Module I – Introduction to Operating Systems

Prof. Ismael H F Santos

Considerações Gerais

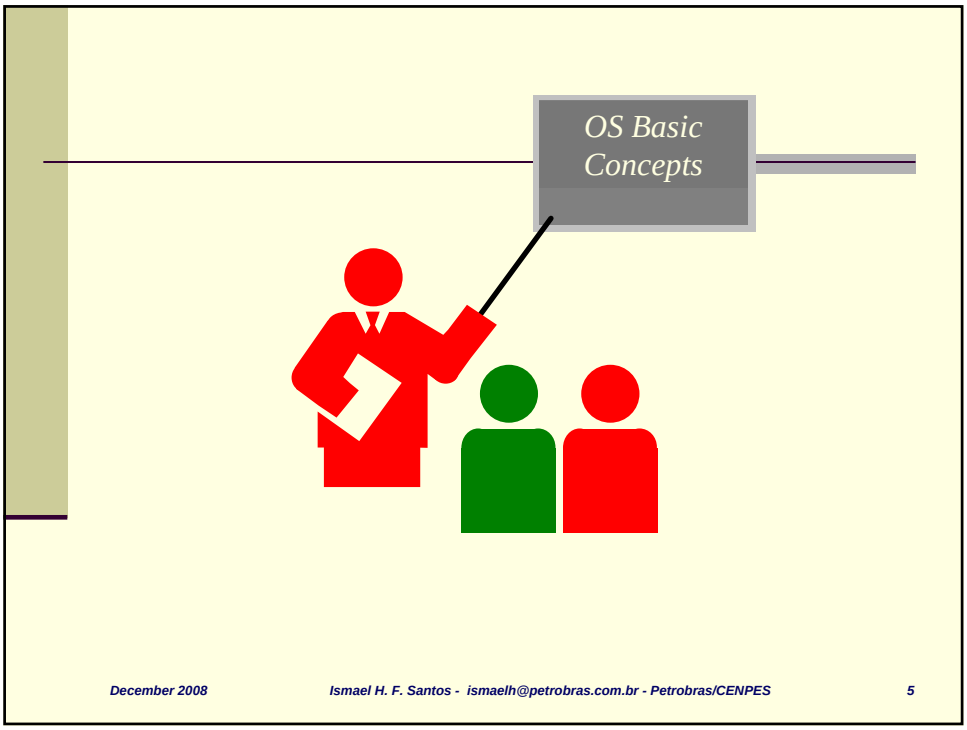
- **Objetivo:** *Discutir os principais conceitos e os princípios básicos da Orientação a Objetos usando a linguagem C++.*
- **A quem se destina :** *Alunos e Profissionais que desejem aprofundar seus conhecimentos sobre Linguagem C++ e suas aplicações*

Bibliografia

- *Sistemas Operacionais*
  - *Santos,F., H., Ismael; Notas de Aula, 2005*
- *Introdução à Arquitetura de Sistemas Operacionais*
  - *Francis B Machado e Luis Paulo Maia , LTC.*
- *Sistemas Operacionais*
  - *Toscani e outros, Serie UFRGS, no 11, Editora sagra-luzzatto*
- *Fundamentos de Sistemas Operacionais*
  - *Silberschatz, Abraham, Galvin, Peter, Gagne, G., LTC*
- *Sistemas Operacionais Modernos*
  - *Andrew S. Tanenbaun; Prentice Hall*
- *Operating System Concepts: Internals and Design Principles*
  - *William Stallings, Prentice Hall 1998 3ª Ed.*

Agenda

- *Operating System Basic Concepts*
- *Multitask Operating Systems*
- *Interruption and Exception*
- *SO Structure*
- *System Call Mechanism*
- *Kernel Structure*
- *Kernel Subsystems*
- *Process*
  - *Subprocess and Threads*
  - *Threads*
  - *Implementation of Threads*
- *Schedulling*
- *Memory Management*
- *Swapping*
- *Virtual Memory*
  - *Paging*
  - *Segmentation*

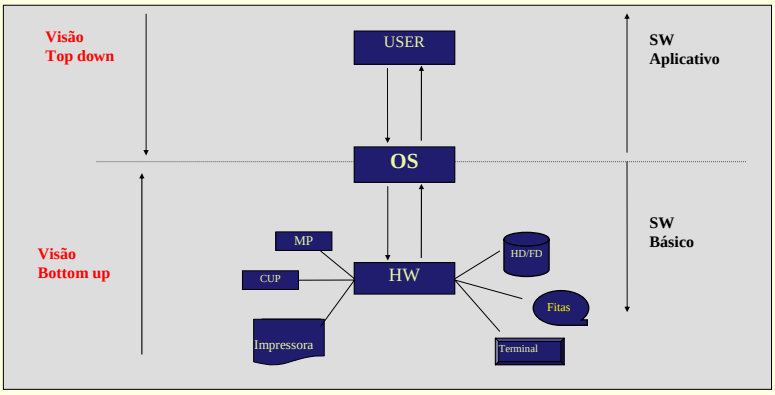


What is the OS ?

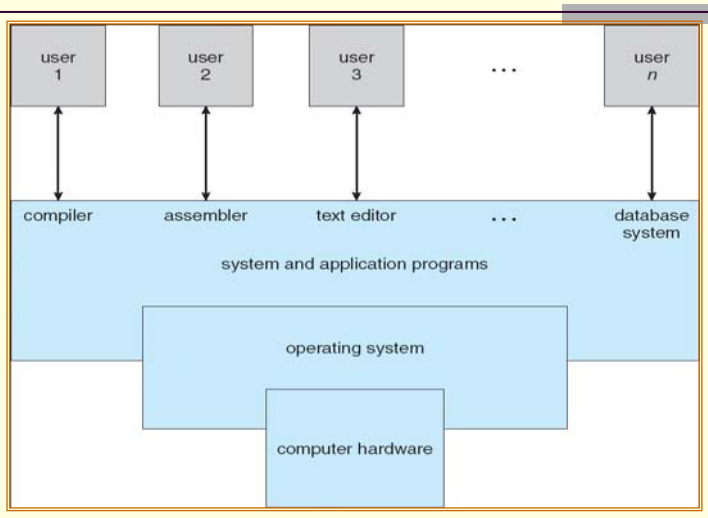
- A program that acts as an intermediary between a user of a computer and the computer hardware.
- Operating system goals:
  - Execute user programs and make solving user problems easier.
  - Make the computer system convenient to use.
- Use the computer hardware in an efficient manner

OS View

- **Top down:** ***Virtual Machine***
- **Bottom up:** ***Resource Manager***

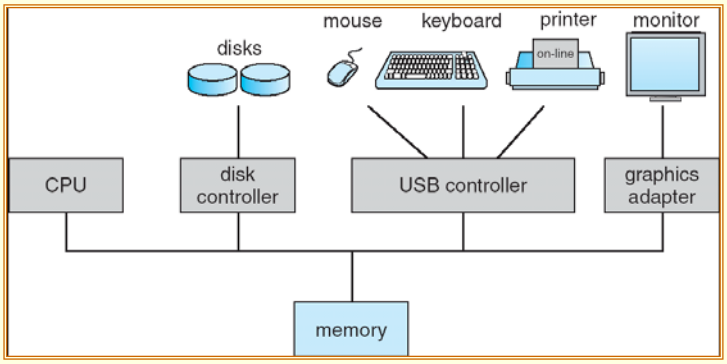


A Computer System basic structure



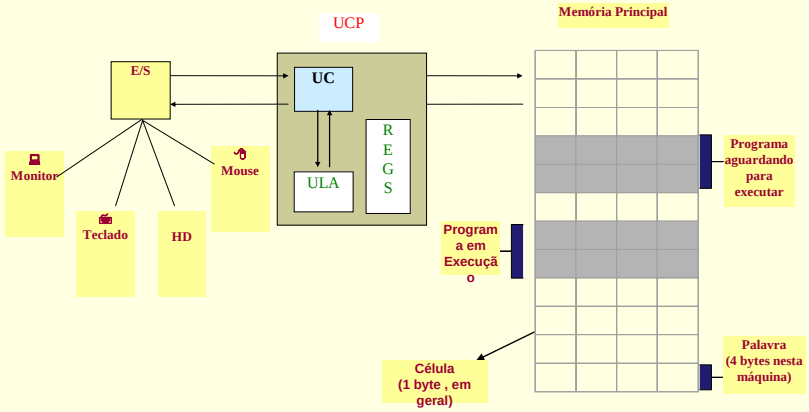
A Computer System basic structure

- **Computer-system operation**
  - One or more CPUs, device controllers connect through common bus providing access to shared memory

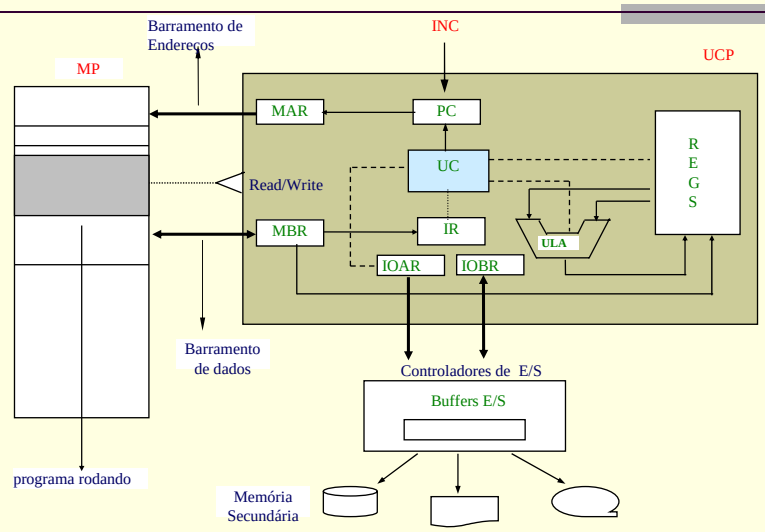


A Computer System basic structure

- **Von-Newman Model**

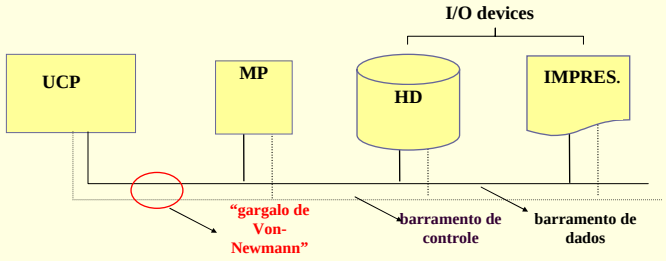


A Computer System basic structure



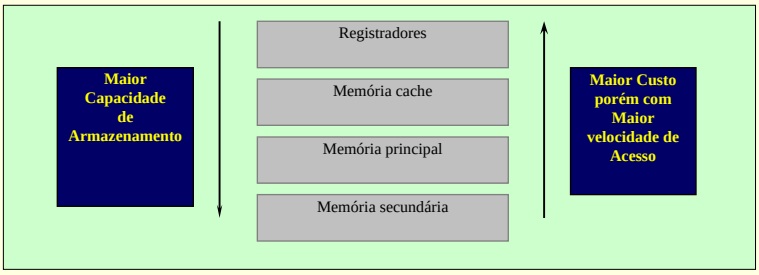
A Computer System basic structure

- **Von-Newman Bottleneck (VNB)**
  - Modelo de Von-Newman apresenta um problema estrutural chamado Gargalo de Von-Newman. Este problema acontece devido ao elevado tráfego de dados e informações entre a UCP e a MP.



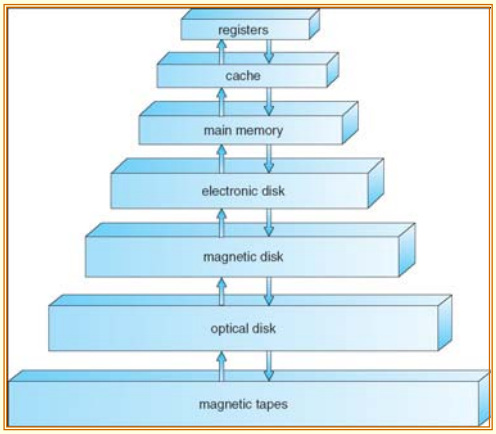
A Computer System basic structure

- **Memória Cache e o problema VN Bottleneck**
  - Para amenizar o problema criou-se a Memória Cache, que é uma memória volátil de alta velocidade e cujo tempo de acesso a um dado nela contido é muito menor que o tempo de acesso caso o dado estivesse na memória principal.



Hierarquia de Memória

- Storage systems organized in hierarchy.
  - Speed
  - Cost
  - Volatility
- *Caching* – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage.



Cache

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy

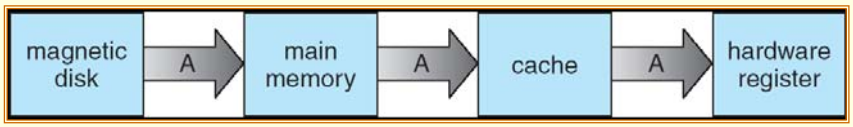
Performance dos diferentes níveis de Memória

- Movement between levels of storage hierarchy can be explicit or implicit

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000.000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

Migration of an Integer "A" from Disk to Register inside the CPU

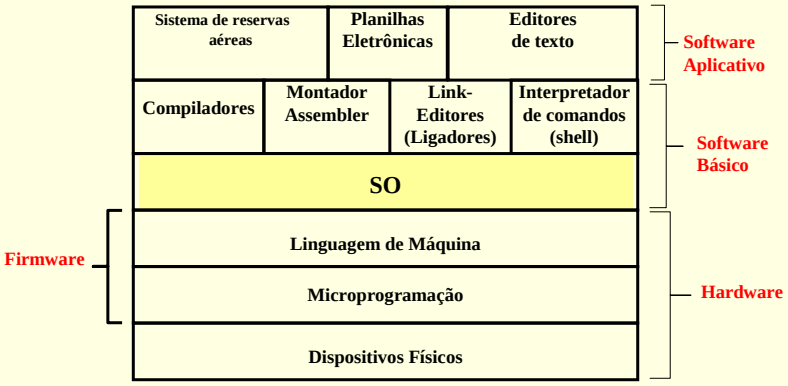
- Multitasking environments must be careful to use most recent value, not matter where it is stored in the storage hierarchy



- Multiprocessor environment must provide cache coherency in hardware such that all CPUs have the most recent value in their cache
- Distributed environment situation even more complex
  - Several copies of a datum can exist
  - Various solutions covered later ….

A Computer System basic structure

- **Tanenbaunn's Layer Model**

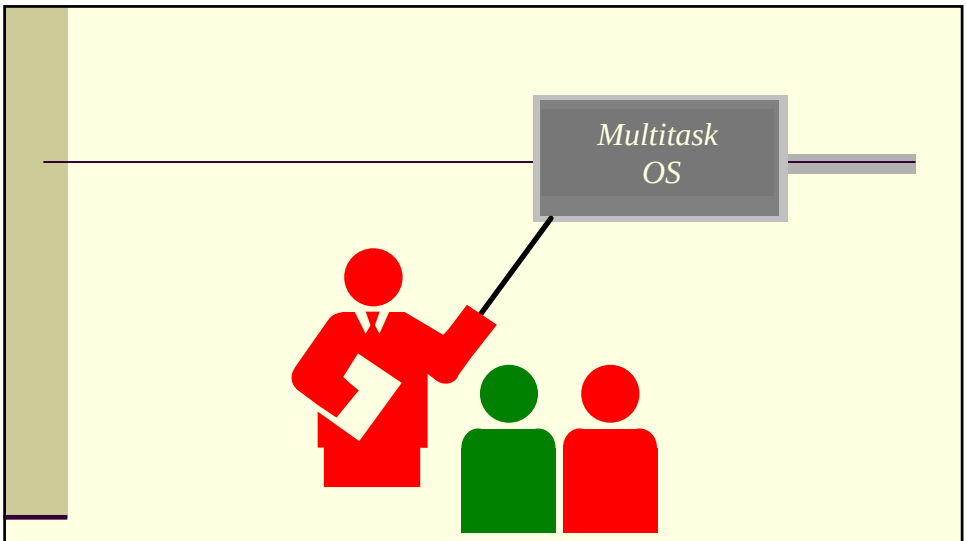


A Computer System basic structure

- Computer-system operation
  - Concurrent execution of CPUs and devices competing for memory cycles I/O devices and the CPU can execute concurrently.
  - Each device controller is in charge of a particular device type.
  - Each device controller has a local buffer.
  - CPU moves data from/to main memory to/from local buffers
  - I/O is from the device to local buffer of controller.
  - Device controller informs CPU that it has finished its operation by causing an *interrupt*

System Boot

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- *Booting* – starting a computer by loading the kernel
- *Bootstrap program* – code stored in ROM or EPROM that is able to locate the kernel, load it into memory, and start its execution
- System Boot
  - Operating system must be made available to hardware so hardware can start it
    - Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
    - Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
    - When power initialized on system, execution starts at a fixed memory location
      - Firmware used to hold initial boot code

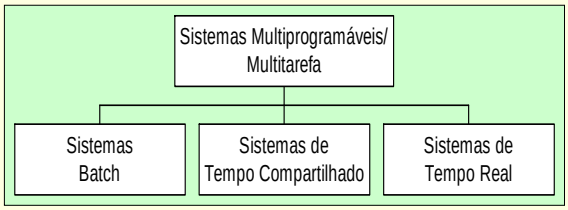


Classificação dos SOs

- **SOs Multiprogramáveis ou Multitarefa**
  - Nestes SOs vários programas compartilham a MP do sistema. As vantagens do uso destes sistemas são o aumento da produtividade dos seus usuários e a redução de custos, a partir do compartilhamento dos diversos recursos do sistema.
  - Podem ser **Multiusuário** ou **Monousuário.** É possível que execute diversas tarefas concorrentemente ou mesmo simultaneamente (**Multiprocessamento**) o que caracterizou o surgimento dos SOs **Multitarefa**.

Classificação dos SOs Multiprogramados

- Os SOs **Multiprogramáveis/Multitarefa** podem ser classificados pela forma com que suas aplicações são gerenciadas, podendo ser divididos conforme mostra o gráfico abaixo.

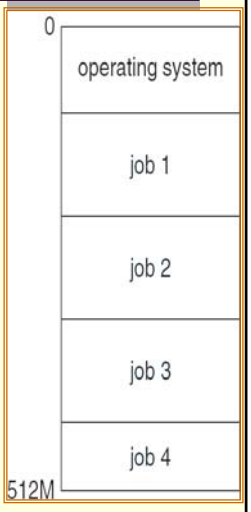


SOs Multiprogramáveis Batch

- Foram os primeiros SOs Multiprogramáveis a serem implementados e caracterizam-se por terem seus programas, quando submetidos, armazenados em disco ou fita, onde esperam ser carregados para execução seqüencial pelo monitor (embrião do SO).

Classificação dos SOs Multiprogramados

- **SOs Multiprogramáveis Batch (cont.)**
  - Quando bem projetados, podem ser bastante eficientes, devido à melhor utilização do processador. Entretanto, podem oferecer tempos de resposta longos, em face do processamento puramente seqüencial e com uma variação alta dos seus **tempos de execução** (**tempo de parede** ou **wall clock time** ou **elapsed time**).
- **Multiprogramming** is needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job



Classificação dos SOs Multiprogramados

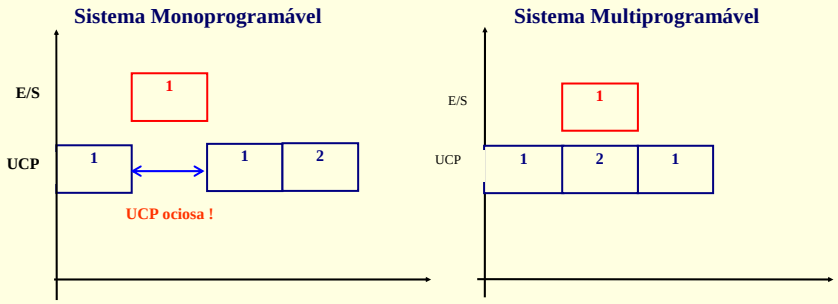
- **SOs Tempo Compartilhado – Time Sharing**
  - Permitem a interação dos usuários com o sistema, basicamente através de terminais de vídeo e teclado (interação on-line).
  - Esses sistemas possuem uma **Linguagem de Controle** que permite ao usuário comunicar-se diretamente com o SO para obter e dar informações diversas.
  - O SO aloca para cada usuário uma fatia de tempo de execução do processador o qual é chamada de **time-slice** (ou quantum de tempo). A execução do programa do usuário é interrompida após este tempo ou caso o programa peça a execução de uma **SVC** (chamada ao supervisor - supervisor call) para a leitura/gravação em algum periférico de E/S.
  - O **processo** (programa do usuário executando) cuja execução foi suspensa por fim do time-slice entra numa "**fila de execução de processos prontos**" para ser processado mais tarde.

Timesharing OS

- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇨**process**
  - If several jobs ready to run at the same time ⇨ **CPU scheduling**
  - If processes don't fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory

Timesharing OS

- **Conseqüência** -> UCP permanece menos tempo ociosa e a MP é utilizada de forma mais eficiente



Interrupção X Exceção

Interrupção e Exceção

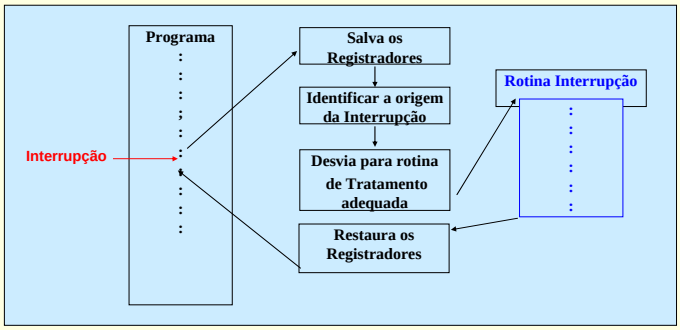
- ***Conceito de Interrupção e Exceção***
  - *Uma **Interrupção** ou **Exceção** é a ocorrência de algum evento durante a execução de um programa obrigando a intervenção do SO.*
  - *Tal evento pode ser resultado de:*
    - *execução de instruções do próprio programa seja devido a um erro ou a um pedido do usuário. Neste caso é chamada de **trap** (sw-generated interrupt);*
    - *gerado pelo SO;*
    - *por algum dispositivo de HW.*

Interrupção e Exceção

- ***Conceito de Interrupção e Exceção (cont.)***
  - *Uma **Interrupção** é gerada pelo SO ou por algum dispositivo e, neste caso, independe do programa que está sendo executado. Um exemplo é quando um periférico avisa à UCP que está pronto para transmitir algum dado. Neste caso, a UCP deve interromper o programa para atender a solicitação do dispositivo. Este tratamento é feito pelo **Mecanismo de Interrupção**, executado pelo HW, definido a seguir.*

Mecanismo de Interrupção

- **Mecanismo de Interrupção**



1. Unidade de controle detecta a ocorrência de interrupção, ela interrompe a execução do programa e salva o PC e a PSW do processo corrente na sua área de STACK.

Mecanismo de Interrupção

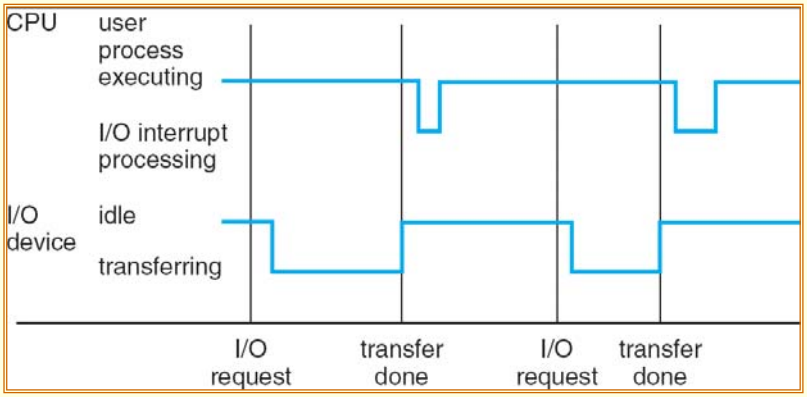
- **Mecanismo de Interrupção (cont.)**
  2. A seguir o controle da execução é passado para o SO que salva o restante das informações do contexto do processo que estava executando.
  3. Feito isto o controle é então desviado para a rotina do SO responsável pelo tratamento da interrupção.
- **Vetor de Interrupção**
  - Existem diferentes tipos de interrupção que são atendidas por diversas rotinas de tratamento. No momento que uma interrupção acontece, a UCP deve saber para qual rotina de tratamento deverá ser desviado o fluxo de execução. Essa informação está em uma estrutura do SO chamado ***Vetor de Interrupção***, que contém a relação de todas as rotinas de tratamento existentes, associadas a cada interrupção.

Mecanismo de Interrupção

- Todo o procedimento para detectar a interrupção, salvar o contexto do programa e desviar para a rotina de tratamento é denominado **Mecanismo de Interrupção.** Este mecanismo é realizado, na maioria das vezes, pelo HW dos computadores, e foi implementado pelos projetistas para criar uma maneira de sinalizar ao processador eventos assíncronos que possam ocorrer no sistema.

- No caso de múltiplas interrupções ocorrerem, o processador deve saber qual interrupção será tratada primeiro, o que é feito através da prioridade que é atribuída pelo sistema operacional a cada interrupção. Normalmente o HW possui um dispositivo denominado **Controlador de Pedidos de Interrupção** que avalia, ordena os pedidos e salva o endereço da instrução interrompida.

Linha de tempo de Interrupção



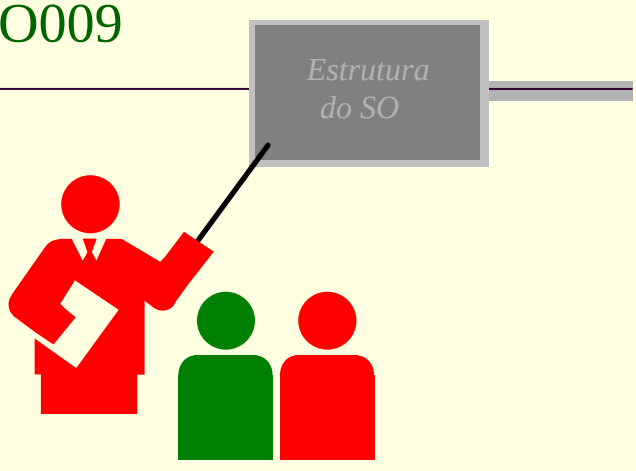
Exceção x Interrupção

- O que diferencia uma **Interrupção** de uma **Exceção** é o tipo de evento que gera esta condição. Uma exceção é resultado direto da execução de uma instrução do próprio programa. Situações como a divisão por zero ou a ocorrência de um overflow caracterizavam essa situação.

- A principal diferença entre **Exceção** e **Interrupção** é que a primeira é gerada por um **evento síncrono**, enquanto que a segunda é gerada por **eventos assíncronos**.

- A interrupção é o mecanismo que torna possível a implementação da concorrência nos computadores, sendo o fundamento básico dos sistemas Multiprogramáveis.

SOP – CO009



Estrutura do SO

- **Introdução**
  - O SO é formado por um conjunto de rotinas que oferecem serviços aos usuários do sistema e suas aplicações. Este conjunto de rotinas é chamado **Núcleo** ou **Kernel**.
- **Principais funções do SO**
  - *Interface com o Usuário;*
    - *CLI – command-Line Interface;*
    - *GUI – Graphical User Interface*
    - *Batch*
  - *Criação e Eliminação de Processos;*
    - *Carga de um programa para a memória e controle de sua execução;*
  -

Estrutura do SO

- **Principais funções do SO (cont.)**
  - *Sincronização e Comunicação entre Processos;*
    - *Permitir a troca de informações entre processos na mesma maquina ou em maquinas diferentes. A comunicação pode ser via memória compartilhada (shared memory") ou troca de mensagens (message passing);*
  - *Escalonamento Processos(Gerência de Processos)*
  - *Gerência do Sistema de Arquivos;*
    - *Criar, remover, ler e escrever arquivos e diretórios*
  - *Gerência de Memória;*
  - *Tratamento de Interrupções;*

Estrutura do SO

- **Principais funções do SO (cont.)**
  - *- Operações de E/S (Gerência de Periféricos);*
    - *O programa em execução pode solicitar uma operação de E/S que pode envolver um arquivo ou um device E/S*
  - *- Detecção e correção de erros;*
    - *O SO deve estar constantemente controlando a ocorrência de possíveis erros:*
      - *Podem ocorrer na CPU, Memória, dispositivos de E/S ou no programa do usuário*
      - *Para cada tipo de erro, o SO deve tomar a ação apropriada para garantir a a correção e consistência da computação executada.*
      - *Facilidades de depuração permitem ao usuário mais facilmente identificar e consertar os erros.*

Estrutura do SO

- **Principais funções do SO (cont.)**
  - *- Contabilização;*
    - *Controlar e contabilizar o uso dos recursos utilizados pelos usuários;*
  - *- Segurança do Sistema - Proteção e Segurança;*
    - *As informações armazenadas em um sistema Multiusuário devem ter as suas informações sigilosas preservadas, de tal forma que processos concorrentes não interfiram uns com os outros:*
      - *Proteção – garantir que o acesso a todos os recursos seja feita de forma controlada;*
      - *Segurança – requer o uso de mecanismos de autenticação para defender o sistema contra a entrada de intrusos;*

SO - CLI

- CLI allows direct command entry
  - Sometimes implemented in kernel, sometimes by systems program
  - Sometimes multiple flavors implemented – **shells**
  - Primarily fetches a command from user and executes it
    - Sometimes commands built-in, sometimes just names of programs
      - If the latter, adding new features doesn't require shell modification

SO - GUI

- User-friendly **desktop** metaphor interface
  - Usually mouse, keyboard, and monitor
  - **Icons** represent files, programs, actions, etc
  - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**)
  - Invented at Xerox PARC
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X as "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)

Kernel do SO

- **Kernel ou Núcleo do SO (cont.)**
  - *A estrutura do **Kernel** do SO, isto é, a maneira como o código do SO é organizado e o inter-relacionamento entre os seus diversos componentes, pode variar conforme a concepção de projeto do SO. Existem basicamente três abordagens: **Monolítico**, **Em Camadas** e **Micro-Núcleo** (**Micro-Kernel**)*

Estrutura do SO

- **Kernel ou Núcleo do SO (cont.)**
  - O SO mantém a estrutura de dados que reflete "o estado" dos recursos do sistema em um dado momento. Esta estrutura é composta por:
    - descritores de processos - *recursos lógicos*
    - descritores de memória - *recursos físicos*
    - descritores de arquivo - *recursos lógicos*
    - descritores de periféricos - *recursos físicos*
  - *Quando um gerente se comunica com outro gerente a comunicação passa pelo núcleo. Os descritores são implementados como listas encadeadas em memória.*

Estrutura do SO

- **Proteção em Sistemas Multiprogramados**
  - *A fim de garantir a integridade dos dados pertencentes a cada usuário o SO deve implementar algum tipo de proteção aos diversos recursos que são compartilhados no sistema, como memória, dispositivos de E/S e UCP.*
  - ***Proteção a memória** : quando o programa tenta acessar uma posição de memória fora de sua área endereçável, um erro do tipo violação de acesso ocorre e o programa é encerrado.*

Estrutura do SO

- **Proteção em Sistemas Multiprogramados**
  - ***Compartilhamento de dispositivos de E/S** : é controlado de forma centralizada pelo SO. Em geral o SO disponibiliza rotinas para trancamento (lock) de arquivos e/ou registros de arquivos para permitir o acesso exclusivo ou compartilhado por diversos usuários. No UNIX temos a **system call** flock.*

Estrutura do SO

- ***Compartilhamento da UCP*** *: Para evitar que um programa em loop aloque o processador por tempo indeterminado, a UCP possui um* ***relógio de tempo real*** *que gera interrupções periódicas (time-slice). A cada interrupção é executada a rotina de tratamento de interrupções de relógio, que entre outras funções, tem a responsabilidade de suspender a execução do processo corrente.*

Estrutura do SO

- Para garantir a proteção do sistema exige-se que toda vez que um usuário desejar utilizar um recurso, ele deverá solicitar o uso do recurso ao SO. O pedido é realizado através de chamadas a rotinas especiais denominadas **system calls**

- **Chamadas ao Sistema - System Call**
  - Uma **system call** permite o acesso a recursos do SO, e por isso possuí um mecanismo de proteção para sua execução denominado **estado de Execução** (**PSW** process status word). O **estado de execução** é uma característica associada ao programa em execução, que determina se ele pode ou não executar outras instruções ou rotinas.

Estrutura do SO

- No **estado usuário**, um programa só pode executar instruções que não afetam diretamente outros programas (**instruções não-privilegiadas**). No **estado supervisor**, qualquer instrução pode ser executada. As instruções que só podem ser executadas por programas no **estado supervisor** são denominadas **instruções privilegiadas.**

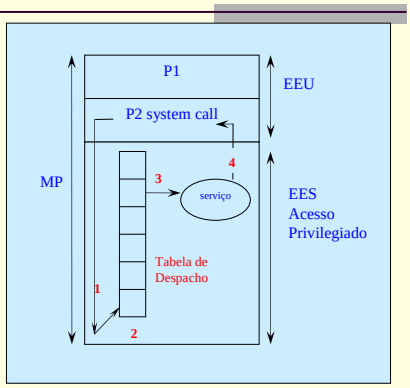
- O **estado de execução** de um processo é determinado por um conjunto de bits, localizado em um registrador especial da UCP que indica o estado corrente (PSW process status word) o qual o HW do sistema acessa para verificar o estado do processo, e se a instrução pode ou não ser executada. Quando um programa que está sendo processado no estado usuário executa uma **system call** o seu estado é alterado pela própria rotina do sistema que se encarrega, ao seu término, de restaurar o estado de execução anterior do processo. Caso um programa tente executar uma **instrução privilegiada**, sem estar no estado supervisor, uma interrupção é gerada e o programa é encerrado.

Estrutura do SO

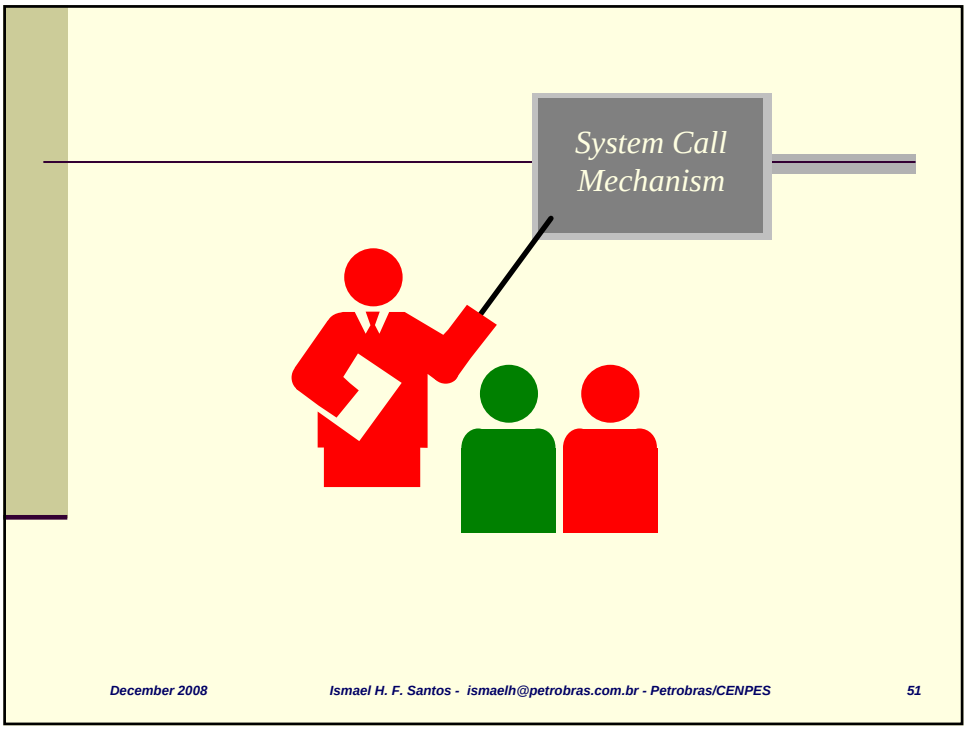
- *O SO divide a MP em dois espaços de enderecamento:*

  EES – EE do Sistema

  EEU – **EE do Usuário**



- *Programas dos usuários só podem ter acesso ao **EES** via mecanismos de* **system call**, *que é invocado sempre que um processo precisa ter acesso a algum serviço disponível.*

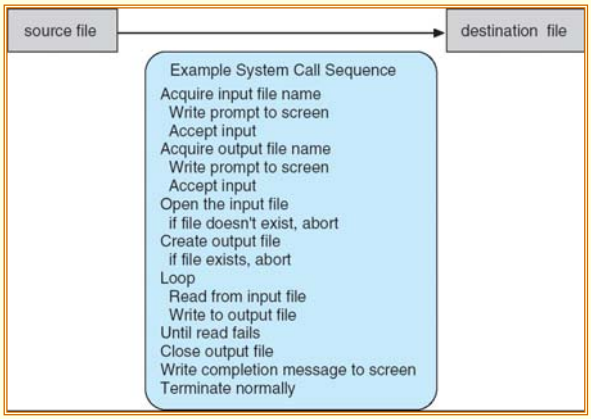


System Calls

- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)

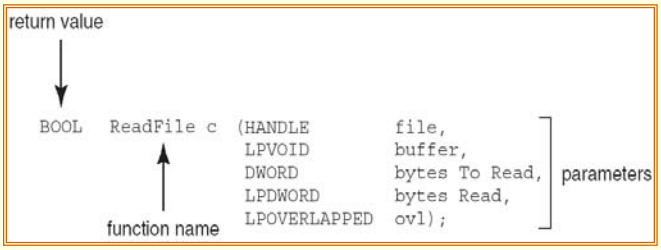
Exemplo de System Call

- System call sequence to copy the contents of one file to another file



Exemplo Standard API

- Consider the ReadFile() function in the
- Win32 API - a function for reading from a file

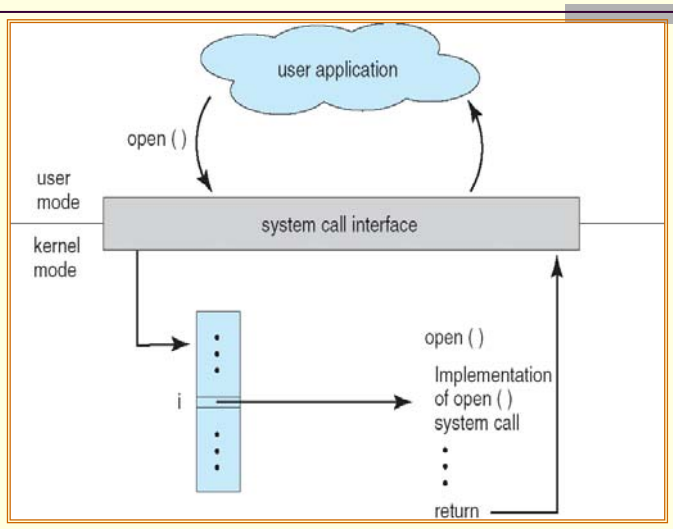


- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

System Call Implementation

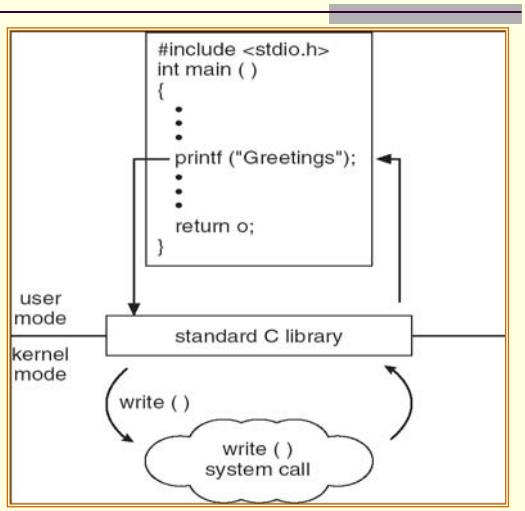
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call



Standard C Library Example

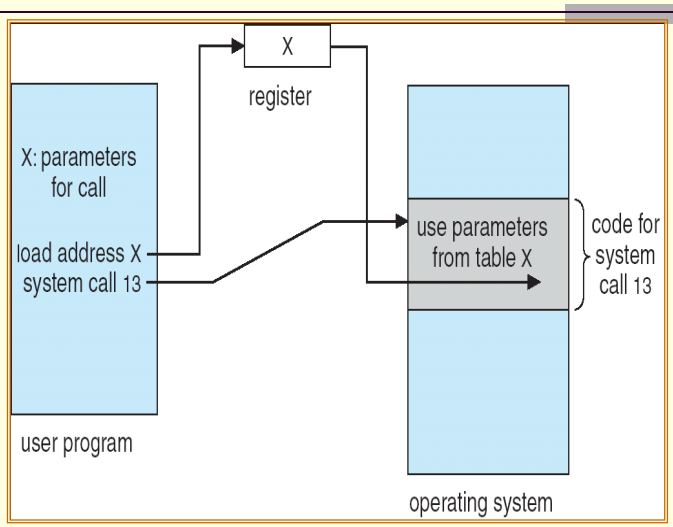
- C program invoking printf() library call, which calls write() system call



Passagem de Parâmetros para System Call

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  - Simplest: pass the parameters in *registers*
    - In some cases, may be more parameters than registers
  - Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system
  - Block and stack methods do not limit the number or length of parameters being passed

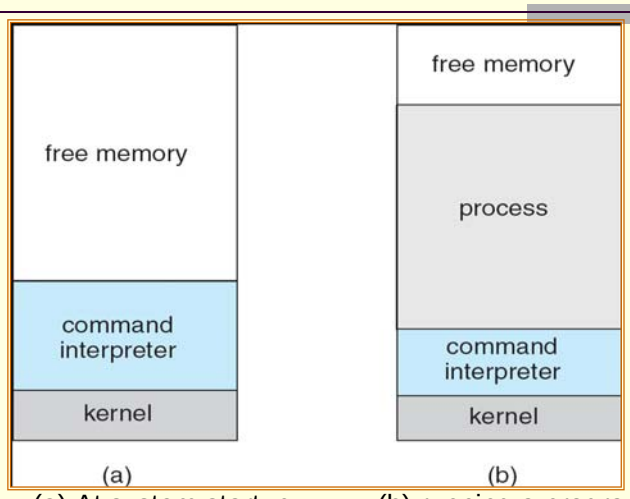
Passagem de Parâmetros via Tabela



Tipos de System Calls

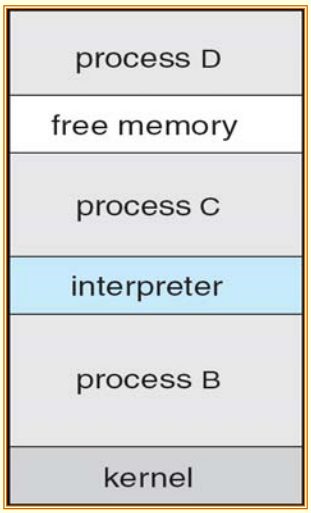
- As **system calls** podem ser divididas em 5 categorias:
  - Controle de processos – criar, terminar, sinalizar, etc.
  - Manipulação de arquivos – criar, remover, ler, gravar, etc.
  - Gerência de dispositivos
  - Comunicação entre processos
  - Informações gerais – obter informações de contabilização, data e hora do sistema, etc.

MS-DOS execution



- (a) At system startup (b) running a program

FreeBSD – Running Multiple Programs

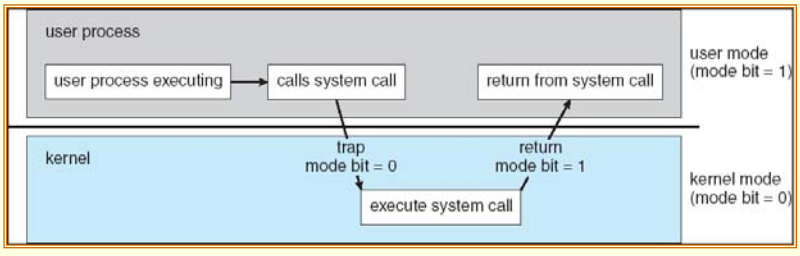


Operação do SO

- Interrupt driven by hardware
- Software error or request creates **exception** or **trap**
  - Division by zero, request for operating system service
- Other process problems include infinite loop, processes modifying each other or the operating system
- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** in the **PSW** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user

Transição User-Mode to Kernel-Mode

- Timer to prevent infinite loop / process hogging resources
  - Set interrupt after specific period
  - Operating system decrements counter
  - When counter zero generate an interrupt
  - Set up before scheduling process to regain control or terminate program that exceeds allotted time



Tratamento de E/S

- After I/O starts, control returns to user program only upon I/O completion.
  - Wait instruction idles the CPU until the next interrupt
  - Wait loop (contention for memory access).
  - At most one I/O request is outstanding at a time, no simultaneous I/O processing.
- After I/O starts, control returns to user program without waiting for I/O completion.
  - *System call* – request to the operating system to allow user to wait for I/O completion.
  - *Device-status table* contains entry for each I/O device indicating its type, address, and state.
  - Operating system indexes into I/O device table to determine device status and to modify table entry to include interrupt

E/S Síncrona e Assíncrona

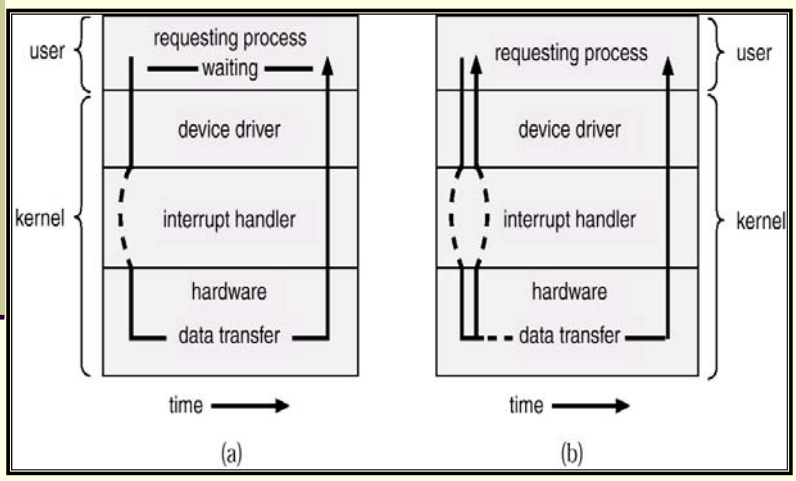
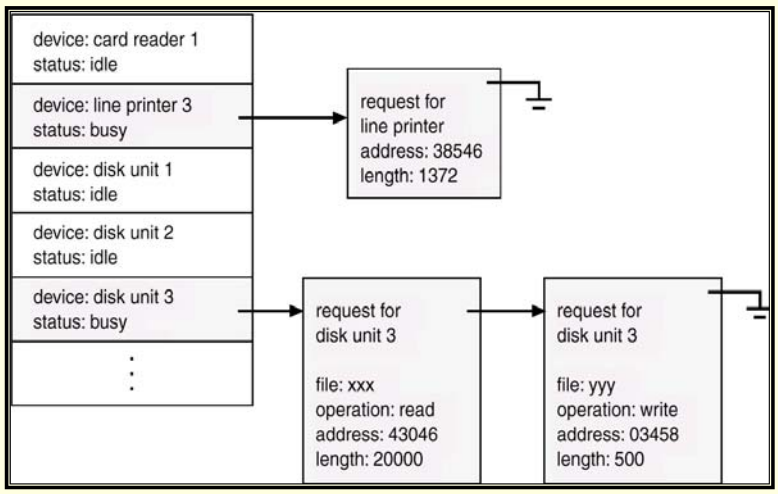


Tabela de Status de cada Device



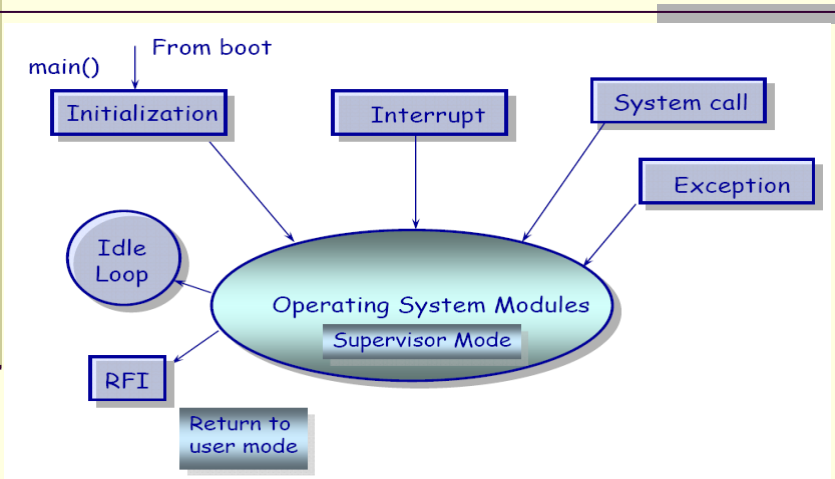
DMA – Direct Memory Access

- Used for high-speed I/O devices able to transmit information at close to memory speeds.
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.
- Only on interrupt is generated per block, rather than the one interrupt per byte.

Conclusão

- *Concluímos então que o SO é um conjunto de programas direcionados por eventos (**event-driven programs**): se não há jobs para executar, nenhum dispositivo de E/S para atender e nenhum usuário para atender o SO permanecerá parado esperando até que algum evento ocorra para ser atendido.*

Control Flow of the OS



Interrupts

- Hardware
  - (sets the PC to) the operating system at a pre-specified location
  - saves a small amount of state in SPRs
  - sets UM "off" in the MSR
  - loads the next instruction
- Operating system
  - saves (remaining) state of the user process
  - identifies the device and/or cause of interrupt
  - Responds to (processes) the interrupt
  - restores state of the user program (if applicable) or some other user program
  - Executes an RFI instruction to return to the user program (RFI changes PC, sets UM "on" in the MSR
- User program continues exactly at the same point it was interrupted.

Key Fact: This is transparent to the user program

Exceptions

- Hardware
  - (sets the PC to) the operating system at a pre-specified location ("Exception" handler)
  - saves a small amount of state in SPRs
  - sets UM "off" in the MSR
  - loads the next instruction
- Operating system
  - identifies the cause of the exception (e.g. divide by 0)
  - If user process has exception handling specified for the specific exception
    - then OS adjusts the user program state so that it calls its handler
    - Execute an RFI instruction (with PC to user's exception handler)
  - else OS kills it and runs some other user program, as available

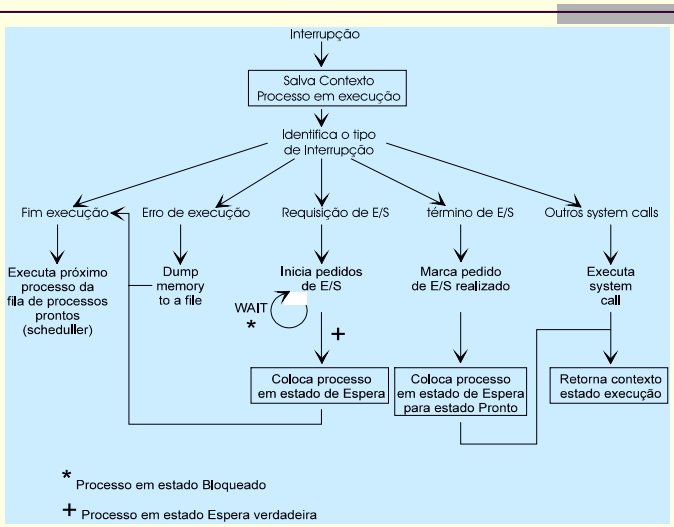
Key Fact: Effects of exceptions can be visible to user programs and cause abnormal execution flow

System Calls

- User program executes a trap instruction (system call)
- Hardware
  - Sets the PC to the operating system system call handler.
  - saves a small amount of state in SPRs
  - sets UM "off" in the MSR
  - loads/executes the instruction at PC
- Operating system
  - identifies the requested service and parameters (e.g. open(filename, O_RDONLY))
  - executes the required service
  - restores user's state (including registers)
  - replaces a register value with indicator of success/failure
  - executes an RFI instruction to return to the user program
- User program receives the result and continues

Key Fact: To the user program, it appears as a function call executed under program control

Operação do SO



Estrutura do Kernel

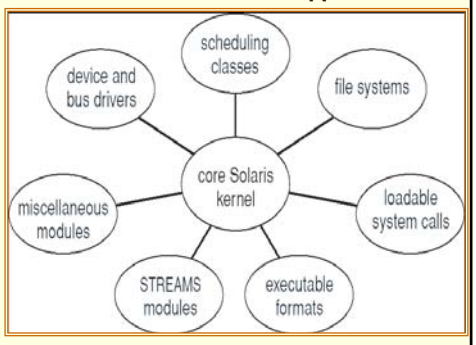
Estrutura do Kernel do SO

- *O Projeto e a Implementação de um SO é uma tarefa complexa e por isso a sua estrutura interna varia de acordo com a escolha do HW e o tipo de arquitetura.*
- *User* goals and *System* goals
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient
- **Policy:** What will be done?
  Mechanism: How to do something ?
  - The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later

Estrutura do Kernel do SO

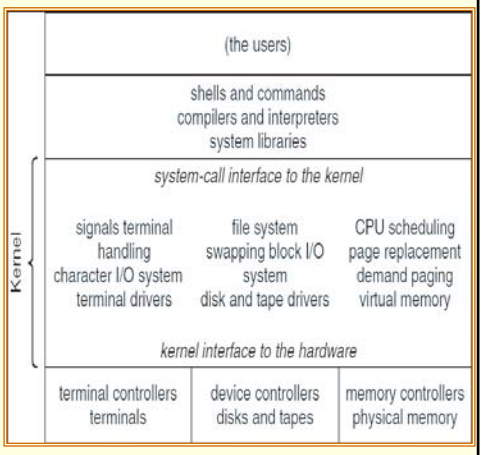
- Sistemas Monolíticos(cont)
  - Most modern operating systems implement kernel modules
    - Uses object-oriented approach
    - Each core component is separate
    - Each talks to the others over known interfaces
    - Each is loadable as needed within the kernel
  - Overall, similar to layers but with more flexible

Solaris Modular Approach



Estrutura do Kernel do SO

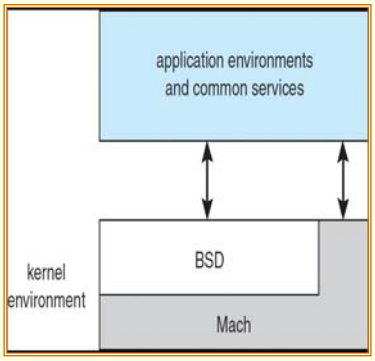
- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - Systems programs
  - The kernel
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level



Estrutura do Kernel do SO

- Sistemas Micro-Kernel (cont)
  - Os Micro-Núcleos disponibilizam uma quantidade mínima de serviços chamados serviços essenciais:
    - comunicação entre processos;
    - gerência de memória básica;
    - gerencia de processos
    - escalonamento;
    - entrada/saída de baixo nível.
  - Benefits:
    - Easier to extend a microkernel
    - Easier to port the operating system to new architectures
    - More reliable (less code is running in kernel mode)
    - More secure
  - Detriments:
    - Performance overhead of user space to kernel space communication

Mac OS X Structure



Estrutura do Kernel do SO

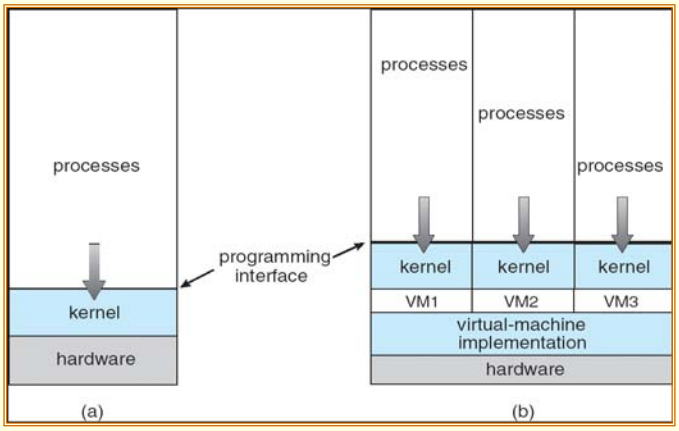
- Máquinas Virtuais
  - A *virtual machine* takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
  - A virtual machine provides an interface *identical* to the underlying bare hardware
  - The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory

Estrutura do Kernel do SO

- Máquinas Virtuais (cont'd)
  - The resources of the physical computer are shared to create the virtual machines
    - CPU scheduling can create the appearance that users have their own processor
    - Spooling and a file system can provide virtual card readers and virtual line printers
    - A normal user time-sharing terminal serves as the virtual machine operator's console

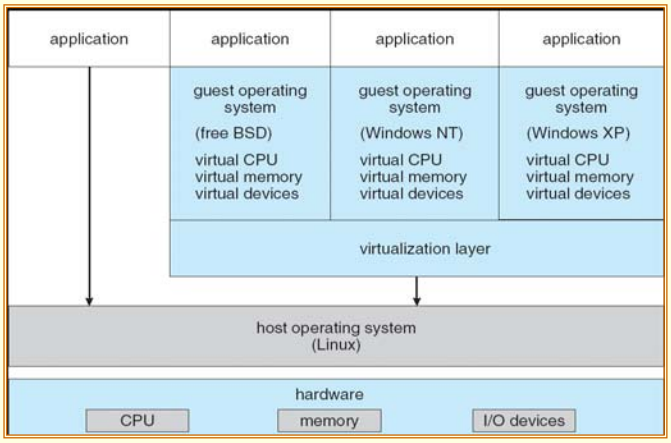
Estrutura do Kernel do SO

- Máquinas Virtuais (cont'd)



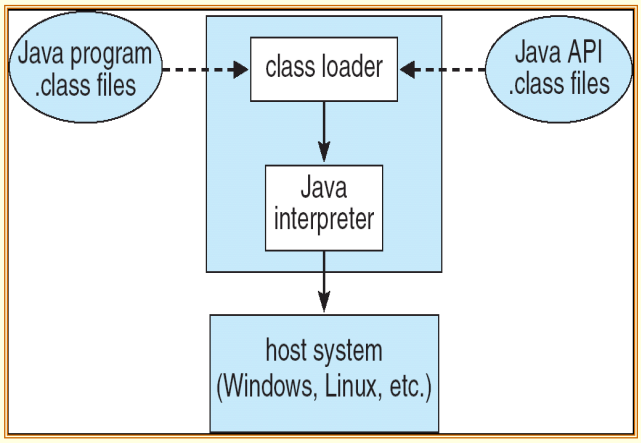
Estrutura do Kernel do SO

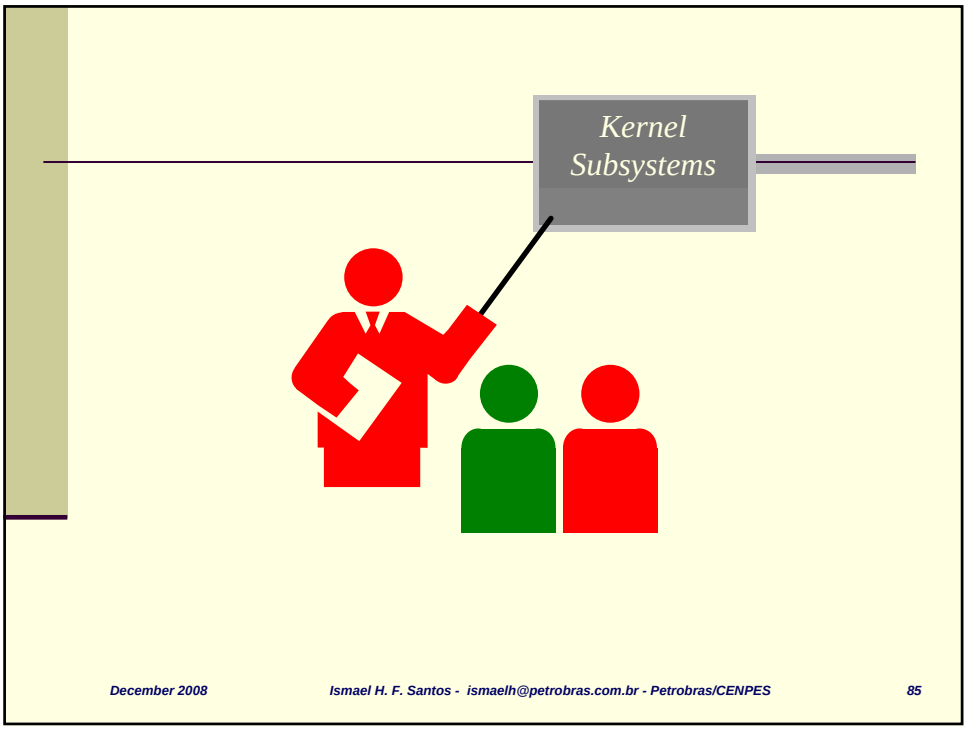
- Máquinas Virtuais (cont'd)



Estrutura do Kernel do SO

- Máquina Virtual Java





Kernel Subsystems

- Gerência de Processos
  - The operating system is responsible for the following activities in connection with process management:
  - Creating and deleting both user and system processes
  - Suspending and resuming processes
  - Providing mechanisms for process synchronization
  - Providing mechanisms for process communication
  - Providing mechanisms for deadlock handling

Kernel Subsystems

- Gerência de Memória
  - All data in memory before and after processing
  - All instructions in memory in order to execute
  - Memory management determines what is in memory when
    - Optimizing CPU utilization and computer response to users
  - Memory management activities
    - Keeping track of which parts of memory are currently being used and by whom
    - Deciding which processes (or parts thereof) and data to move into and out of memory
    - Allocating and deallocating memory space as needed

Kernel Subsystems

- Gerência de Memória
  - All data in memory before and after processing
  - All instructions in memory in order to execute
  - Memory management determines what is in memory when
    - Optimizing CPU utilization and computer response to users
  - Memory management activities
    - Keeping track of which parts of memory are currently being used and by whom
    - Deciding which processes (or parts thereof) and data to move into and out of memory
    - Allocating and deallocating memory space as needed

Kernel Subsystems

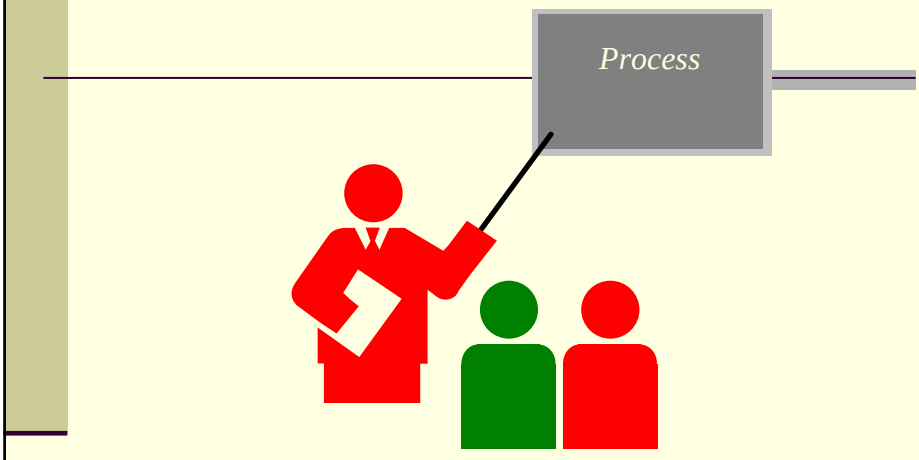
- Gerência de Sistema de Arquivos
  - OS provides uniform, logical view of information storage
    - Abstracts physical properties to logical storage unit - **file**
    - Each medium is controlled by device (i.e., disk drive, tape drive)
      - Varying properties include access speed, capacity, data-transfer rate, access method (sequential or random)
  - File-System management
    - Files usually organized into directories
    - Access control on most systems to determine who can access what
    - OS activities include
      - Creating and deleting files and directories
      - Primitives to manipulate files and dirs
      - Mapping files onto secondary storage
      - Backup files onto stable (non-volatile) storage media

Kernel Subsystems

- Gerência de Sistema de E/S
  - Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time.
  - Proper management is of central importance. Entire speed of computer operation hinges on disk subsystem and its algorithms
  - OS activities
    - Free-space management
    - Storage allocation
    - Disk scheduling
  - Some storage need not be fast
    - Tertiary storage includes optical storage, magnetic tape
    - Still must be managed
    - Varies between WORM (write-once, read-many-times) and RW (read-write)

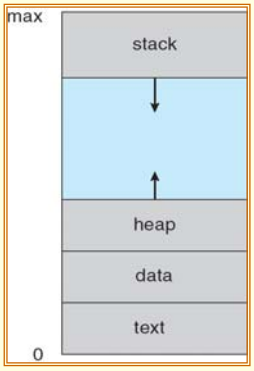
Kernel Subsystems

- Gerência de Sistema de E/S
  - One purpose of OS is to hide peculiarities of hardware devices from the user
  - I/O subsystem responsible for
    - Memory management of I/O including buffering (storing data temporarily while it is being transferred), caching (storing parts of data in faster storage for performance), spooling (the overlapping of output of one job with input of other jobs)
    - General device-driver interface
    - Drivers for specific hardware devices



Conceito de Processo

- **Definição**
  - **Processo** é o ambiente onde se executa um programa. Um mesmo programa pode produzir resultados diferentes, em função do **Processo** no qual ele é executado.
  - Processo pode ser definido também como um programa em execução.
  - A process includes:
    - program counter
    - stack
    - data section



Conceito de Processo

- **Definição**
  - O SO materializa o processo através de uma estrutura chamada **bloco de controle do processo (Process Control Block PCB).** A partir do **PCB**, o SO mantém todas as informações sobre o processo, como:
    - identificação
    - prioridade
    - estado corrente
    - recursos alocados
    - informações sobre o programa em execução, Program Counter, CPU registers.
    - CPU scheduling information
    - Memory-management information
    - Accounting information
    - I/O status information

Process Control Block (PCB)

PCB (1/process) contains info about a process, including its execution state (when not running).

General Info:

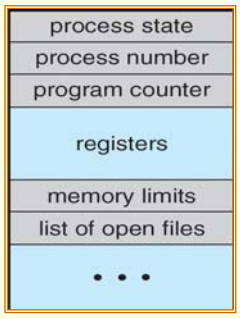
- Process number
- Process ID
- Username/ID
- Open File Info
- Scheduling Info (e.g. priority)
- Queuing fields

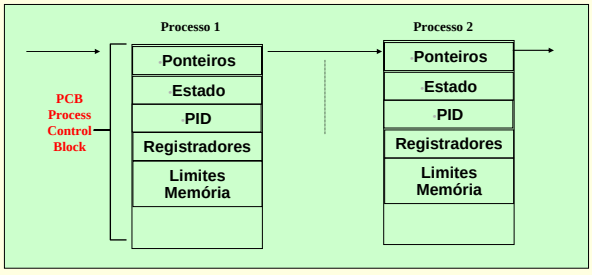
Execution State:

- Process state
- Program Counter (PC)
- Stack Pointer (SP)
- Registers (e.g. GPRs)

14

Conceito de Processo



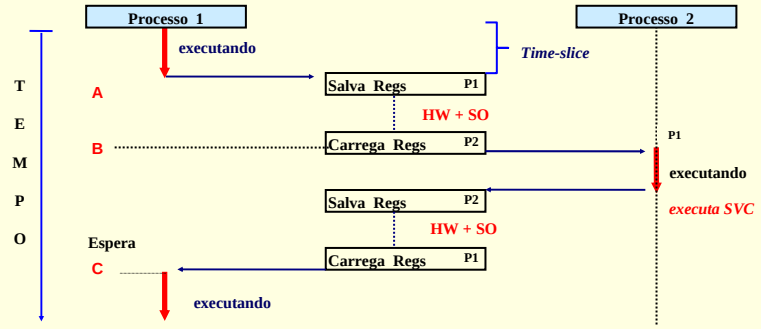


- O processo pode ser dividido em três elementos básicos: contexto de hardware (chw), contexto de software (csw) e espaço de endereçamento (ee), que juntos mantêm todas as informações necessárias à execução do programa.

Conceito de Processo

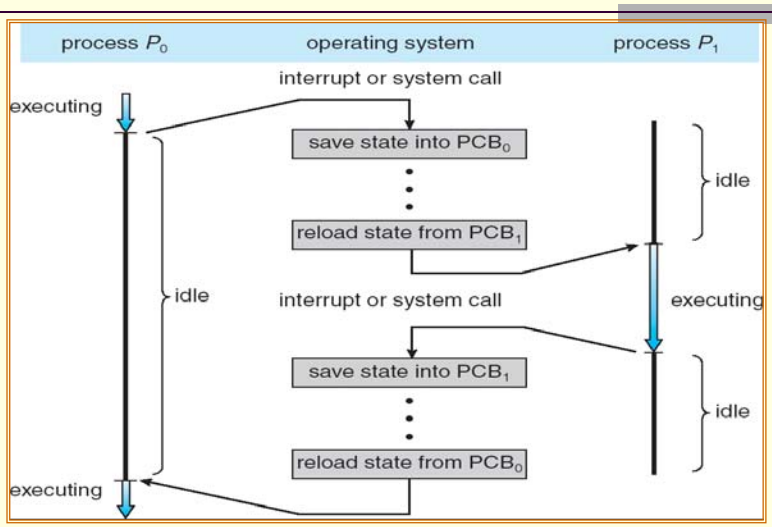
- **Contexto de HW (CHW)**
  - O **Contexto de Hardware** constitui-se, basicamente, do conteúdo de registradores: program counter (PC), stack pointer (SP) e bits de estado. Quando um processo está em execução, o seu contexto de hardware está armazenado nos registradores do processador. No momento em que o processo perde a utilização da UCP, o sistema salva suas informações no seu **CHW.**
  - O **CHW** é fundamental para a implementação dos SOs de tempo compartilhado onde os processos se revezam na utilização do processador. podendo ser interrompidos e, posteriormente, restaurados como se nada tivesse acontecido. A troca de um processo por outro na UCP, realizada pelo SO, é notificada através da **mudança de contexto** (**context switching**).

Salvamento de Contexto

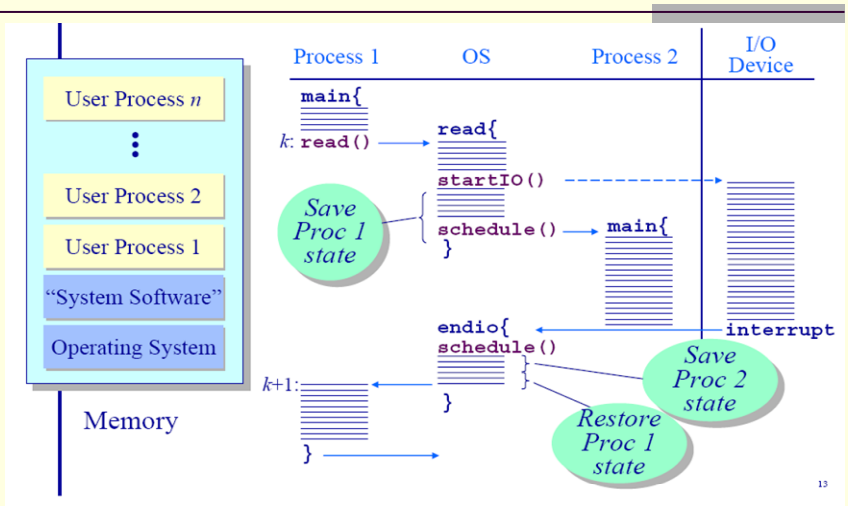


- A mudança de contexto consiste em salvar o conteúdo dos registradores da UCP e carregá-los com os valores referentes ao do processo que esteja ganhando a utilização do processador.

Salvamento de Contexto



Salvamento de Contexto



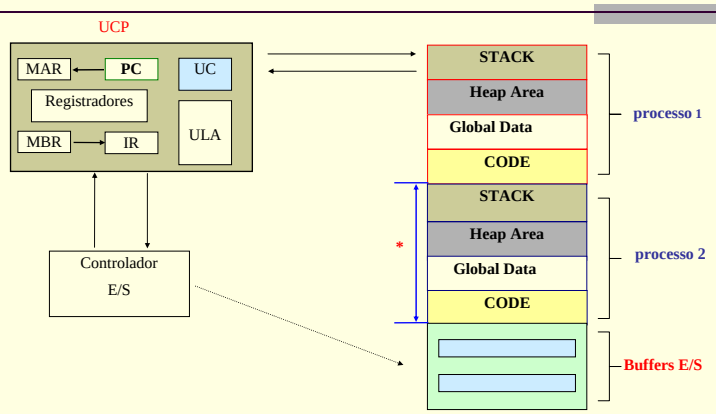
Conceito de Processo

- **Contexto de SW (CSW)**
  - O **Contexto de Software** especifica características do processo que vão influir na execução de um programa. como, por exemplo, o número máximo de arquivos abertos simultaneamente ou o tamanho do buffer para operações de E/S. Essas características são determinadas no momento da criação do processo, podendo algumas ser alteradas durante sua existência.
  - O **CSW** define três grupos de informações sobre um processo:
    1. **Identificação - PID - process identification; UID - user identif.**, atribuídas ao processo no momento de sua criação.
    2. **Quotas -** As quotas são os limites de cada recurso do sistema que um processo pode alocar.
    3. **Privilégios -** Os privilégios definem o que o processo pode ou não fazer em relação ao sistema e aos outros processos.

Conceito de Processo

- **Espaço de Endereçamento**
  - O espaço de endereçamento é a área de memória do processo onde o programa será executado, além do espaço para os dados utilizados por ele. Cada processo possui seu próprio espaço de endereçamento, que deve ser protegido do acesso dos demais processos.
  - Atualmente o modelo mais geral para o processo executando na memória principal é o mostrado a seguir na figura

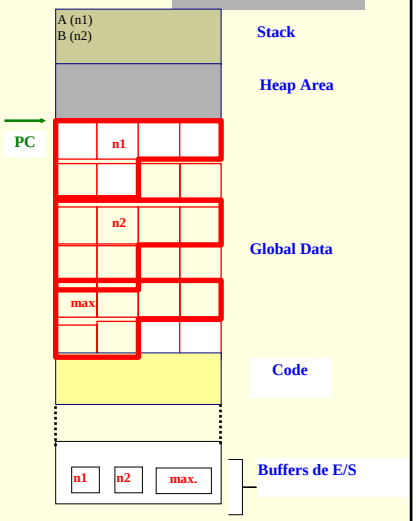
Espaço de Endereçamento



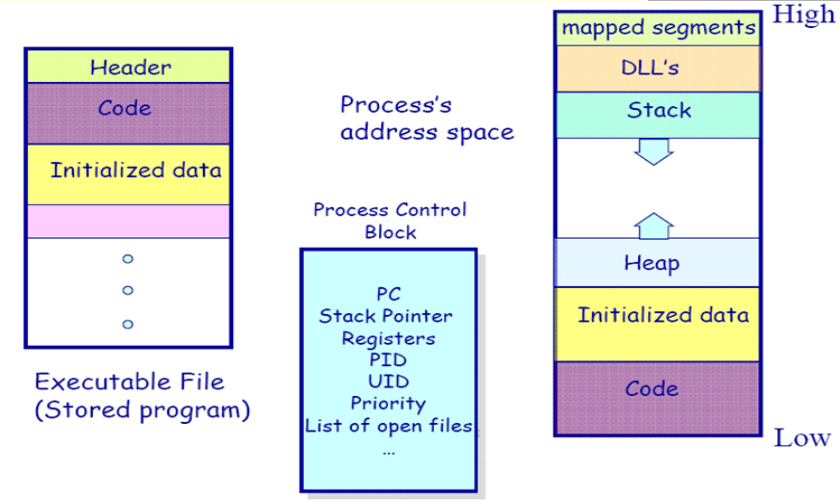
* Vejamos como é feita a execução de um programa que irá imprimir o maior de dois números dados pelo usuário

Process Execution

```
Programa MAIOR; {Imprimir o maior de dois
    números}
VAR {global}
    n1, n2, max: integer;
Function maior (A, B: integer): integer;
BEGIN
    IF A > B then
          maior:= A;
    ELSE
          maior:= B;
END;
BEGIN {pgm principal}
    READLN (n1); READLN (n2);     max:=
    maior (n1, n2);
    WRITELN ('maximo entre', n1, n2, 'é',
    max);
END
```



Process Execution



Conceito de Processo

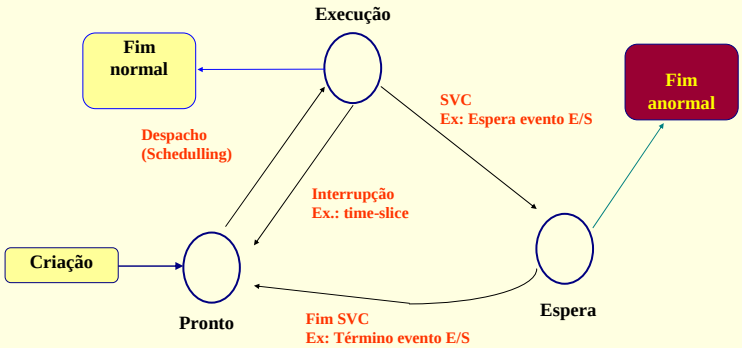
- **Estados de um Processo**
  - Um processo durante a sua existência passa por uma série de estados:
    - **Execução (running) -** quando está sendo processado pela UCP. Em sistemas com apenas um processador, somente um processo pode estar sendo executado num dado instante de tempo. Os processos se revezam na utilização do processador segundo uma política estabelecida pelo sistema operacional. Já em sistemas com múltiplos processadores. vários processos podem estar sendo executado ao mesmo tempo, dependendo do número de processadores. Existe também a possibilidade de um mesmo processo ser executado por mais de um processador (processamento paralelo).

Conceito de Processo

- **Estados de um Processo**
  - **Pronto (ready) -** Um processo está no **estado de Pronto** quando apenas aguarda uma oportunidade para executar, ou seja, espera que o sistema operacional aloque a UCP para sua execução. O sistema operacional é responsável por determinar a ordem pela qual os processos em estado de pronto devem ganhar a UCP. Normalmente existem vá-rios processos no sistema no estado de pronto.
  - **Espera (wait) -** Um processo está no **estado de Espera** quando aguarda algum evento externo ou por algum recurso para poder prosseguir seu processamento. Como exemplo. podemos citar o término de uma operação de entrada/saída ou a espera de uma deter-minada data e/ou hora para poder continuar sua execução.

Conceito de Processo

- **Diagrama de Transição de Estados**

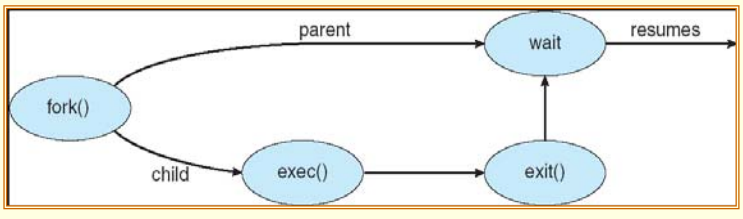


Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program



C Program Forking Separate Process

```
int main() {
    pid_t  pid = fork();          /* fork another process */
    if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
            /* parent will wait for the child to complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
    }
}
```


Unix's fork(): Example Usage

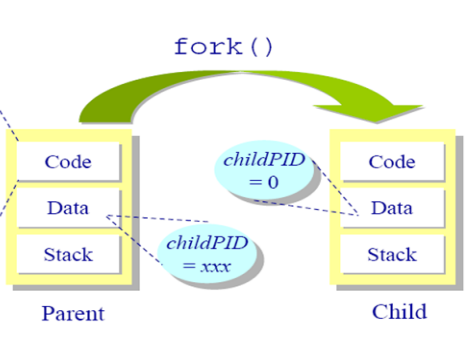
- The execution context for the child process appears as a *copy* of the parent's context at the time of the call

```
main {
  int childPID;
  S1;

  childPID = fork();

  if(childPID == 0)
    <code for child process>
  else {
     <code for parent process>
     wait();
   }

  S2;
}
```



18

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

Program Loading: exec()

- The exec() call allows a process to "load" a different program and start execution at _start
- It allows a process to specify the number of arguments (argc) and the string argument array (argv)
- If the call is successful
  - it is the same process …
  - but it runs a different program !!
- Two implementation options:
  - overwrite current memory segments with the new values
  - allocate new memory segments, load them with the new values, and deallocate old segments

General Purpose Process Creation

```
In the parent process:
main()
...
int pid = fork();                    // create a child
if(pid == 0) {
    // child continues here
    exec("program", argc, argv0, argv1, ...);
}
else {
    // parent continues here
...
}
```

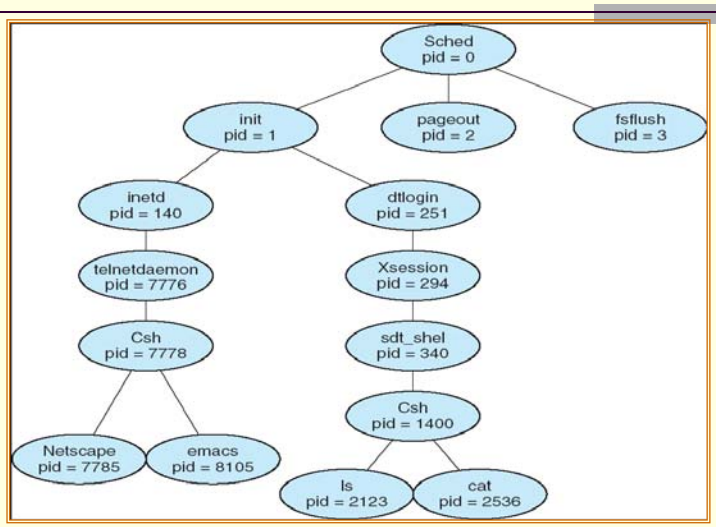
- After the program finishes execution, it calls *exit*(n)
- This ***system call*** :
  - takes the "result" of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies

Tying it All Together: The Unix Shell

```
while(! EOF) {
    read input
    handle regular expressions and file-name expansion
    int pid = fork();                // create a child
    if(pid == 0) {                   // child continues here
        exec("program", argc, argv0, argv1, ...);
}
else {                               // parent continues here
... wait for child to finish
}
```

- Translates <CTRL-C> to the kill() system call with SIGKILL
- Translates <CTRL-Z> to the kill() system call with SIGSTOP
- Allows input-output redirections, pipes, and a lot of other stuff that we will see later

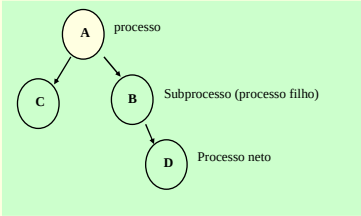
A tree of processes on a typical Solaris



Subprocesso e Thread

Conceito de Processo

- **Subprocesso e Thread**
  - Um processo pode criar outros processos de maneira hierárquica. Quando um processo (processo pai) cria um outro, chamamos o processo criado de **subprocesso** ou processo filho. O **subprocesso**, por sua vez, pode criar outros **subprocessos**.



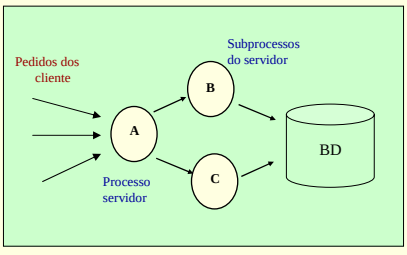
  - A utilização de **subprocessos** permite dividir uma aplicação em partes que podem trabalhar de forma concorrente.

Concorrência em arquitetura cliente cliente-servidor

- **Subprocesso e Thread**

Exemplo:

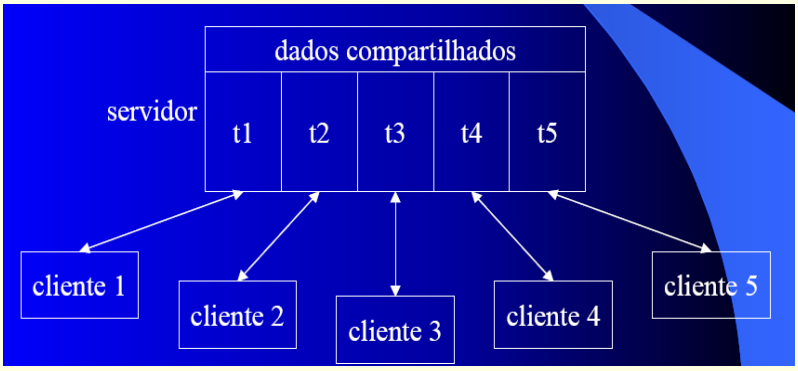
Suponha que um processo seja responsável pelo acesso a um banco de dados e existam vários usuários solicitando consultas sobre esta base.



Caso um usuário solicite um relatório impresso de todos os registros, os demais usuários terão de aguardar até que a operação termine. Com o uso de subprocessos. cada solicitação implicaria a criação de um novo processo para atendê-la, aumentando o throughput da aplicação e, conseqüentemente, melhorando seu desempenho.

Concorrência em arquitetura cliente cliente-servidor: servidor

- Atendimento simultâneo a vários clientes



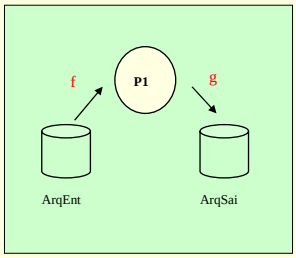
Concorrência em arquitetura cliente cliente-servidor: cliente

- Melhor estrutura da aplicação:
  - resposta a eventos de interface e de rede
- Melhor aproveitamento do tempo:
  - disparo de diversas solicitações simultâneas
  - tratamento local de dados enquanto espera resultado de solicitação

Conceito de Processo

- ***Subprocesso e Thread***
  - *Exemplo: Cópia de Arquivos*

```
Assign(f, 'ArqEnt´);
Assign(g, 'ArqSai´);
Reset (f,ArqEnt);
Rewrite (g,ArqSai);
Read (f,Reg);
While not eof(f) do
 Begin
   Write (g, reg);
   Read (f, g);
 End
```

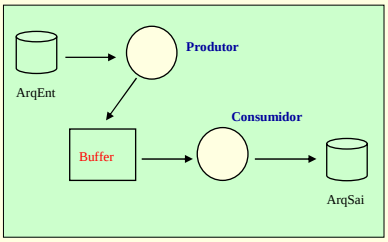


-> *Será que podemos melhorar este programa ?*

Conceito de Processo

- **Subprocesso e Thread**

Uma solução possível seria a criação de dois processos, um processo chamado Produtor que se encarregará de ler o arquivo de entrada e carregar um Buffer intermediário e um segundo subprocesso chamado Consumidor que irá ler as informações do Buffer e gravará os dados no arquivo de saída.



A função do Buffer é a de prover um meio de armazenamento para os dois processos, de tal forma que se o Buffer for infinito ambos nunca ficarão bloqueados (Explique).

Conceito de Processo

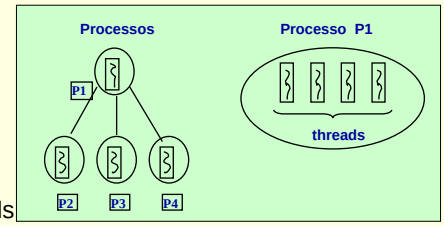
- **Subprocesso e Thread**
  - O uso de subprocessos no desenvolvimento de aplicações concorrentes demanda consumo de diversos recursos do sistema. Sempre que um novo processo é criado o SO deve alocar recursos (contexto de HW, contexto de SW e espaço de endereçamento) para cada processo além de consumir tempo de UCP neste trabalho. No caso de término do processo, o sistema desperdiça tempo para desalocar recursos previamente alocados.
  - Na tentativa de diminuir o tempo gasto na criação/eliminação de processos, bem como economizar recursos do sistema como um todo, foi introduzido o conceito de thread (ou processo leve ou linha de controle ou linha de execução).

Conceito de Processo

- **Subprocesso e Thread**
  - Em um SO com Kernel com capacidade para criar múltiplas threads (multithreaded Kernel) não é necessária a criação de vários processos para se implementar aplicações concorrentes. Em um SO Multithread cada processo pode responder a várias solicitações concorrentes.
  - Threads compartilham o processador da mesma maneira que um processos. Por exemplo, enquanto uma thread espera por uma operação de E/S, outra thread pode estar executando. Cada thread possui seu próprio conjunto de registradores (contexto de HW) , porém compartilha o mesmo espaço de endereçamento com as demais threads do processo.

Conceito de Processo

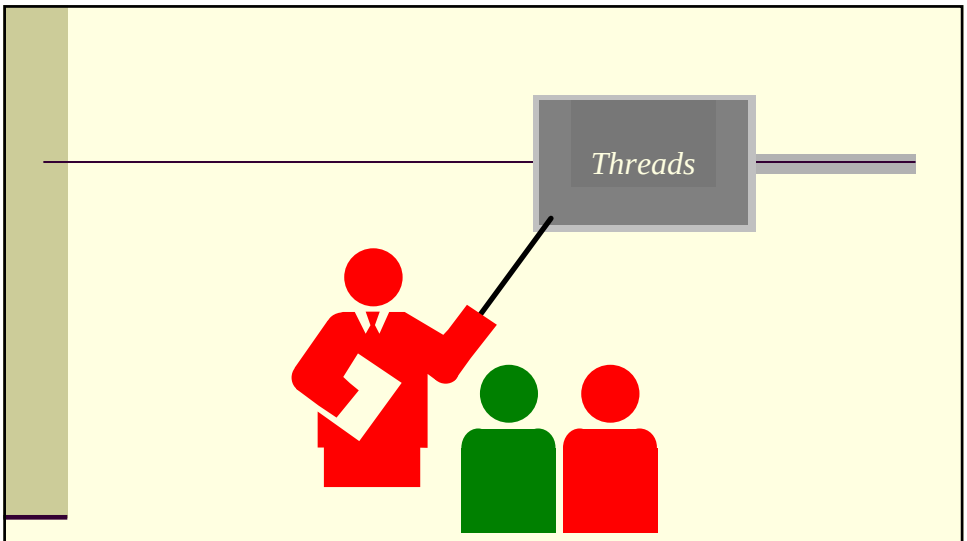
- **Subprocesso e Thread**
  - Na figura ao lado existem quatro processos, cada um com seu próprio contexto de HW, contexto de SW e espaço de endereçamento; e um único processo com 3 threads de execução, cada uma com seu próprio contexto de HW e contexto de SW.



  - O mecanismo de runtime é responsável pelo despacho para execução da thread de maior prioridade, ou da thread que estava esperando o fim de alguma operação de E/S. O controle de execução de threads é feito de forma cooperativa.

Conceito de Processo

- **Subprocesso e Thread**
  - Quando uma thread está sendo executada o contexto de HW da respectiva thread é carregado no processador. No momento em que uma thread perde (fim de time-slice) ou libera (yield) a UCP, o SO salva informações. Threads passam pelos mesmos estados que passam os processos.
  - A grande diferença entre subprocesso e thread é em relação ao espaço de endereçamento. Enquanto subprocessos possuem, cada um, espaços independentes e protegidos, threads compartilham o mesmo espaço de endereçamento do processo, sem nenhuma proteção, permitindo assim que uma thread possa alterar dados de outra thread.



Threads ou Ligth Weighted Process (LWP)

- A thread (or lightweight process) is a basic unit of CPU utilization; it consists of:
  - program counter
  - register set
  - stack space
- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources collectively know as a task.
- A traditional or heavyweight process is equal to a task with one thread

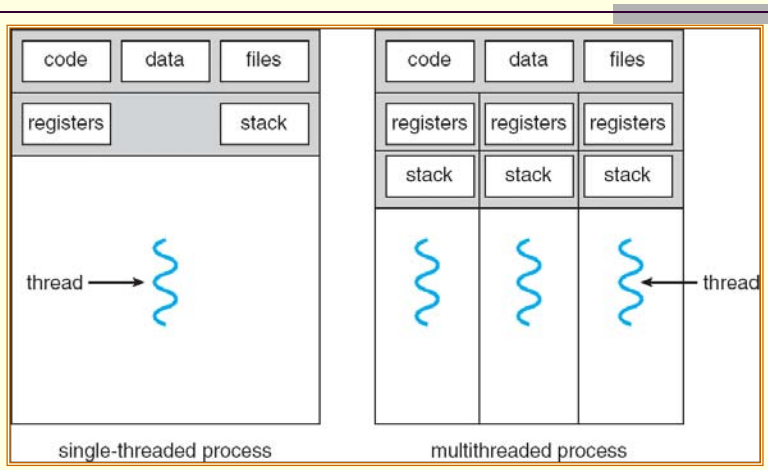
Threads (Cont.)

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
  - Cooperation of multiple threads in same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.
- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.
- Kernel-supported threads (Mach and OS/2).

Threads (Cont.)

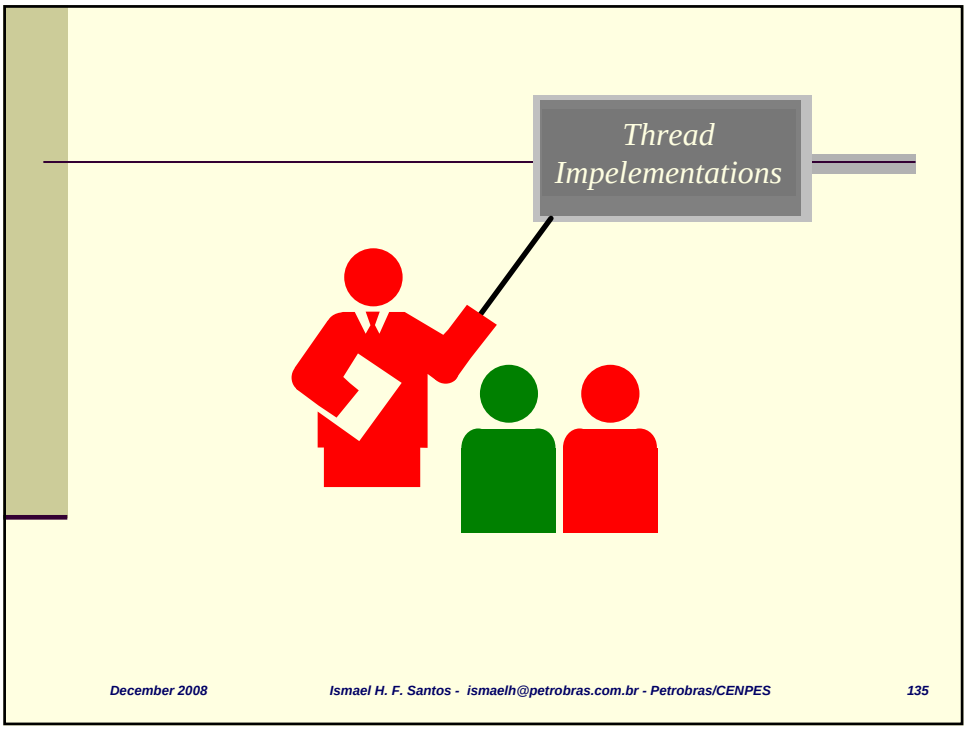
- User-level threads; supported above the kernel, via a set of library calls at the user level (Project Andrew from CMU).
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).
- Threads Benefits
  - Responsiveness
  - Resource Sharing
  - Economy
  - Utilization of MP Architectures

Single and Multithreaded Processes



User and Kernel Threads

- User Threads
  - Thread management done by user-level threads library
  - Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads
- Kernel Threads
  - Supported by the Kernel
  - Examples
    - Windows XP/2000
    - Solaris
    - Linux
    - Tru64 UNIX
    - Mac OS X



Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads – creating 1 thread

```
int sum; /* this data is shared by the thread(s) */
void *runner(void *param) {/* the thread */
   int upper = atoi(param);   int i;   sum = 0;
   if (upper > 0) {
      for (i = 1; i <= upper; i++)  sum += i;
   }
   pthread_exit(0);
}
main(int argc, char *argv[]) {
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of attributes for the thread */
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* now wait for the thread to exit */
   pthread_join(tid,NULL);    printf("sum = %d\n",sum);
}
```

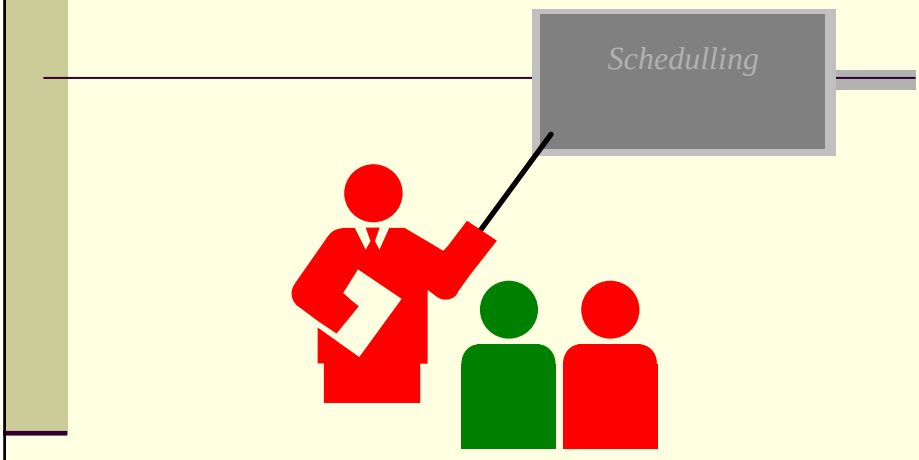

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[]) {
    int i;
    pthread_t tid[NUM THREADS];
    pthread_attr t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD SCOPE SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED OTHER);
```

Pthread Scheduling API

```
   /* create the threads */
   for (i = 0; i < NUM THREADS; i++)
      pthread_create(&tid[i], &attr, runner, NULL);
    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
      pthread_join(tid[i], NULL);
}

 /* Each thread will begin control in this function */
void *runner(void *param) {
    printf("I am a thread\n");
    pthread exit(0);
}
```

Gerência do Processador

- **Objetivos do Escalonamento**
  - manter a UCP ocupada a maior parte do tempo;
  - maximizar o throughput;
  - oferecer tempos de respostas aceitáveis para usuários interativos.
- **Criterios de Escalonamento**
  - Utilização da UCP:
  - Throughput;
  - Tempo de turnaround;
  - Tempo de Resposta.

Gerência do Processador

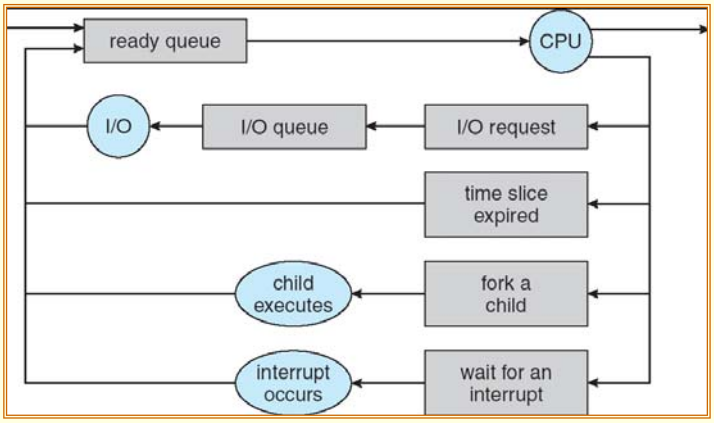
- O algoritmo de escalonamento busca otimizar a utilização da UCP e o Throughput, enquanto tenta diminuir os tempos de turnaround e de resposta.

- O algoritmo de escalonamento não é o único responsável pelo tempo de execução de um processo. Outros fatores, como o tempo de processamento (tempo de UCP) e de espera em operações de E/S. devem ser considerados no tempo total da execução (tempo de parede ou elapsed time ou wall clock time). O escalonamento somente afeta o tempo de espera de processos na fila de pronto.

Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Representation of Process Scheduling

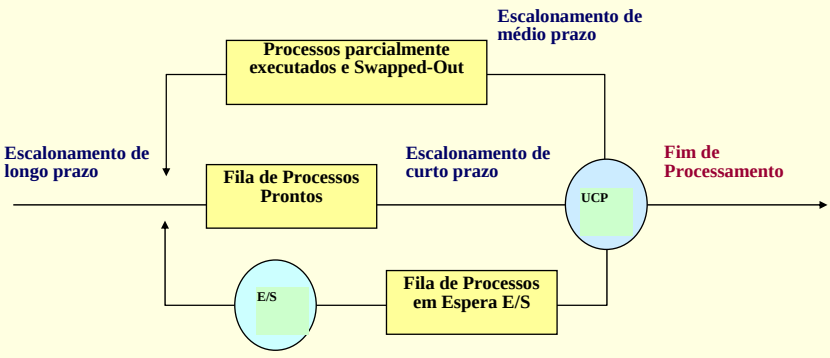


Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Gerência do Processador

- Escalonamentos de Longo, Médio e Curto Prazo



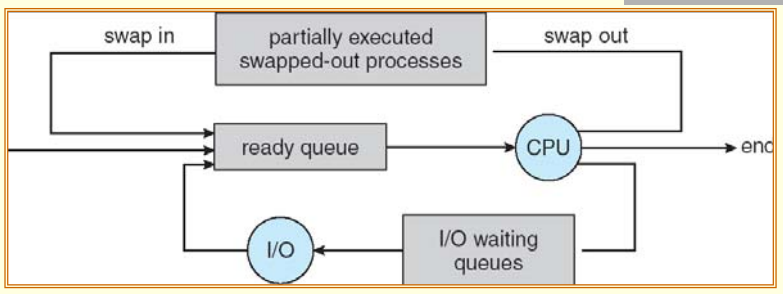
Gerência do Processador

- O escalonamento de **Longo Prazo** determina quais os jobs serão admitidos pelo SO para processamento.

- O de **Curto Prazo** seleciona o próximo processo a executar, dentre todos os processos da fila de processos prontos que estejam residentes em memória.

- O de **Médio Prazo**, mais comum em sistemas com **Memória Virtual** , tem objetivo de liberar espaço na memória, removendo processos que estejam esperando algum evento externo (por exemplo, o fim de uma operação de E/S) para area de **Swap**, para permitir que novos processos sejam executados.

Addition of Medium Term Scheduling



- Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*

Schedulers (Cont.)

- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Context Switch
  - When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process
  - Context-switch time is overhead; the system does no useful work while switching
  - Time dependent on hardware support

CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is *nonpreemptive*
- All other scheduling is *preemptive*

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- *Dispatch latency* – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

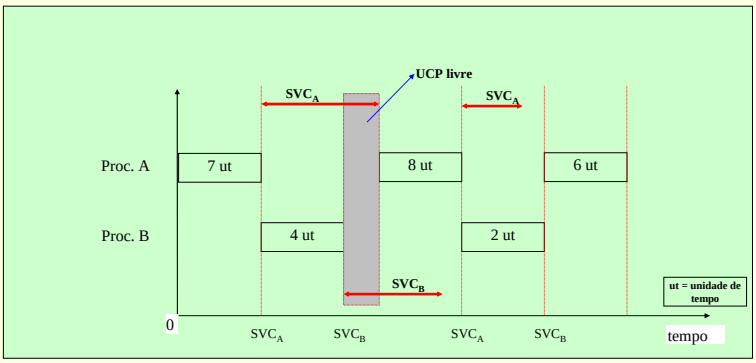
- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

Escalonamento CPU

- **Escalonamento Circular Simples - FIFO ou FCFS**
  - Todos os processos começam a executar segundo a ordem que são chamados para execução. Quando um processo ganha o processador, ele utilizará o processador até o seu final sem ser interrompido. No caso de ser executada uma SVC, o processo, após ter sido atendida a SVC, voltará para o final da fila de processos prontos.
  - O problema do **escalonamento FIFO** é a impossibilidade de se prever quando um processo terá sua execução iniciada, já que isso varia em função do tempo de execução dos processos que se encontram na sua frente.

Escalonamento CPU

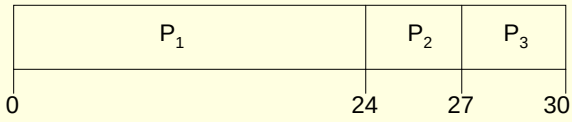
- ***Escalonamento FIFO (FCFS)***



First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1, P_2, P_3 The Gantt Chart for the schedule is:



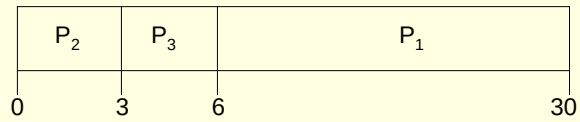
- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



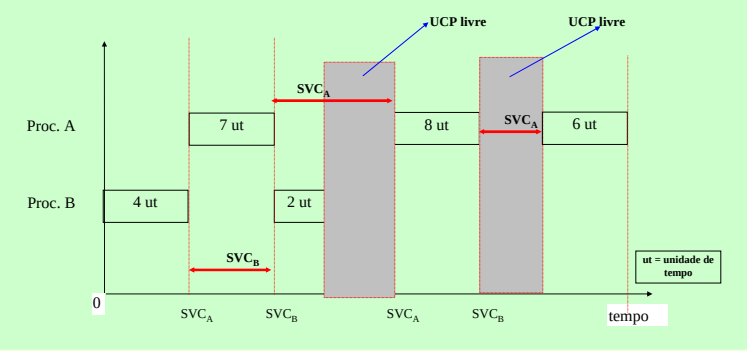
- Waiting time for P_1 = 6; P_2 = 0, P_3 = 3
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case !!!
- *Convoy effect* short process behind long process

Escalonamento CPU

- **Escalonamento Shortest Job First – SJF**
  - Neste escalonamento cada processo tem associado o seu tempo de execução. Desta forma quando a UCP está livre o processo em estado de pronto que tiver menor tempo de execução será selecionado para execução.
  - O escalonamento **SJF** favorece os processos que executam programas menores, além de reduzir o tempo médio de espera (na fila de processos prontos) em relação ao escalonamento FIFO. A dificuldade é determinar, exatamente, quanto tempo de UCP cada processo necessita para terminar seu processamento.

Escalonamento CPU

- **Escalonamento Shortest Job First - SJF**



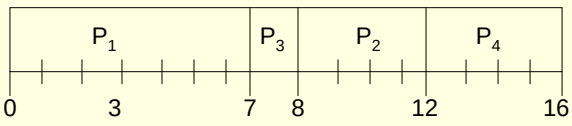
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)
- SJF is optimal – gives minimum average waiting time for a given set of processes

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

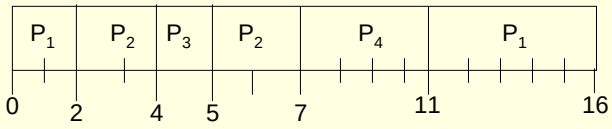


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



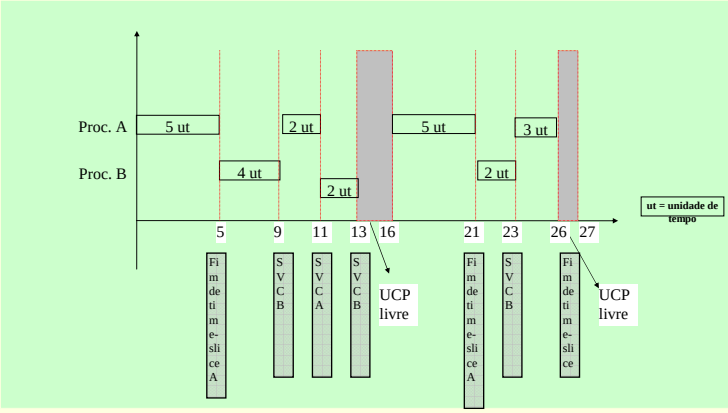
- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Escalonamento CPU

- **Escalonamento Circular - Round-Robin**
  - Algoritmo projetado especialmente para **SOs de Tempo Compartilhado**.O algoritmo é semelhante ao FIFO, porém, quando um processo passa para o estado de execução, existe um tempo limite para a sua utilização de forma continua. Quando este tempo, denominado **time-slice** ou **quantum**, expira sem que antes a UCP seja liberada pelo processo, este volta ao estado de pronto (**preempção por tempo**), dando a vez a outro processo. A fila de processos prontos é tratada como uma fila circular.

Escalonamento CPU

- **Escalonamento Circular - Round-Robin**



Escalonamento CPU

- **Escalonamento Round-Robin**
  - O **Escalonamento Round-Robin (RR)** consegue melhorar a distribuição de tempo de UCP em relação aos escalonamentos não preemptivos, porém não consegue implementar um compartilhamento eqüitativo entre os diferentes tipos de processos. Isso acontece em razão do escalonamento circular tratar os processos igualmente.
  - No **Escalonamento RR** os processos **IO-Bound** são prejudicados em relação aos processos **UCP-Bound**.

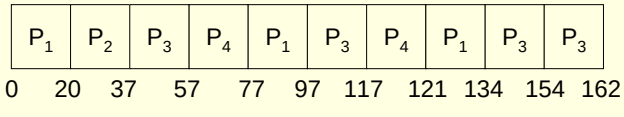
Round Robin (RR)

- Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance
  - *q* large ⇒ FIFO
  - *q* small ⇒ *q* must be large with respect to context switch, otherwise overhead is too high !!!

Example of RR, Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

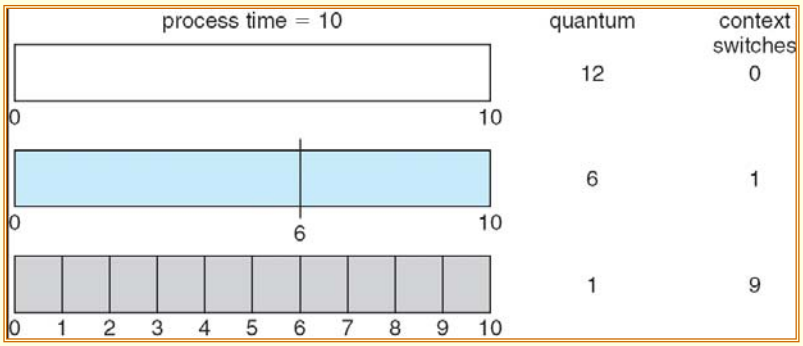
- The Gantt chart is:



- Typically, higher average turnaround than SJF, but better *response*

Time Quantum x Context Switch Time

- How a Smaller Time Quantum Increases Context Switches



Escalonamento CPU

- **Escalonamento por Prioridades**
  - Para compensar o excessivo tempo gasto no **estado de espera**, devemos atribuir alguma compensação aos processos **IO-Bound.** Isto pode ser feito através da variação da prioridade de execução associada a cada processo.
  - No **Escalonamento por Prioridades**, processos de maior prioridade são escalonados preferencialmente. Toda vez que um processo for para a fila de prontos com prioridade superior a do processo em execução, o SO deverá interromper o processo corrente, coloca-lo no estado de pronto e escalonar o processo de maior prioridade para execução. Esse mecanismo é definido como **preempção por prioridade.**

Escalonamento CPU

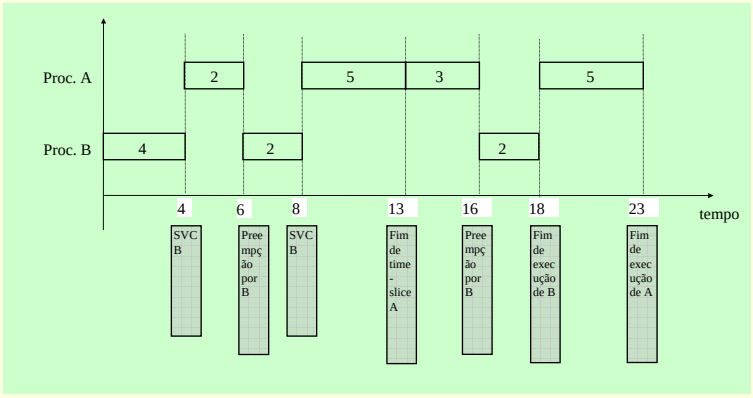
- **Escalonamento por Prioridades**
  - Assim como na **preempção por tempo** a **preempção por prioridade** é implementada mediante um clock, que interrompe o processador em determinados intervalos de tempo, para que a rotina de **Escalonamento de Curto Prazo** (**Escalonador ou Dispatcher**) reavalie as prioridades e, possivelmente, escalone outro processo.
  - A prioridade é uma característica do **contexto de SW** do processo, podendo ser **estática** ou **dinâmica.** A prioridade é dita **estática** quando não é modificada durante a existência do processo. Na prioridade **dinâmica** a prioridade do processo pode ser ajustada de acordo com o tipo de processamento realizado pelo processo e/ou pela carga do sistema.

Escalonamento CPU

- **Escalonamento por Prioridades**
  - Todo processo, ao sair do **estado de espera**, recebe um acréscimop à sua prioridade. Dessa forma, os processos **I/O Bound** terão mais chance de ser escalonados e, assim, compensar o tempo que passam no **estado de espera**. Obeserve que este procedimento não prejudica os processos **CPU Bound**, pois estes podem ser executados enquanto os processos **I/O Bound** esperam por algum evento.
  - Um problema potencial é que um processo pode sofrer um **adiamento indefinido** ou **starvation** quando sempre que ele estiver na fila de processos prontos aparecer outro processo de maior prioridade. A utilização de prioridade **dinâmica** tende a diminuir este problema.

Escalonamento CPU

- **Escalonamento por Prioridades**



Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem ≡ Starvation – low priority processes may never execute
- Solution ≡ Aging – as time progresses increase the priority of the process

Windows XP Priorities

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Linux Scheduling

- Two algorithms: time-sharing and real-time
- Time-sharing
  - Prioritized credit-based – process with most credits is scheduled next
  - Credit subtracted when timer interrupt occurs
  - When credit = 0, another process chosen
  - When all processes have credit = 0, recrediting occurs
    - Based on factors including priority and history
- Real-time
  - Soft real-time
  - Posix.1b compliant – two classes
    - FCFS and RR
    - Highest priority process always runs first

The Relationship Between Priorities and Time-slice length

numeric priority	relative priority		time quantum
0	highest	real-time tasks	200 ms
•			
•			
•			
99			
100		other tasks	
•			
•			
•			
140	lowest		10 ms

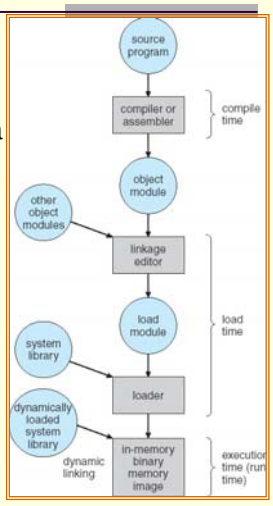
Memory Management

Gerente de Memória

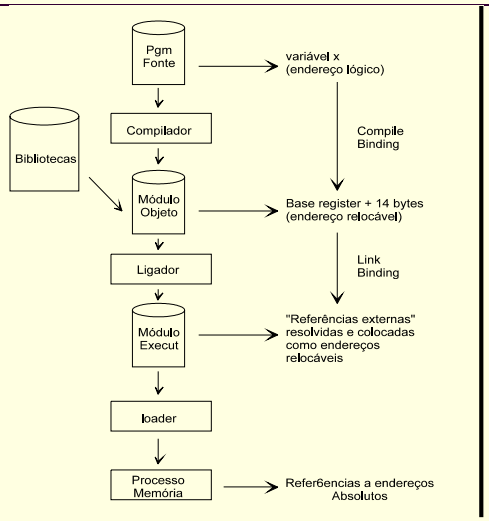
- Objetivos da Gerência de Memória
  - Tornar o mais eficiente possível o compartilhamento de memória entre os processos.
  - Impedir que um processo acesse área de memória que não lhe pertence.
  - Facilitar alocação de memória.
  - Recuperar a memória liberada pelos processos.

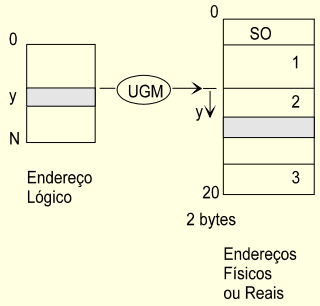
Binding of Instructions and Data to Memory

- **Address binding of instructions and data to memory addresses can happen at three different stages**
  - **Compile time**: If memory location known a priori, *absolute code* can be generated; must recompile code if starting location changes
  - **Load time**: Must generate *relocatable code* if memory location is not known at compile time
  - **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*).



Binding of Instructions and Data to Memory

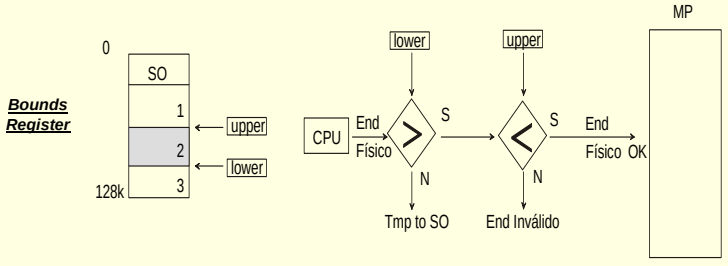




UGM - Unidade de Gerenciamento de Memória. HW para conversão End Lógico → End Real

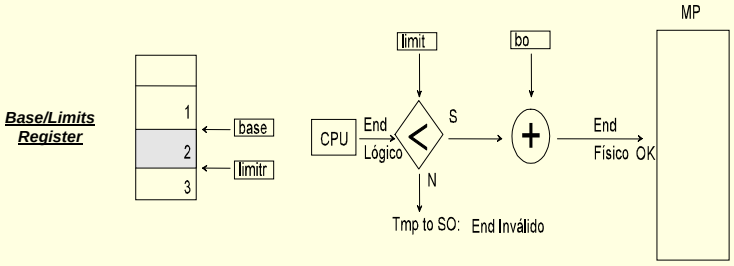
Static Relocation

- **Static Relocation** (gerada em tempo de carga do programa para execução).
  - A proteção, nesse esquema de alocação de memória, é feita com o uso de dois registradores – lower e upper



Dynamic Relocation

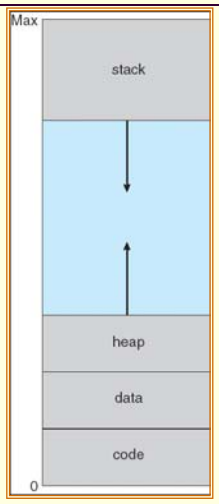
- **Dynamic Relocation**
  - Exige a soma do registrador base a cada endereço referendado o que representa um custo adicional a execução o qual pode ser compensado com técnicas de overlapping de instruções no processador.



Logical vs. Physical Address Space

- The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management
  - **Logical address** – generated by the CPU; also referred to as *virtual address*
  - **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

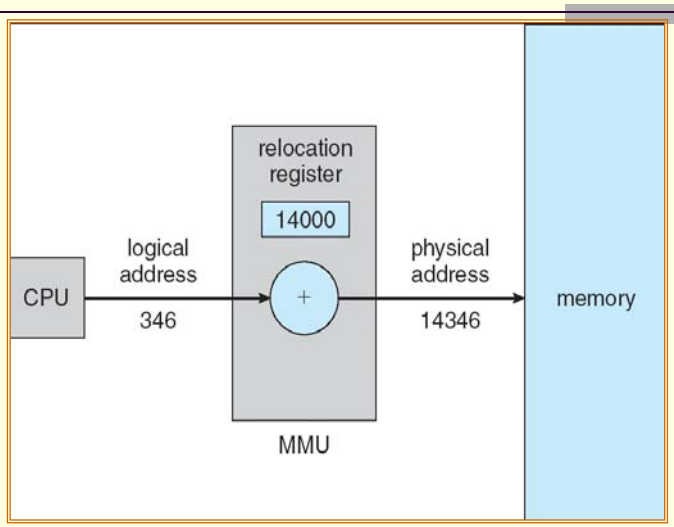
Virtual-address Space



Memory-Management Unit (MMU)

- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register



Dynamic Loading

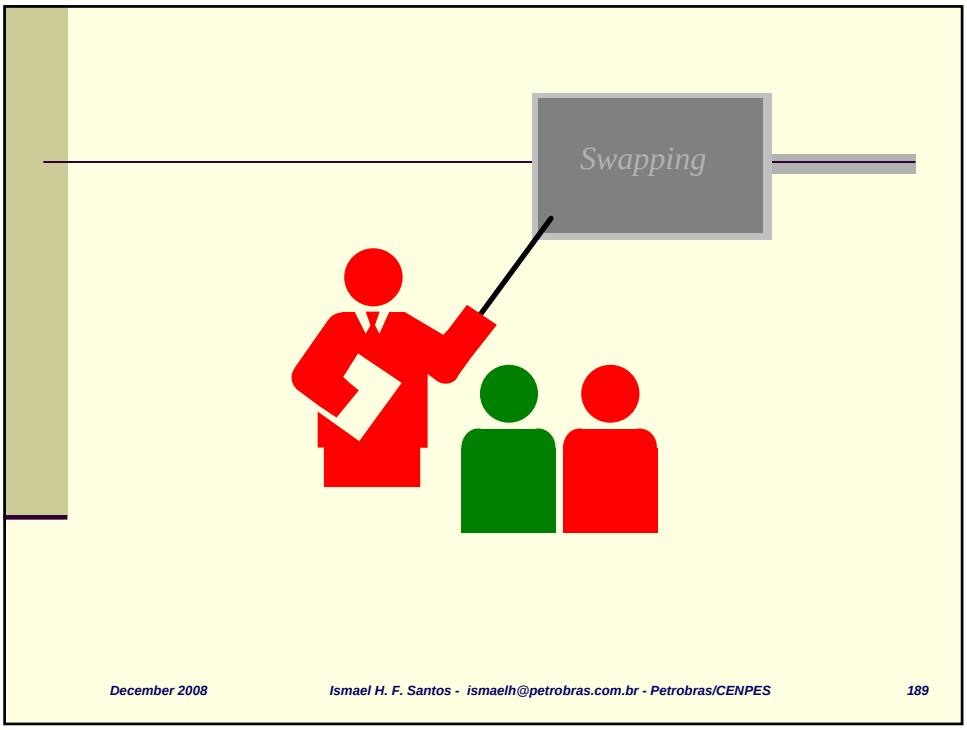
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

Static and Dynamic Linking

- A program whose necessary library functions are embedded directly in the program's executable binary file is *statically* linked to its libraries
- The main disadvantage of static linkage is that every program generated must contain copies of exactly the same common system library functions
- *Dynamic* linking is more efficient in terms of both physical memory and disk-space usage because it loads the system libraries into memory only once

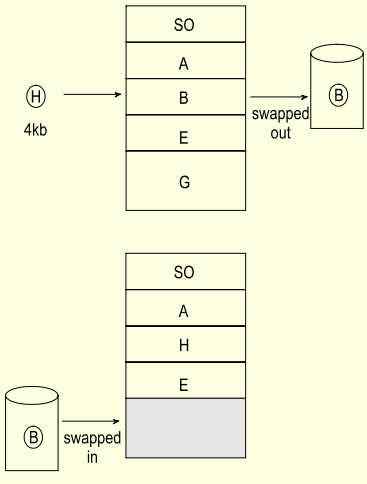
Dynamic Linking

- Linking postponed until execution time
- Small piece of code, *stub*, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries



Swapping

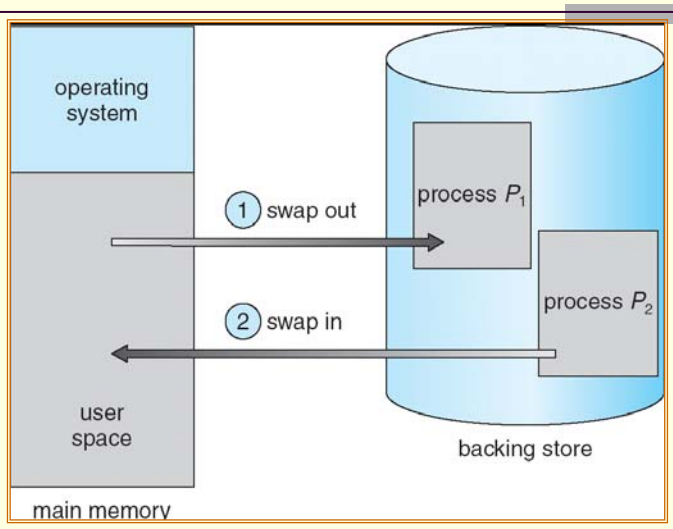
- **Swapping**
  - Técnica para permitir que programas que esperam por memória livre possam ser processados. Nesta situação, o SO escolhe um programa residente que é levado da MP para o disco (**swapped out**) retornando posteriormente para a MP (**swapped in**) como se nada tivesse ocorrido.



Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows)

Swapping



Swapping

- **Swapping**
  - Sempre que o **Escalonador** (**CPU scheduler**) decide executar um processo ele chama o **Dispatcher**. O **Dispatcher** verifica se o processo a ser executado está residente em memória, se não, ele remove (swap out) algum processo que esteja correntemente na MP e carrega (swap in) o processo escolhido para ser executado. Após isto o contexto do processo a ser executado é restaurado e o controle é passado ao processo que passa então a executar.
  - **Exemplo: Calcule o ΔTswap = ?**
    Tamanho do processo = 1 Mb
    Taxa de transferência = 100 Mbits/s
    Tempo de seek= 10 msec, Disco = 7200 rpm

Swapping (cont.)

7200 voltas → 1 minuto = 60 s

t 1volta *→ 60/7200 s*

Assumimos ***Tlatência = ½ t 1volta*** *= ½ (60/7200) = 1/240*

Ttransf disco-memória = Tseek + Tlatência + Ttransf =

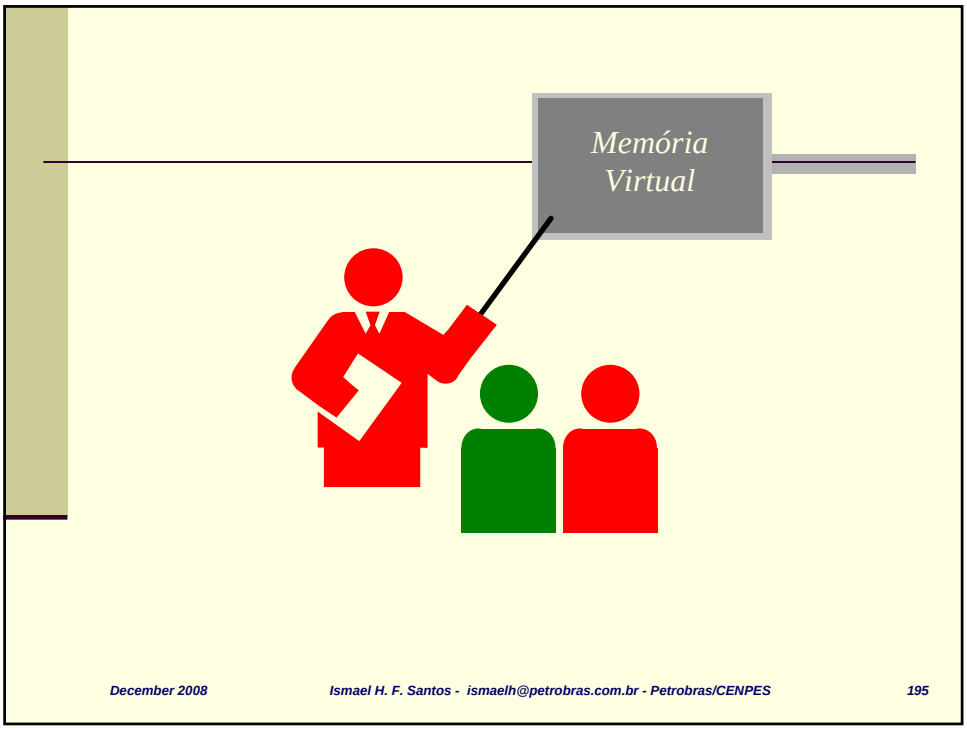
= 10 x 10 $^{-3}$ + 1/240 + (1Mb x 8)/100 Mbits /s =

= 0,01 + 0,042 + 8/100 = 0,01 + 0,042 + 0,08 = 0,132 s

Δ Tswap = 2 * Transf disco-memória *= 0,26 seg*

↓

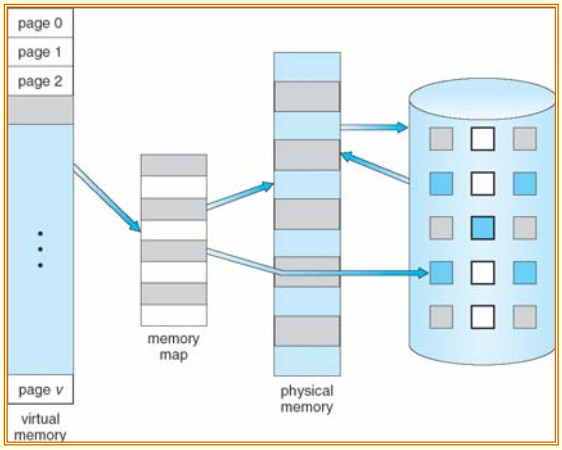
Swap in
Swap out



Background

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows address spaces to be shared by several processes.
  - Allows for more efficient process creation.
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

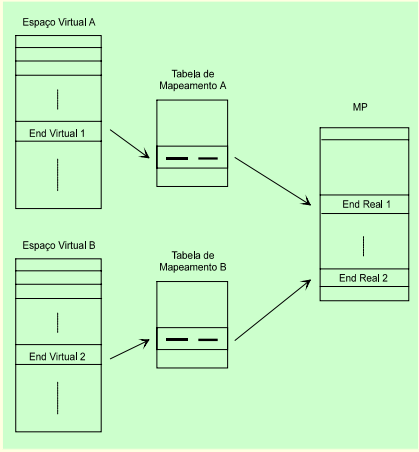
Virtual Memory That is Larger Than Physical Memory



Memória Virtual

- Cada processo tem o mesmo **EEV (espaço endereçamento virtual)** como se possuísse a sua própria MV.

- O **mecanismo de tradução** se encarrega, então, de manter as tabelas de mapeamento exclusivas para cada processo. Quando um programa está sendo executado o SO utiliza a tabela de mapeamento do processo, no qual o programa executa, para realizar a tradução.

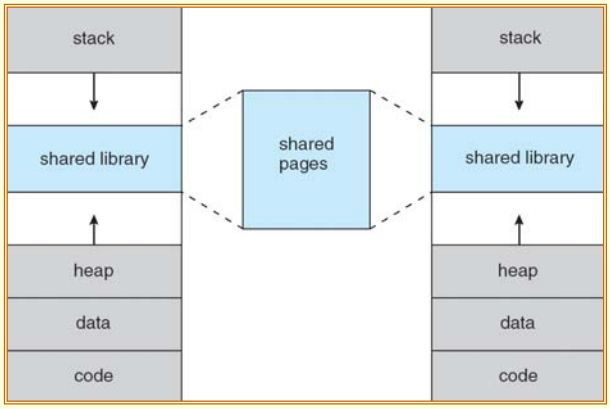


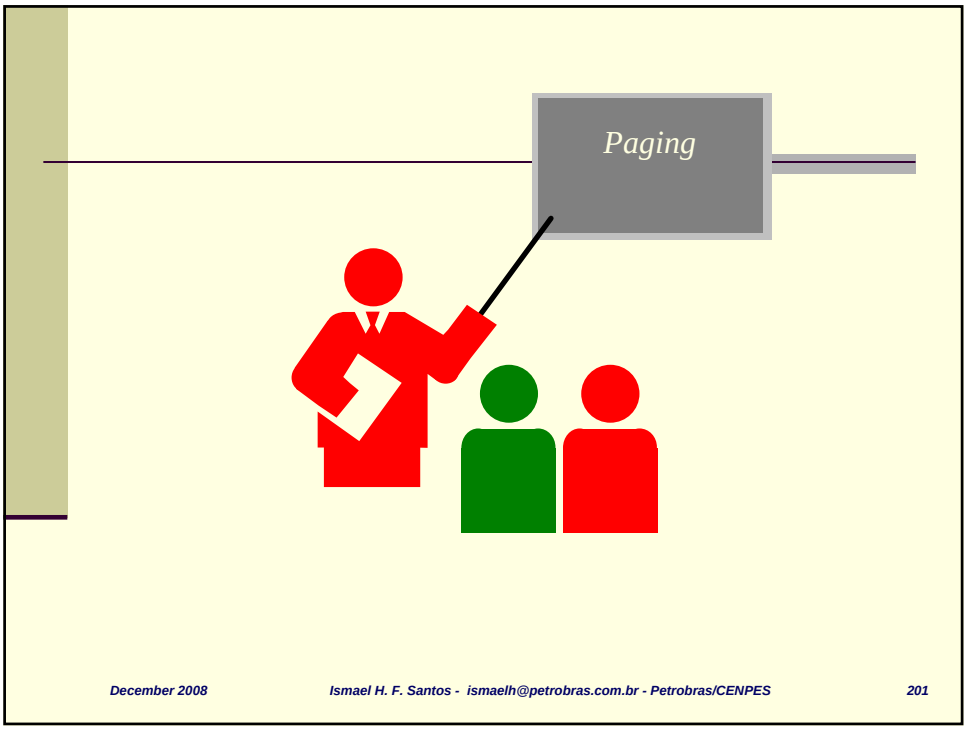
Memória Virtual

- A informação correspondente à posição inicial da tabela de mapeamento é em geral indicada por um registrador, chamado **PTBR** (**page table base register**) e faz parte do contexto do HW do processo.

- A **tabela de mapeamento** mapeia **blocos de informação** cujo tamanho determina o número de entradas necessário na tabela. **Quanto maior o bloco, menor número de entradas na tabela e, conseqüentemente, tabelas menores; entretanto blocos maiores aumentam o tempo de transferência do bloco entre MS e MP**. Existem SOs que trabalham apenas com blocos de mesmo tamanho (**páginas**), outros que utilizam blocos de tamanho diferente (**segmentos**) e, ainda há SOs que trabalham com os dois sistemas.

Shared Library Using Virtual Memory



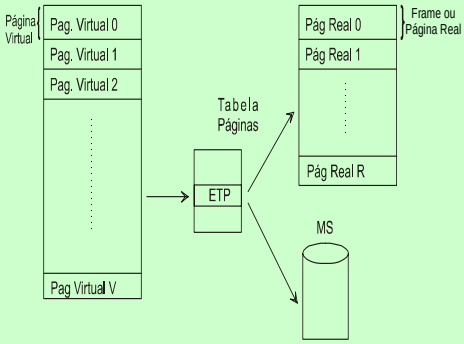


Paging

- Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available
- Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes)
- Divide logical memory into blocks of same size called **pages**.
- Keep track of all free frames
- To run a program of size *n* pages, need to find *n* free frames and load program. Set up a page table to translate logical to physical addresses

Tabela de Paginas

- **Paginação**
  - Nesta técnica o **EEV** e o EER são divididos em Blocos do mesmo tamanho, chamados **páginas.**
  - As **páginas** no **EEV** São chamadas **páginas Virtuais** e as **páginas** no **EER** são chamadas **páginas reais ou Frames.**

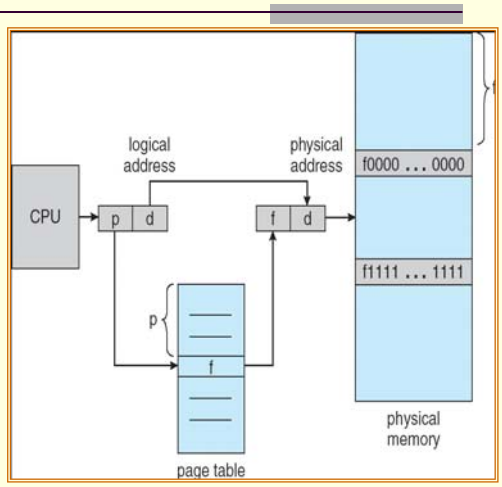


Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Address Translation Architecture

- Address generated by CPU is divided into:
  - Page number (p) – used as an index into a page table which contains base address of each page in physical memory
  - Page offset (d) – combined with base address to define the physical memory address that is sent to the memory unit

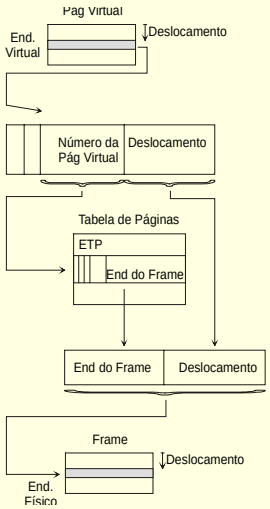


Paginação

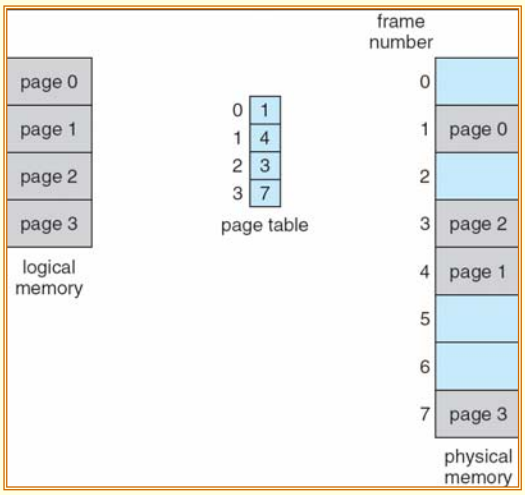
NPV = V div P
d = V mod P

Onde:
P - tamanho da página;
V - Endereço Virtual;
NPV - número de página Virtual;
d - deslocamento dentro da página;
div e mod são a divisão inteira e o resto da divisão.

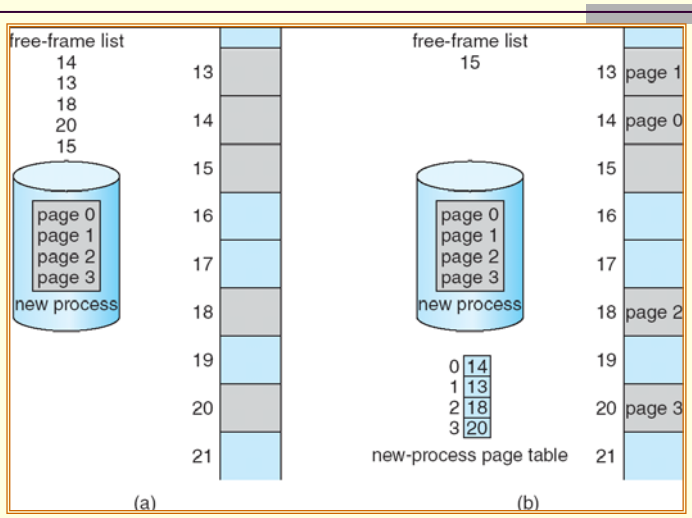
Exemplo: Pagina 66 apostila !



Paging Example



Free Frames



Implementation of Page Table

- Page table is kept in main memory
- *Page-table base register (*PTBR) points to the page table
- *Page-table length register* (PRLR) indicates size of the page table
- In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.
- The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

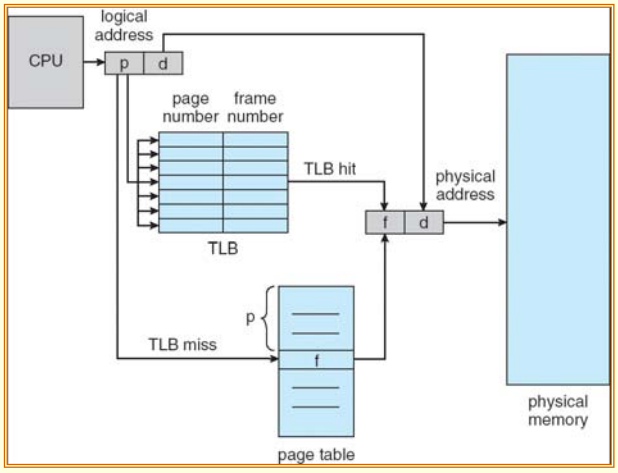
Associative Memory

- Associative memory – parallel search

Page #	Frame #

- Address translation (A´, A´´)
  - If A´ is in associative register, get frame # out
  - Otherwise get frame # from page table in memory

Paging Hardware With TLB



Effective Access Time

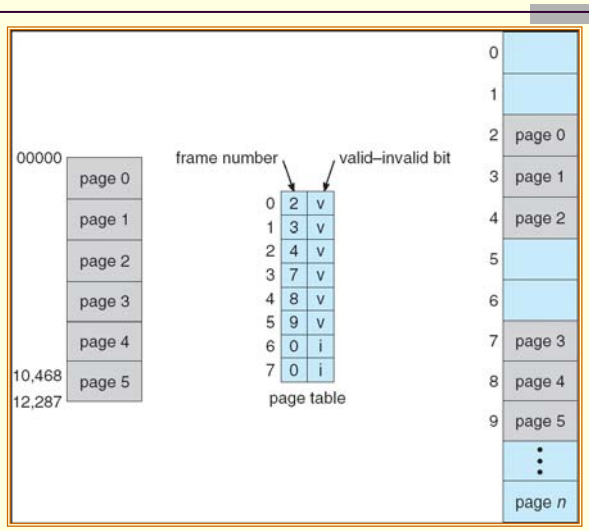
- Associative Lookup = ε time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ration related to number of associative registers
- Hit ratio = α
- **Effective Access Time** (EAT)

$$EAT = (1 + \varepsilon)\,\alpha + (2 + \varepsilon)(1 - \alpha)$$
$$= 2 + \varepsilon - \alpha <= 2 \quad (\text{já que } \alpha > \varepsilon\ !)$$

Memory Protection

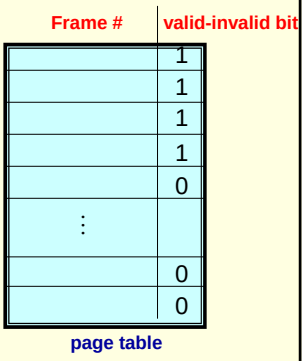
- Memory protection implemented by associating protection bit with each frame
- **Valid-invalid** bit attached to each entry in the page table:
  - "valid" indicates that the associated page is in the process' logical address space, and is thus a legal page
  - "invalid" indicates that the page is not in the process' logical address space

Valid/Invalid Bit In A Page Table

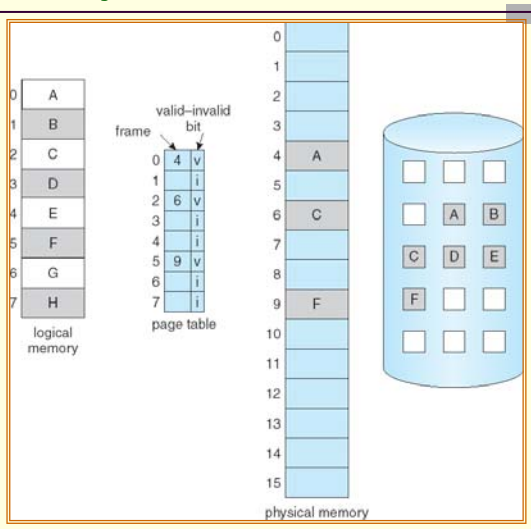


Valid-Invalid Bit

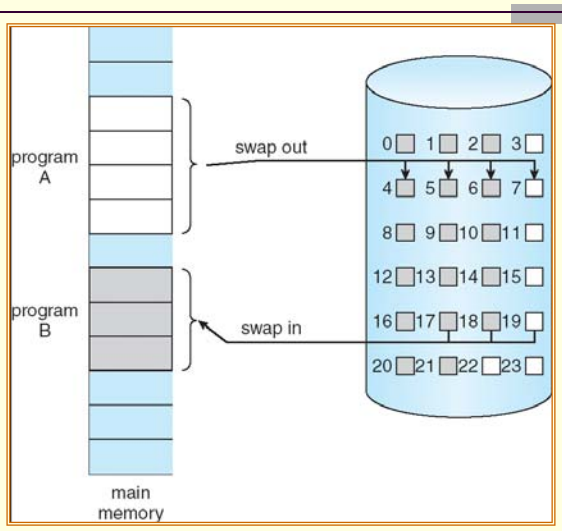
- With each page table entry a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault !



Page Table When Some Pages Are Not in Main Memory



Transfer of a Paged Memory to Contiguous Disk Space

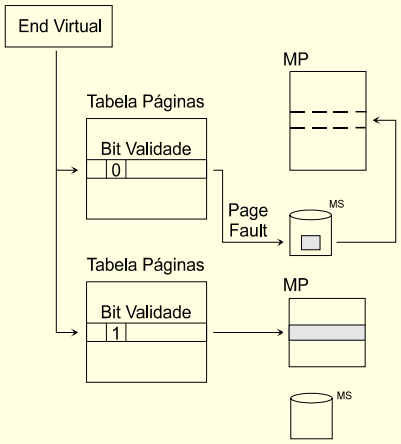


Page-Fault

- **Page-Fault**
  - Além da informação sobre a localização da página virtual, a **ETP** possui outras informações, dentre elas o **bit de validade** que indica se uma página está ou não na MP (valid bit ou reference bit).
  - Sempre que o processo faz referência a um endereço virtual o SO verifica se a página que contém o endereço referenciado está ou não na MP. Caso não esteja, o SO acusa a ocorrência de um **page-fault** (interrupção) e uma página é então transferida da MS para a MP.

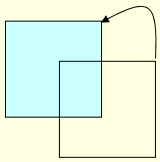
Page-Fault (cont.)

- Page-Fault
  - As páginas dos processos são transferidas da MS para a MP, apenas quando são referenciadas. Este mecanismo, é chamado **paginação por demanda (demand paging)** e é conveniente, na medida em que leva para a MP apenas as páginas realmente necessárias para a execução do programa.

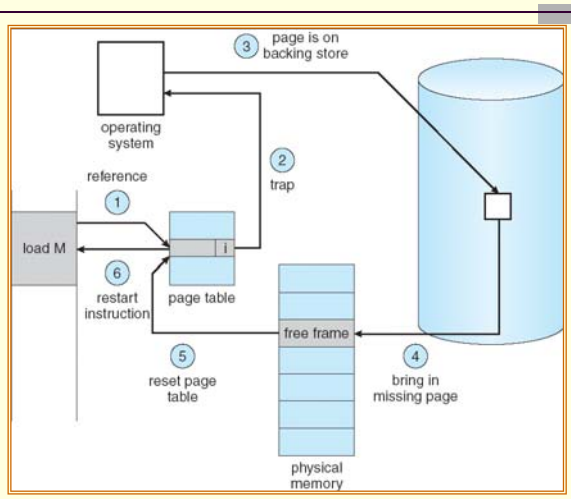


Page Fault

- If there is ever a reference to a page, first reference will trap to OS ⇒ page fault
- OS looks at another table to decide:
  - Invalid reference ⇒ abort.
  - Just not in memory.
- Get empty frame.
- Swap page into frame.
- Reset tables, validation bit = 1.
- Restart instruction: Least Recently Used
  - block move
  - auto increment/decrement location



Steps in Handling a Page Fault



Proteção

- **Proteção em Paginação**
  - O **mecanismo de proteção** para sistemas paginados é feito através de bits de proteção associados a cada página (na ETP). Em geral um bit é reservado para identificar se a **página é read/write ou read only**. Uma tentativa de se escrever em uma página RO ocasiona uma **interrupção chamada violação de memória (memory protection violation).**

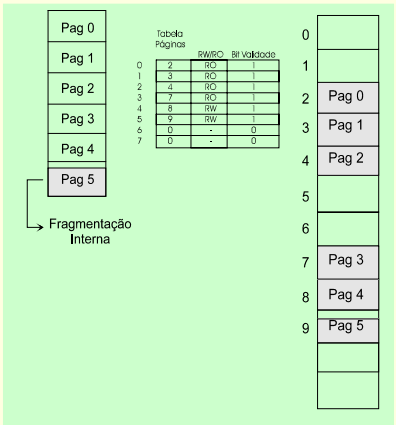
  • A arquitetura da máquina define o range de endereços válidos para um programa. Por exemplo uma máquina com MAR de 16 bits pode gerar endereços virtuais entre 0 e 65535. Com os **registradores de fronteira** (barrier registers) ou **base e limite** (base limit register) nós podemos trapear endereços gerados erroneamente por programas.

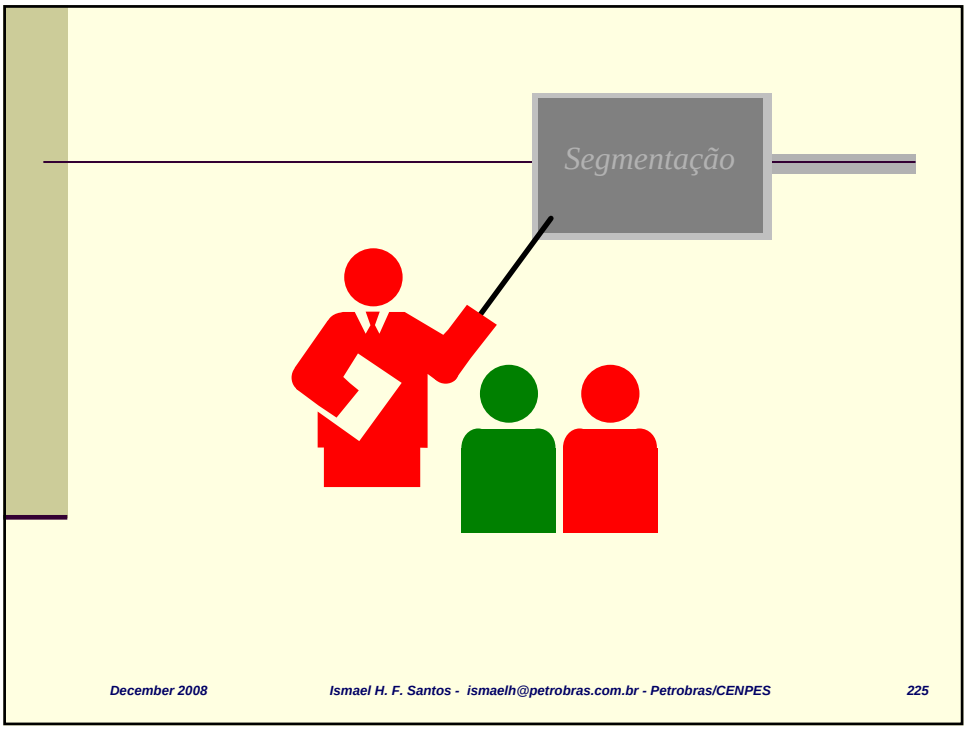
Fragmentação

- Fragmentação
  - **Fragmentação** também está presente em **sistemas paginados**, só que em menor escala, se comparada com a de outras organizações já vistas. A fragmentação só é encontrada, realmente, na última página, quando o código não a ocupa por completo. Maior ou menor fragmentação depende do tamanho da página.
  - O tamanho da página influencia outros fatores tais como:
    - **tamanho das tabelas de mapeamento;**
    - **taxa de paginação, expressa pelo número de page-faults do sistema por unidade de tempo;**
    - **percentual de utilização da MP**

Fragmentação (cont.)

- **Fragmentação**
  - Um problema colateral gerado pela **fragmentação interna** é que referências a pág 5 (conforme figura) mesmo que além da área definida pelo programa não gerarão **violação de endereço!**



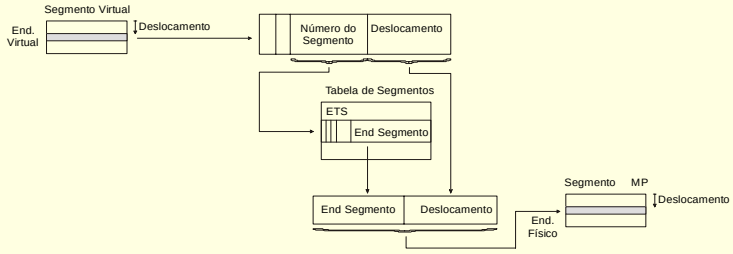


Segmentação

- **Segmentação**
  - Técnica de gerência de MP, onde os programas são divididos logicamente em blocos (segmento de dados, código, stack). Estes blocos têm tamanhos diferentes e são chamados **segmentos**, cada um com seu próprio espaço de endereçamento.
  - A diferença em relação à **paginação** é que a **segmentação** permite uma relação entre a lógica do programa e sua divisão na MP (porque?).

Segmentação

- **Segmentação(cont.)**
  - O mecanismo de mapeamento é semelhante ao de **paginação**. Os **segmentos** são mapeados através de tabelas de mapeamento e os endereços são compostos pelo número do segmento e um deslocamento dentro do segmento.



Alocação de Segmentos

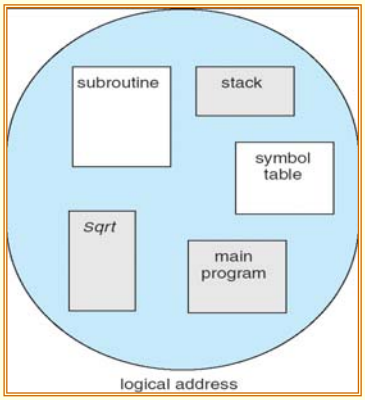
- O SO mantém uma tabela com as áreas livres e ocupadas da MP. Quando um processo é carregado para MP, o SO localiza um espaço livre que o acomode. As estratégias para escolha da área livre podem ser as mesmas do MVT, ou seja, **BEST-FIT, WORST-FIT, FIRST-FIT.**

- Na **segmentação**, apenas os segmentos referenciados são transferidos da MS para MP. Logo, para ser mais eficiente, os programas devem estar bem modularizados.

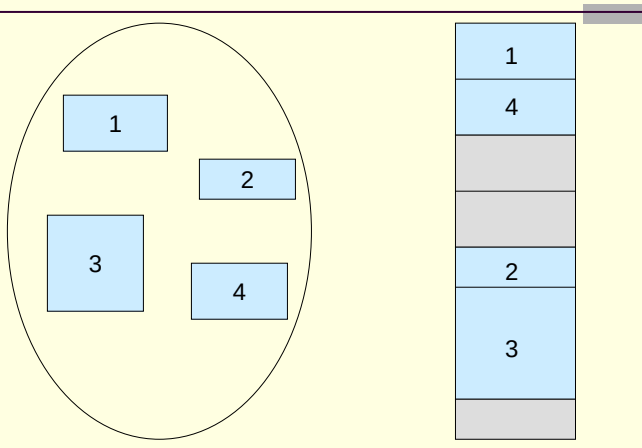
- O problema de **fragmentação** também ocorre nesse modelo, quando as áreas livres são tão pequenas, que não acomodam nenhum segmento que necessite ser carregado.

Segmentation

- Memory-management scheme that supports user view of memory
  - A program is a collection of segments. A segment is a logical unit such as:
    main program,
    procedure,
    function,
    method,
    object,
    local variables, global variables,
    common block,
    stack,
    symbol table, arrays



Logical View of Segmentation



Segmentation Architecture

- Logical address consists of a two tuple:
  <segment-number, offset>,
- **Segment table** – maps two-dimensional physical addresses; each table entry has:
  - base – contains the starting physical address where the segments reside in memory
  - *limit* – specifies the length of the segment
- *Segment-table base register (STBR)* points to the segment table's location in memory
- *Segment-table length register (STLR)* indicates number of segments used by a program;
- segment number *s* is legal if *s* < STLR

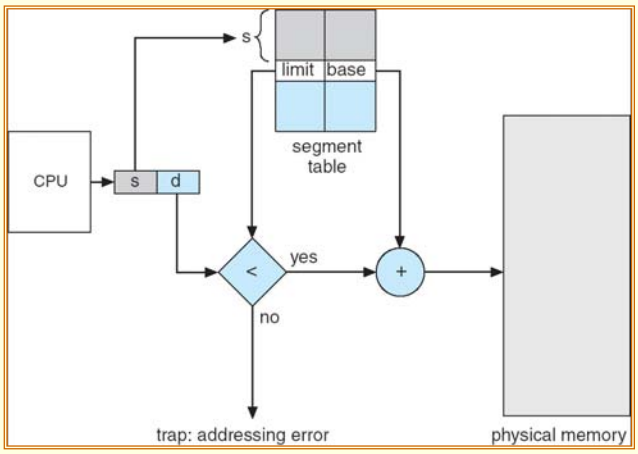
Segmentation Architecture (Cont.)

- **Relocation**.
  - dynamic
  - by segment table
- **Sharing**.
  - shared segments
  - same segment number
- **Allocation**.
  - first fit/best fit
  - external fragmentation

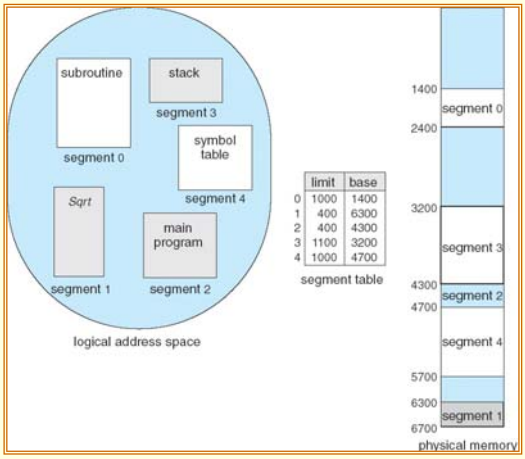
Segmentation Architecture (Cont.)

- Protection. With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level
- Since segments vary in length, memory allocation is a dynamic storage-allocation problem
- A segmentation example is shown in the following diagram

Address Translation Architecture



Example of Segmentation



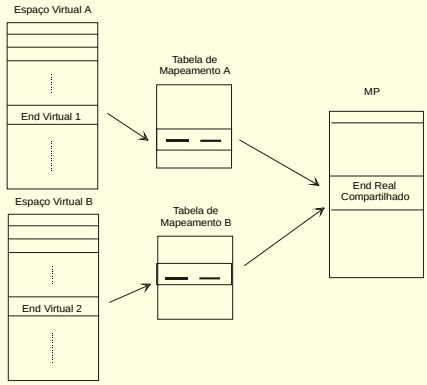
Compartilhamento de Segmentos

- **Compartilhamento de Memória**
  - Em sistemas multiprogramáveis, é comum usuários utilizarem certos programas simultaneamente (**código reentrante**), o que evita que várias cópias de um mesmo programa ocupem a memória desnecessariamente. Exemplos são os utilitários do sistema, como **compiladores**, **editores de texto** ou, mesmo, algumas aplicações de usuários.

  - Para compartilharmos código e/ou dados entre vários processos basta que as entradas das **páginas/segmentos** apontem para os mesmos frames/segmentos na memória principal. Dessa forma, é possível reduzir-se o número de programas na memória e aumentar o número de usuários compartilhando o mesmo recurso conforme mostra a figura.

Compartilhamento de Segmentos

- O **compartilhamento** de **segmentos** é mais simples de implementar do que o de **páginas**. O compartilhamento de estruturas de dados dinâmicas na **paginação** implica alocação de novas páginas e, consequentemente, o ajuste das tabelas de mapeamento, na **segmentação**, as tabelas devem ter ajustado apenas o tamanho do segmento.



Sharing of Segments