya dengan TRUE,key=TRUE,kemudian karena proses ini menang maka ia akan memeriksa while nya,pada pemeriksaan pertama while nya TRUE sehingga ia akan melakukan key = TestAndSet(&lock), return value dari test and set karena lock nya False maka return valuenya False,sehingga key sekarang bernilai false,kemudian ia kembali keatas lagi memeriksa while nya ,waiting[i] masih TRUE sedangkan key nya sudah FALSE sehingga proses ini keluar dari blok,pada saat keluar dari blok ia mengeset waiting[i] =FALSE lalu masuk ke critical sesion.Proses lain tidak akan bisa masuk ke critical sesion pada saat ini karena lock masih bernilai TRUE,jika lock bernilai TRUE itu berarti jika dipanggil menggunakan fungsi testandset return value nya akan TRUE sehingga key=TRUE ,Waiting=TRUE dan key=TRUE sehingga proses tersebut blok diwhile. Pada saat sebuah proses keluar dari critical sesion maka ia akan menicrement indeks j modulo n(n=banyak nya proses),maksudnya mengeset indeks ke tetangganya.Lalu memeriksa

```
while ((j != i) && !waiting[j]),
```

jika J!= i maksudnya proses yang diselidikinya sekarang bukan dirinya sendiri,dan !waiting[j] berarti tetangganya tersebut waiting nya tidak sama dengan TRUE(tidak lagi waiting). Selama whilenya benar maka ia akan mengincrement lagi j(pindah lagi ke tetangga berikutnya) lalu memeriksa nya lagi apakah ia sudah sampai kedirinya sendiri lagi(satu putaran)dan apakah proses tersebut waiting atau tidak. Proses tersebut dapat keluar dari blok while jika ia menemukan ada sebuah proses yang ada di queue tersebut yang lagi menunggu dan jika ia menemukan j==i(kembali kedininya lagi,habismi dai cek tetangganya)

Jika ia keluar dari blok karena j!=i maka ia akan mengeset lock=FALSE dan bila ia keluar karena waiting[j](ada yang menunggu) maka ia akan mengeset lock=FALSE dan mengeset Waiting[j](waiting dari proses j) menjadi TRUE(maksudnya menyuruh proses J untuk segera masuk ke critical sesion)



Mengapa solusi ini memenuhi ke 3 syarat critical session solution?

- 1) Pi akan masuk ke critical session hanya jika waiting[i]=FALSE, atau key=FALSE. Nilai key hanya akan FALSE jika fungsi test and set di eksekusi dengan lock=FALSE. Proses pertama yang berhasil mengeksekusi fungsi test and set akan mendapatkan key=FALSE jika key=FALSE maka lock akan TRUE sehingga proses lain akan blok di While. Variabel waiting[i] hanya akan FALSE jika ada proses lain yang meninggalkan critical session. **Jadi mutual exclusion terpenuhi**
- 2) Jika mutual exclusion terpenuhi biasanya progress juga terpenuhi. Karena setiap proses yang keluar dari critical session akan menset lock=FALSE dan waiting[j]=FALSE maka dijamin proses yang blok di while akan ada salah satu nya yang akan masuk ke critical session. **Jadi progress terpenuhi.**
- 3) Ketika sebuah proses meninggalkan critical session maka ia akan men scan secara cyclic dan proses yang ditemukan pertama kali waiting nya =TRUE maka waiting nya diubah jadi FALSE dan akan memasuki critical session selanjutnya. Jadi setiap proses minimal menunggu n-1 giliran untuk masuk ke critical session(ada batas atas request nya). **Bounded waiting terpenuhi.**

Solusi ini telah menyelesaikan masalah critical session, namun hanya pada yang uni prosesor bagaimana jika solusi ini bekerja di multiprosesor,suatu saat terdapat dua buah proses dimasing-masing prosesor yang mengakses test and set bersamaan ,siapa yang akan menang,jadi diantara prosesor terjadi masalah critical session lagi.

- **Semaphore**

Solusi berbasis hardware yang dijelaskan sebelumnya sangat kompleks bagi programer. Untuk menyederhanakan masalahnya kita menggunakan salah satu tool synchronisasi yaitu Semaphore. Semaphore S adalah variabel integer. Ada dua standar operasi atomic yang dapat memodifikasi S yaitu wait() dan signal().

- **Wait**

```
wait (S) {  
    while S <= 0  
        ; // no-op  
        S--;  
}
```

Satu ketukan/tidak boleh disela

- **Signal**

```
signal (S) {  
    S++;  
}
```

Ada dua jenis semaphore yaitu

- Counting semaphore: nilainya integer dan range nya tidak dibatasi
- Binary semaphore: nilainya integer 0 atau 1 (lebih mudah diimplementasikan)  
Dikenal juga sebagai mutex lock



Semaphore dapat menunjukan mutual exclusion:

```
Semaphore mutex;      // initialized to 1
do {
    wait (mutex);
    // Critical Section

    signal (mutex);
    // remainder section
} while (TRUE);
```

### Implementasi semaphore

Untuk mengimplementasikan semaphore maka kiat harus menjamin terlebih dahulu bahwa tidak ada dua buah proses yang dapat mengakses wait() dan signal() pada semaphore yang sama dan dalam waktu bersamaan(bisa diatur dengan hardware). Hal ini menjadi masalah critical sesion sendiri lagi.salah satu kekurangan dari semaphore adalah busy waiting.sebenarnya busy waiting memeliki keuntungan(lihat buku page 228).Bagaimana caranya agar tidak ada busy waiting?

Semaphore nya kita beri waiting queue.Setiap entry pada queue memiliki nilai:

- Value(integer)
- Pointer ke record berikutnya di list

Juga terdapat dua operasi yaitu blok(menempatkan proses ke waiting queue) dan wake up(mengambil proses dari waiting queue dan ditempatkan ke ready queue)

Implementasi dari wait:

```
wait (semaphore*S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

Implementasi signal:

```
signal(semaphore*S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P); }}
```



### Deadlock dan starvation

Deadlock terjadi jika dua atau lebih proses sedang menunggu sebuah event yang dapat dihasilkan oleh hanya satu proses dari proses-proses yang saling menunggu. Misalkan terdapat dua semaphore S dan Q dan diinisialisasi dengan 1:

P0	P1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

P0 mengeksekusi wait(S) dan menunggu P1 untuk melakukan signal(S) namun P1 sedang mengeksekusi wait(Q) dan menunggu P0 untuk mengeksekusi signal(Q). Begitu seterusnya. Jika telah terjadi deadlock maka proses akan mengalami starvation, proses tersebut tidak akan bisa keluar dari waiting list.

### 6.5. Classic Problems of Synchronization

Ada beberapa masalah syncronisasi clasic yaitu

- **Bounded-Buffer Problem(produser consumer problem)**

Misalkan kita selesaikan dengan semaphore.

Kita misalkan N buffer masing-masing buffer dapat menampung satu item. Terdapat 3 buah semaphore yaitu:

1. Semaphore mutex ,diinisialisai dengan nilai 1`
2. Semaphore full ,diinisialisai dengan nilai 0
3. Semaphore empty ,diinisialisai dengan nilai N

Cara mengimplementasikanya:

- SISI PRODUSER

```
do  {  
    //  produce an item in nextp  
    wait (empty);  
    wait (mutex);
```

```
wait (empty) {  
  
    while S <= 0  
    ; // no-op  
  
    S--;  
}
```

Menunggu buffer ada yang kosong bukan kosong betulan,nilai awalnya N



```
// add the item to the buffer  
  
signal (mutex);  
  
signal (full);  
  
} while (TRUE);
```

- **SISI consumer**

```
do {  
  
    wait (full); → Menunggu buffer ada yang ada isinya  
    bukan full=penuh,nilai awalnya 1  
  
    wait (mutex);  
  
    // remove an item from buffer to nextc  
  
    signal (mutex);  
  
    signal (empty);  
  
    // consume the item in nextc  
  
} while (TRUE);
```

- **Readers and Writers Problem**

Sebuah himpunan data di shared oleh beberapa proses yang berjalan concurrent. Sebuah proses dapat menjadi:

- Readers → hanya membaca data, tidak melakukan update
- Writers → dapat membaca dan menulis

Kita dapat menizinkan banyak reader untuk membaca dalam waktu yang bersamaan namun hanya boleh satu writer yang menulis ke data dalam satu waktu tertentu. Misalkan kita selesaikan dengan semaphore. Misalkan terdapat sebuah data set, semaphore mutex dengan inisialisasi 1, semaphore wrt dengan inisialisasi 1, variabel integer readcount dengan inisialisasi 0.

**WRITERS:**

```
do {  
  
    wait (wrt) ;  
  
    // writing is performed  
  
    signal (wrt) ;  
  
} while (TRUE);
```



## READERS

```
do {  
    wait (mutex) ;  
    readcount++ ;  
    if (readcount== 1)  
        wait (wrt) ;  
        signal (mutex)  
        // reading is performed  
        wait (mutex) ;  
        readcount--;  
    if (readcount== 0)  
        signal (wrt) ;  
        signal (mutex) ;  
} while (TRUE);
```

Permasalahan yang terjadi diatas adalah jika ada terus proses yang ingin membaca(karena reader tidak dibatasi) maka writer tidak akan mendapatkan kesempatan untuk melakuakn writer(writer hanya punya kesempatan jika readcount==0)

- Dining-Philosophers Problem



Gambar 6.5.1.Dining Philosopher Problem

```
do {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i + 1) % 5 ] );  
    // eat  
    signal ( chopstick[i] );  
    signal (chopstick[ (i + 1) % 5 ] );  
    // think  
} while (TRUE);
```



## CHAPTER 6. Syncronisasi

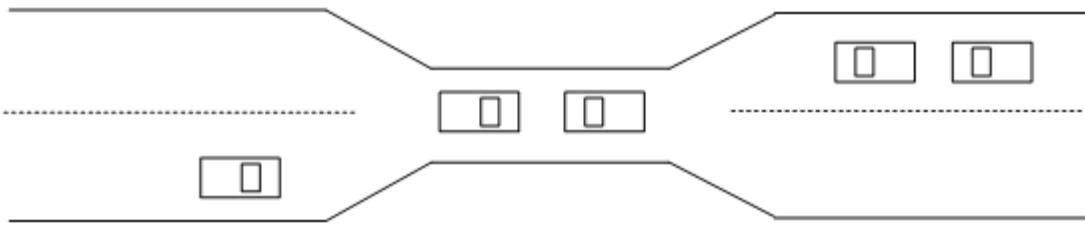
### 6.x.1. Deadlock

Deadlock terjadi ketika sehimpunan proses-proses yang blok, dimana masing-masing proses tersebut memegang sebuah resource dan bersamaan dengan itu juga sedang menunggu resource lain yang sedang dipegang oleh proses lain pada himpunan tersebut.

Contoh 1:

System memiliki dua disk drives. P1 dan P2 memegang masing-masing satu disk drive namun secara bersamaan pula P1 membutuhkan disk drive yang dipegang P2 dan juga P2 membutuhkan disk drive yang dipegang P1.

Contoh real ; **Bridge Crossing Example**



Trafik hanya terjadi dalam satu arah dalam suatu waktu. Jembatan bisa dianggap sebagai resource. Jika deadlock terjadi maka hanya akan bisa diresolve jika satu mobil mengalah dan mundur. Biasanya akan bukan hanya satu mobil yang mundur namun akan banyak. Kemungkinan dapat terjadi starvation kalau sebuah mobil tidak pernah bisa memenangi kompetisi untuk memakai satu jalur tersebut. Pada SO modern tidak mencegah dan deal dengan deadlock(tentu saja SO ini untuk komputer yang general purposes(mengetik,facebook) namun jika SO untuk komputer yang dipasang dipesawat, atau diperbankan dll, haruslah dapat mencegah deadlock). Jika kita harus deal dengan deadlock maka akan ada trade off misalnya performa kan menurun karena harus ada deadlock yang harus selalu diantisipasi.

### System Model

Mari kita melihat deadlock dengan lebih formal, Deadlock dapat digambarkan dengan system model berikut.

Asumsikan ada beberapa type resource R1,R2...,Rm(contoh: CPU cycles, memory sapace,I/O device). Setiap resource Ri juga memiliki Wi instance(misalnya kita memiliki dua buah prosesor, berarti kita resource CPU cycles memiliki dua instance).

Setiap proses yang ingin menggunakan resource harus melakukan 3 intruksi berikut(sebenarnya bukan hanya 3 tapi hanya untuk penyederhanaan/abstraksi):

- **Request**
- **use**
- **release**



**Deadlock** dapat terjadi jika ke 4 hal dibawah ini terjadi secara bersamaan dan jika deadlock terjadi maka ke 4 hal ini pasti terjadi(jadi ini adalah syarat perlu deadlock,if only if)

1. **Mutual exclusion :**

resource hanya dapat dipake oleh sebuah proses dalam suatu waktu,jika resource dapat dipake bareng-bareng deadlock tak akan terjadi.

2. **Hold and wait:**

Pada saat bersamaan sebuah proses memegang sebuah resource namun ia juga menunggu resource dari proses lain. Artinya proses tersebut mebutuhkan dua-duanya. Jika suatu proses ingin menggunakan sebuah resource ia melepaskan dulu resource yang sedang dipeganya sekarang maka deadlock tidak akan terjadi.

3. **No preemption:**

Tidak ada cara untuk mengambil sebuah resource yang sedang dipegang oleh sebuah proses lain sampai proses itu sendiri yang melepaskannya. Namun sebenarnya hanya SO yang dapat mengambilnya. Kalau penggunaan resource bersifat preemptive maka deadlock tidak akan terjadi.

4. **Circular wait:**

Jika terdapat himpunan  $\{P_0, P_1, \dots, P_n\}$  yang sedang saling menunggu sedemikian sehingga  $P_0$  menunggu resource yang dipegang oleh  $P_1$ ,  $P_1$  sedang menunggu resource yang sedang dipegang oleh  $P_2$  begitu seterusnya hingga diujung nanti  $P_n$  sedang menunggu resorce yang dipegang oleh  $P_0$ .

Jika salah satu saja dari ke empat hal tersebut tidak terjadi maka Deadlock tidak akan terjadi.

### 6.x.2. Resource allocation graph

Untuk memudahkan kita melihat hubungan proses,resource dan deadlock maka kita akan menggunakan **Resource Allocation Graph**. Resource alocation graph adalh grap yang menggabarkan alokasi resource kepada beberapa proses.

**Resource Allocation Graph** memiliki dua buah jenis vertex yaitu

- $P=\{P_1, P_2, \dots, P_n\}$  yaitu himpunan proses dalam system



- $R=\{R_1, R_2, \dots, R_m\}$  yaitu himpunan resource dalam system

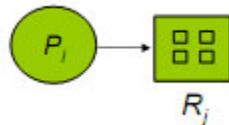


*Resource dengan 4 buah instance*



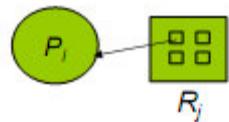
Resource Allocation Graph memiliki dua buah jenis edge yaitu

- Request edge



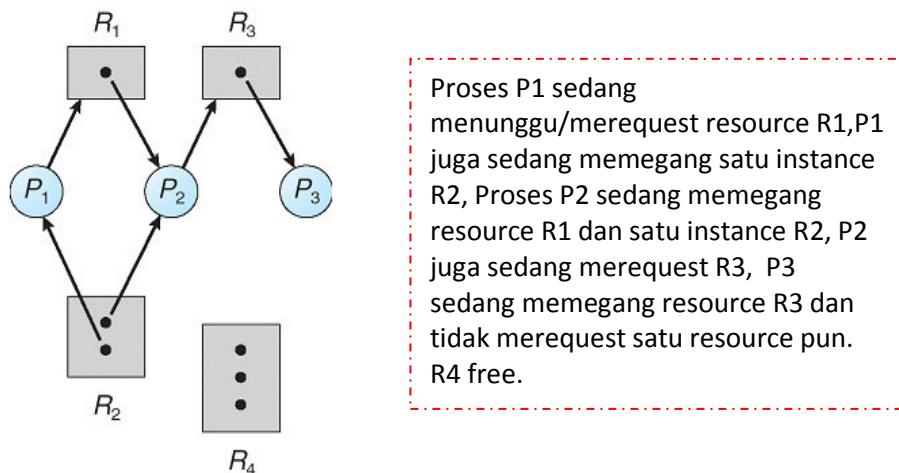
*Pj me-request instance resource Rj*

- Assignment edge



*Pj memegang instance resource Rj*

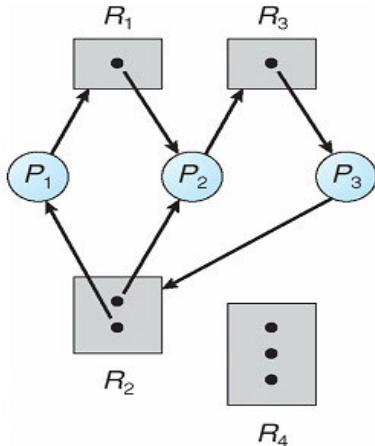
Contoh resource allocation graph



Sekarang mari melihat resource allocation graph dengan Deadlock. Asumsikan mutual exclusion, hold and wait, dan not preemptive terjadi. Maka sekarang kita tinggal melihat circular wait.



Contoh resource allocation graph dengan **Deadlock**

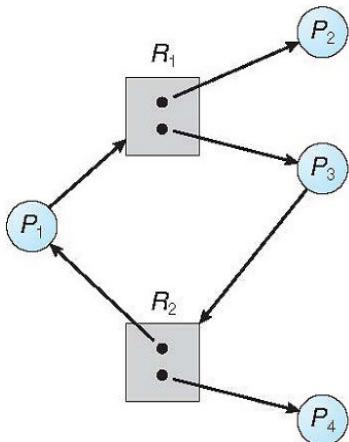


P1 sedang menunggu R1 yang sedang dipegang oleh P2, P2 sedang menunggu R3 yang sedang dipegang oleh P3, P3 sedang menunggu R2 yang sedang dipegang oleh P1 dan P2.  
Terjadi **sircular wait**

P1 → R1 → P2 → R3 → P3 → R2 → P1

Tidak selama jika kita melihat cycle di graph maka artinya telah terjadi circular wait dan akhirnya deadlock.

Contoh:



Mula-mula kita dapat melihat bahwa ada cycle disini, P1 sedang menunggu R1 yang sedang dipegang oleh P3, P3 sedang menunggu R2 yang sedang dipegang oleh P1. ini memang cycle.

Namun jika dilihat lagi sebenarnya R1 yang sedang ditunggu oleh P1 tidak hanya dipegang oleh P3 namun juga dipegang oleh P2. Begitupula R2 yang sedang ditunggu oleh P1 tidak hanya dipegang oleh P1 namun juga dipegang P4. Nah suatu ketika P2 ataupun P4 akan selesai dan dia akan melepas Resource yang dipegangnya (resource selalu dipegang dalam waktu yang finite) sehingga tidak akan terjadi deadlock. Salah satu saja dari P2 dan P4 selesai maka deadlock tak akan terjadi

#### Basict Fact:

- Jika graph tidak mengandung cycle maka pasti tidak terjadi deadlock
- Jika graph mengandung cycle maka
  - ✓ Jika setiap resource (resource yang terdapat dalam cycle) hanya memiliki satu instance maka pasti terjadi deadlock
  - ✓ Jika setiap resource (resource yang terdapat dalam cycle) mememiliki beberapa instance maka mungkin terjadi deadlock.



### 6.x.3. Methods for Handling Deadlocks

- **Pastikan bahwa system tidak akan pernah memasuki state deadlock**  
Untuk memastikan ini maka sistem dapat menerapkan salah satu dari deadlock prevention atau Deadlock avoidance
- **Izinkan saja sistem memasuki deadlock state, deteksi bahwa terjadi deadlock dan kemudian recover**(biasanya komputer hang kita hanya bisa merestart, merestart adalah salah satu cara recover dari deadlock). Namun cukup sulit juga untuk membuat algoritma mendeteksi deadlock.
- **Abaikan saja deadlock dan pura-pura tidak terjadi deadlock pada sistem.**

Mengignore tentu saja mudah sehingga yang akan kita bahas adalah dua bagian teratas.

#### 6.x.3.1. Deadlock Prevention and Deadlock Avoidance

Prevention dan avoidance berbeda. **Deadlock prevention** adalah himpunan metode untuk memastikan bahwa paling sedikit satu dari 4 syarat perlu deadlock tidak terjadi. Deadlock prevention mencegah deadlock dengan memberikan batasan ke proses bagaimana melakukan request resource. Sedangkan **Deadlock avoidance** adalah metode yang mewajibkan SO untuk memiliki priori informasi tentang resource mana, yang sebuah proses akan request dan gunakan selama dia dieksekusi. Dengan informasi tersebut SO akan dapat memutuskan request mana yang harus di penuhi atau di delay dengan mempertimbangkan resource yang sedang tersedia saat itu, resource yang sedang digunakan oleh proses, resource yang akan direquest dan resource yang akan dilepas oleh masing-masing proses pada waktu yang akan datang.

##### 6.x.3.1.1. Deadlock Prevention

Jadi bagaimana mencegah/prevention deadlock? Caranya kita cukup mencegah salah satu dari ke 4 syarat perlu dari deadlock yang telah dijelaskan sebelumnya, satu saja dari ke empat syarat tersebut tidak terpenuhi maka deadlock tidak akan terjadi.

- **Mutual exclusion**  
Menghalangi mutual exclusion artinya membiarkan sebuah resource dapat digunakan bersama dalam satu waktu oleh beberapa proses berbeda. Namun ini mustahil bisa terus dilakukan karena misalnya kita memiliki disk, tentu saja kita tidak akan mengizinkan dua buah proses melakukan proses write ke file disk dalam waktu bersamaan karena akan terjadi inkonsistensi data nantinya.
- **Hold and Wait**  
Kita harus dapat menggaransi bahwa setiap proses yang merequest resource tidak sedang memegang resource. Caranya dengan meminta proses untuk merequest dan meangalokasikan semua resource yang akan dibutuhkannya sebelum dia dieksekusi. Namun ini susah untuk dilakukan karena kita tidak bisa menebak resource apa yang akan dibutuhkan oleh sebuah proses. Satu saja resource yang dibutuhkannya tidak ada maka proses ini tidak bisa dieksekusi. Overheadnya akan tinggi dan mungkin saja akan terjadi starvation karena bisa saja ada proses yang tidak pernah bisa terpenuhi kebutuhan resourcenyaa
- **Not preemptive**  
Menghalangi not preemptive artinya mengizinkan proses untuk bersifat preemptive. Jika sebuah proses sedang memegang beberapa resource dan merequest resource lainnya yang tidak bisa didapatnya sesegera mungkin, maka semua resource yang



dipegangnya sekarang harus dilepaskannya. Kesulitanya adalah kita harus membuat list resource yang sedang direquest oleh proses-proses dan juga menentukan bagaimana SO mengalokasikan resource ke proses-proses yang sedang menunggu itu dan resource mana yang akan dialokasikannya.

Proses hanya akan direstart jika hanya ketika dia mendapat resource lamanya dan resource baru yang dia telah request.

- **Circular Wait**

Untuk menghalangi circular wait kita harus menerapkan sistem total ordering terhadap permintaan resource.

Contoh: jika P1 memegang resource dengan nomor 2 maka dia tidak boleh merequest resource dengan nomor 1 dan 0. Yang bisa direquestnya adalah resource dengan nomor resource diatas 2 misalnya 3,4, dan seterusnya.

#### 6.x.3.1.2. Deadlock Avoidance

Seperti yang disebutkan diatas untuk melakukan deadlock avoidance SO harus memiliki priori information seperti:

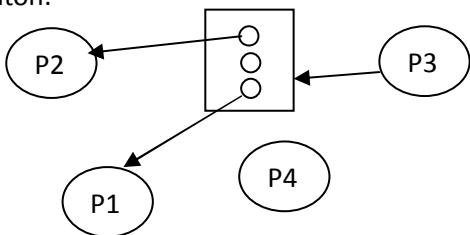
- Jumlah maksimum resource dari setiap type yang sebuah proses mungkin butuhkan. Informasi ini diberikan ke SO sebelum proses tersebut dieksekusi.
  - Informasi apakah **resource allocation state** akan memasuki circular wait atau tidak. Deadlock-avoidance algorithm akan membuat resource allocation graph. Kemudian Deadlock-avoidance algorithm secara dynamic, sebelum SO memberikan resource akan memeriksa apakah kedepannya **resource allocation state** akan membentuk circular wait atau tidak. Jika menghasilkan circular wait maka pemberian resource akan ditunda.
- Resource-allocation state** adalah didefinisikan oleh jumlah resource yang available, jumlah resource yang telah dialokasikan, dan maksimum kebutuhan resource oleh proses.

#### 6.x.3.1.2.1. Safe State

Algoritma deadlock avoidance berjalan setiap terjadi request, pada saat terjadi request dia akan mengecek apakah jika proses diberikan resource maka apakah system akan memasuki unsafe state atau tidak.

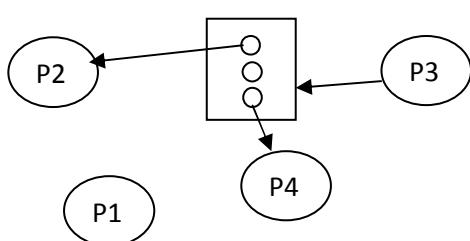
Unsafe state adalah system yang tidak safe state. **Safe state adalah** jika terdapat barisan(urutan berlaku)  $\langle p_1, p_2, \dots, p_n \rangle$  dari **semua** proses yang ada disistem sedemikian sehingga untuk setiap  $p_i$ , resource yang masih dapat direquest oleh  $p_i$  dapat dipenuhi oleh resource yang masih available(bebas) + resource yang sedang dipegang oleh semua  $p_j$  dimana  $j < i$ .

Contoh:



P3 membutuhkan 3 resource, Masih dalam **safe state**, karena permintaan P3 masih dapat dipenuhi oleh 1 resource yang masih bebas +2 resource yang





P3 membutuhkan 3 resource ,dalam **unsafe state**, karena permintaan P3 sudah tidak dapat di penuhi oleh 1 resource yang masih bebas +1 resource yang dipegang oleh P2

Jika kebutuhan resource Pi tidak dapat dipenuhi sesegera mungkin maka Pi dapat menunggu hingga semua Pj selesai.

Ketika Pj selesai maka Pi dapat mengambil resourcennya ,mengeksekusi,mengembalikan resource, kemudian terminate

Ketika Pi terminate ,Pi+1 dapat mengambil resource Pi dan seterusnya.

#### Kesimpulan:

1. Jika sebuah sistem dalam saf state pasti tidak deadlock
2. Jika system dalam unsafe state maka mungkin terjadi deadlock
3. Avoidance memastikan system tidak pernah ke dalam unsafe state.

Ada dua algoritma yang dapat kita gunakan

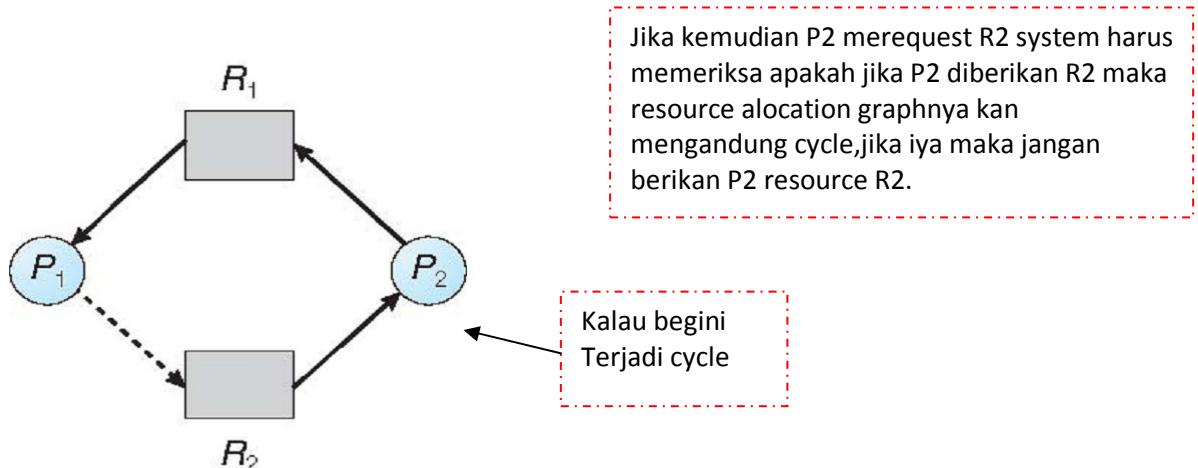
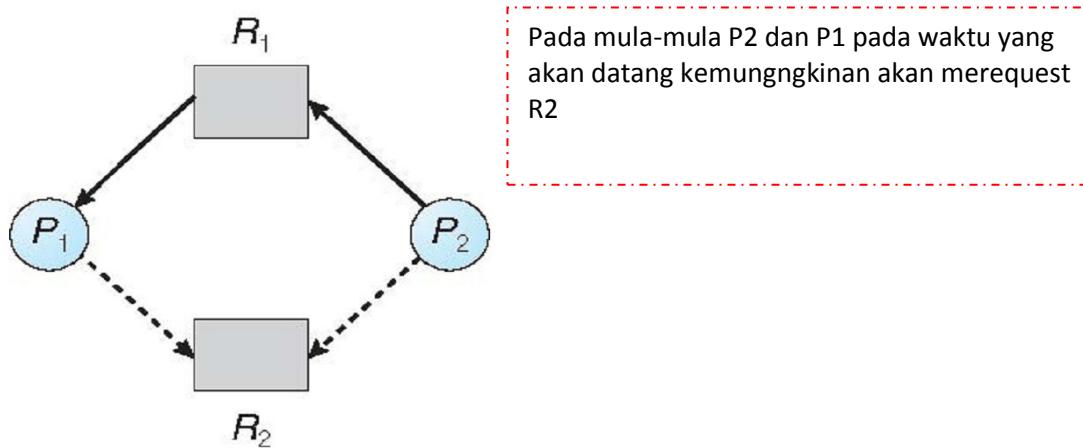
1. Resource allocation graph, namun setiap type resource hanya boleh memiliki satu instance
2. Banker Algoritma,jika setiap type resource memiliki lebih dari satu instance

#### 6.x.3.1.2.1.1.Resource allocation graph

Sama seperti yang sudah dibahas sebelumnya namun kita menambahkan satu edge lagi yang disebut **Claim edge**. Claim edge mengidentifikasi bahwa sebuah proses tersebut diajukan yang akan datang mungkin akan merequest resource tersebut. Claim edge digambarkan dengan garis panah terputus-putus



Contoh



#### 6.x.3.1.2.1.2.Banker Algoritma

Algoritma ini digunakan jika setiap type resource memiliki lebih dari satu instance. Sebelum eksekusi setiap proses harus menklaim berapa maksimum jumlah resource yang akan dibutuhkannya. Jika sebuah proses merequest resource mungkin saja dia harus menunggu dan jika ia telah mendapatkan semua resource yang dibutuhkanya maka dia harus melepas kembali dalam waktu yang terhingga.

Data struktur dari banker algoritma :

- Misalkan  $n$  = jumlah semua proses dalam system dan  $m$  = jumlah type resource
- **Available:** sebuah vektor dengan panjang  $m$ . Jika  $available[j]=k$ , maka ada  $k$  instance dari resource  $R_j$  yang available.
- **Max :** matriks  $n \times m$ . Jika  $Max[i,j]=k$  maka Proses  $P_i$  mungkin akan merequest paling banyak  $k$  instance dari resource  $R_j$ .



- **Allocation:** matrik  $n \times m$ . Jika  $\text{Allocation}[i,j]=k$  maka  $P_i$  sedang memegang  $k$  instance dari resource  $R_j$ .
- **Need:** matrik  $n \times m$ . Jika  $\text{Need}[i,j]=k$  maka  $P_i$  masih membutuhkan  $k$  instance dari resource  $R_j$  untuk dapat menyelesaikan tugasnya.

$$\text{Need } [i,j] = \text{Max}[i,j] - \text{Allocation } [i,j]$$

### Safety Algorithm

1. Let  $Work$  and  $Finish$  be vectors of length  $m$  and  $n$ , respectively.

Initialize:

$$Work = Available$$

$$Finish[i] = \text{false} \text{ for } i = 0, 1, \dots, n-1$$

2. Find an  $i$  such that both:

- (a)  $Finish[i] = \text{false}$

- (b)  $\text{Need}_i \leq Work$

If no such  $i$  exists, go to step 4

3.  $Work = Work + Allocation_i$

$$Finish[i] = \text{true}$$

go to step 2

4. If  $Finish[i] == \text{true}$  for all  $i$ , then the system is in a safe state

Setelah dieksekusi kemudian resource akan dilepas dan kembali ditambahkan jadi available lagi

Kalau tidak **all** berarti unsafe state



Jika sebuah proses melakukan request lakukan algoritma berikut

*Request* = request vector for process  $P_i$ . If  $Request_i[j] = k$  then process  $P_i$  wants  $k$  instances of resource type  $R_j$

1. If  $Request_i \leq Need_i$ , go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If  $Request_i \leq Available$ , go to step 3. Otherwise  $P_i$  must wait, since resources are not available
3. Pretend to allocate requested resources to  $P_i$  by modifying the state as follows:

$$Available = Available - Request;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe  $\Rightarrow$  the resources are allocated to  $P_i$
- If unsafe  $\Rightarrow P_i$  must wait, and the old resource-allocation state is restored

Contoh:

- 5 processes  $P_0$  through  $P_4$ :

3 resource types:

A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Max</u>	<u>Available</u>	<u>Need</u>			
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	3	3	2
$P_1$	2	0	0	3	2	2			
$P_2$	3	0	2	9	0	2			
$P_3$	2	1	1	2	2	2			
$P_4$	0	0	2	4	3	3			

Catatan:

- Max tidak akan berubah-ubah diset memang dari awal



- Mula-mula need diperoleh dari max-allocation, kemudai seterusnya  
$$\text{Need}_i = \text{Need}_i - \text{Request}_i$$
- Available itu vektor dengan panjang m, jadi seperti variabel, yang akan ditimpakan (hanya satu variabel bukan matrik kaya max allocation dan need yang ada entri-entrinya).

Pada Time T<sub>0</sub> ini system masih dalam safe state karena jika kita menjalankan safety algorithm maka

.....

### 6.x.3.2. Deadlock Detection

Seperti halnya deadlock avoidance maka Deadlock Detection dapat dilakukan dengan dua cara

- Membentuk wait-for graph
- Detection algorithm

#### 6.x.3.2.1 Wait For Graph

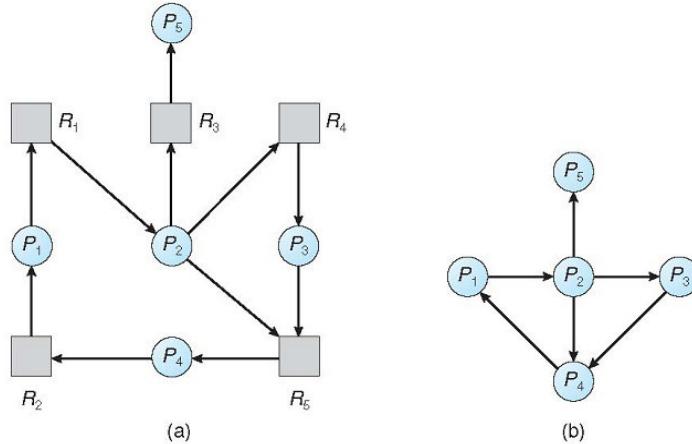
Node akan dianggap sebagai proses.  $P_i \longrightarrow P_j$  jika  $P_i$  menunggu  $P_j$

Secara periodik panggil algoritma untuk mencari cycle di graph. Jika ada cycle maka ada deadlock. Algoritma pencarian cycle memerlukan  $n^2$  operasi, dimana n adalah jumlah vertex di graph.

Bagaimana cara mebentuk Wait-for graph?

Pada dasarnya wait for graph adalah sama dengan resource allocation graph. Jika kita memiliki Resource allocation graph maka kita tinggal menghilangkan Node Resource dan memperthankan anak panah nya saja.

Contoh



Resource-Allocation Graph

Corresponding wait-for graph



### 6.x.3.2.2 Detection Graph

Detection Graph ini sangatlah mirip dengan Banker algorithm. Data strukturnya hampir sama namun Need ditiadakan.

- **Available:** A vector of length  $m$  indicates the number of available resources of each type.
- **Allocation:** An  $n \times m$  matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An  $n \times m$  matrix indicates the current request of each process. If  $\text{Request}[i][j] = k$ , then process  $P_i$  is requesting  $k$  more instances of resource type  $R_j$ .

Algoritmanya adalah:

1. Let  $Work$  and  $Finish$  be vectors of length  $m$  and  $n$ , respectively  
Initialize:
  - (a)  $Work = Available$
  - (b) For  $i = 1, 2, \dots, n$ , if  $\text{Allocation}_{i,i} \neq 0$ , then  $\text{Finish}[i] = \text{false}$ ; otherwise,  $\text{Finish}[i] = \text{true}$
2. Find an index  $i$  such that both:
  - (a)  $\text{Finish}[i] == \text{false}$
  - (b)  $\text{Request}_{i,i} \leq Work$

If no such  $i$  exists, go to step 4
3.  $Work = Work + \text{Allocation}_{i,i}$   
 $\text{Finish}[i] = \text{true}$   
go to step 2
4. If  $\text{Finish}[i] == \text{false}$ , for some  $i$ ,  $1 \leq i \leq n$ , then the system is in deadlock state. Moreover, if  $\text{Finish}[i] == \text{false}$ , then  $P_i$  is deadlocked

**Algorithm requires an order of  $O(m \times n^2)$  operations to detect whether the system is in deadlocked state**



- Five processes  $P_0$  through  $P_4$ ; three resource types A (7 instances), B (2 instances), and C (6 instances)

- Snapshot at time  $T_0$ :

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	0	0	0	0	0	0
$P_1$	2	0	0	2	0	2			
$P_2$	3	0	3	0	0	0			
$P_3$	2	1	1	1	0	0			
$P_4$	0	0	2	0	0	2			

#### 6.x.3.Detection –algorithm Usage

Kapan dan seberapa sering kita menggunakan Detection algorithm tergantung pada:

- Seberapa sering deadlock dapat muncul
- Berapa proses yang harus dirole back jika terjadi deadlock  
Biasanya satu proses untuk satu cycle.



#### 6.x.4. Recovery from deadlock

Dapat dilakukan dengan beberapa cara diantaranya

- Process Termination
- Resource preemption

##### 6.x.4.1. Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

##### 6.x.4.2. Resource preemption

- Selecting a victim – minimize cost
- Rollback – return to some safe state, restart process for that state
- Starvation – same process may always be picked as victim, include number of rollback in cost factor



## CHAPTER 7.MAIN MEMORY

### 7.1.Background

Main memori dan register adalah dua buah media penyimpanan yang dapat diakses oleh CPU. Semua program yang akan dieksekusi oleh CPU harus dibawah terlebih dahulu ke memory. Kemudian CPU akan membawa data atau intruksi yang diperlukan ke register nya. CPU tidak dapat mengubah-ubah data pada memory yang dapat diubahnya hanya data pada register.Yang dilakukan CPU ke memory adalah hanya sebatas meload data atau intruksi ke registernya dan menyimpan/store data dari registernya ke main memori. CPU dapat mengakses register dalam satu cycle CPU clock bahkan kurang.Sedangkan untuk mengakses main memori CPU biasanya membutuhkan lebih dari satu cycle CPU clock. Hal ini mengakibatkan CPU terkadang harus menunggu(stall) ketika membaca/menulis data dari/ke memori.Solusinya adalah dengan menambahkan satu memry berkecepatan tinggi diantara CPU dan main memory yang disebut cache(buffer dari main memory).Cache lebih cepat dari memory namun lebih lambat dari register.

Selain masalah kecepatan akses ke phisical memori,kita juga harus memastikan lokasi memori dari system operasi tidak terganggu oleh *user proses*.Beginu pula lokasi memori user proses yang satu tidak diganggu oleh proses user yang lain. Jadi kita harus membatasi range alamat memory yang legal yang dapat diakses oleh sebuah proses. Mekanisme dasar yang digunakan adalah penggunaan register base dan limit. Setiap proses memiliki base dan limit yang berbeda. Pada saat terjadi proses switching maka base dan limit untuk sebuah proses dirubah. Base dan limit adalah ekstensi logis dari memory,setiap proses akan menganggap dirinya dapat mengakses semua alamat memori dari 0 sampai n(seakan-akan diberjalan sendiri tidak ada proses lain di memory) ,namun sebenarnya alamat yang diaksesnya tersebut akan dipetakan lagi (nanti dipelajari pemetaanya) menggunakan base dan limit oleh SO untuk mengakses alamat phisik memory.

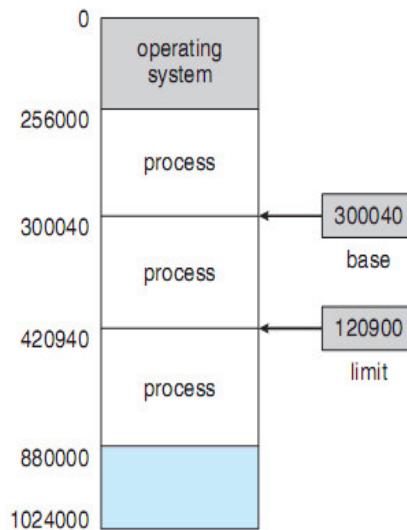


Figure 7.1 A base and a limit register define a logical address space.



Misalkan perhatikan gambar 7.1 diatas, sebuah proses memiliki base 300040 dan limit 120900, berarti proses tersebut hanya akan dianggap legal jika mengakses alamat 300040 sampai 420940. Jika sebuah proses mengakses diluar dari base dan limitnya maka akan ditrap dan akan terjadi error. Hanya SO yang dapat menyeting base dan limit serta dapat pula merubahnya.

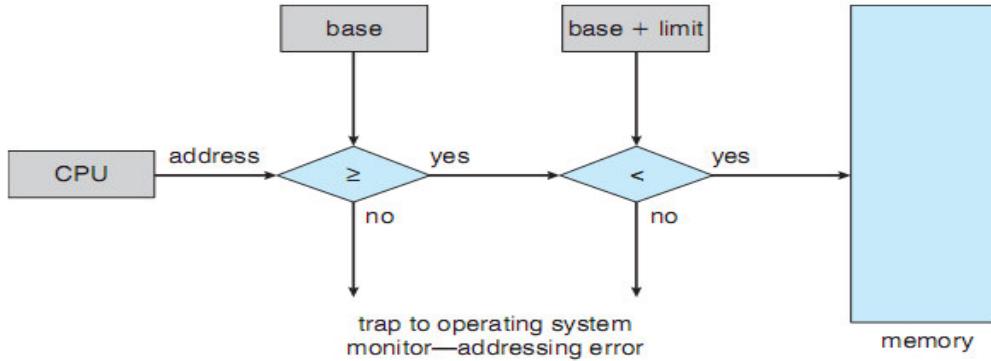


Figure 7.2 Hardware address protection with base and limit registers.

Selain menggunakan register base dan limit, dikenal juga yang namanya **binding**. Sebuah program biasanya terdiri dari data dan intruksi. Seperti yang disebutkan diatas untuk dapat dieksekusi data dan intruksi tersebut harus diload terlebih dahulu ke suatu lokasi dimemory. Penempatan lokasi program di memory biasanya dilakukan secara symbolic, compiler akan **membind** alamat yang berbentuk symbolic ini menjadi relocatable address misalnya 14 byte dari awal alamat memori. Jadi walaupun alamat memori dimulai dari 0000 maka sebuah program tidak harus penempatan di memorinya start dari alamat 0000. Setelah itu loader atau linkage editor akan **membind** relocatable address menjadi absolute address (alamat phisik memory yang sebenarnya) misalnya 74014. Sebenarnya proses binding program(intruksi dan data) ke alamat memory dapat dilakukan pada beberapa waktu:

- **Compile time**  
Jika pada saat compile dicode program kita sudah sebutkan alamat phisik memory dimana kita ingin program kita diletakan (kita sudah mengetahui dimana lokasi/alamat memory program akan disimpan) maka bind dapat dilakukan pada saat itu, yaitu dengan menciptakan **absolute code**(kode alamat memory sebenarnya/phisik). Jika pada suatu waktu lokasi memory dinginkan dirubah maka program harus dicompile ulang. Cara seperti ini biasanya diterapkan pada compiler bersifat MS DOS tempo dulu.
- **Loading Time**  
Jika pada saat compile kita tidak mengetahui alamat dimana program akan ditempatkan maka pada saat load, tugas compiler haruslah menciptakan **relocatable code**, yaitu mengubah symbol variabel yang kita tuliskan diprogram menjadi sebuah relocatable address seperti yang dicontohkan diatas. Misalnya intruksi  $i:=j+1$ , i dan j haruslah diubah oleh compiler menjadi relocatable address. Jika pada suatu waktu kita ingin mengubah lokasi memory kita hanya perlu mereload user code kita.
- **Execution time**  
Jika sebuah program pada saat eksekusinya kita inginkan dapat berpindah-pindah dari suatu segment ke segment lain pada memory maka **binding** haruslah dilakukan pada saat eksekusi. Cara seperti ini biasanya dilakukan dengan dukungan hardware(nanti dibahas pada 7.1.3 dibuku)

Berikut ini fase-fase sebuah program sebelum dieksekusi

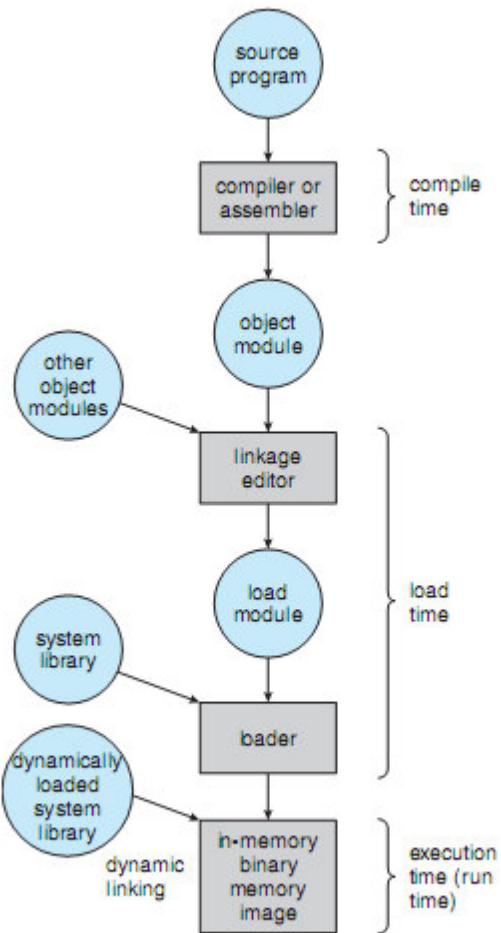


Figure 7.3 Multistep processing of a user program.

**Dynamic loading** adalah salah satu cara untuk meningkatkan utilitas Memory, dengan cara ini sebuah routin yang dibutuhkan oleh proses tidak akan di load ke memory sebelum dipanggil oleh routine didalam proses tersebut. Biasanya routine yang belum dipanggil berada di disk.

**Dynamic linking** adalah juga salah satu cara untuk meningkatkan utilitas memory. Library yang digunakan oleh proses akan dilinking pada saat execution time. Sebuah proses dapat menggunakan library yang terdapat pada proses lain (dengan melink). **Statistik linking**, librarynya digabung dalam program.

**Perbedaan dengan dynamic loading** adalah pada dynamic loading tidak memerlukan bantuan SO secara langsung, SO hanya membantu menyediakan librarynya, namun pada dynamic linking SO sangat berperan jika library yang dibutuhkan berada pada proses lain yang tidak mungkin diakses (dilink) oleh proses lain, maka SO lah yang akan mengaseskannya untuk dynamic linking.

Intinya kayanya begini ☺ Dynamic loading untuk mengeload ke memory, dynamic linking spesial untuk melink.



## 7.2. Logical Versus physical address

Address yang diciptakan oleh CPU disebut **logical address**, sedangkan address yang hanya dapat dilihat oleh unit memory dan merupakan alamat yang diload ke register memory-address disebut **physical address**. Binding pada compile time dan load time mengenerate phisical dan logical address yang sama. Namun pada execution time binding menghasilkan logical dan phisical address yang berbeda. Biasanya logical address pada execution time disebut **virtual address**.

Pemetaan dari alamat logical address ke phisical address dilakukan oleh hardware yang disebut **memory-management unit(MMU)**. Ada banyak metode yang dapat digunakan untuk melakukan mapping ini seperti yang akan dibahas pada chapter 7.3 sampai 7.7 di buku (contigues, segmentation, paging dll).

Untuk sementara kita akan melihat dulu contoh cara kerja MMU menggunakan skema pemetaan sederhana menggunakan base register yang pada kasus ini disebut relocation register seperti gambar dibawah ini.

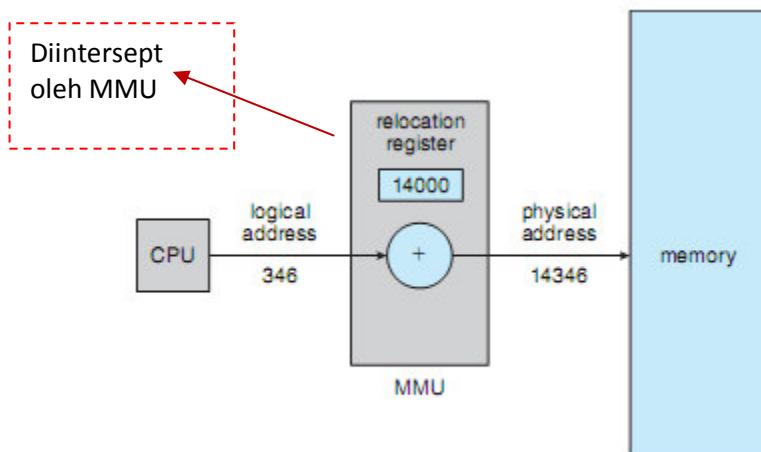


Figure 7.4 Dynamic relocation using a relocation register.

User program tidak pernah berhubungan dengan phisical address, user program berpikir bahwa proses yang dibuatnya berjalan pada lokasi memory 0 sampai max.

## 7.3. Swaping

Nanti dibahas.....

## 7.4. Contiguous Memory Allocation

Teknik ini adalah salah satu teknik dari manajemen memory, manajemen memori pada dasarnya adalah mekanisme bagaimana logical address dipetakan ke phisical address.

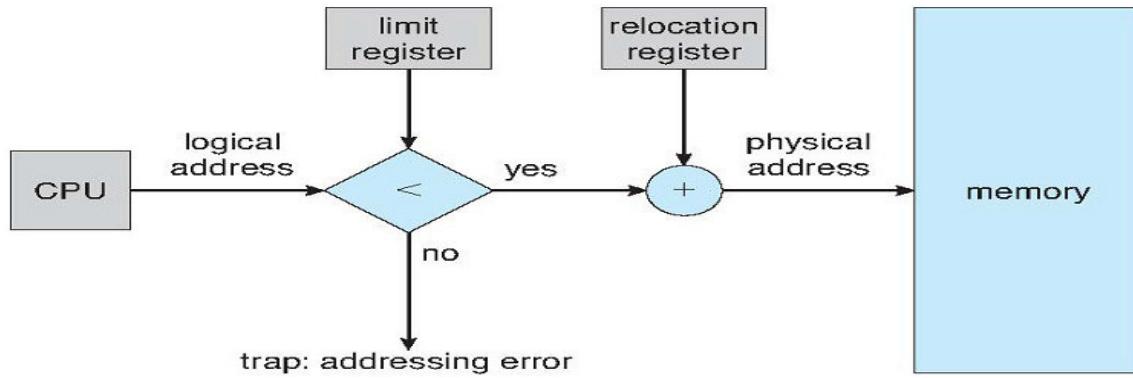
Main memory dibagi dalam dua bagian:

- Area dimana operating system berada, biasanya berada di area bawah memory bersama interrupt vektor.
- Area dimana user prosess berada, biasanya berada di area atas memory.

Setiap proses berada pada satu bagian contiguous dari memory.



Untuk melakukan proteksi,pada teknik ini digunakan relocation register(base register) dan limit register seperti yang telah dibahas sebelumnya.Relocation berisi alamat phisical memory terkecil yang dapat digunakan oleh proses yang bersangkutan. Limit register berisi range dari logical address yang dapat digunakan oleh proses bersangkutan.cara kerjanya seperti gambar dibawah ini:



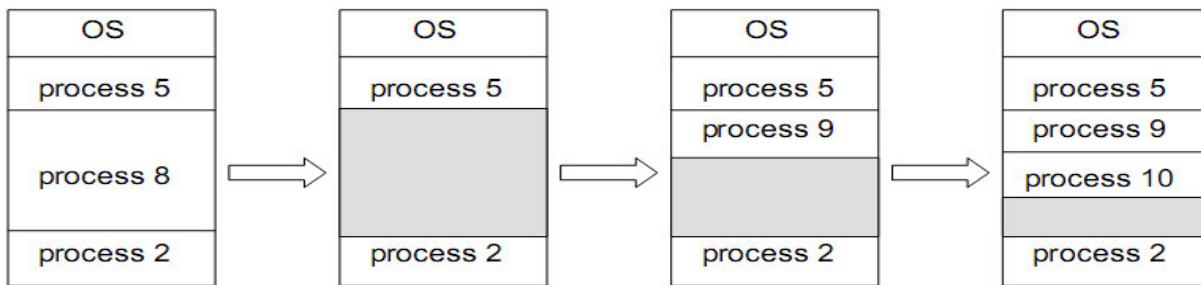
Mekanisme paling sederhana dari contiguous allocation adalah dengan membagi-bagi memory menjadi partisi-partisi dengan ukuran yang fixed,teknik ini disebut **multiple partition method**. Oleh karena itu dengan menggunakan teknik ini degré dari multiprograming ditentukan oleh banyaknya partisi. Pada saat sebuah partisi kosong maka sebuah proses diambil dari input queue dan diisikan ke partisi tersebut,jika proses ini selesai dijalankan maka partisi ini dikosongkan lagi dan siap digunakan oleh proses lain. Teknik ini pernah digunakan pada IBM SO/360(sudah tidak digunakan lagi).

Mekanisme yang lain adalah pengembangan dari mekanisme **multiple partition method** yaitu **metode variable-partition**. Perbedaannya adalah partisi pada teknik ini tidak fixed jadi dapat berubah-ubah (variabel). Pada metode ini System operasi memiliki tabel yang menampung partisi yang kosong(available) dan partisi yang sedang digunakan. Mula-mula memory dianggap sebagai satu partisi kosong yang besar (hole) yaitu tempat yang kosong dimana proses bisa ditempatkan.Pada saat proses masuk ke input queue maka system operasi menghitung ukuran memory yang dibutuhkan oleh proses tersebut dan memilih proses mana yang akan diberikan alokasi memory.

Jika pada suatu saat SO memiliki beberapa blok size yang available/hole dan beberapa proses di input queue. Maka SO memilih proses dari input queue menggunakan algoritma scheduling.Sebuue proses akan dialokasikan ke memory jika ada hole pada memory yang dapat menampung kebutuhan memory proses tersebut. Jika ada sebuah proses terlalu besar dan SO tidak mendapatkan tempat/hole yang cukup untuk menampung proses tersebut maka system operasi harus menunggu hingga ada hole yang cukup ukuranya,atau SO bisa menskip proses tersebut dan mencari proses lain yang masih dapat ditampung oleh hole.

Seperti yang dapat dilihat teknik ini memunculkan hole, menyebar di keseluruhan memory. Pada saat suatu proses membutuhkan lokasi memory, maka SO akan mencarikan hole yang cukup buat proses tersebut,jika hole yang didapat oleh SO terlalu besar(tidak adami yang pas,hanya ada yang lebih besar dari ukuran proses) maka hole tersebut dibagi dua,satu bagian digunakan untuk menempatkan proses tersebut dan satu bagian lagi kembali menjadi hole. Jika pada saat sebuah proses diterminated atau

selesaimi runingnya maka lokasi memorynya menjadi hole kembali. Jika hole tersebut bertetangga dengan hole lagi maka hole-hole tersebut dimerge.



Systen operasi harus mampu melakukan algoritma yang dapat mensatisfied sebuah permintaan ukuran n untuk dialokasikan ke suatu hole dari beberapa hole sehingga hole yang dihasilkan akhirnya jumlahnya seminimal mungkin. Masalah ini disebut **dynamic storage-allocation problem**. Ada beberapa algoritma yang dapat dilakukan diantaranya.

- **First Fit**

Metode ini mengalokasikan proses ke hole **yang pertama kali ditemukan** dan cukup untuk memuat proses tersebut. Pencarian akan distop jika sudah menemukan hole yang cukup untuk proses tersebut.

- **Best Fit**

Metode ini mengalokasikan proses ke hole yang **terkecil** dan dapat memuat proses tersebut. Oleh karena itu pencarian harus dilakukan ke semua hole. Metode ini menghasilkan hole yang ukuranya kecil.

- **Worst Fit**

Metode ini mengalokasikan proses ke hole **terbesar** dan memuat proses tersebut. Oleh karena itu pencarian harus dilakukan ke semua hole. Metode ini menghasilkan hole yang ukuranya besar.

Berdasarkan penelitian First Fit dan Best Fit lebih baik jika dipandang dari segi kecepatan dan storage utilization.

Kelemahan utama dari metode manajemen memory yang contiguous adalah munculnya external dan internal fragmentation. **External fragmentation** terjadi ketika terdapat space memory yang kosong yang sebenarnya mungkin dapat memuat sebuah proses, namun space kosong tersebut tersebar menjadi hole-hole yang kecil yang tidak contiguous.

Alokasi memory tidak selalu sama dengan kebutuhan proses sebenarnya. Misalnya sebuah proses membutuhkan 11.257.486 byte, namun karena pengalokasian memory dilakukan dengan kelipatan tertentu misalkan 1kB(1000 Byte) maka alokasi memory yang akan dialokasikan ke proses tersebut adalah 11.258.000. Rasio/perbedaan ukuran memory yang dibutuhkan sebuah proses dan alokasi memory yang dapat dialokasikan tersebut disebut **Internal Fragmentation**. Tidak ada cara untuk mengetasi Internal fragmentation.

Salah satu solusi untuk mereduksi external fragmentation adalah dengan **compaction**. Tujuan utama dari compaction adalah untuk menggabungkan semua hole-hole yang terpisah-pisah(tidak contiguous) menjadi satu hole/blok besar. Compaction hanya dapat dilakukan jika proses relocation dynamic dan proses



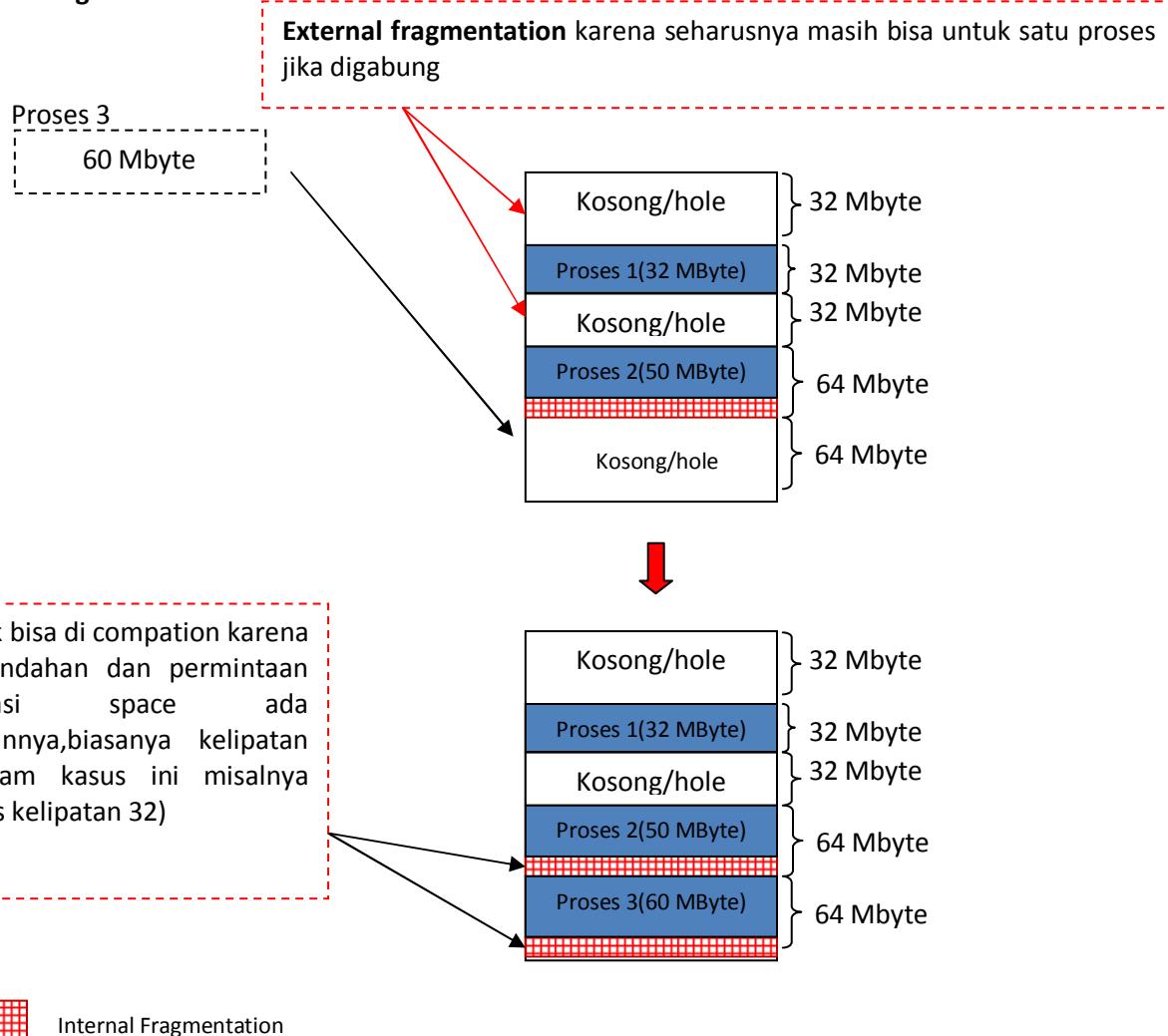
binding dilakukan pada saat execution time(knp? karena pada kondisi tersebut proses dapat dipindah-pindah).

Selain itu cara lain adalah dengan menggunakan metode manajemen memory yang lain yaitu segmentasi dan paging ataupun gabungan keduanya.

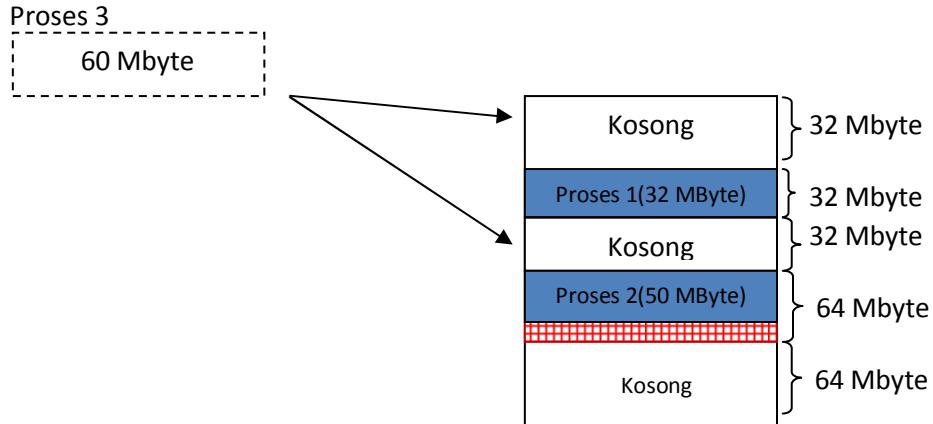
## 7.5.Paging

Perbedaan secara umum contiguous dan paging(dalam gambar)....

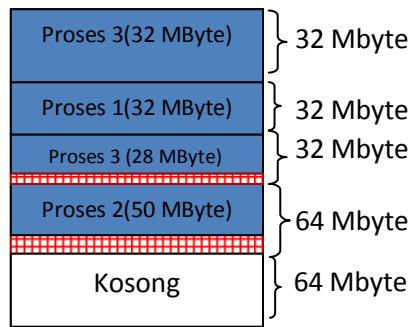
### Contiguous



## Paging



TIDAK ADA EXTERNAL FRAGMENTATION



Internal Fragmentation

Alamat memory logis pada metode paging tidak harus contiguous (boleh melompat-lompat seperti pointer dan array). Memory phisik dibagi dalam blok-blok dengan ukuran yang tetap yang disebut **frame**. Ukuranya biasanya adalah kelipatan dua dari 512 byte sampai 16 Mbyte. Memori logis dibagi-bagi kedalam blok dengan ukuran yang tetap yang disebut **page**. System operasi akan memaintenace semua frame-frame yang kosong. Untuk menjalankan sebuah program dengan ukuran N page, maka SO harus menemukan N frame yang kosong kemudian meload program tersebut. Pada metode paging masih terdapat **internal fragmentation** tapi tidak dengan **external fragmentation**.

Untuk melakukan pemetaan dari memory logis ke memory phisik dibutuhkan **page table**. Alamat yang digenerate oleh CPU dibagi dua bagian

- Page Number( $p$ )  
Digunakan untuk mengindeks ke page table, page table terdiri dari base address dari setiap page pada phisical memory.
- Page offset( $d$ )  
Akan dikombinasikan dengan base address untuk mendefiniskan alamat dari page pada phisical memory. Alamat ini yang dikirim ke memory unit.

Untuk logical address space  $2^m$  dan ukuran page  $2^n$

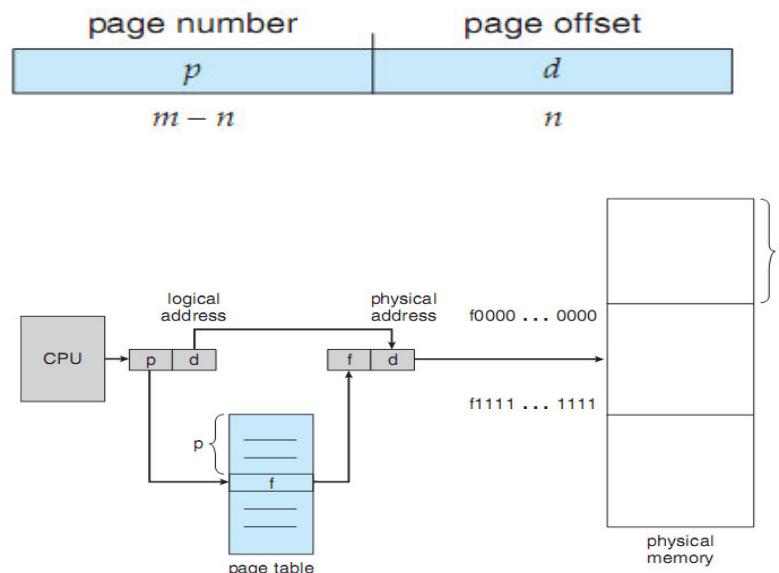
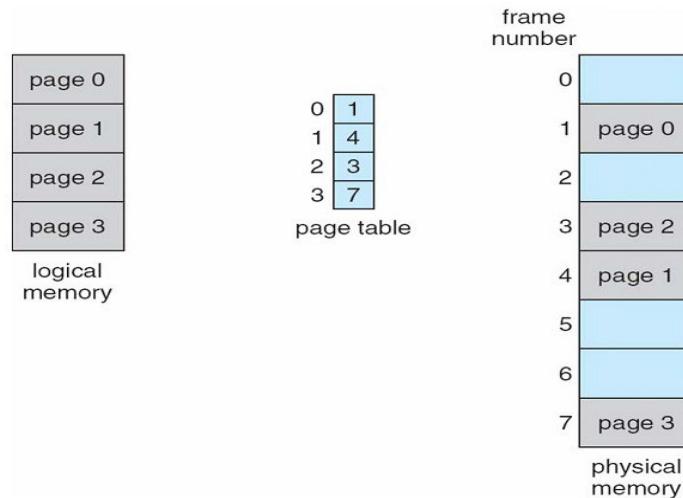
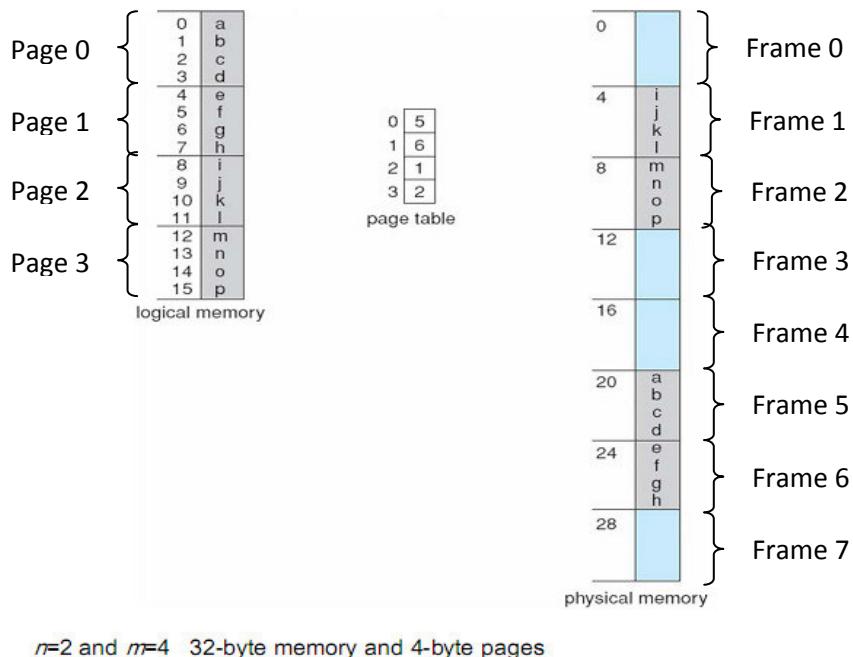


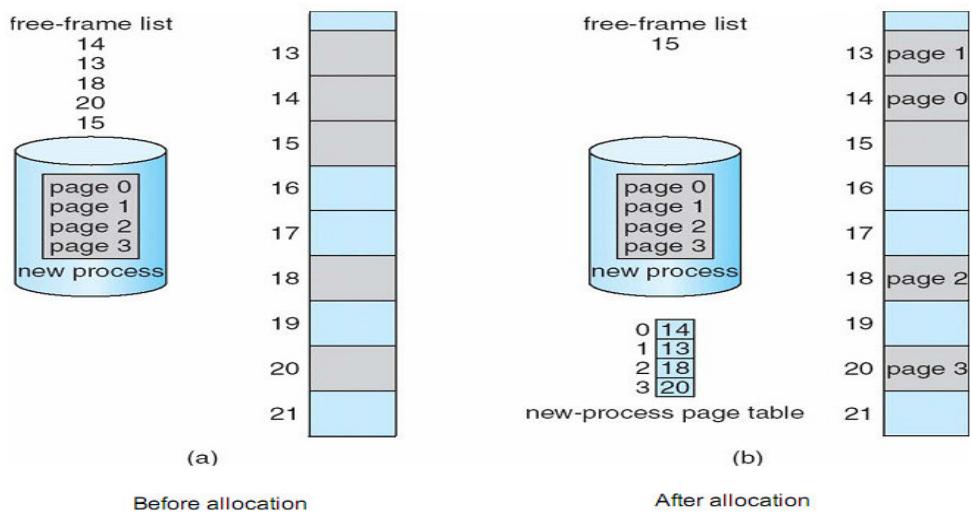
Figure 7.7 Paging hardware.





$n=2$  and  $m=4$  32-byte memory and 4-byte pages

### Bagaimana free frame diatur



### Implementasi page tabel

Setiap proses memiliki page table sendiri. Page table diletakan di memory, jadi jika sebuah program ingin diakses maka hal yang pertama dilakukan adalah mencari page tablenya. Setiap page tabel memiliki **Page-table base register (PTBR)**, yaitu pointer ke lokasi page table dimemory **dan Page-table length register (PTRLR)**, mengindikasikan ukuran dari page table. Oleh karena itu untuk mengakses sebuah



alamat diperlukan dua kali akses ke memory hal ini menjadikan proses translasi itu sendiri dua kali lebih lambat dari memory(power of 2). Solusinya adalah dengan menggunakan fast-lookup hardware cache yang disebut **associative memory or translation look-aside buffers (TLBs)**. Namun sayangnya hardware ini sangat mahal.Cara kerja TLB:

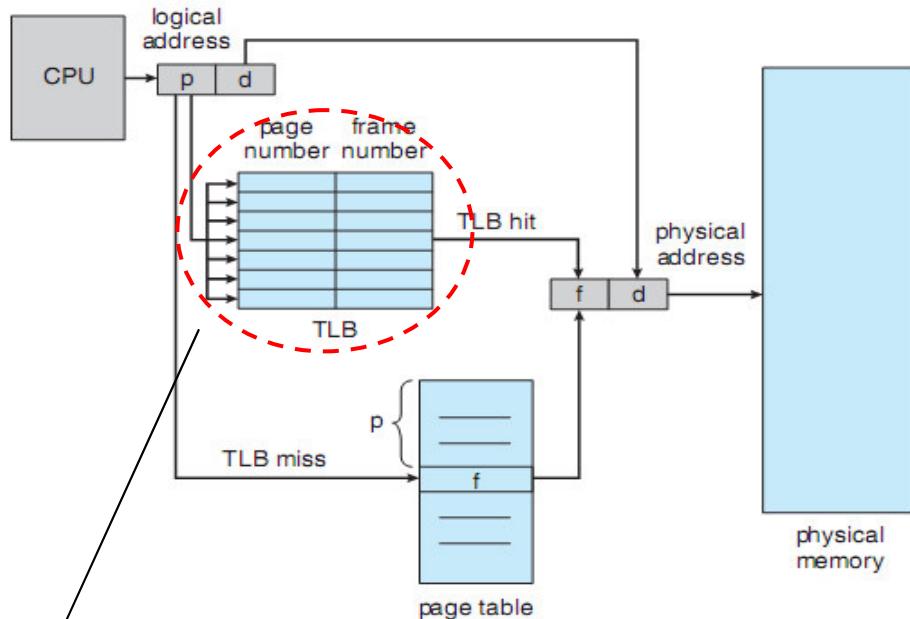


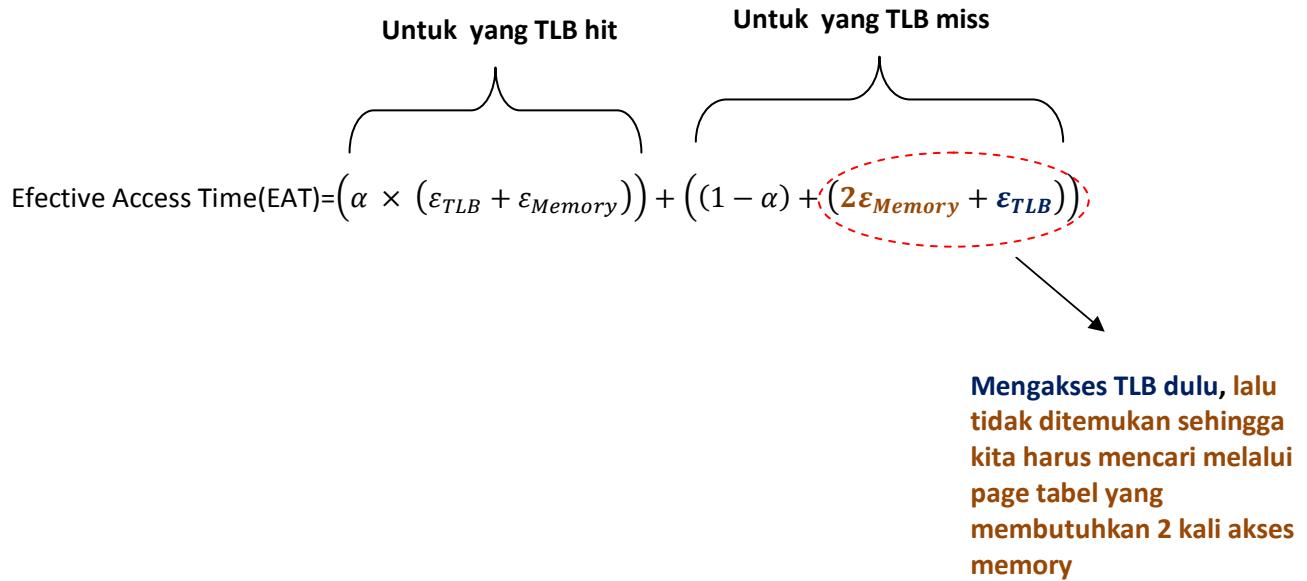
Figure 7.11 Paging hardware with TLB.

Pencarian dilakukan secara paralel(mis: satu entri (p) dimasukan langsung dicari di keseluruhan list dalam waktu bersamaan)

Ukuran TLB sangatlah kecil (64 sampai 1024 entries) sehingga biasanya tidak semua frame physical memory dapat ditampungnya. Sehingga terkadang ada entri yang tidak ditemukan(**TLB miss**) sehingga jika hal ini terjadi maka kita harus kembali mencari dengan menggunakan page table (dengan dua kali akses). Jika frame telah ditemukan dengan menggunakan page table biasanya frame tersebut dimasukan ke TLB untuk kebutuhan pengaksesan selanjutnya. Jika TLB penuh maka biasanya ada yang diganti dengan menggunakan algoritma pergantian(bisa secara random atau list recently uses). Namun pada TLB ada entry yang sifatnya wired down artinya tidak bisa diganti biasanya berhubungan dengan kernel kode, ini dimaksudkan untuk keperluan mempercepat akses ke kernel.

### Menghitung Efektive Memory Access Time

Percentase sebuah page number ditemukan di TLB disebut Hit Ratio. Untuk menghitung **Efektive Memory Access Time** yaitu:



Contoh:

Consider  $\alpha = 80\%$ ,  $\epsilon = 20\text{ns}$  for TLB search,  $100\text{ns}$  for memory access

- $\text{EAT} = 0.80 \times 120 + 0.20 \times 220 = 140\text{ns}$

Consider slower memory but better hit ratio  $\rightarrow \alpha = 98\%$ ,  $\epsilon = 20\text{ns}$  for TLB search,  $140\text{ns}$  for memory access

- $\text{EAT} = 0.98 \times 160 + 0.02 \times 300 = 162.8\text{ns}$



### Memory Protection

Karena pada metode paging tidak ada base dan limit register maka memory protection dilakukan dengan **valid-invalid bit**. Valid-invalid bit biasanya ditempatkan bersama page table. Valid-invalid menentukan apakah page yang bersangkutan adalah page yang dapat diakses oleh proses tersebut atau tidak. Berikut contoh penggunaan

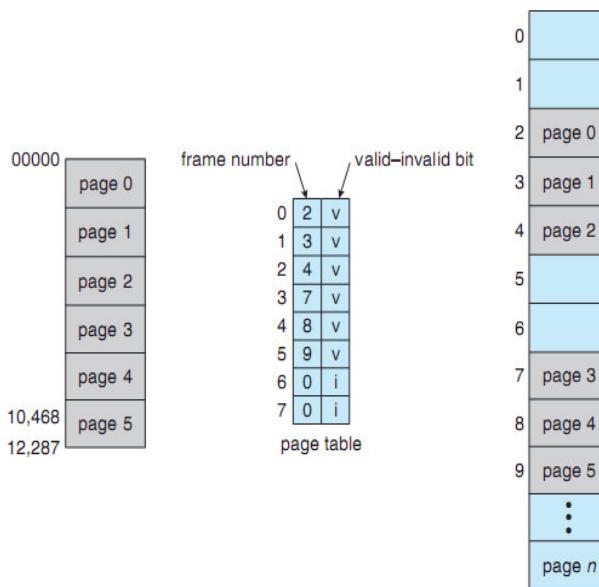


Figure 7.12 Valid (v) or invalid (i) bit in a page table.

Jadi proses tersebut hanya dapat mengakses page 0,1,2,3,4,5 karena pagenya diset valid.Jika proses mencoba mengakses(melakukan proses write) ke page 6,7 maka akan ditrap.

Setiap proses memiliki kebutuhan page yang terkadang berubah-rubah. Untuk itu biasanya SO meberikan alokasi page yang tidak pas dengan kebutuhan proses tersebut. Jika kebutuhan proses tersebut tersebut berkurang maka SO tinggal mengeset ivalid bit kepada page-page yang tidak digunakan proses tersebut(tidak perlu didelete). Jika suatu waktu proses tersebut membutuhkan page tersebut lagi maka tinggal diset valid lagi oleh SO.

Selain menggunakan valid-invalid bit untuk proteksi, pada metode paging juga bisa menggunakan **Page Table Length Register(PTLR)** seperti yang dijelaskan sebelumnya



### Shared Page.

Beberapa proses dapat menggunakan bersama-sama sebuah page. Misalnya pada kasus jika beberapa proses menggunakan beberapa notepad dalam suatu waktu, setiap proses tidak akan memiliki page masing-masing untuk notepad tersebut namun cukup satu page untuk satu notepad kemudian kesemua proses tersebut menggunakan bersama page tersebut (dshare). Namun data untuk setiap proses tetap terpisah-pisah page nya.

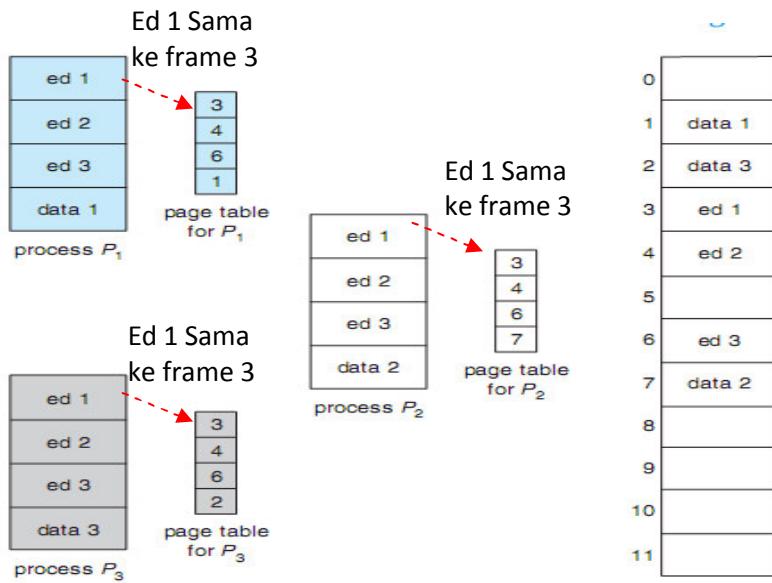


Figure 7.13 Sharing of code in a paging environment.

### 7.6. Struktur Page Table

Beberapa teknik untuk memstruktur page table adalah:

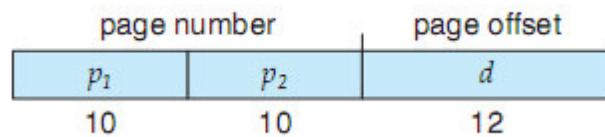
- **Hierarchical Paging**
- **Hashed Page Tables**
- **Inverted Page Tables**

#### 1. Hierarchical Paging

Struktur memory untuk paging akan membesar jika menggunakan metode biasa (lurus-lurus saja). Misalkan computer kita adalah komputer dengan logical address 32 bit. Satu Page berukuran = 4 KB ( $2^{12}$ ). Jadi terdapat sekitar 1 juta ( $2^{32} / 2^{12}$ ) buah page yang harus ditampung oleh page tabel (ada 1 juta entry pada satu page tabel). Jika setiap entry itu adalah 4 byte maka keseluruhan ukuran page table adalah 4 Byte x 1 juta entry = 4 MB. Jadi phisical address harus menampung 4 MB hanya untuk suatu page tabel. Hal itu tentu saja sangat besar dan tentu saja kita tidak ingin mengalokasikan 4 MB secara contiguous di memory.

Salah satu solusinya adalah dengan mempage lagi tabel page itu sendiri dengan menggunakan algoritma two-level paging. Kita kembali ke contoh diatas misalkan pada system dengan 32 bit logical address dan ukuran page 4 KB. Logical address kita bagi menjadi seperti dibawah ini





$p_1$  adalah indeks ke outer page table dan  $p_2$  adalah displacement ke dalam page dari inner page table sedangkan  $d$  adalah displacement ke physical address. Skema page tabel menjadi seperti ini

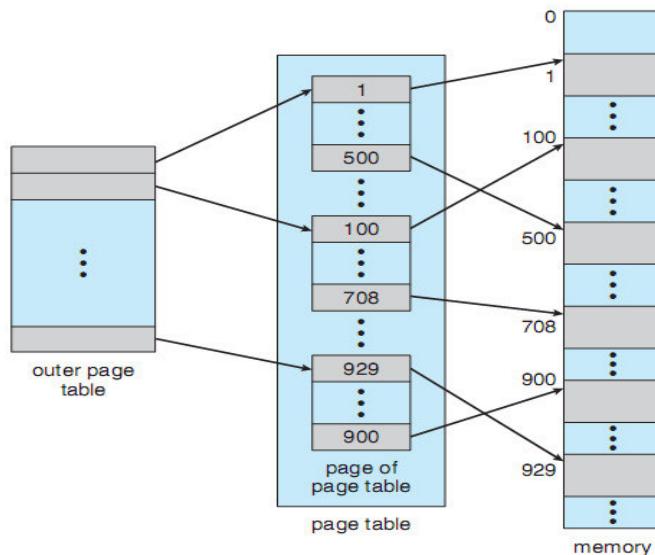


Figure 7.14 A two-level page-table scheme.

Dan skema translasinya pengelamatanya menjadi

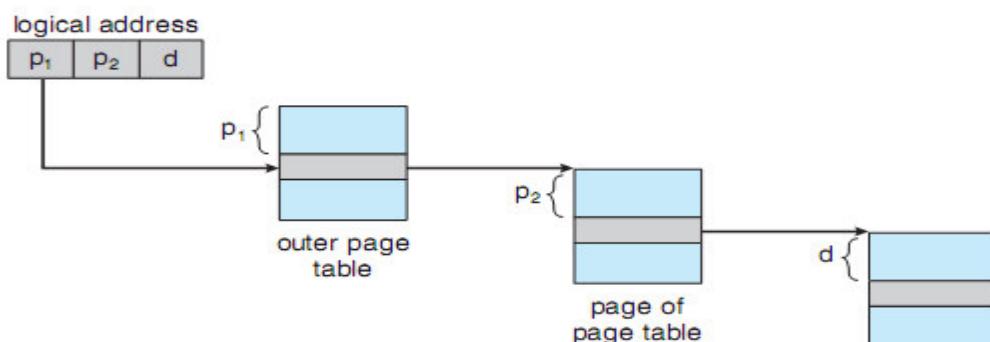
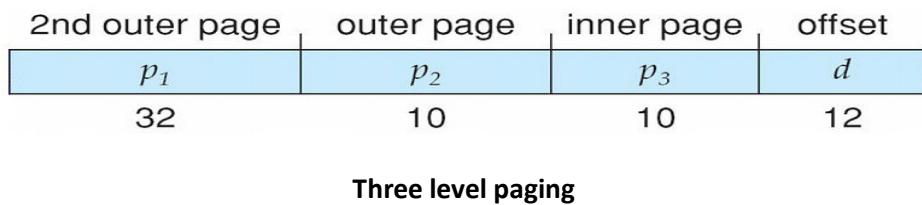
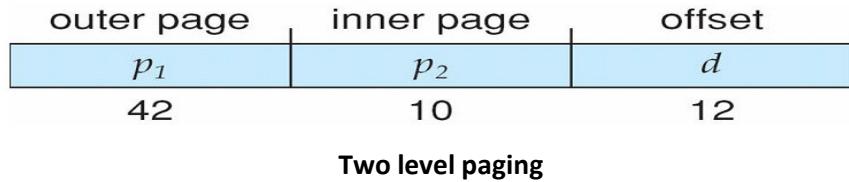


Figure 7.15 Address translation for a two-level 32-bit paging architecture.



### Untuk yang 64 bit

Jika digunakan yang two level paging maka outer page table nya masih terlalu besar jadi sebaiknya dipakai yang three level paging



## 2. Hashed Page Tables

Kelemahan dari page tabel bentuk hirarki adalah kita harus mengakses page tabel beberapa kali untuk mendapatkan alamat frame dari page di phisical memory. **Hashed Page Tables digunakan pada system > 32 bit**. Nomor dari page virtual akan dihash dan nilai hash nya disimpan pada page table. Untuk nomor page virtual memory yang nilai hash nya sama di dalam page table pada entry yang bersesuaian dengan nilai hash tersebut terdapat element linked list (untuk mencegah kolision). Setiap elemen linked list tersebut terdapat (1) virtual page number (2) nilai pemetaan dari page frame(frame yang bersesuai dengan page),(3) pointer ke elemen berikutnya.

Cara kerjanya:

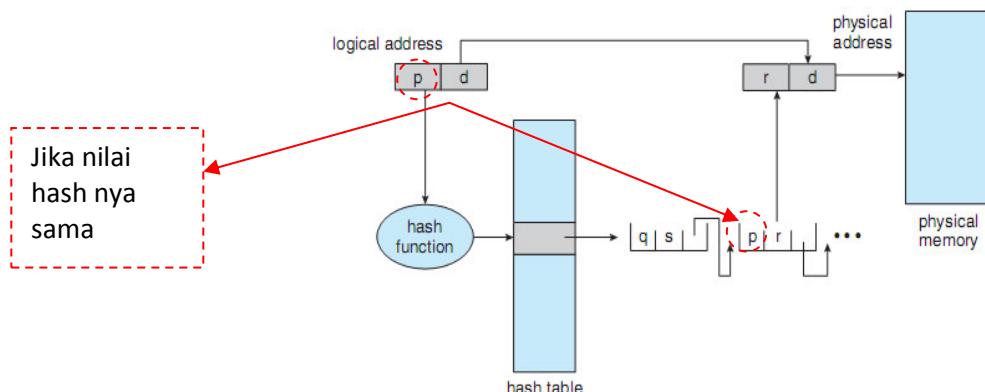


Figure 7.16 Hashed page table.



### 3. Inverted page table

Pada pembahasan sebelumnya setiap proses memiliki satu page tabel yang menyimpan semua page tabel dan frame yang mungkin digunakan oleh proses tersebut. Pada inverted page table semua proses menggunakan satu page table. Pada inverted page table setiap entrynya berkorespondensi dengan real page pada physical memory.

Cara kerja :

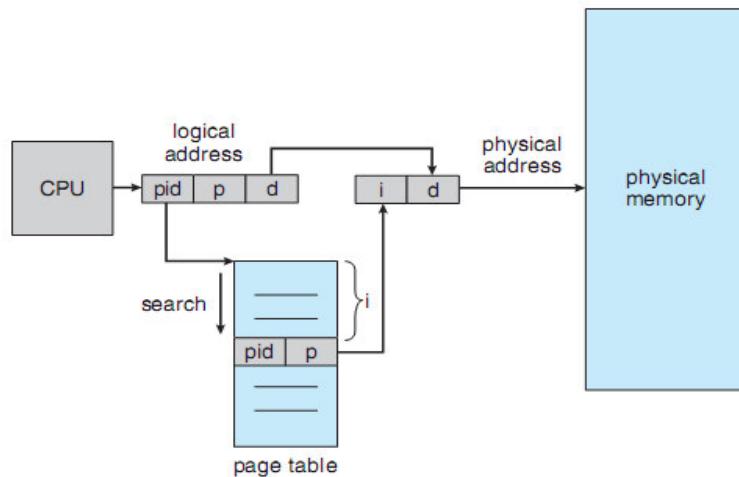


Figure 7.17 Inverted page table.

**Keunggulanya** dari teknik ini dapat menghemat menyimpan page table untuk setiap proses.  
**Kelemahan** dari teknik ini adalah harus melakukan pencarian ke semua entry untuk menemukan pid dan p yang cocok, namun hal ini bisa diatasi dengan menggunakan TLB

## 7.7. Segmentasi

Segmentasi adalah teknik manajemen memory yang mendukung user view, cara user melihat memory. Bagaimana user melihat memory? User tidak menganggap bahwa memory sebagai sebuah array linear dari byte, yang isi sebagian intruksi dan sebagian lagi data. Yang biasa kita pikirkan adalah memory merupakan tempat penyimpanan apa-apa(segment-segment) yang kita gunakan saat menulis program. Kita juga tidak pernah memusingkan apakah dipenyimpanannya (memory) segment-segment yang kita gunakan itu di urut berdasarkan aturan tertentu atau tidak. Segment-segment yang kita gunakan saat menulis program biasanya, yaitu: main program, procedure, function, method, object, local dan global variabel, beberapa blok, symbol tabel (menghubungkan variabel yang kita gunakan dan letaknya di memory), arrays dll. (segment-segment ini nantinya yang akan dipetakan ke memory)

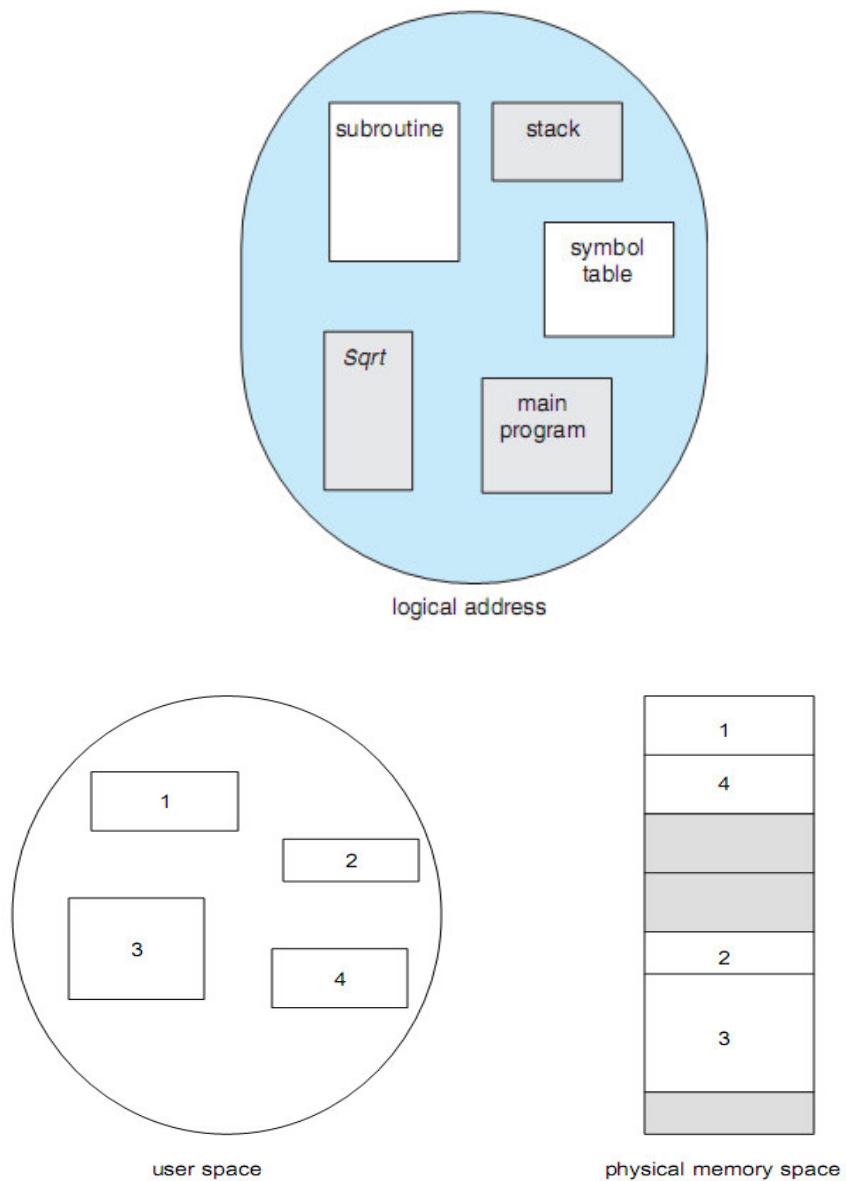


Figure 7.18 User's view of a program.



Oleh karena itu logical address terdiri dari segment-number, setiap segment memiliki nama (kita sebut saja segment number untuk lebih mudahnya) dan panjang segment (offset nya). User akan menspesifikasi (**dapat melihat pembagian**) segment number dan offset nya (offset = byte ke berapa dalam segment tersebut yang kita inginkan). Jadi berbeda dengan paging yang logical address nya oleh user dispesifikasi hanya berupa single address yang nantinya oleh hardware akan dibagi menjadi page number dan offset (**tidak kelihatan oleh user pembagiannya**).

**Logical address = <segment-number,offset>.**

Walupun user pada segmentasi dapat mereferensi alamat sekarang dengan segment number dan offset namun physical memory address masih berbentuk satu /single address. Sehingga masih diperlukan pemetaan. Untuk tujuan itu dalam segmentasi diperlukan pula **segment table**. Setiap entry pada segment table terdiri dari

- Segment base: mengindikasikan alamat permulaan dari sebuah segment
- Segment limit : mengindikasikan panjang segment tersebut

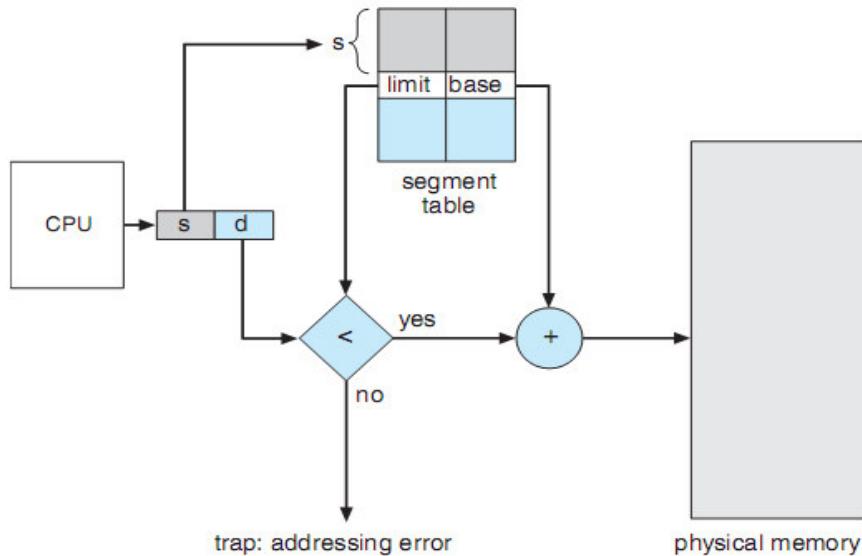


Figure 7.19 Segmentation hardware.

### Contoh penggunaan segment table

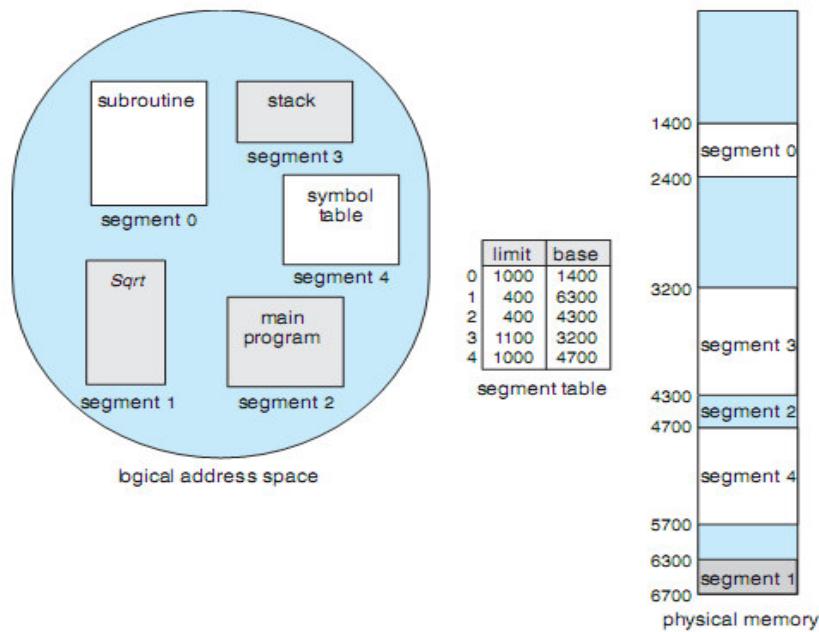


Figure 7.20 Example of segmentation.

Segment table diletakan dimemory, sehingga diperlukan **Segment-table base register (STBR)** yang menunjuk ke alamat dimana segment table disimpan dan **Segment-table length register (STLR)** yang mengindikasikan jumlah segment yang digunakan oleh program.

### Protection

Pada segmentasi protection terhadap lokasi memory (suatu segment) dilakukan dengan menambahkan beberapa bit proteksi pada setiap entry yang terdapat di segment table, diantaranya adalah validation bit 0 yang berarti segment tersebut ilegal untuk diakses, juga ada yang namanya previleged read/write dan execution.

Protection bit dapat digunakan untuk proses segment share seperti pada page sharing.



## CHAPTER 8. VIRTUAL MEMORY

{demand segmentation tidak dibahas disini karena sama dengan demand paging)

### 8.1. Background

Pada chapter 7 kita telah membahas mengenai logical address dan hubungannya dengan phisical address, pada chapter ini kita akan lebih memandang secara khusus bagaimana sebenarnya logical address(logical memory) itu. Pada virtual address kita akan mebahas logical address tampa ada hubungannya dengan phisical address jadi dengan kata lain kita akan melihat logical address lebih eksterem/eksklusif.

Latarbelakang penggunaan virtual memory:

- Untuk dapat dieksekusi sebuah program harus berada di memory,namun terkadang tidak semua bagian program tersebut yang digunakan. Sebagai contoh error code(hanya digunakan saat terjadi error)
- Biasanya array,list dan tables sering dialokasikan banyak namun sebenarnya tidak digunakan/dibutuhkan. Biasanya kita mendefinisikan/memesan array dengan 100 element namun terkadang yang digunakan hanya sampai 10(sehingga banyak mubazir dimemory)
- Beberapa option dan feature dari program terkadang digunakan sangat jarang. Contoh nya microsoft word tidak semuanya toolnya digunakan biasanya hanya beberapa.
- Bahkan jika digunakan bagian-bagian program tidak digunakan dalam waktu bersamaan. Idak lag

**Virtual Memory digunakan untuk memisahkan user logical memory dari phisical memory.** Dengan adanya virtual memory:

- User tidak lagi concern dengan jumlah/ukuran phisical memory yang tersedia

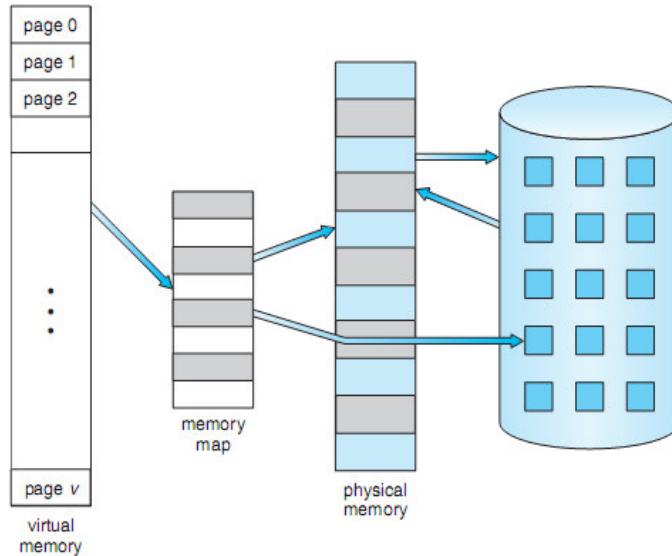


Figure 8.1 Diagram showing virtual memory that is larger than physical memory.



- Hanya bagian program yang dibutuhkan yang akan ada dimemory untuk dieksekusi
- Logical address space dapat lebih banyak dari phisical address space  
Kita dapat memiliki program yang dijalankan berukuran 4 GB walaupun memory phisik kita hanya 512 MB
- Beberapa proses dapat menshare address space
- Membuat proses creation lebih mudah  
Kita tidak harus mengetahui apakah masih ada phisical memory yang tersedia jika kita ingin memcreate sebuah proses.
- Karena setiap program hanya bagian-bagian tertentunya saja yang ada dimemory (menghemat penggunaan memory phisik) maka akan lebih banyak program yang akan dapat berada dimemory dalam suatu waktu. Meningkatkan degree multi programing, CPU utilization dan throughput, walupun tidak meningkatkan response time atau turnaround time.
- Karena penggunaan phisical memory lebih hemat maka akan sangat jarang kita menggunakan I/O (penggunaan I/O lambat) untuk load dan swap program dari phisical memory.

### **Virtual address space (ruang alamat virtual)**

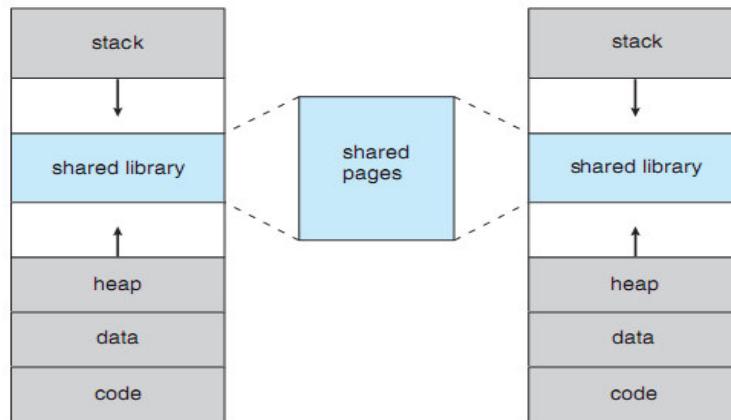
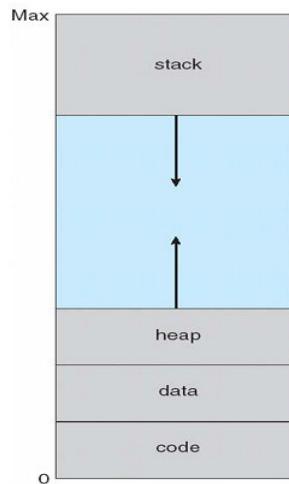


Figure 8.3 Shared library using virtual memory.



Virtual memory diimplementasikan dengan dua cara :

- Demand paging
- Demand Segmentation

### 8.2.Demand Paging

Ada dua pilihan ketika sebuah program akan diload ke memory untuk dieksekusi. Pilihan pertama adalah meload semua program ke memory. Namun pendekatan ini memiliki masalah karena terkadang diawal tidak semua bagian dari program yang kita gunakan. Pilihan kedua adalah meload page dari program hanya ketika page tersebut dibutuhkan (demand) oleh program tersebut (meload sebagian program). Pendekatan inilah yang disebut dengan **demand paging**. Contohnya ketika kita membuka excel tidak semua fungsi-fungsi di excel akan diload ke memory hanya yang penting-penting saja dulu kemudian nanti ketika sebuah fungsi ingin kita gunakan baru fungsi itu diload ke memory. Pendekatan demand paging memiliki efek:

- Sedikitnya penggunaan I/O, jika harus meload semua page program maka akan banyak sekali I/O yang dibutuhkan, dan juga bisa saja kita menggunakan I/O tersebut untuk meload sebuah data tapi tidak digunakan.
- Penggunaan memory fisik yang hemat, dibanding harus meload semua.
- Response yang cepat, pertama karena tidak banyak I/O yang ikut campur kemudian juga hanya page-page yang penting diload sehingga program terasa ringan/response nya cepat
- Karena memory lebih hemat, maka lebih banyak user yang dapat ditampung.

Nah, ketika sebuah page dibutuhkan maka page tersebut akan direferensi, jika terjadi referensi yang invalid (didepan akan dipelajari) maka proses referensi akan diabort, dan jika page yang dibutuhkan tidak ada dimemory maka page tersebut akan diswap dari disk. Untuk proses swap kita kenal yang namanya **lazy swapper**, lazy swapper hanya akan menswap suatu page dari disk ke memory hanya ketika page tersebut benar-benar dibutuhkan oleh program. Swaper dalam konsep paging lebih dikenal dengan sebutan **pager**.

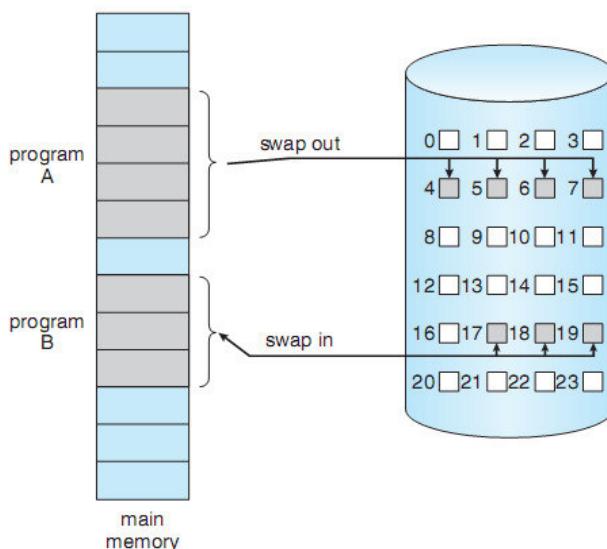
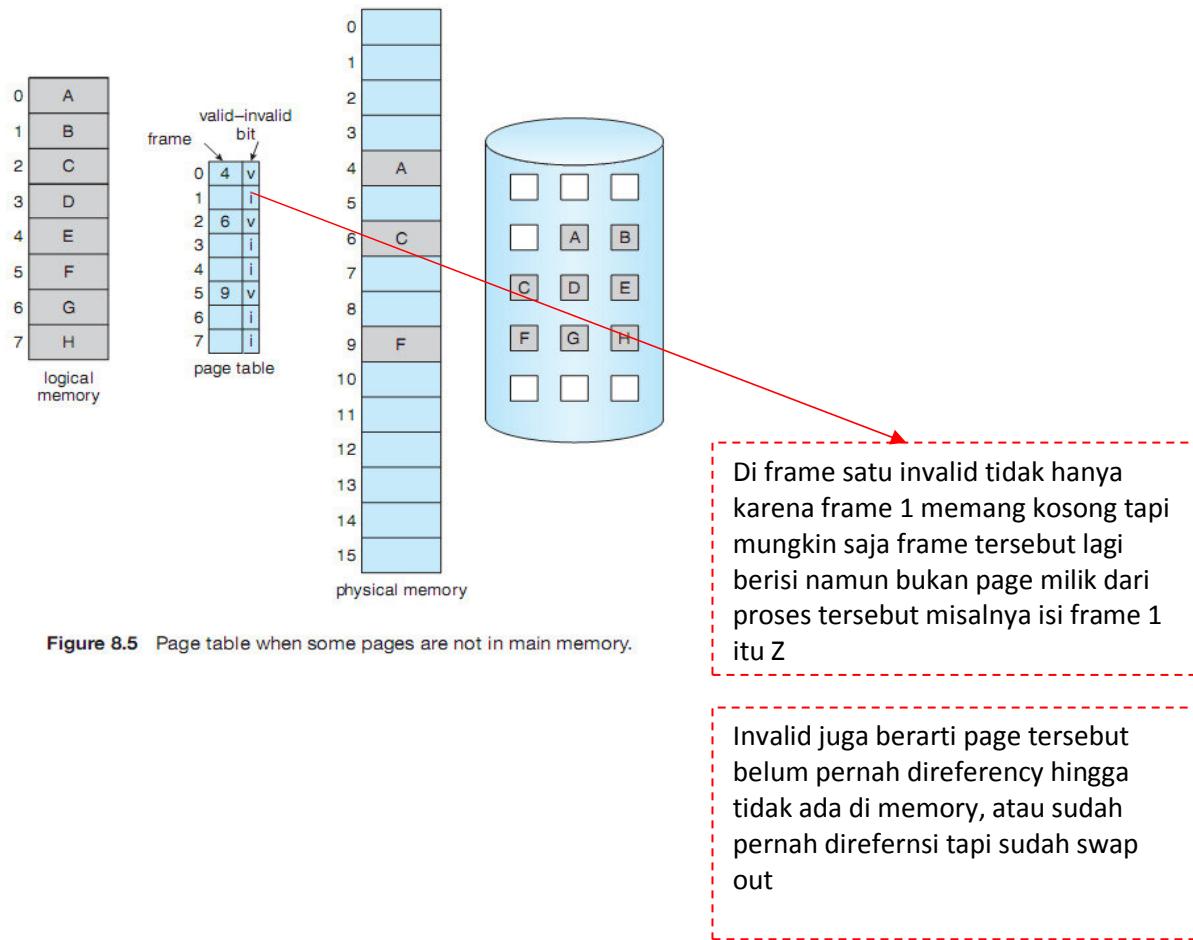


Figure 8.4 Transfer of a paged memory to contiguous disk space.



Proses swap out dan swap in biasanya terjadi di level page bukan pada level program. Biasanya program tidak memiliki semua page nya di memory, hanya sebagian, yang diswap ini dan out itu page nya(bukan programnya). Biasanya page yang diswap out adalah page yang lagi tidak dieksekusi.

Karena adanya swap in dan swap out tersebut maka pada page table ada yang namanya **valid-invalid bit**. Setiap entry diberi bit V(valid) jika page pada entri tersebut berada di memory(**memory resident**), dan I (invalid) jika page tersebut tidak ada di memory atau diframe yang bersesuaian dengan page tersebut ada isinya tapi bukan page milik proses tersebut tapi milik proses lain. Mula-mula semua entri pada page table valid-invalid bit nya diset I (invalid) semua. Jika pada saat translasi untuk page mengacu ke frame ternyata didapat bahwa valid-invalid bitnya, invalid(i) maka akan terjadi **page fault**.



### Page Fault Handling

Jika terjadi page fault maka akan ada intrupsi/trap ke SO bahwa telah terjadi page fault kemudian SO akan:

1. SO akan mengecek ke internal table dari proses(biasanya ada dalam PCB) apakah memang referensi nya valid atau telah terjadi invalid memory access.

Jika terjadi invalid memory access (referensinya invalid) maka proses akan di abort/terminate, namun jika referensinya valid namun hanya karena page tersebut tidak ada dalam memory, masih ada di disk lanjutkan ke langkah 2.

2. SO akan mencari frame yang kosong di memory (asumsikan ada frame yang kosong dulu)
3. Jika frame yang kosong didapat maka page yang berada di disk akan di swap in ke frame tersebut
4. Kemudian SO akan mereset page table dan menset valid-invalid bit pada page yang telah diswap in tadi menjadi valid.
5. Kemudian SO akan merestart instruksi yang menyebabkan page fault.

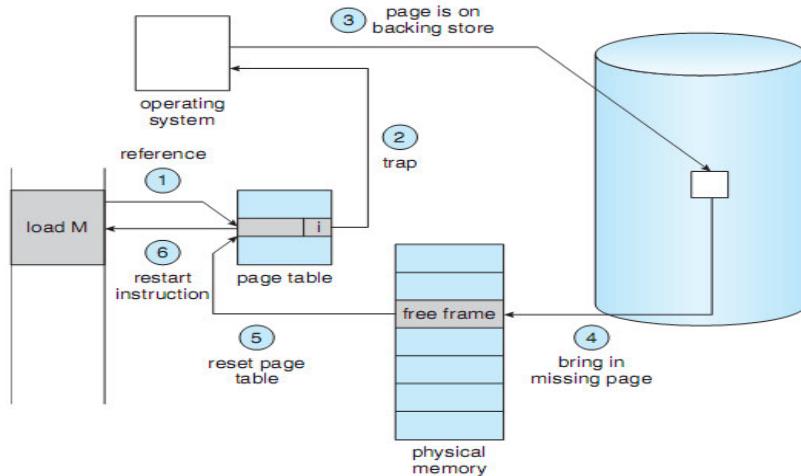


Figure 8.6 Steps in handling a page fault.

### Menghitung performa dari demand paging

Nilai Persentase Page fault terletak pada  $0 \leq p \leq 1$ . Jika  $p = 0$  maka artinya tidak terjadi page fault dan jika nilai  $p=1$  maka terjadi page fault disetiap proses kita mereferensi memory.

$$\text{Effective Access Time (EAT)} = ((1 - p) \times \text{memory access})$$

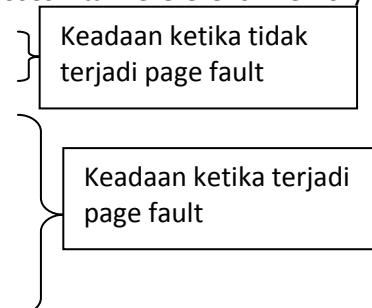
+  $p$  (page fault overhead)

+ swap page out

+ swap page in

+ restart overhead

)



Ket:

**Page fault overhead** = proses mereferensi (mula-mula no 1 pada gambar), interupsi ke SO bahwa telah terjadi page fault, mengecek page pada disk

**Swap page out** = jika tidak ada lagi frame yang kosong di memory, maka SO akan memilih salah satu frame untuk diswap out dengan menggunakan algoritma tertentu



**Swap page in**= melakukan swap in terhadap page yang menyebabkan page fault, ke free frame

**Restart overhead**= mereset page table , mereset intruksi penyebab page fault

Contoh:

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds

$$EAT = ((1 - p) \times 200) + p (8 \text{ milliseconds})$$

$$= ((1 - p) \times 200) + p \times 8,000,000$$

$$= 200 - 200p + 8,000,000p$$

$$= 200 + 7,999,800p$$

$$1 \text{ milisecond} = 10^6 \text{ nanosecond}$$

Kemudian misalkan terjadi 1 page fault pada 1000 kali access maka  $p=1/1000=0.001$

$$EAT = 200 + 7,999,800p$$

$$= 200 + (7,999,800 \times 0.001)$$

$$= 8199.8 \text{ nanosecond} = 8.2 \text{ microsecond}$$

$$1 \text{ nanosecond} = 10^{-9} \text{ second}$$

Dibanding memory access yang tanpa page fault 200 nanosecond atau 0.2 microsecond maka hanya dengan satu page fault akan memperlambat access memory sebanyak 40 kali

### Copy On Write

Salah satu keuntungan dari paging adalah beberapa proses dapat menggunakan page yang sama (share page). Salah satu fasilitas di page share adalah Copy On Write. Copy on Write mengizinkan parent dan child menshare page yang sama dalam memory. Jadi tidak perlu setiap kita membuat proses baru harus selalu mengalokasikan page baru. Jika salah satu proses ingin memodifikasi data pada page tersebut barulah page tersebut dibuatkan copynya (copy on write). Dan yang dibuat copynya hanya page yang dimodifikasi saja.

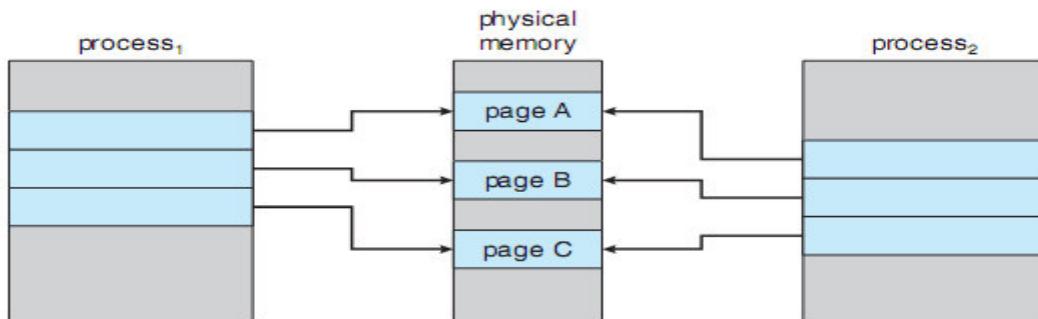


Figure 8.7 Before process 1 modifies page C.



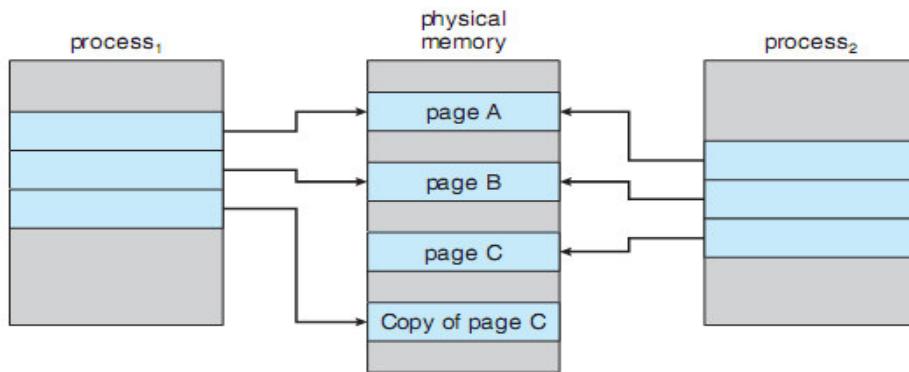


Figure 8.8 After process 1 modifies page C.

### 8.3.Page Replacement

Pada proses handling page fault pada langkah ke 4 SO akan mencari frame yang kosong untuk tempat page yang akan di swap in, nah bagaimana jika frame yang kosong sudah tak ada lagi. Solusinya adalah kita harus mengantikan frame yang page nya sudah tidak digunakan lagi(menterminate,atau swap out, ataupun mereplacenya). Algoritma yang digunakan disebut algoritma page replacement. Algoritma yang kita inginkan adalah algoritma yang dapat meminimalisasi jumlah page fault.

Page replacement mencegah kita melakukan **over-allocation**,maksudnya misalnya kita memiliki 6 proses yang masing-masing memiliki 10 page yang harus diload ke memory namun sebenarnya kita tinggal memiliki 40 frame yang kosong di memory phisik.

Untuk menampung page replacement maka kita mengubah algoritma handling page fault yang telah kita sebut diatas

1. SO akan mengecek ke internal table dari proses(biasanya ada dalam PCB) apakah memang referensi nya valid atau telah terjadi invalid memory access.  
Jika terjadi invalid memory access (referensinya invalid) maka proses akan di abort/terminate, namun jika referensinya valid namun hanya karena page tersebut tidak ada dalam memory,masih ada di disk lanjutkan ke langkah 2.
2. SO akan mencari frame yang kosong di memory
  - Jika ada frame yang kosong gunakan frame tersebut
  - Jika sudah tidak ada lagi frame yang kosong maka panggil algoritma page replacement untuk memilih frame yang akan dikorbankan(victim frame) .Swap out frame tersebut ke disk dan kemudian reset page table untuk frame yang bersesuaian(dirubah ke invalid).
3. Page yang berada didisk di swap in ke frame memory yang telah kosong tersebut
4. Kemudian SO akan mereset page table dan menset valid-invalid bit pada page yang telah diswap in tadi menjadi valid.
5. Kemudian SO akan merestart intruksi yang menyebabkan page fault.

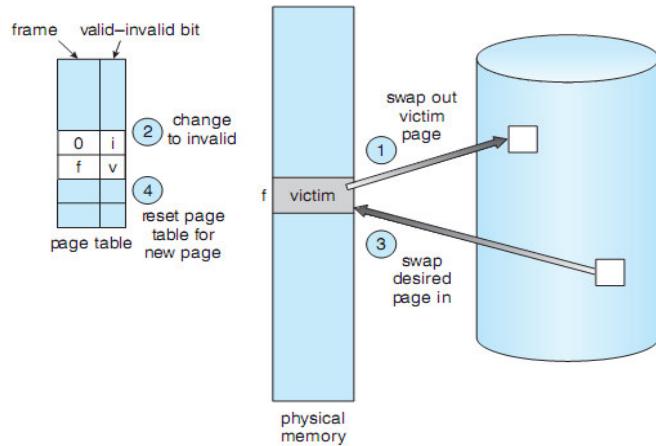


Figure 8.10 Page replacement.

Namun kita lihat terjadi dua kali proses transfer page (swap out dan swap in) hal ini akan lebih memperkecil EAT sehingga memperlambat access memory. Solusi untuk mereduksi permasalahan dua kali transfer tersebut adalah dengan menggunakan **dirty bit**.

Dengan skema dirty bit setiap page atau frame diberikan bit yang menandai apakah page tersebut telah dimodifikasi semenjak dia terakhir kali diswap in ke memory. Jika telah dimodifikasi maka page tersebut ditandai dengan dirty bit (biasanya 1 untuk yang sudah dimodifikasi dan 0 untuk yang belum dimodifikasi). Pada saat page replacement ingin menswap out sebuah frame dari memory maka dia harus memeriksa dirty bitnya jika diset 1 maka frame tersebut harus tetap diswap out ke disk karena page yang didisk harus diupdate perubahannya (kosistensi data). Namun jika frame tersebut dirty bit nya 0 maka tidak perlu diswap out timpai saja frame tersebut dengan frame yang baru dari disk yaitu frame yang menyebabkan page fault. Kenapa ditimpai? Karena tidak ada perubahan, berarti masih sama dengan yang di disk. Disini terlihat kita dapat mengurangi satu proses transfer.

### Algoritma-Algoritma Page Replacement

Sebelum menentukan algoritma page replacement kita harus juga memperhatikan algoritma bagaimana menentukan jumlah frame yang akan dialokasikan ke sebuah proses.

Algoritma page replacement yang diinginkan adalah algoritma yang jumlah page faultnya sedikit. Ada banyak algoritma page replacement, cara mengevaluasi apakah algoritma itu baik adalah dengan menjalankannya pada string dari referensi memory kemudian menghitung berapa banyak page fault yang terjadi.

Biasanya dengan menambahkan frame kesebuah proses dapat mengurangi page fault.

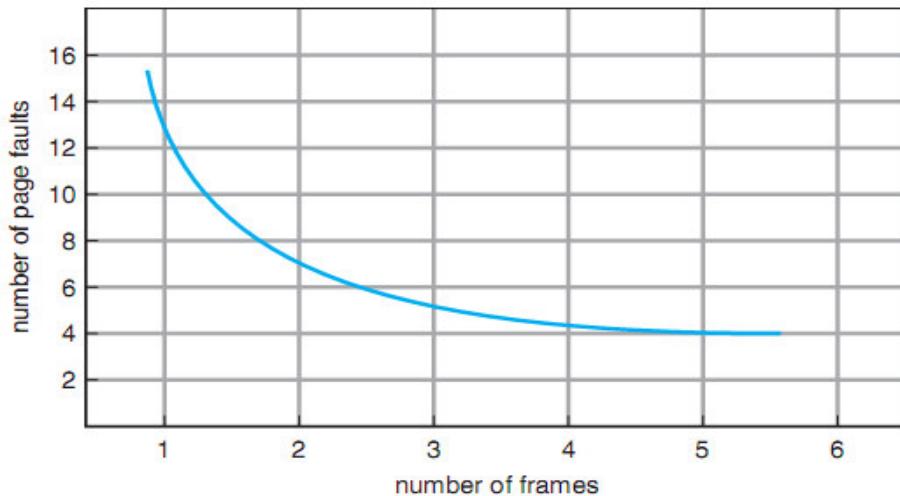


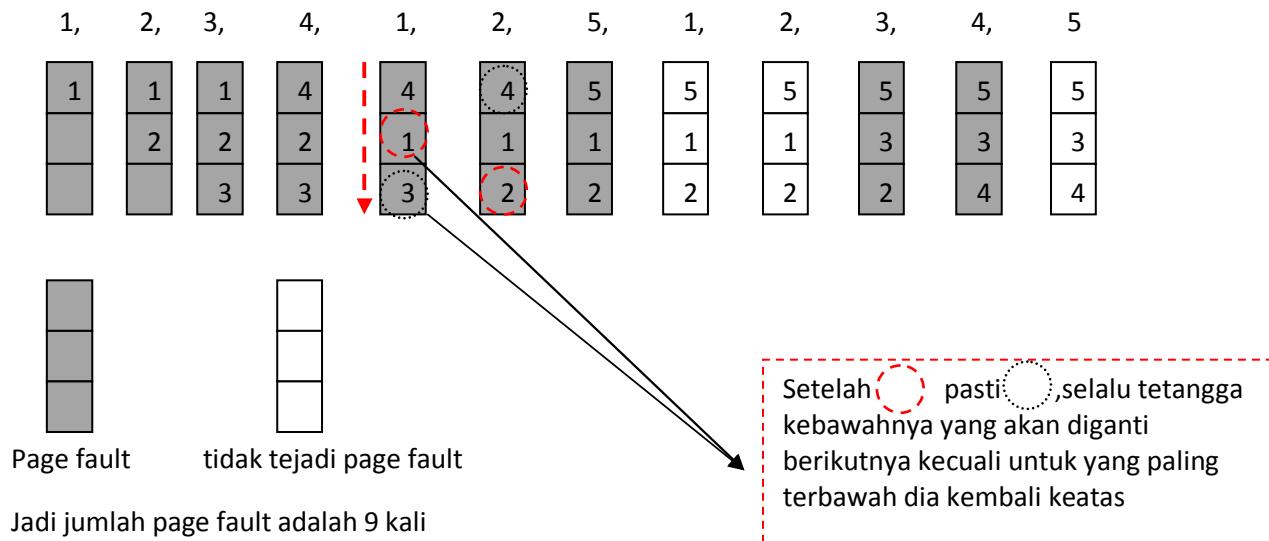
Figure 8.11 Graph of page faults versus number of frames.

### 1. FIFO Page Replacement

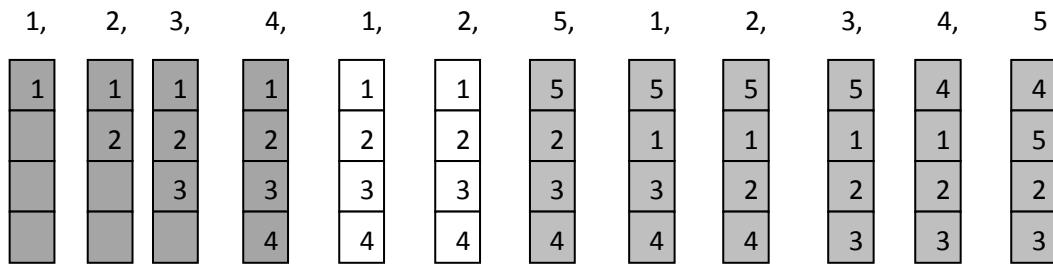
Ide dari algoritma ini adalah mengganti page yang paling lama resident di memory.

Misalkan per proses diberikan 3 frame dan mereka mereferensi alamat memory berikut(string referensi): 1,2,3,4,1,2,5,1,2,3,4,5(perlu diingat referensi ini sebenarnya datangnya satu-satu )

Maka proses page replacement dan jumlah page faultnya adalah

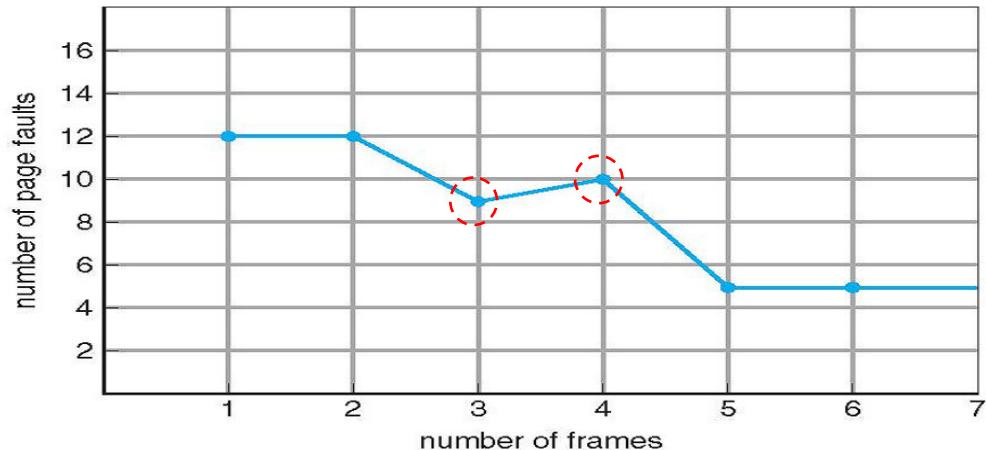


Sebelumnya dikatakan bahwa jika frame ditambah maka akan mengurangi page fault namun menurut penelitian beberapa algoritma (salah satunya adalah FIFO) penambahan frame kadang-kadang menyebabkan jumlah page fault bertambah. Fenomena ini disebut **Belady's Anomaly**. Contoh dengan menggunakan reference string diatas namun frame nya kita tambah menjadi 4 frame.



Jumlah page faultnya menjadi **10 kali**

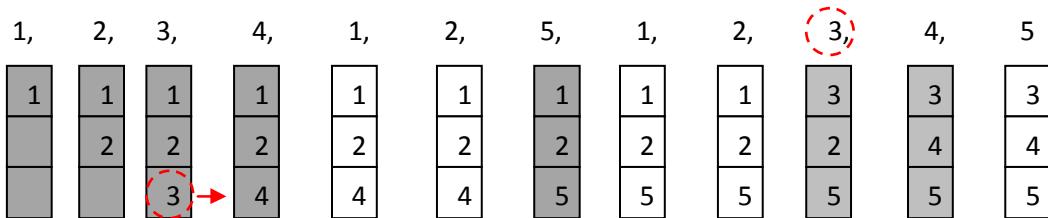
Berikut ini grafik yang menunjukkan Belady's Anomaly di FIFO



## 2. Optimal Page replacement Algorithm

Algoritma ini adalah algoritma yang paling optimal diantara algoritma-algoritma page replacement yang lain. Ide dasarnya adalah mereplace frame yang tidak akan dipakai dalam waktu yang paling lama.

Contoh cara kerja algoritma



Diantara 1,2 dan 3, Yang akan paling lama tidak **akan** digunakan adalah 3,maka 3 yang diganti begitu seterusnya

Jadi hanya akan terjadi **7 kali** page fault dibanding FIFO yang sebanyak 9 kali.

Namun algoritma ini tidak dapat diterapkan dikomputer karena syarat penggunaan algoritma ini adalah SO harus mengetahui semua referency string yang **akan** digunakan oleh proses. SO harus dapat mengetahui masa depan. Hal ini mustahil karena pada saat kita menjalankan program banyak faktor dan kondisi yang cukup kompleks(misalnya percabangan) yang membuat mustahil untuk mengetahui alamat referensi yang mana yang akan direferensi oleh program sampai program ini selesai. Misal contoh diatas



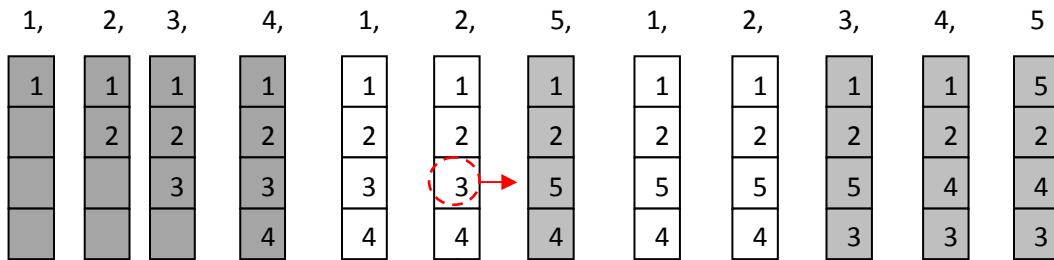
Lalu kenapa contoh diatas kita diberitahu referensi selanjutnya? Karena hal itu hanya untuk menevaluasi kerja algoritma page replacement

3. **Least-recently-used(LRU) algorithm.**

LRU algorithm adalah algoritma yang lebih baik dibandingkan algoritma FIFO namun tidak lebih baik dari optimal algoritma. Idenya adalah mengganti page/frame yang paling lama tidak digunakan(paling lama tidak direferensi lagi). Jadi LRU menggunakan pengetahuan masa lalu dibanding masa depan(optimal algoritma). Biasanya referensi yang baru saja digunakan peluang untuk digunakan lagi berikutnya sangatlah besar(lokalisitas memory) untuk itu sebaiknya jangan diswap out(diganti), sebaiknya yang diswap yang paling lama telah digunakan.

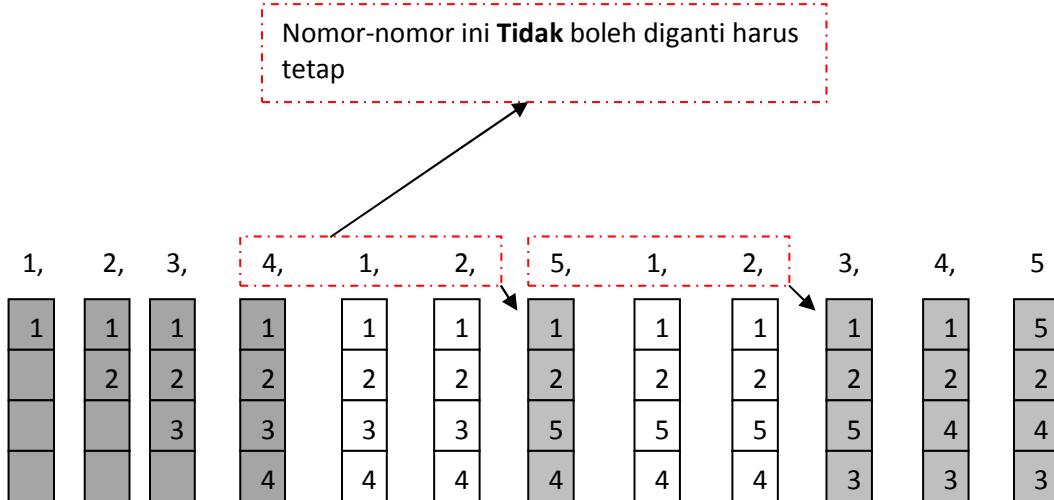


Contoh:



Urutan dari yang paling baru digunakan ke yang paling lama adalah 2,1,4,3 jadi yang harus diganti adalah 3, begitu seterusnya

**Cara cepat:** pada saat ingin mengganti page, lihat urutan 3 angka sebelumnya(namu 3 angka tersebut harus beda-beda), 3 angka berbeda tersebut tidak boleh diganti, ganti page dengan nomor selain dari 3 angka tersebut.

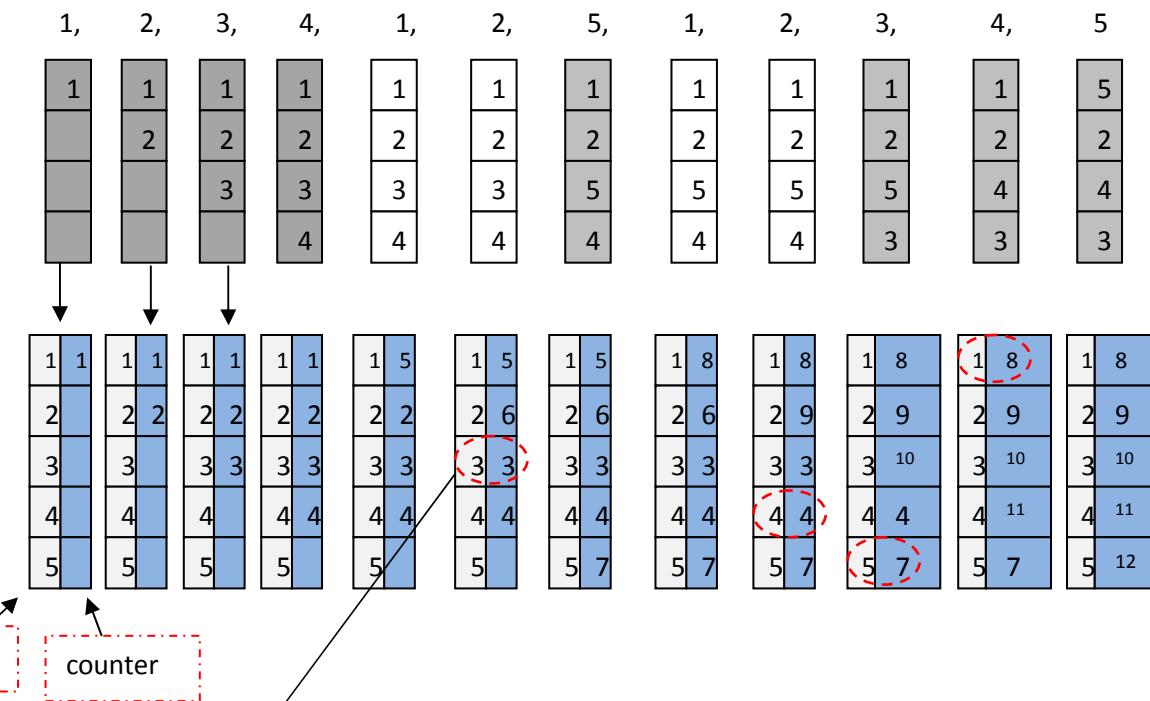


Cara tersebut mudah kalau manusia yang kerjakan karena kita bisa lihat polanya , bagaimana cara jika diterapkan dikomputer? Ada dua cara yang umum digunakan yaitu

- Dengan counter
- Dengan stack

**Teknik dengan Counter** caranya setiap page direferensi maka kita tambahkan clock ke page tersebut, kemudian untuk menyeleksi page mana yang akan diganti kita tinggal mencari clock yang paling kecil diantara page tersebut.

Contoh:



Karena page 3 yang paling kecil counternya  
maka page 3 yang diganti, sebenarnya 5  
dengan counter 0 tapi lima tidak sedang  
berada di memory

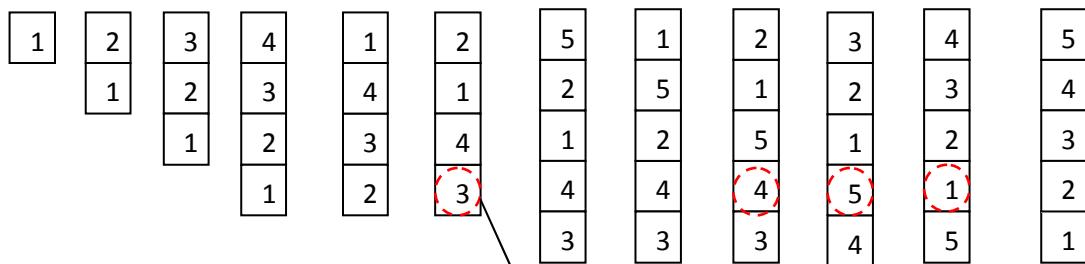


**Teknik dengan Stack**, idenya kita akan selalu meletakan string yang baru direferensi selalu berada pada bagian atas stack, sehingga kita tahu urutan page mana yang recently uses, oleh karena itu page yang paling bawah **diantara page yang ada di memory saat itu** adalah page yang harus digantikan karena paling lama telah digunakan .

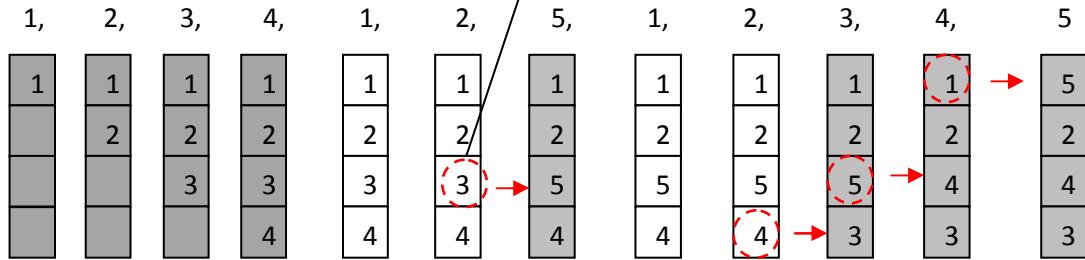
Contoh :

1,    2,    3,    4,    1,    2,    5,    1,    2,    3,    4,    5

Buat dulu stacknya



Karena page 3 yang paling bawah  
maka page 3 yang diganti



Perhatikan stack nya dari atas kebawah adalah urutan list recently uses, yaitu semakin ke bawah semakin lama telah digunakan.



## CHAPTER 8. VIRTUAL MEMORY

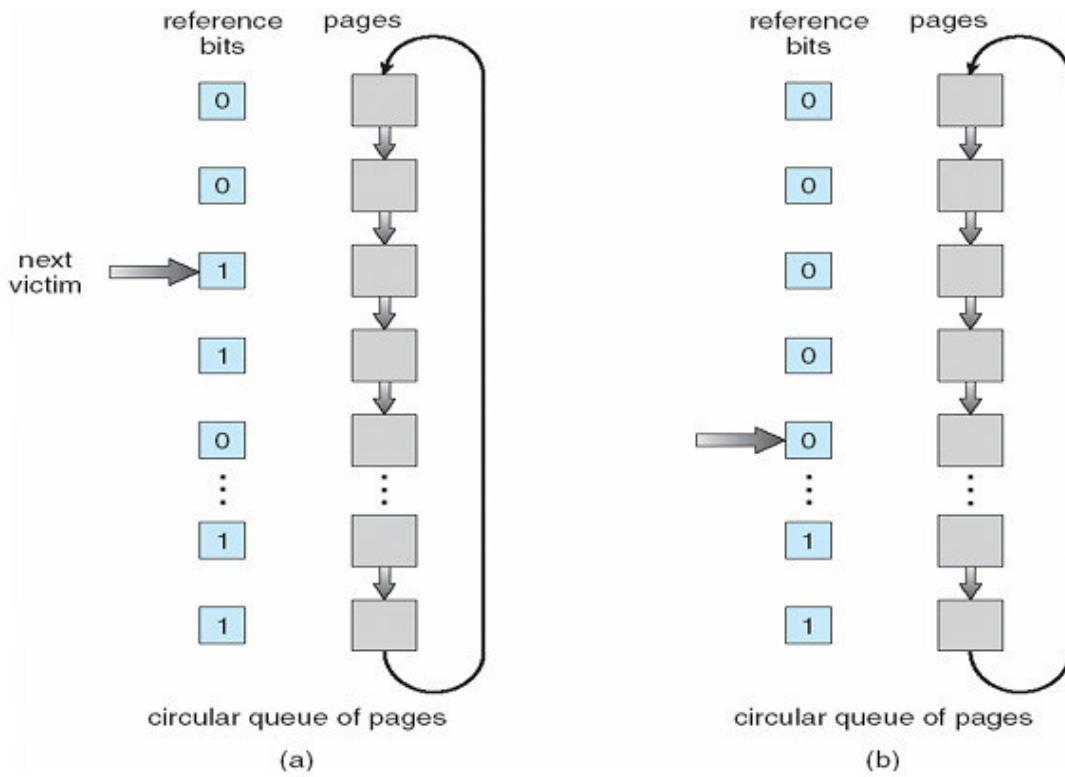
### 4. LRU approximation algoritma

Penggunaan algoritma LRU baiknya diterapkan dengan menggunakan hardware khusus,karena jika harus diterapkan dengan menggunakan memory saja maka hanya akan menambah access time kita ke memory(kita harus mengakses memory beberapa kali dulu untuk mendapatkan apa yang kita inginkan dari memory). Untuk itu ada yang namanya **Referency bit**. Reference bit ini akan diletakan dipage tabel, hal ini tidak masalah karena memang mau tidak mau kita harus mengakses page tabel

Mula-mula semua page diinisialisasi dengan **Referency bit 0**, ketika sebuah page di referensi maka referensi bit di set ke 1

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - ▶ We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - Clock replacement
  - If page to be replaced has
    - ▶ Reference bit = 0 -> replace it
    - ▶ reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules





## 5. Counting algoritma

Algoritma lain yang juga sering digunakan adalah algoritma counting. Ideanya adalah setiap page direferensi count nya akan ditambah satu. Kemudian untuk memilih page mana yang direplace dapat menggunakan dua pilihan berikut:

- **LFU (Least Frequent Uses) Algorithm:** page yang digantikan adalah page dengan count paling kecil. Diharapkan karena page ini jarang dibutuhkan maka dikemudian waktu page ini pun akan jarang dibutuhkan
- **MFU Algorithm:** page yang digantikan adalah page dengan count paling besar. Karena kalau yang count nya kecil mungkin saja baru diswap in ke memori dan belum digunakan.

## 8.6. Alocation Frame

Setiap proses membutuhkan sejumlah minimum frame agar bisa di eksekusi. Sebagai contoh pada IBM 370 untuk suatu intruksi SS moves dibutuhkan minimum 6 page:

2 page untuk menampung intruksi, karena bisa saja intruksi itu harus dibagi dua (misalnya komputer tersebut 32 bit jadi harus dibawah dua kali intruksinya), 2 page lagi untuk menghandle from, dan 2 page lagi untuk menghandle to.

Ada dua teknik alokasi frame yaitu;

- Fixed allocation
- Priority allocation

### 1. Fixed allocation

Mekanisme ini terdiri dari beberapa bentuk pula

#### - Equal allocation

Kita membagikan frame dengan jumlah yang sama ke pada semua proses(equal). Misalnya memory kita terdapat 100 frame(setelah SO mendapat bagian frame), maka jika ada 5 proses maka setiap proses akan mendapatkan 20 frame. Jika sebuah proses kebutuhan frame nya bertambah berarti proses tersebut harus banyak melakukan algoritma replacement.

Kelemahan system ini adalah misalkan sebuah system ukuran 1 frame adalah 1 KB, terdapat sebuah proses A dengan ukuran 10 KB dan satu proses B 127 KB, dan total frame yang tersedia adalah 62 frame. Maka tidak masuk akal untuk memberikan proses A 31 frame karena sebenarnya dia hanya membutuhkan tidak lebih dari 10 frame (21 frame siasanya akan sia-sia)

#### - Proportional allocation

Pada teknik ini alokasi frame diadaskan pada size dari proses(walaupun mustahil sebenarnya dapat memastikan ukuran proses, karena suatu proses dapat tumbuh membesar atau mengecil pada saat dieksekusi)

$$s_i = \text{ukuran proses } p_i$$

$$S = \sum s_i$$

$m = \text{jumlah frame yang tersedia}$

$$a_i = \text{jumlah alokasi frame untuk proses } p_i = \frac{s_i}{S} \times m$$

**contoh**  $m=64$ ,  $s_1=10$ ,  $s_2=127$  maka

$$a_1 = \frac{10}{137} \times 64 \approx 5 \text{ dan } a_2 = \frac{127}{137} \times 64 \approx 59$$

### 2. Priority allocation

Sebenarnya hampir sama dengan propotional allocation namun disini kriterianya bukan size tapi priority.

### 8.6. Global versus local alocation

Sebelumnya kita telah membahas page replacement algoritma, pada replacement algoritma intinya kita menswap in dan swap out frame. Frame dapat berasal dari proses lain yang disebut **global alocation** atau hanya dapat berasal dari proses itu sendiri(**local alocation**). Global alocation meningkatkan throughput(sehingga paling umum digunakan) namun proses tidak dapat mengontrol rate dari page faultnya sendiri karena page fault nya juga dipengaruhi oleh page dari proses lain. Selain itu juga waktu eksekusi dari satu waktu ke waktu yang lain dapat berbeda-beda karena frame dapat diambil dari proses



lain yang berbeda-beda. Pada local allocation kelebihanya performance proses lebih konsisten tidak berubah-ubah, namun disatu sisi utilitas memory akan berkurang karena misalnya ada frame yang tidak sedang digunakan oleh proses yang memiliki frame tersebut seharusnya bisa digunakan oleh proses lain tapi tidak bisa digunakan karena local allocation

### **8.7 Thrashing.**

Thrashing terjadi karena proses tidak memiliki cukup page, sehingga rate page fault sangat tinggi dan proses hanya akan sibuk menswap in dan swap out..

Ketika sebuah proses membutuhkan page maka dia akan memunculkan page fault untuk mendapatkan page yang dibutuhkannya. Karena alokasi frame yang dia miliki sudah penuh semua maka dia memanggil page replacement untuk menggantikan framenya yang berada dimemory dengan frame dari page yang diinginkannya. Namun ketika proses tersebut mendapatkan page tersebut ternyata page tersebut tidak dapat diproses karena membutuhkan page yang lain lagi yang ternyata tidak ada lagi di memory jadi proses tersebut page fault lagi.Begitu terus menerus, proses tersebut akan hanya sibuk meswap in dan swap out, dimana aktivitas ini tidak membutuhkan CPU, utilitas CPU yang kurang akan dideteksi oleh SO, SO berpikir bahwa CPU tidak memiliki proses yang harus dieksekusi jadi untuk kembali meningkatkan degree multiprograming SO meload proses baru lagi ke Memory. Otomatis alokasi-alokasi frame proses lain akan dikurangi, sehingga thrashing menjadi lebih parah lagi. Begitu terus menerus.

### **Demand Paging dan Thrashing**

Mengapa demand paging dapat bekerja ?(menghasilkan sesuatu yang baik di komputer)

Karena kalau diperhatikan Program-program yang kita miliki memiliki model yang lokal. Misalnya kita sedang mengeksekusi sebuah fungsi maka kebutuhan memory yang dimiliki oleh fungsi tersebut(letak intruksi dan data pada fungsi)letaknya berdekatan. Jadi pada saat proses-proses bekerja pada fungsi ini page-page yang dibutuhkannya letaknya tidak berjauhan.Sehingga dapat terlihat bahwa area penggunaan page oleh proses-proses tersebut membentuk sesuatu yang lokal terkadang dapat overlap.Ada pola yang dibentuk oleh sebuah proses ketika dieksekusi , letak page di memory yang mereka gunakan saling berdekatan.

Kapan thrashing terjadi? Ketika page fault terlalu banyak.Mengapa terjadi page fault yang banyak karena jumlah keseluruhan locality dari seluruh proses lebih besar dari pada ukuran memory. Jika sebuah proses terlalu sering berpindah-pindah locality maka semakin sering pula ia mengalami page fault. Semakin sering berpindah lokality(lokality makin besar) semakin sering dia harus melakukan page replacement karena page yang diberikan kepadanya terbatas dan tidak berubah-ubah.

Bagaimana mencegah thrashing? Salah satunya adalah menjaga ukuran dari lokality tiap proses. Misalnya kita memberikan nilai batas bagi jumlah lokality sebuah proses(batasnya biasanya jumlah frame yang tersedia). Jika melebihi batas tersebut maka proses tersebut harus di suspend terlebih dahulu nanti di restart lagi.

Ada dua metode yang digunakan adalah **working set model** dan **page fault frequency**



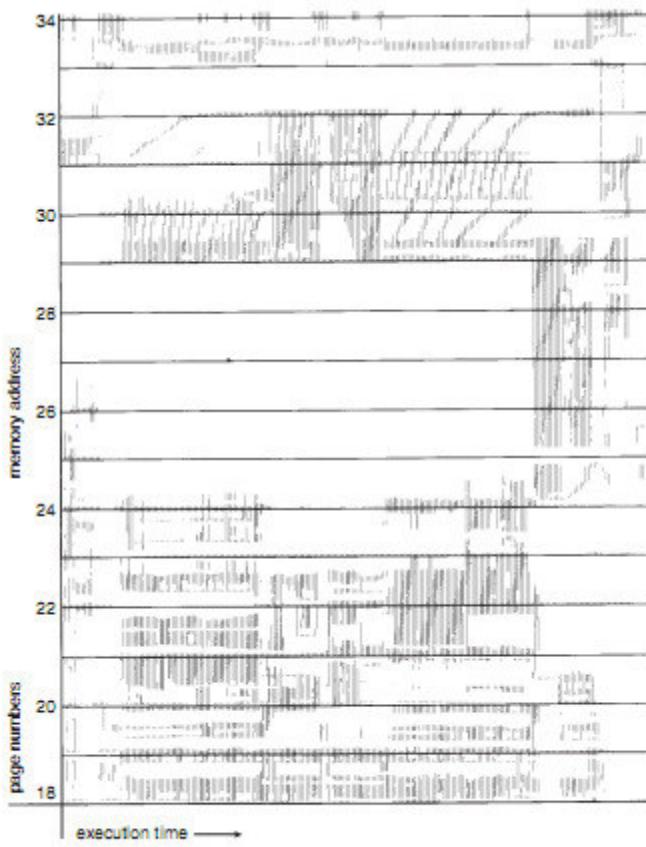


Figure 8.19 Locality in a memory-reference pattern.

### 1. Working Set Model

Parameter working set model yaitu

$\Delta$ =working-set window(range pengamatan kita setiap berapa kali page reference)

Misalnya: 10.000 intruksi(10.000 page reference)

Sebenarnya saat kita mengeksekusi program di komputer hal yang kita lakukan adalah memindah-mindahkan sebuah data atau intruksi dari suatu page ke page lain atau jika intruksi aritmatika dari page –register-page lagi.

$WSS_i$ (workin set dari proses ke i)= jumlah page yang direferensi pada range pengamatan  $\Delta$ (bisa berbeda-beda dari waktu ke waktu). Sering juga dianggap sebagai jumlah frame yang dibutuhkan oleh proses i.

Contoh: dengan  $\Delta = 10$

page reference table

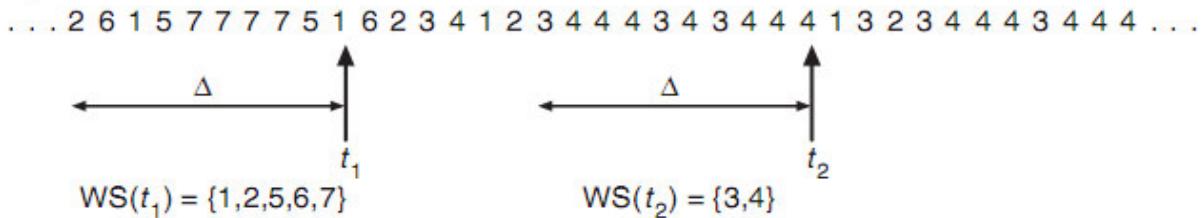


Figure 8.20 Working-set model.

Working set adalah approximasi dari lokality sebuah program. Keakurasiannya dipengaruhi oleh pemilihan  $\Delta$

- Jika  $\Delta$  terlalu kecil maka SO tidak akan dapat mengamati satu lokality
- Jika  $\Delta$  terlalu besar SO mungkin akan mengamati beberapa lokality
- Jika  $\Delta$  tak terhingga itu artinya SO akan mengamati working set sepanjang program tersebut berjalan.

$D = \sum WSS_i$  = jumlah keseluruhan  $WSS_i$  dari semua proses yang sedang dieksekusi. Atau dengan kata lain adalah total frame yang dibutuhkan oleh proses-proses tersebut.

Jika  $D > m$ ,  $m$  adalah jumlah keseluruhan frame yang tersedia, maka akan terjadi **thrashing** karena ada beberapa proses yang kekurangan frame.

#### Cara kerja SO dan working set:

Setelah  $\Delta$  diset maka SO akan memonitor working set dari setiap proses dan mengalokasikan ke proses tersebut frame yang cukup sesuai dengan ukuran working set nya. Jika setelah semua proses yang dieksekusi frame nya cukup, dan ternyata masih ada frame yang nganggur maka SO akan meload proses baru. Jika suatu waktu working set dari sebuah proses meningkat dan mengakibatkan  $D$  lebih besar dari jumlah frame yang tersedia saat itu maka SO akan memilih sebuah proses dan mensuspend proses tersebut dengan men swap out page nya dan bekas frame nya digunakan untuk proses lain. Proses yang di suspend tersebut akan direstart lagi nanti.

Hal yang sulit dengan model working set adalah menjaga track dari working set window. Karena working set window berjalan.



We can approximate the working-set model with a fixed-interval timer interrupt and a reference bit. For example, assume that  $\Delta$  equals 10,000 references and that we can cause a timer interrupt every 5,000 references. When we get a timer interrupt, we copy and clear the reference-bit values for each page. Thus, if a page fault occurs, we can examine the current reference bit and two in-memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used, at least one of these bits will be on. If it has not been used, these bits will be off. Those pages with at least one bit on will be considered to be in the working set. Note that this arrangement is not entirely accurate, because we cannot tell where, within an interval of 5,000, a reference occurred. We can reduce the uncertainty by increasing the number of history bits and the frequency of interrupts (for example, 10 bits and interrupts every 1,000 references). However, the cost to service these more frequent interrupts will be correspondingly higher.

## 2. Page Fault Frequency

Thrashing adalah tingginya rate dari page fault. Sehingga kita ingin mengontrol rate dari page fault tersebut. Caranya adalah dengan menggunakan **Page Fault Frequency**. Teknik ini mengeset batas atas dan batas bawah rate page fault. Jadi SO akan memonitor rate page fault setiap proses. Jika sebuah proses rate page fault nya melebihi batas atas maka proses tersebut akan diberi tambahan frame, namun jika rate page faultnya berada dibawah batas bawah maka sebuah frame dari proses tersebut akan diambil.

Jika pada suatu saat semua proses rate faultnya melebihi batas atas dan sudah tidak ada lagi frame yang bebas maka SO akan memilih proses yang akan disuspend. Setelah proses tersebut disuspend bekas frame nya akan diberikan ke proses lain.

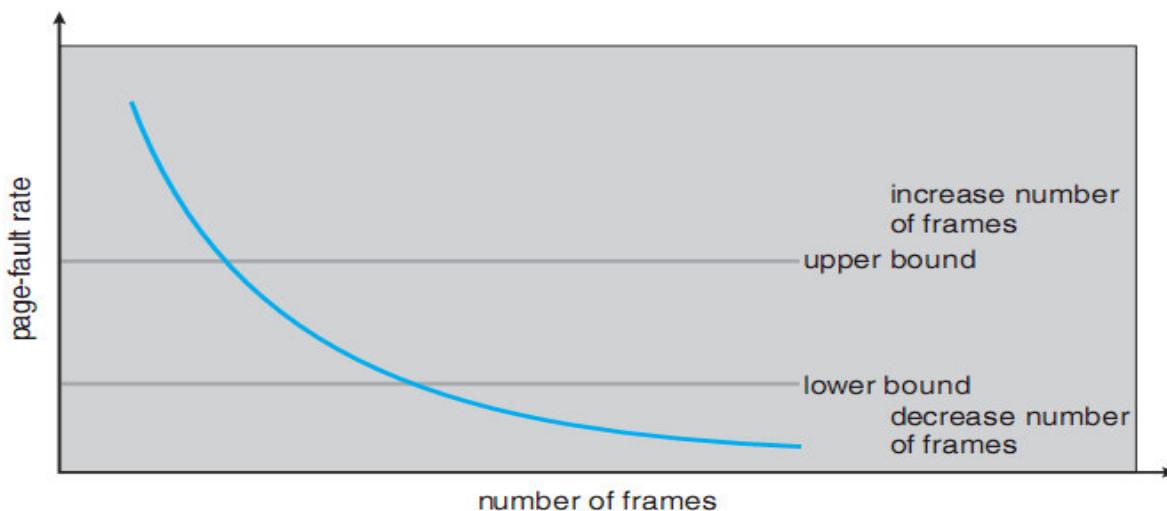


Figure 8.21 Page-fault frequency.



### 8.8. Memory Mapped File

Memory mapped file adalah teknik yang membuat file I/O dapat dipetakan ke Memory sehingga pengaksesanya seperti kita mengakses memory, zaman dahulu ada yang namanya RAM disk. Jadi kita tidak perlu lagi memanggil sistem call(seperti open(), write(), read() dll) untuk mengakses sebuah file di disk kita hanya perlu mengakses file tersebut seperti halnya mengakses page di memory. Jika terjadi modifikasi ke file maka setiap kali file di close file akan kembali ditulis ke disk. Untuk mengetahui apakah file tersebut telah dimodifikasi atau belum maka dapat digunakan **dirty bit**.

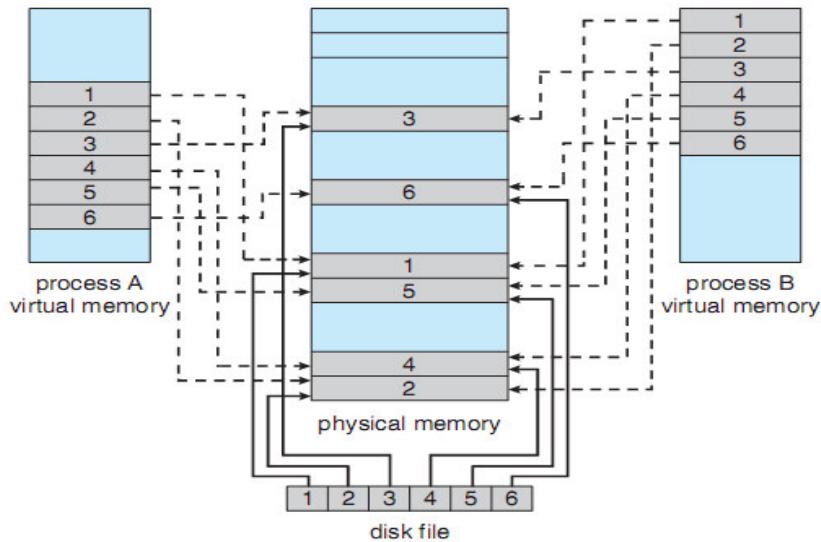


Figure 8.23 Memory-mapped files.

### 8.8. Allocating Kernel Memory

Alokasi memory untuk kernel dilakukan berbeda dengan yang untuk user. Biasanya alokasi frame untuk user di ambilkan dari list frame yang masih bebas yang dikelola oleh kernel. Sedangkan kernel alokasi frame nya diambil dari free memory pool(tidak dalam bentuk frame-frame). Mengapa?

- Kernel mereques memori untuk struktur data(object di SO operasi berbasis OOP) yang ukuranya berbeda-beda, yang biasanya lebih kecil dari ukuran page. Sehingga kernel harus diberikan kebebasan dalam mengalokasikan kebutuhan memorinya dan terbebas dari permasalahan fragmentasi.
- Beberapa kernel harus dialokasikan secara contiguous, misalnya I/O device. Karena beberapa hardware menakses memory tidak dengan virtual memory tapi langsung ke phisical address. Sehingga kernel tersebut memerlukan lokasi yang contiguous di phisical memory(tidak berpisah-pisah dimana-mana)

Ada dua teknik yang sering digunakan untuk mengalokasikan memory ke kernel yaitu **buddy sistem** dan **slab allocation**.

## 1. Buddy Sistem

Mekanisme ini mengalokasikan memory ke kernel dalam bentuk segment yang berukuran tetap dimana didalamnya terdapat page phisical memory yang contiguous. Alokasi nya dilakukan oleh **power of 2 alocator**. Setiap permintaan akan di penuhi ke segment dengan ukuran kelipatan dua. Jika permintaan kurang dari kelipatan dua maka permintaan tersebut akan dipenuhi oleh segment dengan size kelipatan dua diatas size permintaan tersebut

Contoh:

Sebuah PCB meminta alokasi memory sebanyak 25 KB, dan mula-mula segment yang tersedia berukuran 256 KB maka segment tersebut akan dibagi menjadi dua buddy sehingga masing masing buddy(kita sebut saja A<sub>L</sub> dan A<sub>R</sub>) berukuran 128 KB. Begitu seterusnya hingga ukuran buddy menjadi cukup untuk memenuhi ukuran PCB tersebut(32 KB).

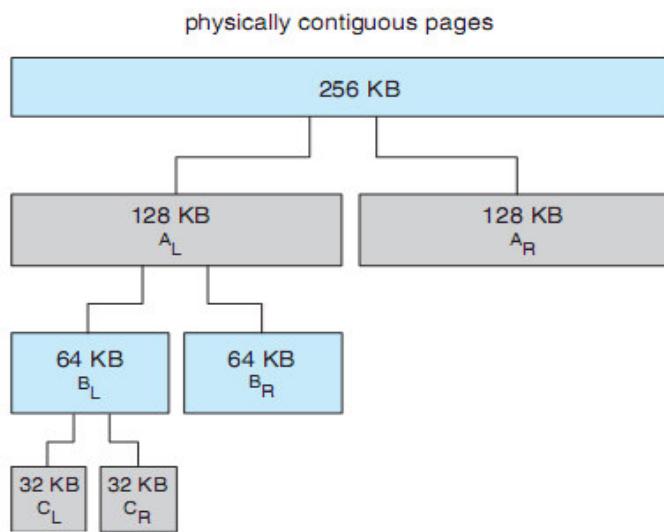


Figure 8.27 Buddy system allocation.

Kelebihan dari sistem ini dua buah buddy yang saling bertetangga dapat dengan mudah digabungkan menjadi segment yang lebih besar dengan metode **coalescing**. Misalnya jika Buddy Cl pada gambar diatas dilepas oleh kernel maka kita dapat menggabungkan nya dengan Cr menjadi buddy yang lebih besar(64 KB) begitu seterusnya.

Kelemahan dari sistem ini adalah adanya fragmentasi. Sehingga biasanya kita memakai mekanisme lain yaitu slab alocator.

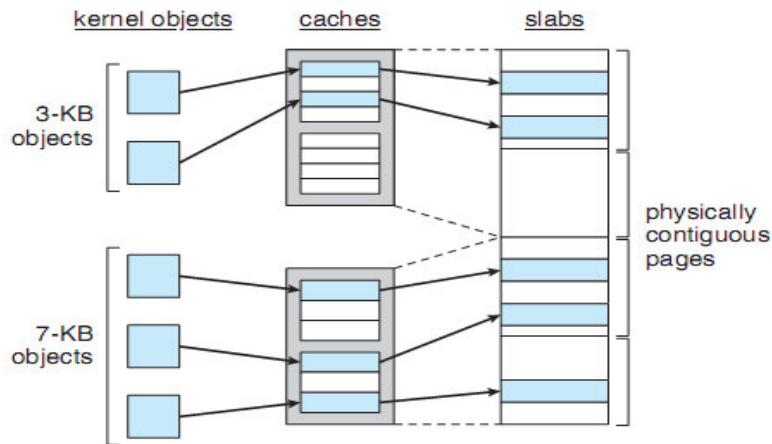
## 2. Slab alocator

Slab adalah satu atau lebih page contiguous pada phisical memory. Cache terdiri dari satu atau lebih slab. Satu buah cache akan dikhusukan untuk menampung satu jenis data struktur dari kernel. Setiap cache terdiri dari object –object yang akan menampung data struktur kernel. Sebagai contoh cache ada yang khusus menampung semaphore akan diisi dengan object semaphore, cache yang khusus menyimpan proses deskriptor akan diisi dengan object proses deskriptor dll. Pada saat cache di buat, semua object akan ditandai dengan **free**. Pada saat struktur data dari kernel disikan maka object ditandai dengan **used**. Jika sebuah slab penuh



maka object akan dialokasikan ke slab lain yang kosong namun masih dalam satu cache. Jika sudah tidak ada slab kosong maka akan dibuat slab baru.

Kelebihannya adalah tidak ada fragmentasi karena setiap data struktur sudah disiapkan cache khusus untuknya dimana ukurannya sesuai dengan ukuran object dan juga metode ini memiliki fast memory request



**Figure 8.28 Slab allocation.**

## 8.9. Isu-Isu Pada Virtual Memory

### 1. Prepaging

Biasanya untuk mereduksi jumlah dari page fault yang muncul pada saat start up sebuah program, SO akan meload page-page dari program yang akan dibutuhkan(yang penting) ke memory sebelum page tersebut direferensi. Namun tidak ada algoritma yang deterministik untuk melakukan ini yang ada kita hanya bisa melakukan probabilistik saja. Kekurangannya adalah jika ternyata page yang kita prepaged itu tidak digunakan maka usaha kita untuk meload page tersebut ke memori akan sia-sia.

Asumsikan  $s$  adalah page yang diprepaged dan  $\alpha$  adalah page yang digunakan dimana  $0 \leq \alpha \leq 1$ . Sehingga kita dapat menghitung manakah cost yang lebih kecil diantara kita melakukan prepaged dan berhasil(banyak page yang digunakan) dengan  $s * \alpha$  atau melayani operasi page fault(karena page yang kita prepaged tidak digunakan otomatis nanti  $n=banyak$  page fault) dengan  $s * (\alpha - 1)$ . Jika  $\alpha$  mendekati 0 maka prepaged kalah dan jika  $\alpha$  mendekati 1 maka prepaged menang.

### 2. Page size

Ada beberapa hal yang harus diperhatikan ketika menentukan page size:

- Fragmentation, semakin besar page size maka semakin besar pula fragmentasi internal yang akan dihasilkan.
- Page table size, semakin besar page semakin kecil page size nya dan sebaliknya
- Resolusi, semakin kecil page size maka semakin baik resolusi karena program akan menyalokasikan proses yang penting saja. Sebaliknya kalau makin besar page size maka semua proses pada program akan ditransfer walaupun mungkin tidak digunakan.

- I/O overhead, semakin kecil page size semakin tinggi lokality sehingga total I/O yang akan dipanggil berkuarang.
- Number of page fault
- Locality
- TLB size

### 3. TLB reach

**TLB Reach** adalah jumlah page pada memory yang dapat diakses dari TLB

$$\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$$

Salah satu cara sederhana meningkatkan TLB reach adalah dengan meningkatkan page size namun seperti telah dibahas sebelumnya bahwa page size yang besar menghasilkan fragmentasi yang besar pula.

### 4. Program struktur

Misalkan:

**Int [128, 128] data;**

**Each row is stored in one page**

Program 1

```
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i, j] = 0;
```

**128 x 128 = 16,384 page faults**

Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i, j] = 0;
```

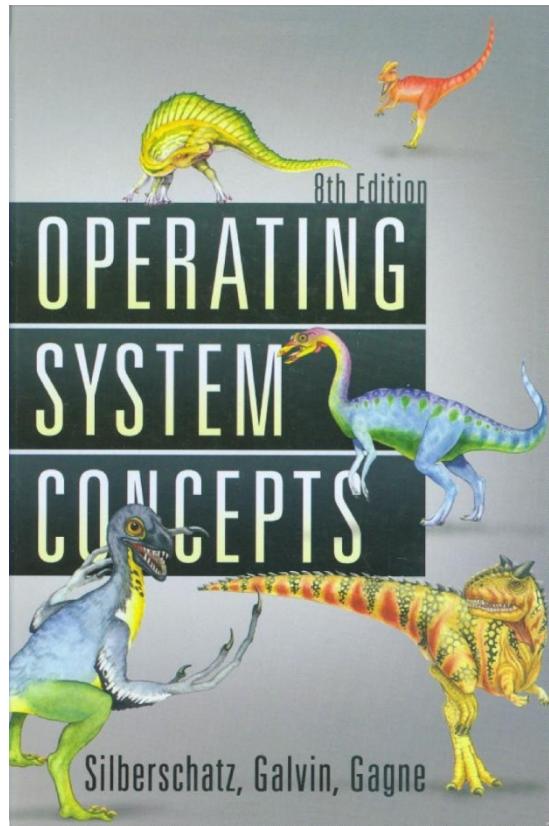
**128 page faults**

### 5. I/O interlock

Page yang digunakan untuk mencopy file dari device misalnya dengan mekanisme DMA harus dilock sehingga tidak dipilih oleh algoritma page replacement. Karena jika sementara mencopy lalu page nya di ganti maka akan terjadi error.



# CATATAN KULIAH SEMESTER 2



LA SURIMI

10/310845/PPA/03471

ILMU KOMPUTER FMIPA  
UNIVERSITAS GADJAH MADA  
YOGYAKARTA

2011