

Sistemas Operacionais

parte I

Professor: Emerson Rogério de Oliveira Jr.

O material apresentado aqui (conjunto de slides) foi fortemente baseado nos slides de aula da disciplina de Sistemas Operacionais, da UFRGS, elaborado pelos professores Rômulo Silva de Oliveira, Alexandre da Silva Caríssimi e Simão Sirineo Toscani, de acordo com o livro destes mesmos professores

OLIVEIRA, R. S.; CARÍSSIMI, A. S. e TOSCANI, S. S. Sistemas Operacionais, 1^a ed. Ed. Sagra-Luzzatto. 2001.

Sumário

- ❑ Introdução aos Sistemas Operacionais
- ❑ Multiprogramação
- ❑ Programação concorrente
- ❑ Gerência do processador
- ❑ Gerência da memória
- ❑ Memória virtual

Introdução aos Sistemas Operacionais

Conceitos Iniciais

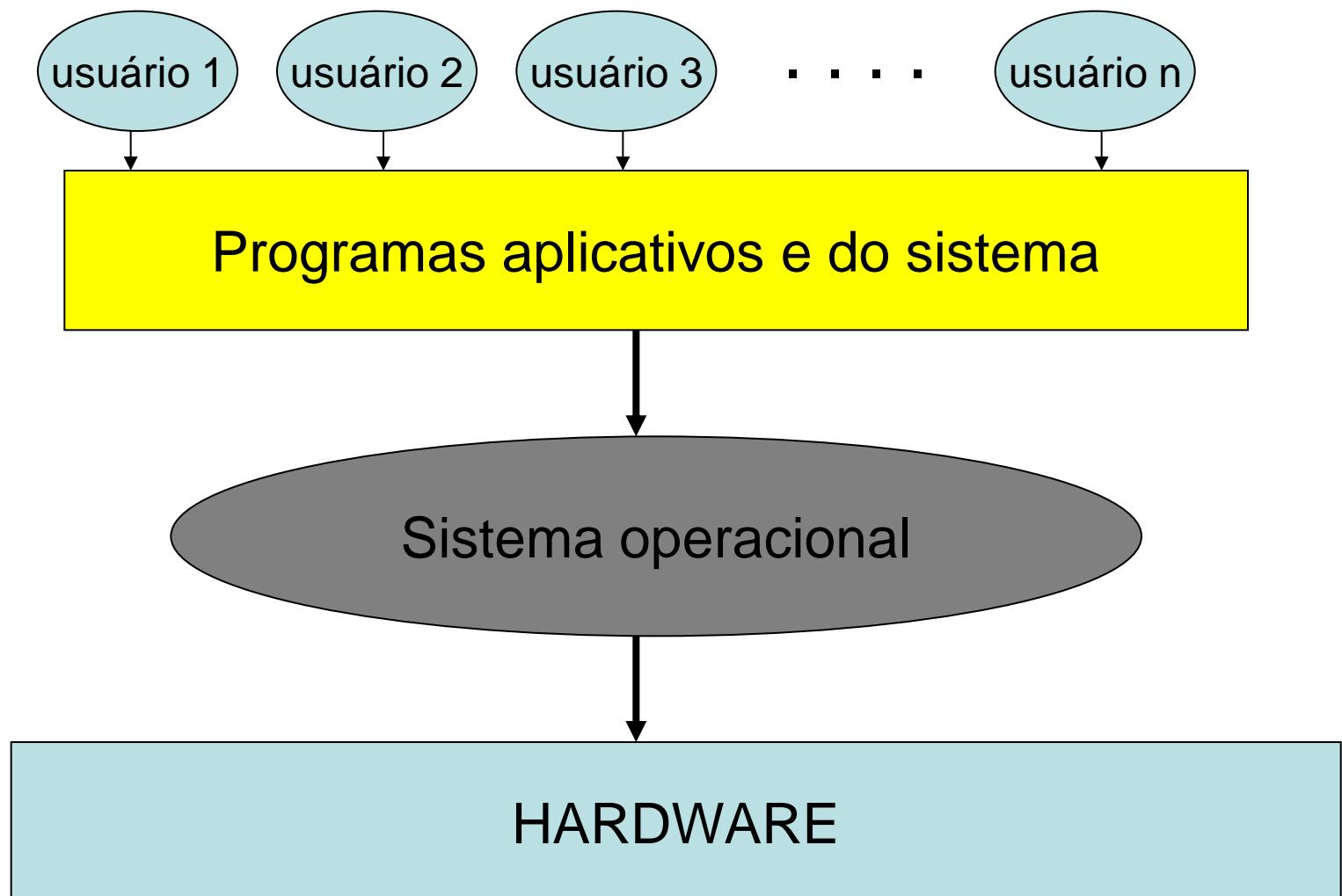
- ❑ Hardware: CPU, memória, HD
- ❑ Software: programas em geral
 - Aplicativos: desenvolvidos para atender à uma necessidade do usuário (contas a pagar, folha de pagamento.)
 - De sistema: usados para o desenvolvimento dos aplicativos (compilador, editor de textos)
- ❑ Programa: qualquer conjunto de instruções.
- ❑ Processo: programa em estado de execução.

Conceitos Iniciais

□ Sistema operacional:

- Década de 40: não existia sistema operacional.
- Década de 50: surge o primeiro sistema operacional para rodar em um IBM 701
- Década de 60: “software que controla o hardware”.
- Atualmente:
 - “software que habilita as aplicações a interagir com o hardware do computador” [Deitel, 2005]
 - “camada de software colocada entre o hardware e os programas que executam tarefas para os usuários” [OLIVEIRA, 2001]
 - “programas que gerenciam as partes de um sistema complexo” [TANENBAUM, 2003]

Relação usuário x hardware



Serviços oferecidos pelos sistemas operacionais

- Alguns serviços são: [OLIVEIRA, 2001]
 - Criação de programas
 - compiladores, editores, depuradores.
 - Execução dos programas
 - Acesso a dispositivos de E/S
 - Controle de acesso a arquivos
 - Acesso a recursos de sistema
 - Detecção de erros
 - Erros na memória, falha em dispositivo de E/S

Chamadas de sistema

- ❑ Maneira pela qual os programas solicitam serviços ao sistema operacional.
- ❑ Serviços executados pelo núcleo (*kernel*)
 - Gerência do processador
 - Gerência da memória
 - Gerência de arquivos
 - Gerência de E/S

Programas de sistema

- Programas executados fora do *kernel* do sistema operacional (utilitários)
- Implementam tarefas básicas
 - Compiladores, *assemblers*, ligadores
 - Interpretador de comandos (ativado sempre que inicia-se uma sessão de trabalho no sistema operacional)

Sistema operacional na visão de projeto

- ❑ Forma pela qual o sistema operacional implementa os serviços
- ❑ Sistema operacional ativado por eventos
 - Chamadas de sistema
 - Interrupção
- ❑ A interrupção é essencial para a construção de sistemas operacionais
 - Sinaliza término e ocorrência de eventos
 - Exceções
 - Auxilia atividades de gerência

Histórico dos sistemas operacionais

- ❑ Inicialmente (anos 40)
 - somente hardware
 - máquinas grandes com uma console
 - o programador operava o sistema
 - devia-se reservar horário para uso do sistema

- ❑ Problemas
 - não usar o tempo todo
 - o tempo ser insuficiente

Histórico dos sistemas operacionais

❑ Evolução

- desenvolvimento de softwares (compiladores, montadores etc)
- Novos hardwares foram desenvolvidos - *drivers* de dispositivos

❑ Consequências

- tornou o sistema difícil de ser operado
- tempo de execução de um programa muito longo

Histórico dos sistemas operacionais

□ Sistema em lote – *batch*

- *Job* – programa a ser compilado e executado, junto com os seus dados de entrada, tudo em cartões perfurados. A passagem de um job para outro é feita manualmente.

□ Monitor residente

- Sequenciamento automático de *jobs* – primeiro sistema operacional
- Monitor residente – fica permanentemente na memória
- Existem cartões de controle – quando termina um job, outro é iniciado

❑ Multiprogramação

- tenta fazer com que a CPU esteja sempre ocupada
- muitos programas na memória ao mesmo tempo
- Sistemas *timesharing*
- Usuários possuem um terminal, podendo interagir com o programa
- Divisão do tempo do processador entre usuários
 - o tempo de resposta torna-se importante

Tipos de sistemas operacionais

- Sistemas operacionais monousuário
 - Um único usuário pode usar por vez
 - MS-DOS, Windows 3.x, Windows 9x, Millenium

- Sistemas operacionais multiusuário
 - Suporta várias sessões de usuários em um mesmo computador
 - Windows NT(2000), unix, linux

Tipos de sistemas operacionais

- Sistemas operacionais monotarefa
 - Executam apenas uma tarefa por vez
 - MS-DOS

- Sistemas operacionais multitarefa
 - Executam várias tarefas ao mesmo tempo
 - Sistemas multitarefa cooperativa – windows 3.x, Windows 9x (aplicativos 16 bits)
 - Sistemas multitarefa preemptiva: Windows NT, OS/2, unix, Windows 9x (aplicativos 32 bits)

Tipos de sistemas operacionais

- Sistemas operacionais distribuídos
 - Distribuem a realização de uma tarefa entre vários computadores.
 - Transparência na existência de várias máquinas
 - Compartilhamento de recursos
 - Balanceamento de carga
 - Aumento da confiabilidade

Tipos de sistemas operacionais

- Sistemas operacionais de tempo real
 - Usado para procedimentos que devem responder dentro de um certo intervalo de tempo
 - Noção de tempo real é dependente da aplicação: milissegundos, minutos, horas

- Sistemas operacionais paralelos
 - Possuem mais de um processador
 - Aumento do *throughput*
 - Aspectos econômicos
 - Cada processador pode executar uma determinada tarefa

Multiprogramação

Multiprogramação

- ❑ Objetivo: tornar mais eficiente o aproveitamento dos recursos do computador
- ❑ Execução simultânea de vários programas
 - Diversos programas estão na memória
 - Conceitos importantes são processo, interrupção e proteção entre processos
- ❑ Programa
 - Entidade estática e permanente
 - Sequência de instruções
 - Passivo, sob o ponto de vista do sistema operacional

❑ Processo

- Entidade dinâmica
- Altera o seu estado à medida que avança na execução
- Composto por código, dados e contexto
- Identificado por um número único (PID)
- Abstração que representa um programa em execução
- Forma pela qual o sistema operacional atende um programa e possibilita sua execução

Ciclos de um processo

- ❑ Processos são:
 - Criados
 - Destruídos
- ❑ Processos executam
 - Programas do usuário
 - Programas do sistema (*daemons*)
- ❑ Ciclos básicos de um processo
 - Ciclo de processador: tempo de uso de CPU
 - Ciclo de E/S: tempo em espera pelo atendimento de uma requisição de E/S

Ciclos de um processo

- ❑ Primeiro ciclo é sempre de processador
 - Chamada de sistema ($\text{CPU} \rightarrow \text{E/S}$)
 - Interrupção ($\text{CPU} \rightarrow \text{E/S}$ ou $\text{E/S} \rightarrow \text{CPU}$)
 - Ocorrência de evento ($\text{E/S} \rightarrow \text{CPU}$)

- ❑ Processos CPU bound e I/O bound

Relação entre processos

❑ Processos independentes

- Não tem relação uns com os outros

❑ Grupo de processos

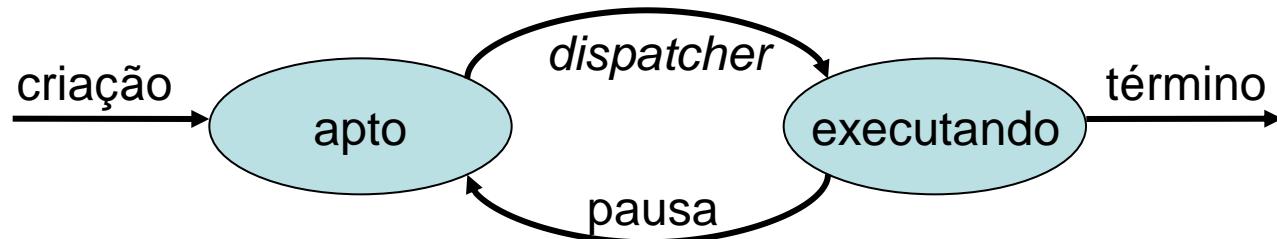
- Apresentam relação
- Podem compartilhar recursos deve-se definir uma hierarquia criador-criado, representado através de uma árvore

❑ Após criado, o processo deve entrar na fila pronto (*ready queue*)

Relação entre processos

❑ Modelo com dois estados

- Tem-se uma fila com os processos aptos a executar
- Aguarda o escalonador (*dispatcher*) para atribuir o processador a um processo da fila
- Modelo impróprio para tratar processos bloqueados



Criação de processos

- ❑ Submissão de um *job*
- ❑ Logon de usuários
- ❑ Processo criado para execução de um determinado serviço (*daemons*)
- ❑ Processo criado a partir de um processo já existente (*spawn*)

Término de processos

- ❑ Em circunstâncias especiais um processo poderá acabar antes de executar seu último comando
 - Um processo pode causar a morte de outro com uso de instruções do tipo *kill ID*
 - Quem mata um processo é o pai dele, a partir de sua identificação (PID)

- ❑ Por quê matar um processo?
 - O filho excedeu o uso de recursos
 - O trabalho atribuído ao filho não é mais necessário
 - O filho de encontra em “LOOP”

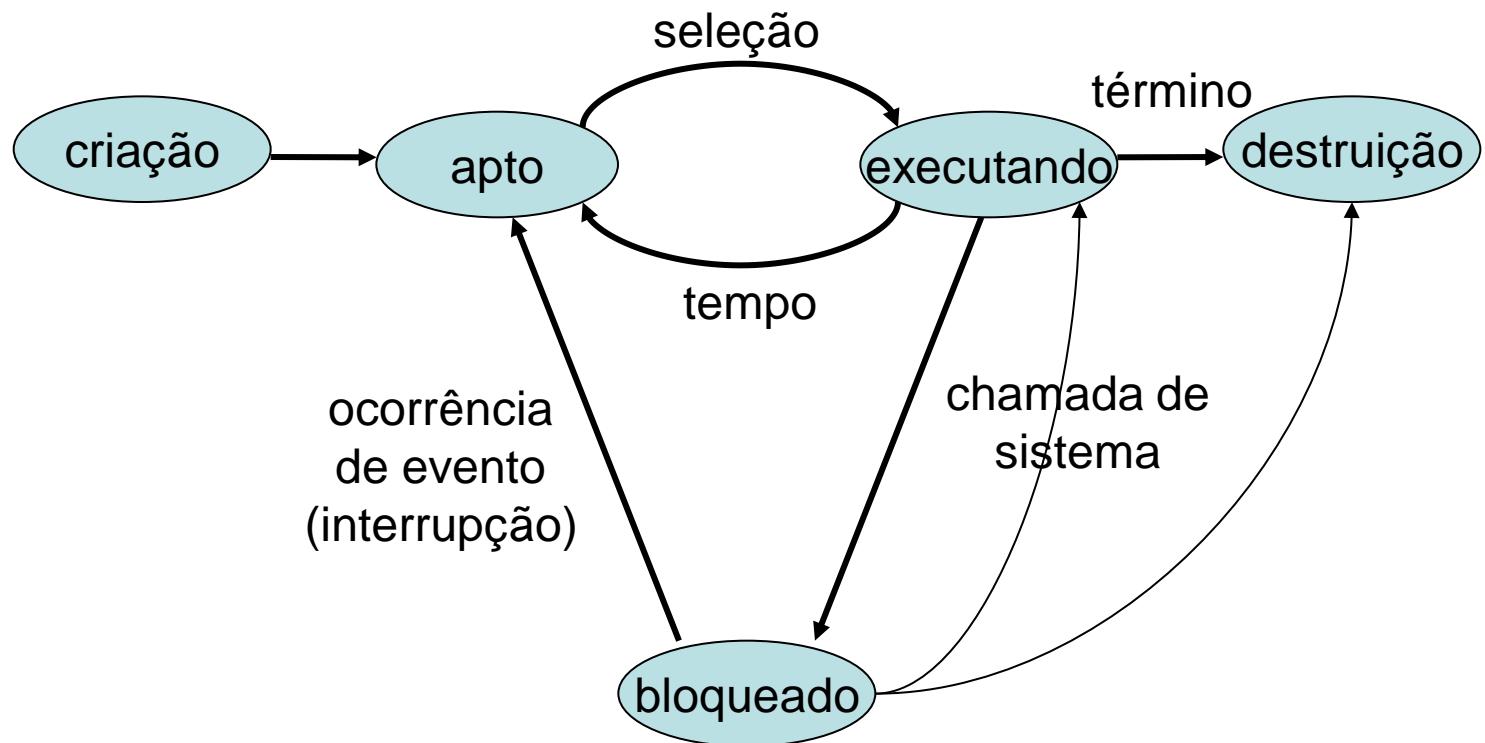
Modelo a 5 estados

- ❑ Estados:

- Executando (*running*)
- Apto (*ready*)
- Bloqueado (*blocked*)
- Criação (*new*)
- Destruição (*exit*)

- ❑ Associam-se eventos na transição de um estado a outro

Modelo a 5 estados



Processos suspensos

- ❑ Processador é mais rápido que operações de E/S
 - Liberar memória ocupada por estes processos
 - Estado bloqueado pode ser com processo no disco ou na memória
 - Necessidade de novos estados
 - Bloqueado, suspenso (*blocked, suspend*)
 - Apto, suspenso (*ready, suspend*)

Razões para suspender processo

- ❑ *Swapping*

- Sistema operacional necessita liberar memória para executar um novo processo

- ❑ Solicitação do usuário

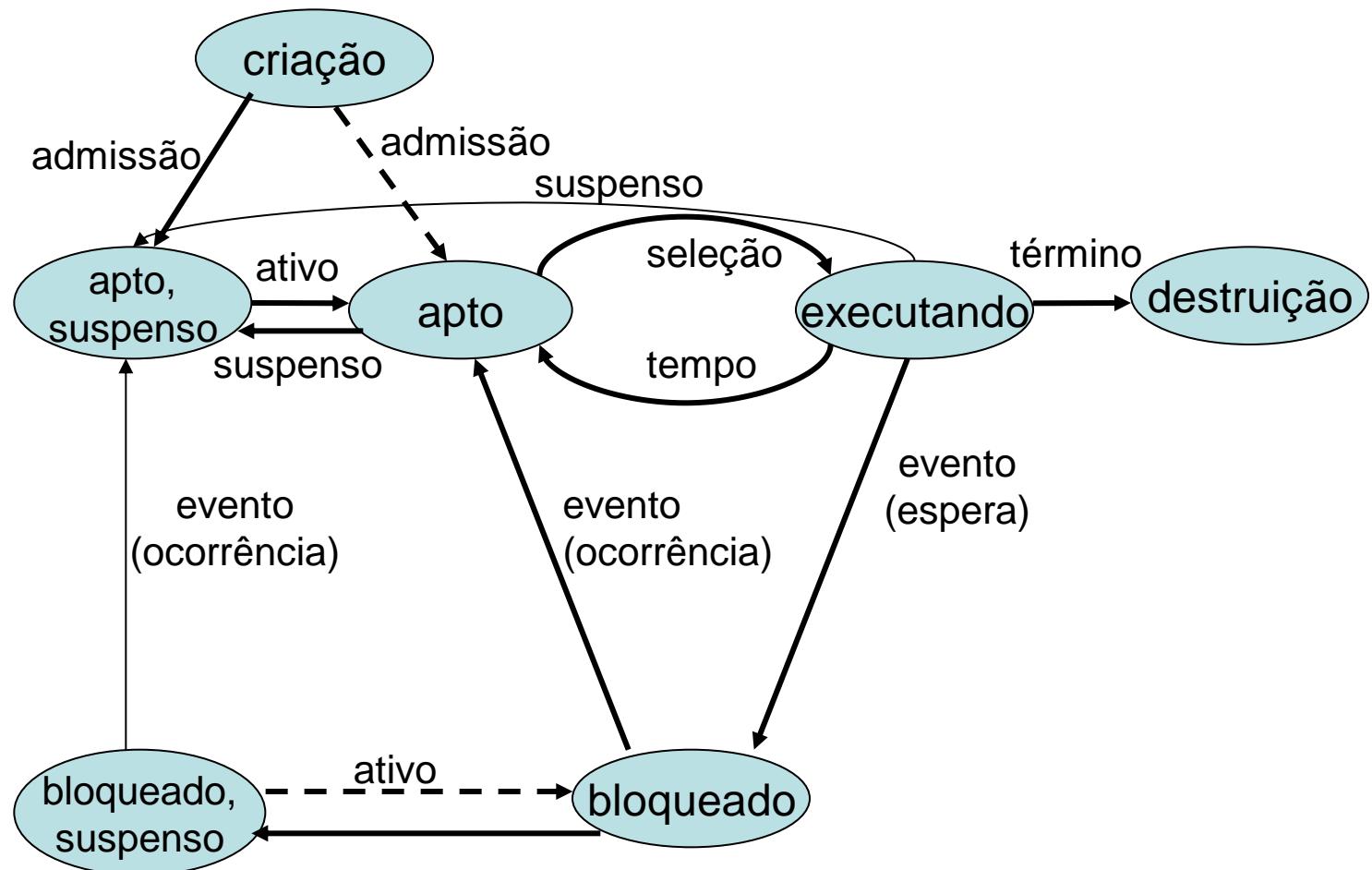
- ❑ Temporização

- Interromper um processo por um certo tempo

- ❑ Processo suspender outro processo

- Para sincronização

Diagrama de estados de processos



Interrupção

- ❑ Multiprogramação necessita de apoio do hardware dos processadores
 - Interrupção
 - Modo de operação protegido (supervisor)
 - Modo de operação do usuário
 - Proteção de periféricos, memória e processador

- ❑ A interrupção sinaliza ao processador para tratar um evento
 - Chama a rotina de interrupção

Interrupção

- ❑ Tipos de interrupção
 - Hardware: ocorrência de evento externo
 - Software: execução de *traps*
 - Exceção: erros de *underflow* e *overflow*
- ❑ Existe um vetor de interrupções
- ❑ Problema: buscar o vetor de interrupções do pentium dual core

Programação concorrente

Conceitos iniciais

- ❑ Programa executado por apenas um processo é chamado de programa sequencial
- ❑ Programa executado por mais de um processo é chamado programa concorrente
 - Existem vários fluxos de controle
 - Necessidade de trocas de informações entre os processos – sincronização
 - Processos disputam recursos comuns
 - Variáveis, periféricos, ...
 - Um processo é dito cooperante quando é capaz de afetar, ou ser afetado, pela execução de outro processo.

Por quê programação concorrente?

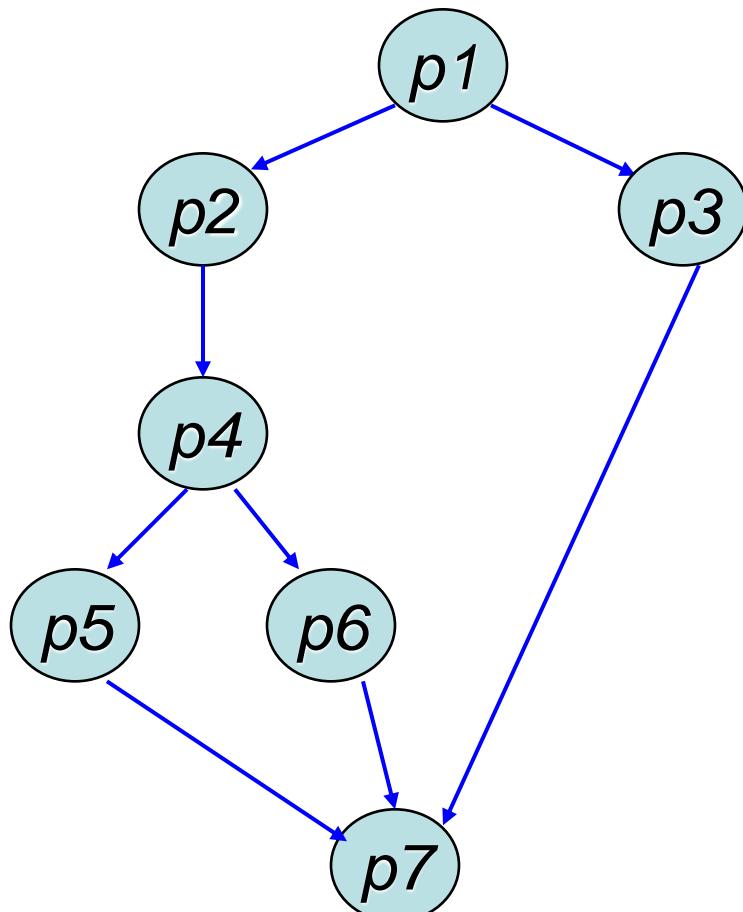
- ❑ Aumento de desempenho
 - Sobreposição de operações de E/S com processamento
- ❑ Facilidade de desenvolvimento de aplicações que possuem um paralelismo intrínsico
- ❑ Desvantagens
 - Programação complexa
 - Diferenças de velocidades de processamentos
 - Dificuldade na depuração

Notações para expressar paralelismo

❑ Fork / join

- Também conhecida como *fork/wait*
- utiliza um grafo de precedência para a construção “*fork*”
- “Um grafo de precedência é um grafo sem ciclos, dirigido, cujos nodos correspondem a um comando individual”
- “Uma aresta de um nodo S_i para um nodo S_j significa que o comando S_j somente poderá ser executado após o comando S_i ter completada sua execução”

Fork/Join



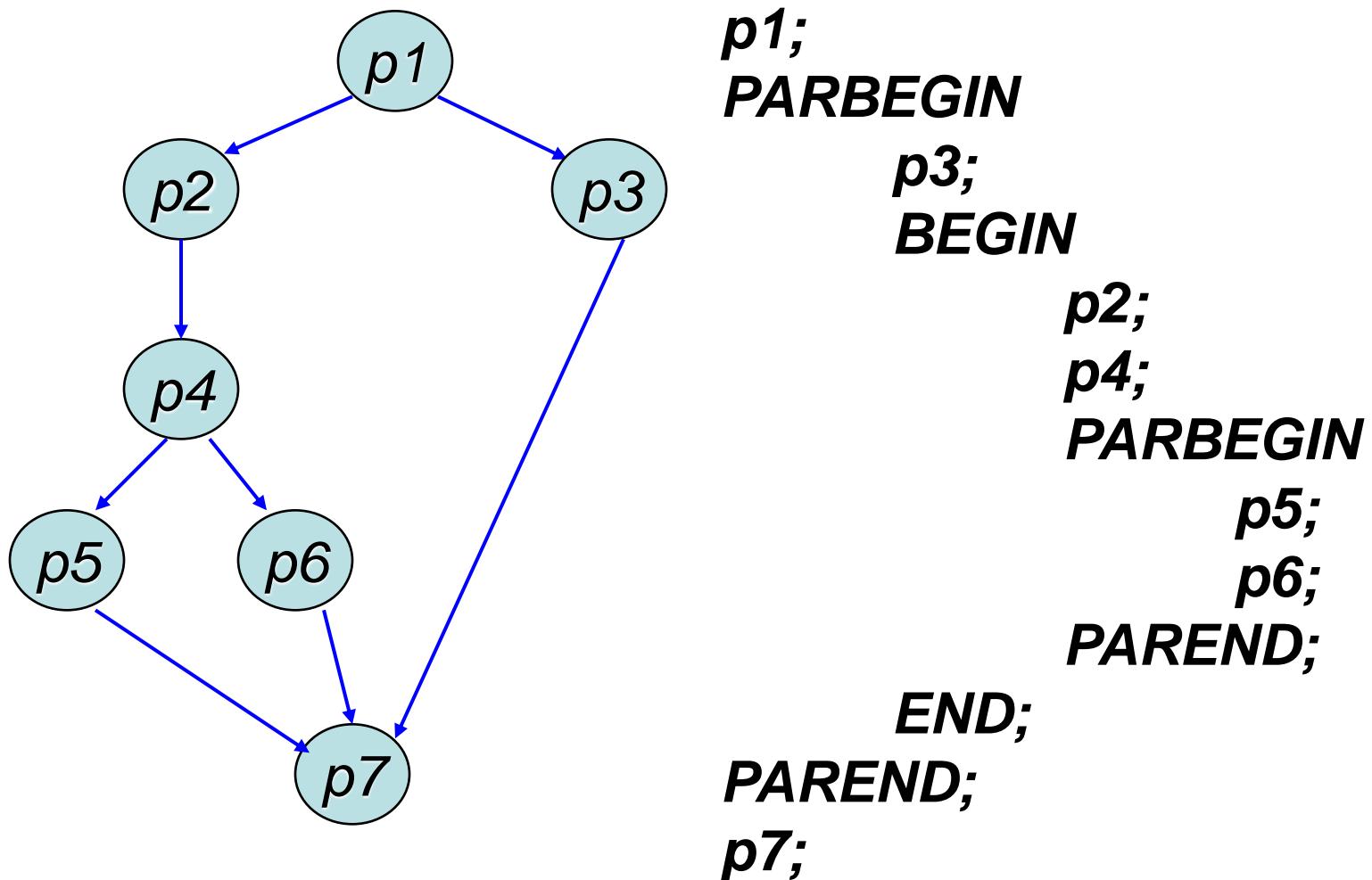
cont=3;
p1;
fork L1;
p2;
p4;
fork L2;
p5;
Go To L3;
L1: p3;
Go To L3;
L2: p6;
L3: join cont;
p7;

Parbegin/parend

- Forma: Parbegin p1, p2, ..., pn, Paren;

- Todos os comandos entre parbegin – parend são executados em paralelo
- O comando seguinte ao PAREN somente será executado após todos os anteriores terem terminados
- Pode ser adicionado a uma linguagem orientada a blocos

Parbegin/parend



O problema do compartilhamento de recursos

- ❑ A programação concorrente implica um compartilhamento de recursos
 - Variáveis, periféricos
- ❑ Um problema clássico é o do produtor-consumidor
- ❑ Exemplo: servidor de impressão
 - Processos usuários produzem impressões que são organizadas em uma fila, de onde um processo consumidor as lê e envia para a impressora

Exemplo: servidor de impressão

- A fila de impressão é um buffer circular
- Existe um ponteiro (*in*) que aponta para uma posição onde a impressão é inserida
- Existe um ponteiro (*out*) que aponta para a impressão que está sendo realizada
- O que acontecerá se as operações seguirem a sequência abaixo?
 - P1 vai imprimir; lê valor de *in* (5); perde processador
 - P2 ganha processador; lê valor de *in* (5); insere arquivo; atualiza *in* (6)
 - P1 ganha processador; insere arquivo (5); atualiza *in* (7)

Exemplo: servidor de impressão

- Estado incorreto
 - Impressão de P2 é perdida
 - Na posição 6 não há uma solicitação válida de impressão

Problema da seção crítica

- Corrida (*race condition*): situação que ocorre quando vários processos manipulam o mesmo conjunto de dados concorrentemente e o resultado depende da ordem em que os acessos são feitos
- Seção crítica: segmento de código no qual um processo realiza a alteração de um recurso compartilhado

Propriedades da exclusão mútua

❑ Exclusão mútua:

- Dois ou mais processos não podem estar simultaneamente em uma seção crítica

❑ Progressão

- Nenhum processo fora da seção crítica pode bloquear a execução de um outro processo

❑ Espera limitada

- Nenhum processo deve esperar infinitamente para entrar em uma seção crítica

❑ Não fazer considerações sobre o número de processadores, nem de suas velocidades relativas

Obtenção de exclusão mútua

- ❑ Desabilitação de interrupções
 - Não há troca de processos com ocorrência de interrupções de tempo ou de eventos externos
 - ❑ Variáveis do tipo *lock*
 - Variável compartilhada com apenas dois valores
 - 0: livre
 - 1: ocupado
- While (*lock*==1);

seção crítica {
 lock=1;
 lock=0;

Obtenção de exclusão mútua

❑ Primitiva *mutex*

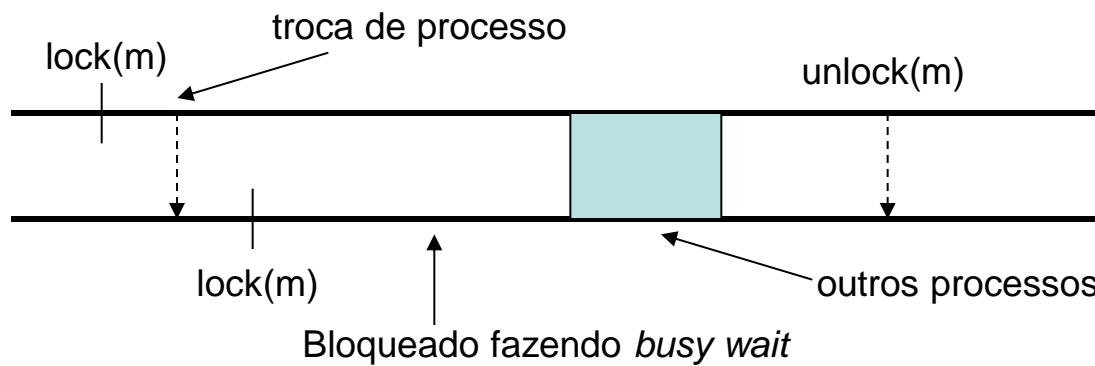
- Variável compartilhada para controle de acesso à seção crítica
- CPUs são projetadas para executar vários processos
- Inclusão de duas instruções *assembly* para leitura e escrita de posições de memória de forma atômica
 - CAS (*Compare and Store*): copia o valor de uma posição de memória para um registrador interno e escreve nela o valor 1
 - TSL (*Test and Set Lock*): lê o valor de uma posição de memória e coloca nela o valor zero
- O uso de mutex necessita de *lock* e de *unlock*

seção crítica {
 lock;
 unlock;

Problema do mutex

O uso de *lock* e *unlock* podem causar *busy waiting* (*spin lock*)

- Após fazer o *lock* e resolver suas necessidades na seção crítica, o processo deve aguardar até que o processador possa estar livre para ele, para fazer o *unlock*.
- Solução: bloquear o processo ao invés de executar *busy waiting*, usando duas novas primitivas
 - *sleep*: bloqueia um processo à espera de uma sinalização
 - *wakeup* sinaliza um processo



Semáforos

- ❑ Mecanismo proposto por Djikstra (1965)
- ❑ Usa duas primitivas: P (testar) e V (incrementar)
- ❑ Necessidade de garantir a atomicidade nas operações de incremento e decremento teste da variável compartilhada
 - Uso de *mutex*
- ❑ Tipos de semáforos:
 - Binários: valores 0 e 1
 - Contadores: n

Semáforos

❑ Implementação de semáforo contador:

P(s): s.valor = s.valor – 1
se s.valor < 0 {

bloqueia processo (*sleep*);
insere processo em s.fila;
}

V(s): s.valor = s.valor + 1
se s.valor <= 0 {

retira processo de s.fila;
acorda processo (*wakeup*);
}

Semáforos

- ❑ Implementação de semáforo binário (valor inicial de s = 1)

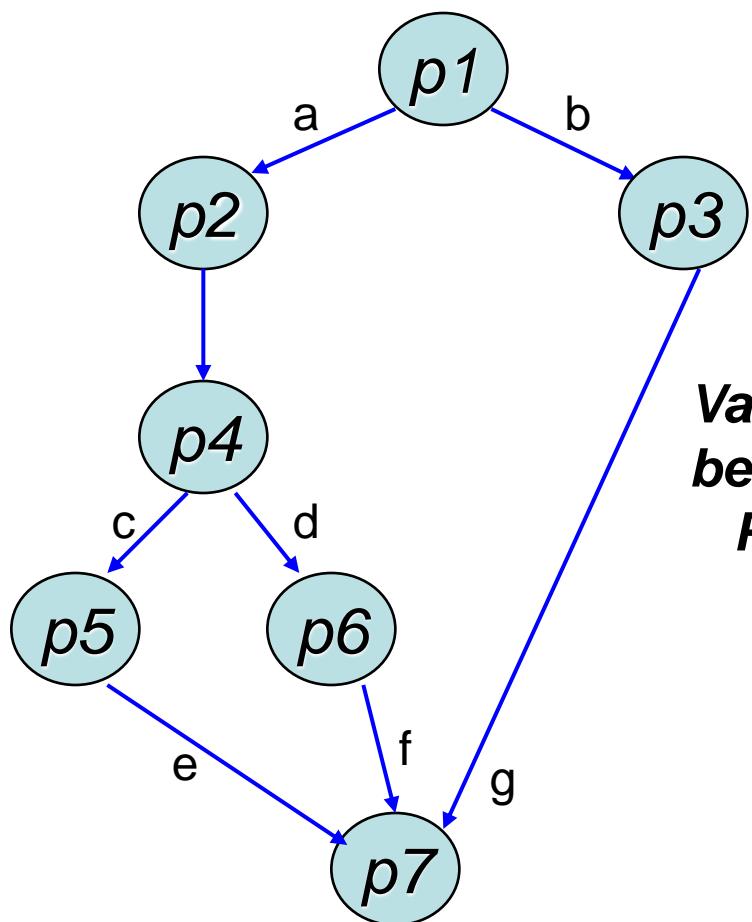
P(s):

```
se s = 1 { s = 0;  
            coloca processo na seção crítica  
        }  
senão  { bloqueia processo (sleep);  
            insere processo em s.fila;  
        }
```

V(s):

```
se tem processo em s.fila { retira processo de s.fila;  
                            acorda processo (wakeup);  
                        }  
senão s = 1;
```

Semáforos



```
Var a,b,c,d,e,f,g: semáforo = 0;  
begin  
    PARBEGIN  
        begin p1; V(a); V(b); end;  
        begin P(a); p2; p4; V(c); V(d); end;  
        begin P(b); p3; V(g); end;  
        begin P(c); p5; V(e); end;  
        begin P(d); p6; V(f); end;  
        begin P(e); P(f); P(g); p7; end;  
    PAREN;  
end;
```

Problema 1: Produtor-Consumidor

- É comum em programação concorrente a existência de um processo que tem como função produzir informações para que outro processo as consuma
 - Este problema é conhecido como Produtor-Consumidor
 - Existe uma variação conhecida como “Problema dos Leitores e Escritores”
- Solução: usar semáforos
 - O produtor produz uma mensagem
 - O consumidor consome uma mensagem
 - semáforo $s = 1$ (para a seção crítica)
 - semáforo $n > 0$ ou $n = 0$ (contador das mensagens)

Problema 1: Produtor-Consumidor

Produtor

↓
produz mensagem
 $P(s)$
coloca mensagem
no buffer

$V(n)$
 $V(s)$

Consumidor

↓
 $P(n)$
 $P(s)$
retira mensagem
 $V(s)$
consome mensagem

O semáforo s terá apenas os valores 0 e 1
O semáforo n terá valor maior ou igual a 0

Problema 2: O Jantar dos Filósofos

- ❑ Cinco filósofos pensam e comem
- ❑ Compartilham uma mesa com cinco lugares
- ❑ Existem cinco pauzinhos (*chopstick*)
- ❑ Um prato de arroz para cada filósofo
- ❑ De tempos em tempos o filósofo sente fome e tenta pegar o *chopstick* à esquerda e o *chopstick* à direita
- ❑ De posse de ambos os *chopsticks*, ele se alimenta
- ❑ Quando termina de comer, libera os *chopsticks* e volta a pensar, novamente

Problema 2: O Jantar dos Filósofos

□ Solução Simples:

- representar cada *chopstick* por um semáforo

```
var chopstick: array 0..4 of semaphore = 1;
var i: integer;
CADA_FILOSOFO (i)
repeat
    P(chopstick (i-1));
    P(chopstick (i mod 5));
        come
    V (chopstick (i-1));
    V (chopstick (i mod 5));
        pensa
until false;
```

Semáforos *versus* *mutex*

- Primitivas *lock* e *unlock* são feitas por um mesmo processo
 - Acesso à seção crítica

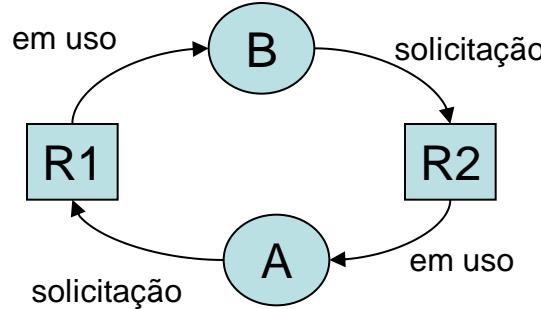
- Primitivas P e V podem ser realizadas por processos diferentes
 - Facilita a gerência dos recursos associados aos processos

Deadlock

- ❑ Situação na qual um ou mais processos ficam impedidos de prosseguirem sua execução porque cada um está aguardando acesso a recursos já alocados por outro processo.
- ❑ Para ocorrer *deadlock* têm-se que, simultaneamente:
 - 1- Exclusão mútua
 - Todo recurso ou está disponível ou está atribuído a um único processo
 - 2- Segura / espera
 - Os processos que detêm um recurso podem solicitar novos recursos
 - 3- Recurso não-preemptível
 - Um recurso concedido não pode ser retirado de um processo por outro

4- Espera circular

- Existência de um ciclo de dois ou mais processos, cada um esperando por um recurso já adquirido (em uso) pelo próximo processo do ciclo



Estratégias para implementação de *deadlock*

- Ignorar
- Detecção e recuperação
 - Monitoração dos recursos liberados e alocados
 - Eliminação de processos
- Impedir ocorrência cuidando na alocação de recursos
 - Algoritmo do banqueiro
- Prevenção por construção
 - Evitar a ocorrência de pelo menos uma das quatro condições necessárias

Gerência do processador

Introdução

- ❑ O uso da multiprogramação prevê a existência simultânea de vários processos disputando o processador
- ❑ Processo é um programa em execução
- ❑ Há a necessidade de gerenciar os processos
 - Filas de processos (uma para cada estado)
 - Eventos e transições de um estado para outro
 - Quando e como um processo usará o processador

Bloco Descritor de Processo

- ❑ Um processo é representado pelo Bloco Descritor de Processos (*Process Control Block-PCB*)
- ❑ Informações presentes no PCB:
 - Prioridade
 - Localização e tamanho na RAM
 - Identificação de arquivos abertos
 - Informações de contabilidade (tempo de CPU, espaço de memória)
 - Estado do processo (apto, executando, bloqueado)
 - Contexto de execução
 - Apontadores para encadeamento dos PCBs
 - etc.

Filas de processos

- ❑ Um processo sempre faz parte de uma fila
 - Fila de livres
 - Fila de aptos
 - Fila de bloqueados
- ❑ Eventos realizam a transição de uma fila para outra

Implementação de processos

❑ Estrutura de dados de um PCB

```
struct desc_proc{  
    char          estado_atual;  
    int           prioridade;  
    unsigned      inicio_memória;  
    unsigned      tamanho_mem;  
    struct {  
        unsigned tempo_cpu;  
        unsigned proc_pc;  
        unsigned proc_sp;  
        unsigned proc_acc;  
        unsigned proc_rx;  
        struct desc_proc *proximo;  
    } arquivos_arquivos_abertos[20];  
};  
struct desc_proc tab_desc[MAX_PROCESS];
```

Implementação de processos

□ Estruturas de filas e inicialização

```
struct desc_proc *desc_livre;
struct desc_proc *espera_cpu;
struct desc_proc *usando_cpu;
struct desc_proc *bloqueados;

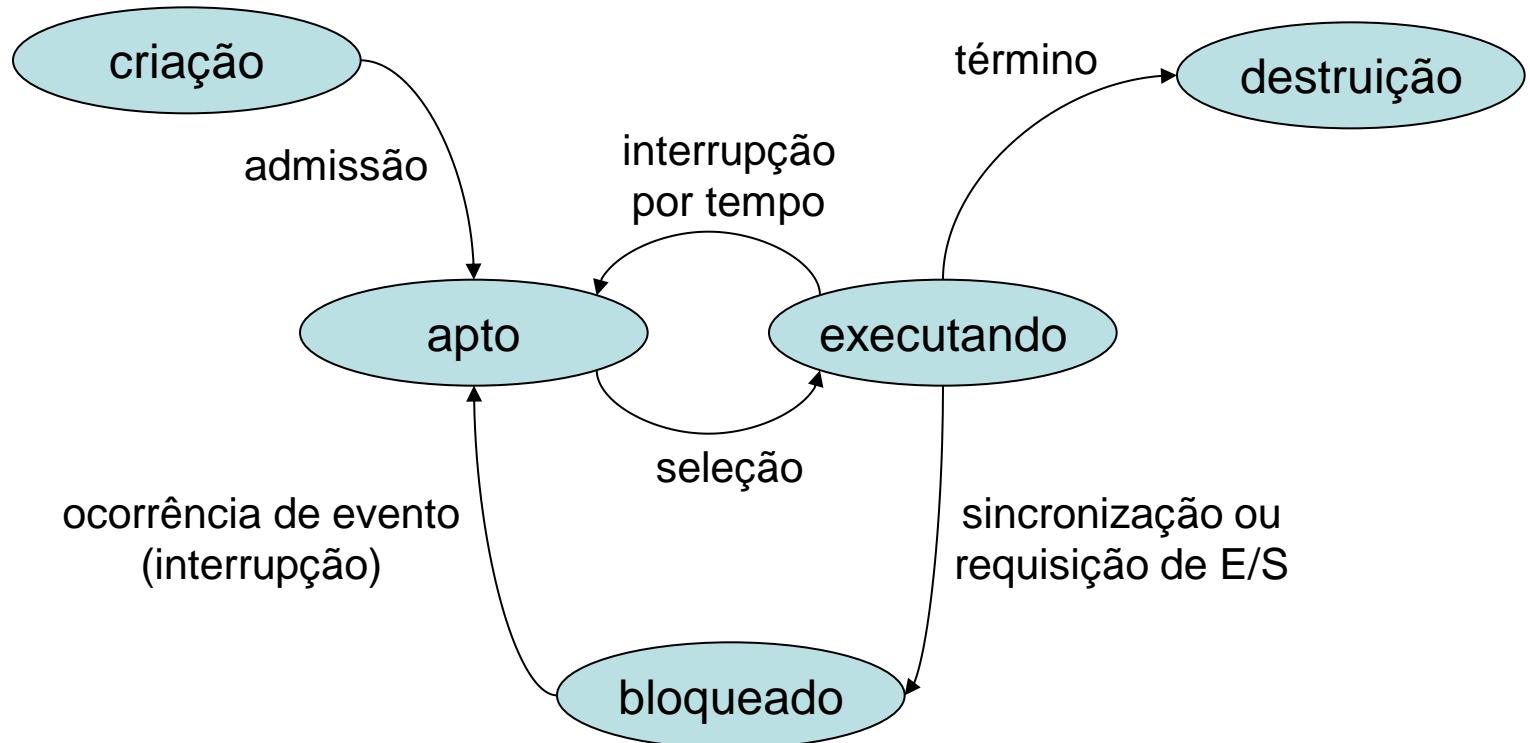
/* inicialização das estruturas de controle */

for (i=0; i< MAX_PROCESS; i++)
    tab_desc[i].prox = &tab_desc[i+1];

tab_desc[i].prox = NULL;
desc_livre = &tab_desc[0];

espera_cpu = NULL;
usando_cpu = NULL;
bloqueado = NULL;
```

Eventos de transição de estados



Criação de processos

- ❑ Alocar um descritor de processo
- ❑ Obter um identificador único (pid)
- ❑ Inicializar o descritor de processo
- ❑ Inserir apontadores para este descritor em estruturas de controle
 - Para inserir na lista de aptos

Seleção de processo para execução

- ❑ Segundo um critério, selecionar um processo na lista de aptos para usar a CPU
- ❑ A seleção ocorre sempre que a CPU estiver livre
- ❑ A seleção é realizada pelo procedimento de escalonamento
 - Conhecido como *dispatcher*
 - Faz o chaveamento do contexto do processo

Critérios usados para a seleção de processos

- Hora de criação do processo
- Tempo de serviço requisitado
- Tempo já gasto processando
- Tempo já gasto sem processar
- Recursos requisitados ou utilizados

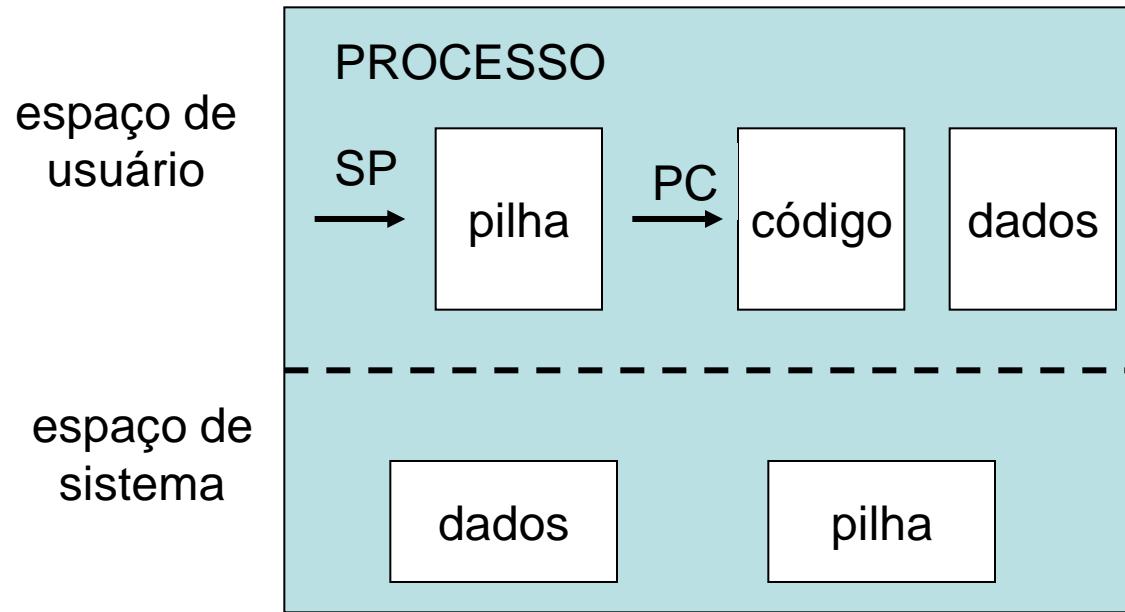
Quando retirar um processo do processador

- Término
- Processo de mais alta prioridade está apto
(se o escalonador for preemptivo)
- Interrupção de tempo
- Interrupção de dispositivos de E/S
- Interrupção por falta de página (segmento)
 - Endereço acessado não está mais carregado na memória
 - Memória virtual
- Interrupção por erros

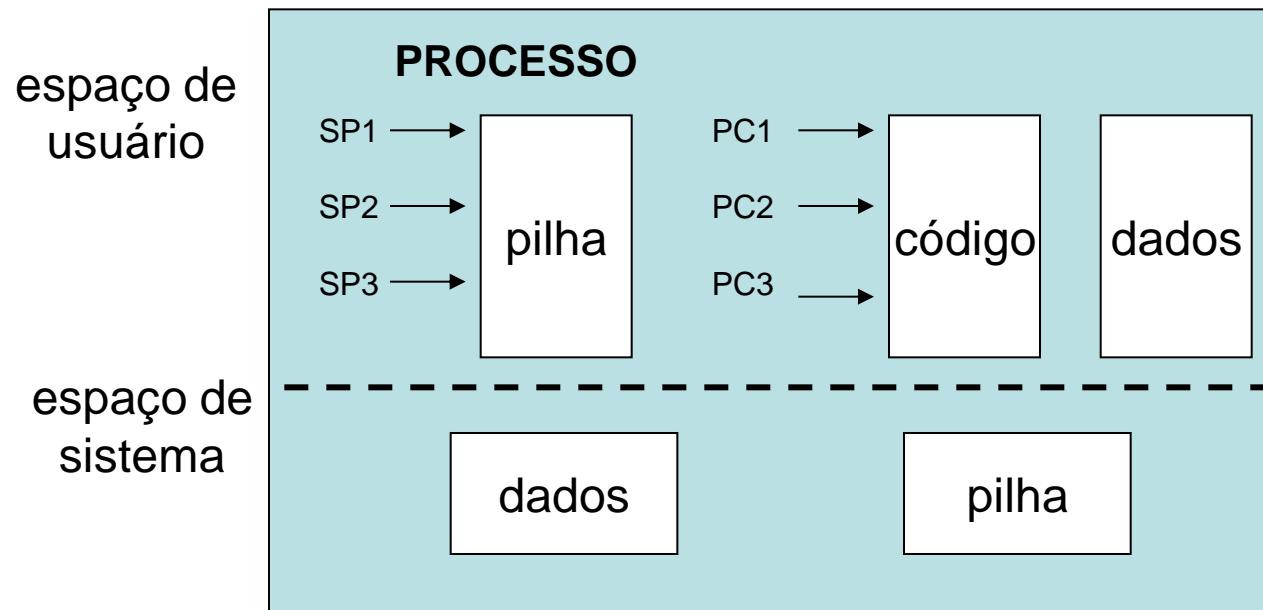
Operações de chaveamento de um processo

- ❑ Salvar o contexto do processo que está executando
- ❑ Atualizar o estado do processo que está em execução no seu descritor do processo
- ❑ Inserir o descritor do processo na fila apropriada
- ❑ Selecionar outro processo para execução
- ❑ Atualizar o descritor de processos do processo selecionado
- ❑ Atualizar estruturas de dados associadas com a gerência de memória e de dispositivos de E/S
- ❑ Restaurar o contexto do processo selecionado

Modelo de processo com um fluxo de controle (*thread*)

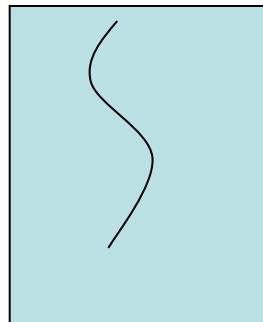


Modelo de processo com vários fluxos de controle

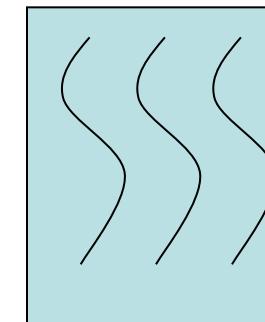
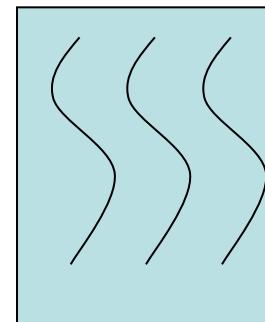
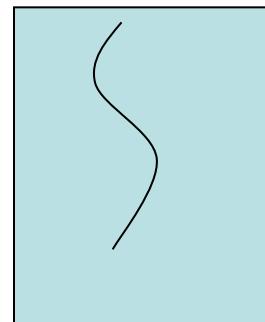
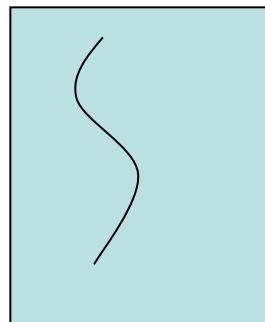
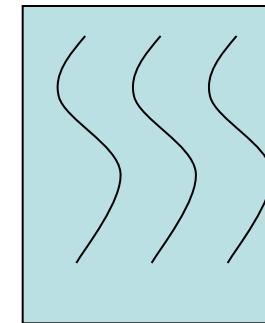


Threads e processos

um processo – uma *thread*



um processo – várias *threads*



vários processos
uma *thread* por processo

vários processos
várias *threads* por processo

Por quê usar *threads*

- ❑ Permitir a exploração do paralelismo real das máquinas multiprocessadas
- ❑ Aumentar o número de atividades executadas por unidade de tempo – *throughput*
- ❑ Melhorar o tempo de resposta
- ❑ Sobrepor operações de cálculo com operações de E/S

Vantagens de *multithreading*

- ❑ Tempo de criação e de destruição de *thread* é menor que de processo
- ❑ Chaveamento de contexto de *threads* é menor que o de processos
- ❑ Como *threads* compartilham o descritor do processo que as porta, elas dividem o mesmo espaço de endereçamento, o que permite a comunicação por memória compartilhada, sem a interação do núcleo do sistema operacional

Implementação de *threads*

- ❑ Todas as tarefas de gerenciamento de *threads* são feitas pela aplicação
- ❑ *Threads* são implementadas através de uma biblioteca que é ligada ao programa
- ❑ Existem interfaces de programação (API) para criação e destruição de *threads*
- ❑ O sistema operacional não sabe que existem *threads*
- ❑ Mas existem *threads* do *kernel*, que são monitoradas pelo sistema operacional

Escalonamento

- ❑ O escalonador é responsável por selecionar um processo para execução
- ❑ Divide o tempo do processador de forma justa entre os processos
- ❑ Dividido em duas partes:
 - Escalonador: política de seleção
 - *Dispatcher*: efetua a troca de contexto

Objetivos do escalonamento

- Maximizar a utilização do processador
- Maximizar o *throughput*
 - Número de processos executados por unidade de tempo
- Minimizar o tempo de execução
 - *Turnaround*
- Minimizar o tempo de espera
 - Tempo do processo na fila de aptos
- Minimizar o tempo de resposta
 - Tempo entre uma requisição e a sua realização

Tipos de escalonador

- ❑ Escalonador longo prazo
 - Executa quando um processo é criado
 - Determina quando um processo está apto
 - Controla o grau de multiprogramação do sistema
- ❑ Escalonador médio prazo
 - Participa do mecanismo de swap
 - Suporte ao grau de multiprogramação
- ❑ Escalonador curto prazo (mais importante)
 - Indica qual processo apto ganha o processador
 - Executado quando ocorrem eventos importantes (interrupção de relógio, de E/S, de software – sinais e chamadas de sistema).

Tipos de escalonador

- ❑ Uma vez escalonado, o processo utiliza o processador até que:
 - Não preemptivo e preemptivo
 - Término da execução
 - Execução de uma requisição de E/S ou de sincronização
 - Liberação voluntária do processador a outro processo (*yield*)
 - Ocorre apenas com o preemptivo
 - Interrupção de relógio
 - Processo de mais alta prioridade está apto

Algoritmos de escalonamento

- ❑ Algoritmo de escalonamento seleciona qual processo deve executar em um determinado instante de tempo
- ❑ Algoritmos não-preemptivos
 - FIFO
 - SJF
- ❑ Algoritmos preemptivos
 - SRT
 - *Round-Robin* (circular)
- ❑ Outros algoritmos
 - *High Response Ratio Next* (HRRN)
 - *Shortest Process Next* (SPN)
 - etc.

FIFO – *First In First Out*

- ❑ Atende os processos aptos na ordem de chegada no sistema
 - Utilizado para processos bloqueados também
- ❑ Vantagem
 - Simplicidade
 - Processo executa até que:
 - Libere explicitamente o processador
 - Realize uma chamada de sistema – fica bloqueado
 - Termine sua execução
- ❑ Desvantagens:
 - Processos pequenos esperam tanto quanto os grandes
 - Processo utilizando o processador sem se bloquear causa uma espera intolerável nos demais processos

SJF – Shortest Job First

- A prioridade de ativação de um processo é baseada no seu tamanho de criação
- Vantagens:
 - É considerado o algoritmo ótimo, pois fornece o menor tempo de espera médio para um conjunto de processos
 - Processos I/O *bound* são favorecidos
 - SJF tende a diminuir o número de processos na fila pronto, pois diminui o número de processos pequenos
- Desvantagens:
 - Espera média para grandes processos é maior do que no FIFO

SRT – *Shortest Remaining Time*

- A prioridade de ativação se baseia no tempo que falta para terminar um processo (tempo restante)
- Vantagens:
 - Vai dar mais prioridade aos processos próximos do fim
 - Processos pequenos são ainda mais favorecidos que os grandes
 - Alcança a menor média global possível, pois termina rapidamente o processo e passa o processador para outro
- Desvantagens:
 - SJF e SRT necessitam de uma estimativa inicial de tempo de execução, logo, são inapropriados para ativar processos interativos
 - O usuário que estimar o menor tempo para o processo terá maior prioridade, logo, pode-se, deliberadamente, “enganar” o sistema

RR – Round Robin

- ❑ Política de gerência de fila muito simples
- ❑ Estabelece uma fatia de tempo para cada processo
- ❑ No final da sua fatia de tempo, o processo é removido do processador em favor do próximo da fila de processos apto
- ❑ Processos removidos, desbloqueados ou que iniciaram são incluídos no final da fila apto
- ❑ A prioridade na fila fica sendo a de ordem ascendente de espera desde a última fatia de tempo
- ❑ Com “n” processos executáveis, cada um recebe $1/n$ do tempo do processador

□ Vantagem:

- Favorece processos curtos, que terminam primeiro, mas sem penalizar em excesso os processos grandes

□ Desvantagem:

- Devido ao tempo de troca de processos (*preempção*), a Espera Média Global é maior que no FIFO

RR – Round Robin

- ❑ Qual o tamanho da fatia de tempo “q”?
- ❑ “q” muito grande:
 - O tempo total de um processo é aumentado pela chegada de novos processos ou desbloqueios e diminuído pelo bloqueio de outros
- ❑ “q” muito pequeno:
 - O processo vai muitas vezes para o final da fila demorado cada vez mais para acabar. Maior frequência de troca de processos no processador causa um maior *overhead* do sistema

Mecanismos de fila múltipla

- Mecanismos de fila única não permitem dar diferentes tratamentos para processos com prioridade
 - Com prioridades, um processo de baixa prioridade pode não executar (*starvation*)
- Sistemas multiprogramados podem querer privilegiar processos “I/O *bound*” em relação a “CPU *bound*”
- Múltiplas filas possuem mais de uma fila para cada Estado e permite usar mais de um mecanismo de ativação, um por fila

Múltiplas filas com realimentação

- ❑ Baseado em prioridades dinâmicas
- ❑ Em função do tempo de uso da CPU, a prioridade do processo aumenta ou diminui
- ❑ Sistema de envelhecimento evita a postergação indefinida
 - O processo pode trocar de uma fila apto para outra

Estudo de caso: escalonamento no unix

- ❑ Múltiplas filas com realimentação empregando round robin em cada uma delas
- ❑ Prioridades são reavaliadas uma vez por segundo, em função de:
 - Prioridade atual
 - Prioridade do usuário
 - Tempo recente de uso da CPU
 - Fator *nice*
- ❑ Prioridades são divididas em faixas de acordo com o tipo de usuário
- ❑ A troca dinâmica das prioridades respeita os limites da faixa

Estudo de caso: escalonamento no unix

- Prioridades recebem valores entre 0 e 127, sendo que 0 representa a maior prioridade possível
 - 0 – 49 para os processos do *kernel*
 - 50 – 127 para os processos dos usuários
- Na ordem decrescente de prioridade:
 - *Swapper*
 - Controle de dispositivos de E/S orientados a bloco
 - Manipulação de arquivos
 - Controle de dispositivos de E/S orientados a caracteres
 - Processos de usuário

Gerência da memória

Introdução

- ❑ CPU e periféricos de E/S interagem com a memória
- ❑ Um programa deve ser carregado na memória principal para executar
- ❑ Memória secundária pode ser utilizada como extensão da memória principal
- ❑ Memória física é a memória do hardware, começa no endereço físico “0” e continua até um endereço qualquer
 - também é chamada de memória principal
 - o núcleo do S.O. ocupa parte da memória principal
- ❑ Memória lógica é aquela que o processo “enxerga”
 - Endereços lógicos são os manipulados pelos processos

□ *Memory Management Unit – MMU*

- Hardware que faz o mapeamento entre o endereço lógico e o físico
- Para um processo executar é necessária a “amarração” de endereços
 - Conhecido como *Binding*
 - Ocorre em tempo de compilação ou de carga ou de execução

Carregando o processo

- ❑ O programador não sabe onde o seu processo será carregado na memória
- ❑ O endereço só é conhecido no momento da carga
 - Durante execução do processo, sua localização física pode ser alterada
 - *Swapping*
- ❑ Há a necessidade de traduzir endereços lógicos a endereços físicos
- ❑ Relocação é a técnica que realiza a tradução
 - Via software: carregador relocador
 - Via hardware: carregador absoluto

Mecanismos de gerência de memória

❑ Alocação contígua de memória

- Alocação simples
- Alocação múltipla
 - Partições fixas
 - Partições móveis (ou Partições variáveis)
 - Regiões móveis

❑ Alocação não contígua de memória

- Paginação
- Segmentação
- Segmentação com paginação (ou segmentação paginada)

Alocação contígua simples

❑ Apenas um programa por vez na memória principal

- O próximo processo entra na memória somente quando terminar a execução do que está na memória

❑ Vantagens:

- Custo de implementação baixo
- Facilita a carga dos programas em endereços fixos
- Não se necessita relocar o S.O.
- Implementação barata e simples pois contém carregador absoluto e monitor de filas é dispensado
- Proteção de memória é esquecida, pois há apenas um programa na memória

❑ Desvantagens:

- Processador ocioso durante operação de E/S
- A memória não utilizada é desperdiçada

Alocação contígua múltipla

- ❑ Implica em multiprogramação
- ❑ Otimiza o uso do processador não deixando-o ocioso muito tempo
- ❑ Otimiza a utilização da memória
- ❑ Os objetivos procurados são:
 - economia de tempo
 - o processador fica mais ocupado, pois quase sempre há outro processo apto a executar, assim que um parar
 - economia do espaço
 - A memória é preenchida ao máximo, pois tem-se vários processos simultaneamente na memória

Alocação contígua múltipla

- Problemas a serem analisados:
 - Como colocar todos os programas na memória?
 - dividindo-a em regiões ocupadas por processos
 - Como proteger as diversas regiões de possíveis interferências?
 - utilizando hardware (MMU) e gerência de SO mais precisos, para evitar a ocorrência desta situação

Partições fixas

- ❑ Número, tamanho e posição das partições são fixos
 - Porém, em alguns sistemas é possível reconfigurar a memória
- ❑ Algumas partições são maiores que outras
 - Permite a execução de processos maiores que a média
- ❑ Vantagem:
 - Quantidade e tamanho das partições é fixada pelo programador
- ❑ Desvantagens:
 - Não há a possibilidade de mudar a configuração do sistema
 - Há desperdício de memória quando um processo menor que a partição é carregado
 - Fragmentação interna: Perda da memória dentro da partição
 - Fragmentação externa: Partição não utilizada

Partições fixas

- ❑ Implementação de Partições Fixas
 - uma fila de processos por partição para permitir uma separação de classes de processos
 - processo alocador de memória para n partições
 - A partição onde o processo vai residir é determinada na compilação pelo usuário
 - uma opção é adiar a alocação de memória para o momento da execução e ter um carregador-relocador, assim qualquer partição pode ser alocada
 - qualquer partição com tamanho suficiente pode ser usada
 - é possível utilizar só uma fila de processos.

- ❑ As partições fixas não se adaptam às necessidades de um conjunto variável de processos (apenas específico)
 - quando se necessita um conjunto variável de processos então a melhor solução é a utilização de Partições Variáveis

Partições variáveis

- ❑ O objetivo é evitar o desperdício da memória e processador tendo-se mais processos
- ❑ Os processos, ao serem carregados, eliminam a fragmentação interna de memória que ocorre em partições fixas
 - Carrega-se o máximo de processos na memória
- ❑ Pressupõe relocação, ou seja, localizações variáveis

❑ Implementação

- Gerente de Espaço

- localiza o espaço necessário para carregar o processo
- acerta partições para delimitar regiões de processos
- chama o carregador-relocador para carregar o processo para a nova partição
- necessita de um poço de memória disponível para fazer a reserva de uma partição na carga e para liberar esta partição ao término da execução do processo

Partições variáveis

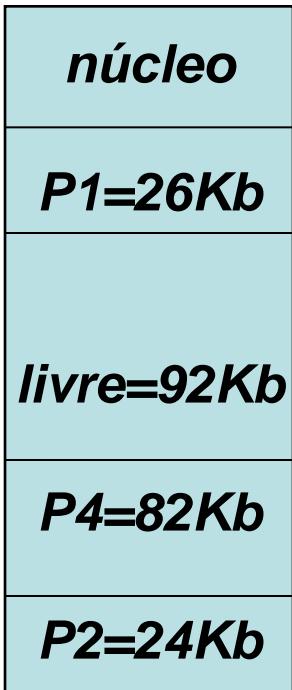
núcleo
P1=26Kb
P3=50Kb
livre=42Kb
P4=82Kb
P2=24Kb

Fila de processos:

P5=62Kb, P6=38Kb, P7=60Kb

Tem-se que esperar até que algum processo acabe para alocar outro processo, pois não há espaço suficiente para o P5.

Partições variáveis



Fila de processos:

$P_5=62\text{Kb}$, $P_6=38\text{Kb}$, $P_7=60\text{Kb}$

*Acabou o P_3 e agora pode-se colocar
o P_5 em execução.*

Partições variáveis

núcleo
P1=26Kb
P5=62Kb
livre=30Kb
P4=82Kb
P2=24Kb

Fila de processos:

P6=38Kb, P7=60Kb

*Existem 30Kb livres
(poço de memória).*

Partições variáveis

núcleo
livre=26Kb
P5=62Kb
livre=30Kb
P4=82Kb
P2=24Kb

Fila de processos:

P6=38Kb, P7=60Kb

Acabou o P1, porém não se pode juntar os dois pedaços de memória livres. Deve-se esperar que o P5 acabe.

Partições variáveis



Fila de processos:

P6=38Kb, P7=60Kb

Acabou o P5 e se tem 118Kb livres

Partições variáveis

núcleo
P6=38Kb
P7=60Kb
livre=20Kb
P4=82Kb
P2=24Kb

P6 e P7 foram carregados

Partições variáveis

- ❑ Esta forma de gerência é bem melhor que a de partições fixas, pois permite uma maior entrada de processos
 - Porém ainda desperdiça a memória
- ❑ Um processo deve esperar que outro libere área p/ poder crescer
- ❑ Há um aumento da sobrecarga (*overhead*) do processador na troca de processos
- ❑ Não é uma solução, apenas um "quebra-galho"
- ❑ Para resolver este problema deve-se utilizar o esquema de Regiões Móveis.

Algoritmos para alocação de partições variáveis

- ❑ Determinar em que bloco de memória livre um processo será alocado. Existem três políticas:
 - BEST-FIT
 - Seleciona do poço de memória livre, o menor bloco que possa satisfazer à requisição
 - Requer pesquisa em toda a lista
 - Aumenta o número de pequenos blocos na memória
 - WORST-FIT
 - Seleciona o maior bloco que satisfaça à requisição
 - Requer pesquisa de toda a lista para ver qual bloco é o maior
 - Distribui mais uniformemente os blocos
 - FIRST-FIT
 - Seleciona o primeiro bloco que satisfaça à requisição
 - A pesquisa na lista é muito menor
 - Causa rápida subdivisão dos blocos grandes, mas cria um número muito menor de blocos pequenos que o BEST-FIT
 - Muitas vezes atende requisições melhor que os outros métodos

Algoritmos para alocação de partições variáveis

Pedido	best-fit	worst-fit	first-fit
	6100,5800	6100,5800	6100,5800
5000	6100,800	1100,5800	1100,5800
5400	700,800	1100,400	1100,400

- ❑ Quando a reserva é feita, o pedaço do bloco que não foi alocado é devolvido ao poço
- ❑ Opções para devolver ao poço:
 - Colocar o resto no mesmo lugar que o bloco original
 - Tratar o resto como um novo bloco liberado

Regiões móveis

- ❑ Tenta evitar a fragmentação da memória em pedaços muito pequenos e processos que não podem ser carregados porque a memória está fragmentada
- ❑ São utilizadas regiões que se movem quando o processo está em execução
- ❑ Evita a fragmentação compactando os espaços livres. Esta compactação é realizada pelo Gerente de Espaço
- ❑ É utilizada relocação dinâmica para relocar processos em execução
- ❑ Busca uma utilização global da memória

Regiões móveis

núcleo
livre=26Kb
P5=62Kb
livre=30Kb
P4=82Kb
P2=24Kb

Fila de processos:
 $P_6=38\text{Kb}$, $P_7=60\text{Kb}$

Agora não é mais preciso ficar esperando que o P5 termine para colocar o P6 em execução

Regiões móveis

- ❑ A grande desvantagem é que, para diminuir a fragmentação da memória, gasta-se muito tempo de cópia de um processo
- ❑ *Swapping* é muito utilizado em sistemas de tempo compartilhado
 - Memória secundária é utilizada como extensão da principal
 - Tem-se a memória carregada com processos e com espaços livres
 - Há um processo querendo executar, porém não há memória para ele, o que fazer?
 - transfere um processo da memória para o disco, até se obter o espaço necessário
- ❑ Existem variantes do *swapping* nos sistemas operacionais, como o windows e o unix

Regiões móveis

- ❑ Se o processo do disco necessitar memória, esta é liberada para ele
 - swap-out: memória p/ disco
 - swap-in: disco p/ memória
- ❑ Ocasiões nas quais se usa *Swapping*:
 - processos trocando de regiões p/ coleta de espaço livre
 - Regiões trocando de processo, de posição ou de tamanho
 - Para que um processo ceda lugar a outro sem precisar ser reiniciado permitindo carregar um processo com maior prioridade e evitando a situação de *dead lock*

❑ Preempção

- Alocação de um recurso que ainda está sendo utilizado por outro processo

❑ Principais razões para usar preempção

- Permitir a execução dos processos na memória que utilizam pouco os recursos
- Abrir espaço de memória p/ processo mais prioritário
- Para diminuir a quantidade de I/O no *swapping* pode-se implementar código reentrante
 - Os usuários que utilizam mesma área de texto (pois estão executando mesmo processo), tendo cada um sua própria área de dados

❑ Desvantagens:

- Perda de tempo quando faz *swapping*, pois realiza muito E/S
- Com *swapping*, o problema de fragmentação de memória fica resolvido em partes, mas ficará sempre fazendo o *swapping*, consumindo muito tempo de E/S

❑ Pode-se ter soluções melhores

- ❑ Solução para eliminar a fragmentação externa e limitar a interna
- ❑ O espaço físico de endereçamento não precisa ser contíguo
- ❑ A memória física (sistema) e a lógica (processo) são divididas em blocos de tamanho fixo e idênticos
 - Memória física em páginas físicas (*frames*)
 - Memória lógica em páginas lógicas (páginas)

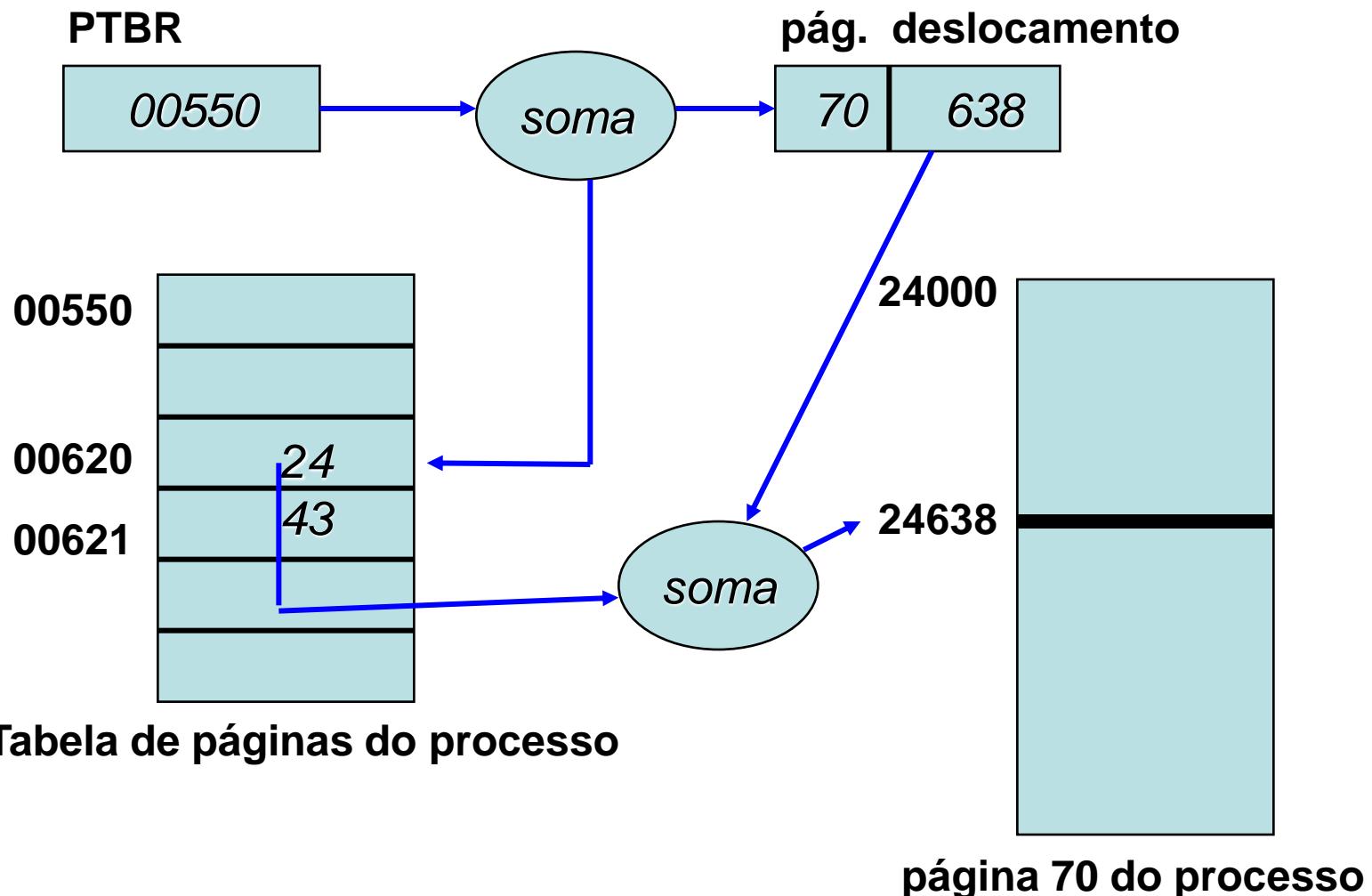
Paginação

- ❑ Para executar um processo de n páginas, basta localizar n *frames* livres
- ❑ Necessita-se traduzir páginas para *frames*
- ❑ O endereço lógico é dividido em duas partes
 - Número da página
 - Deslocamento dentro da página
- ❑ Tradução é realizada através de uma tabela de páginas
- ❑ O tamanho da página é imposto pelo hardware (MMU)
 - Valores variam entre 1Kb a 8Kb

□ Quanto ao tamanho de páginas:

- Páginas grandes implicam
 - Processo composto por menos páginas (tabela com poucas páginas)
 - Aumento da fragmentação interna da última página
- Páginas pequenas implicam
 - Processos compostos por mais páginas (tabela com muitas páginas)
 - Diminuição da fragmentação interna da última página

Paginação



☐ Implementação da tabela de páginas

- Registradores

- Cada página em um único registrador
- No descritor do processo devem ser mantidas cópias dos registradores
- Desvantagem é o número de registradores

- Memória

- *Page-table base register* (PTBR) aponta para a tabela de páginas
- *Page-table lenght register* (PTLR) indica o tamanho da tabela em número de entradas
- Desvantagem é que cada acesso a dado/instrução necessita de dois acessos à memória

□ *Translation look-aside buffers (TLBs)*

- Meio termo entre implementação via registrador e via memória
- Usa memória cache especial (TLB) composta por um banco de registradores (memória associativa)
 - Busca por conteúdo e não por endereço
 - Acesso paralelo e não sequencial
- Tamanho limitado pelo custo
 - De 8 a 2048 entradas
 - Tempo de acesso é superior em 10 vezes o acesso à RAM
 - Acessa a TLB, se o dado está presente busca-o, senão, consulta a RAM e atualiza a TLB
 - *Hit-ratio*: probabilidade da entrada da página referenciada estar na TLB
 - *Miss-ratio*: taxa de erro no acesso

□ Paginação multinível

- As tabelas de páginas possuem tamanho variável
 - Ex. processadores Intel 32 bits, páginas de 4Kb resulta em tabelas de 1 Mega entradas
 - Muito grandes para armazenar contiguamente
 - Solução: diretórios de tabelas de páginas (níveis)
 - Cada entrada do diretório aponta para uma tabela de páginas diferente

☐ Tabela de páginas invertida

- Problema é o tamanho grande da tabela de páginas
- Tabela invertida
 - Uma tabela de páginas para todo o sistema (não mais uma por processo)
 - Uma entrada para cada *frame*
- Endereço lógico é composto por
 - Process_id
 - Página
 - Deslocamento
- Cada entrada da tabela possui process_id e página

Segmentação

- ❑ Leva em consideração a visão de programadores e compiladores
- ❑ Um programa é uma coleção de segmentos
 - Código
 - Dados alocados estaticamente
 - Dados alocados dinamicamente
 - Pilha
- ❑ Um segmento pode ser uma unidade lógica inteira
 - Uma rotina, uma biblioteca

□ Endereço lógico em segmentação

- Composto por número do segmento e deslocamento dentro do segmento
- Os segmentos não possuem o mesmo tamanho
- Existe um tamanho máximo de segmento
- A tradução é feita de forma similar à de paginação

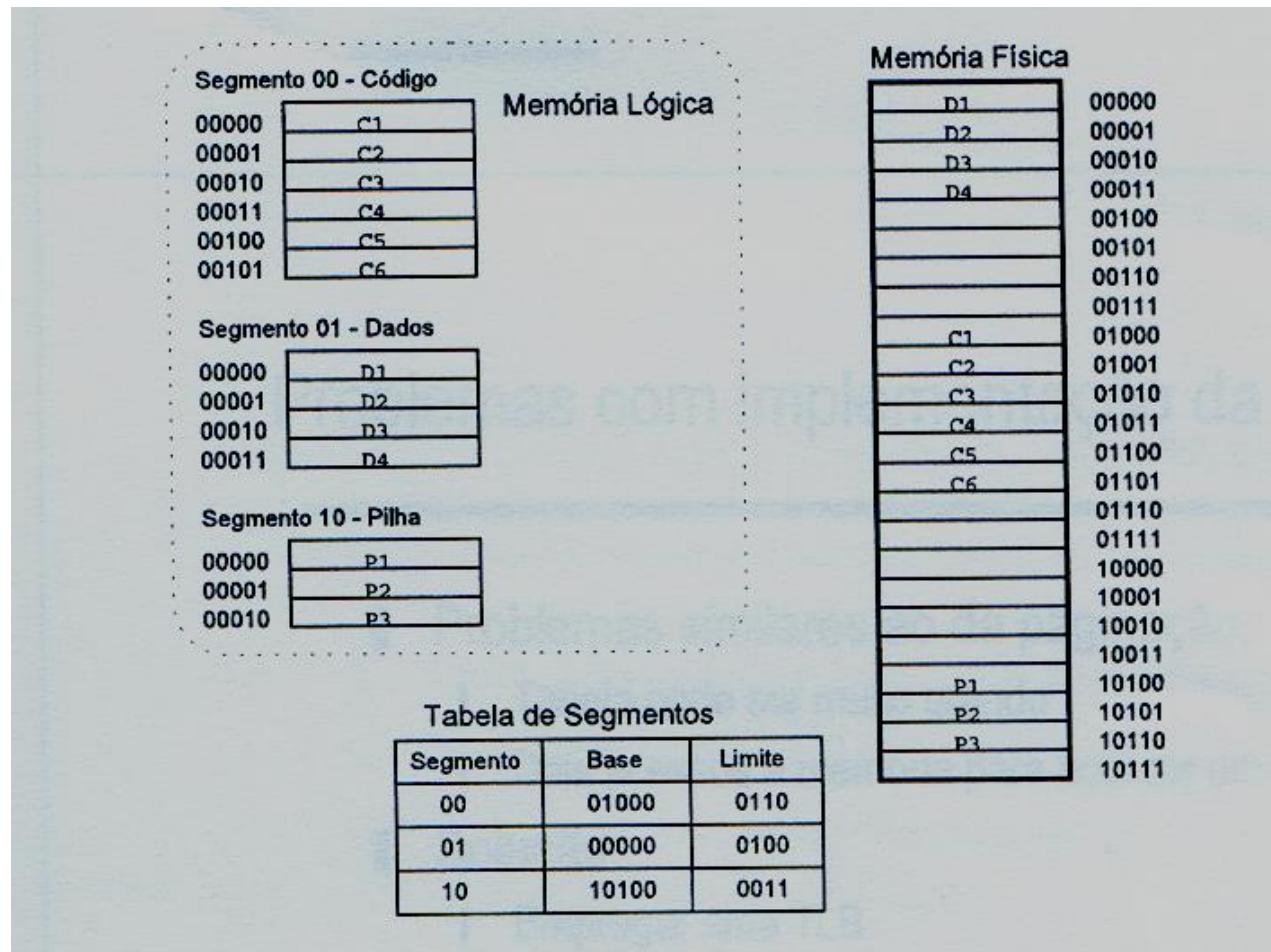
Segmentação

□ Tradução de endereço lógico em endereço físico

- Feita de forma similar à paginação via tabela de segmentos
- Entrada na tabela de segmentos
 - Base: endereço inicial (físico) do segmento na memória
 - Limite: tamanho do segmento

Segmentação

Tradução de endereço lógico para endereço físico



☐ Implementação da tabela de segmentos

- Cada segmento corresponde a uma entrada na tabela
- Cada segmento necessita armazenar dois valores
 - Limite e base
- Análogo à tabela de páginas
 - Registradores
 - Memória

- ❑ Implementação da tabela de segmentos via registradores
 - Cada segmento dois registradores (base e limite)
 - No descritor do processo devem ser mantidas cópias dos registradores
 - Na troca de contexto ocorre atualização dos registradores
 - Número de registradores impõe uma restrição ao tamanho da tabela de segmentos (como na paginação)

- Implementação da tabela de segmentos via memória
 - Tabela de segmentos é armazenada em memória
 - *Segment-table base register* (STBR): localização do início da tabela de segmentos na memória
 - *Segment-table lenght register* (STLR): indica o número de segmentos de um processo

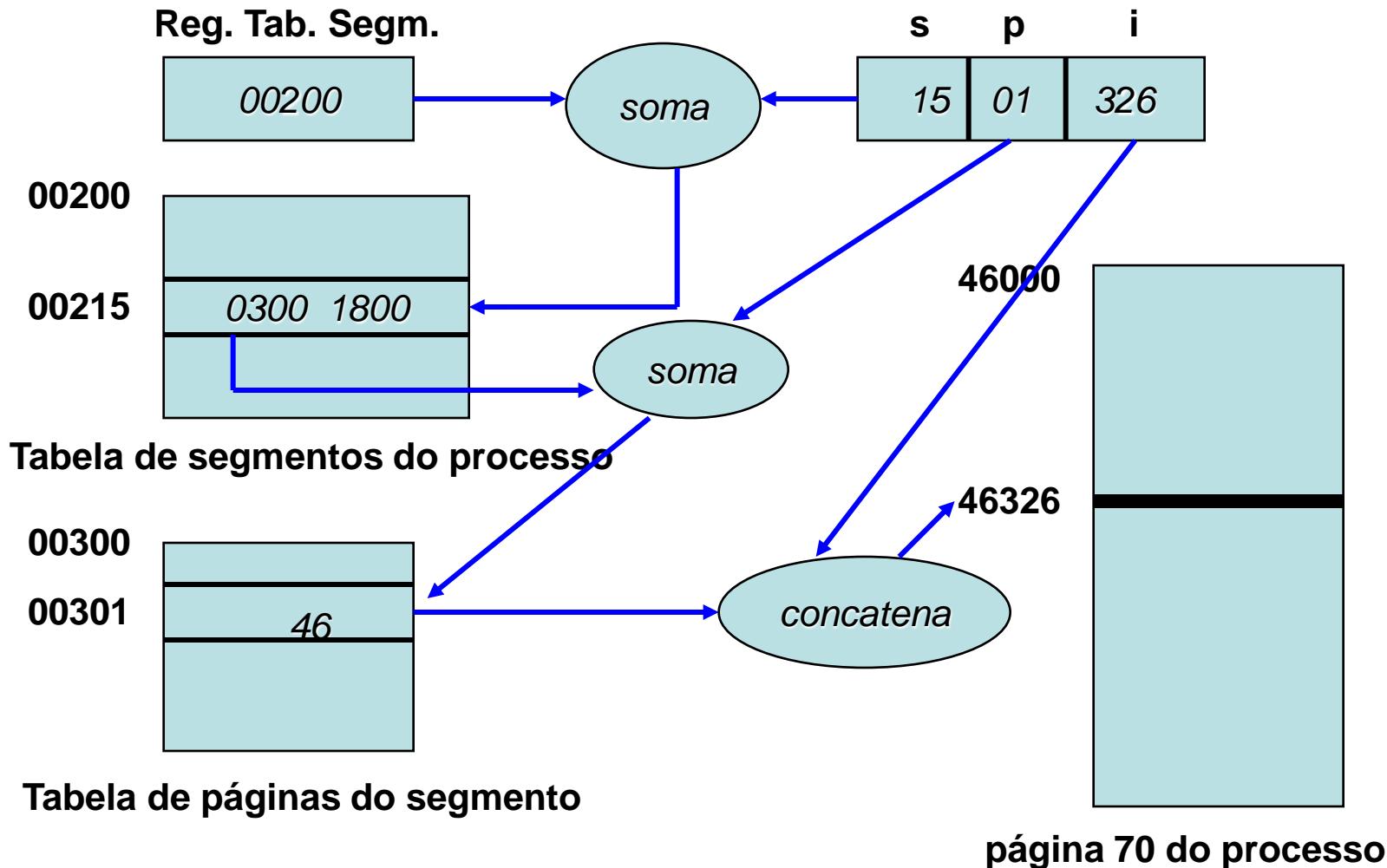
- Problemas com a implementação da tabela de segmentos via memória
 - Tabela pode ser muito grande
 - Dois acessos à memória para acessar um dado/instrução
 - Solução: usar TLB
 - Observação
 - Como na paginação, a consulta à tabela em memória provoca no mínimo dois acessos à memória, pois uma entrada na tabela pode representar mais de um acesso.

- Desvantagem da segmentação:
 - Provoca fragmentação externa quando segmentos começam a liberar memória
 - Mesmo problema de partições variáveis com as mesmas soluções:
 - Concatenação de segmentos adjacentes
 - Compactação da memória

Segmentação com paginação

- Também conhecida como segmentação paginada
 - O endereço lógico é composto por segmento, página e deslocamento dentro da página
 - O segmento é paginado
 - Existe uma tabela de segmentos e uma tabela de páginas por segmento

Segmentação com paginação



Memória Virtual

□ Problemas da gerência de memória

- Todo o espaço lógico é mapeado no espaço físico
- O tamanho do programa é limitado pelo tamanho da memória
- Desperdício de memória por manter armazenado código não mais utilizado
- Um programa que tem memória alocada poderia liberá-la para a execução de outros programas, mas não é possível fazê-lo

Memória virtual: conceito

- ❑ É a técnica que permite a execução de um processo sem que ele esteja completamente em memória
 - Separação do vínculo de endereço lógico do endereço físico
- ❑ Princípios básicos:
 - Carregar uma página ou segmento na memória principal apenas quando ela for necessária
 - Paginação sob demanda
 - Segmentação sob demanda

Memória virtual: vantagens

- ❑ Aumento do grau de multiprogramação
- ❑ Reduz o número de operações de E/S para carga do programa
- ❑ Capacidade de executar programas maiores que o tamanho da RAM

Princípio da localidade

- Base de funcionamento da memória virtual
- Na execução de um processo existe a probabilidade de que acessos a instruções e a dados sejam limitados a um mesmo trecho de código
 - Execução da instrução $j+1$ segue a de j
 - Laços *for*, *while*, *do-while*
 - Acessos a elementos de vetores
- Em um determinado instante de tempo apenas esses trechos do processo necessitam estar na memória

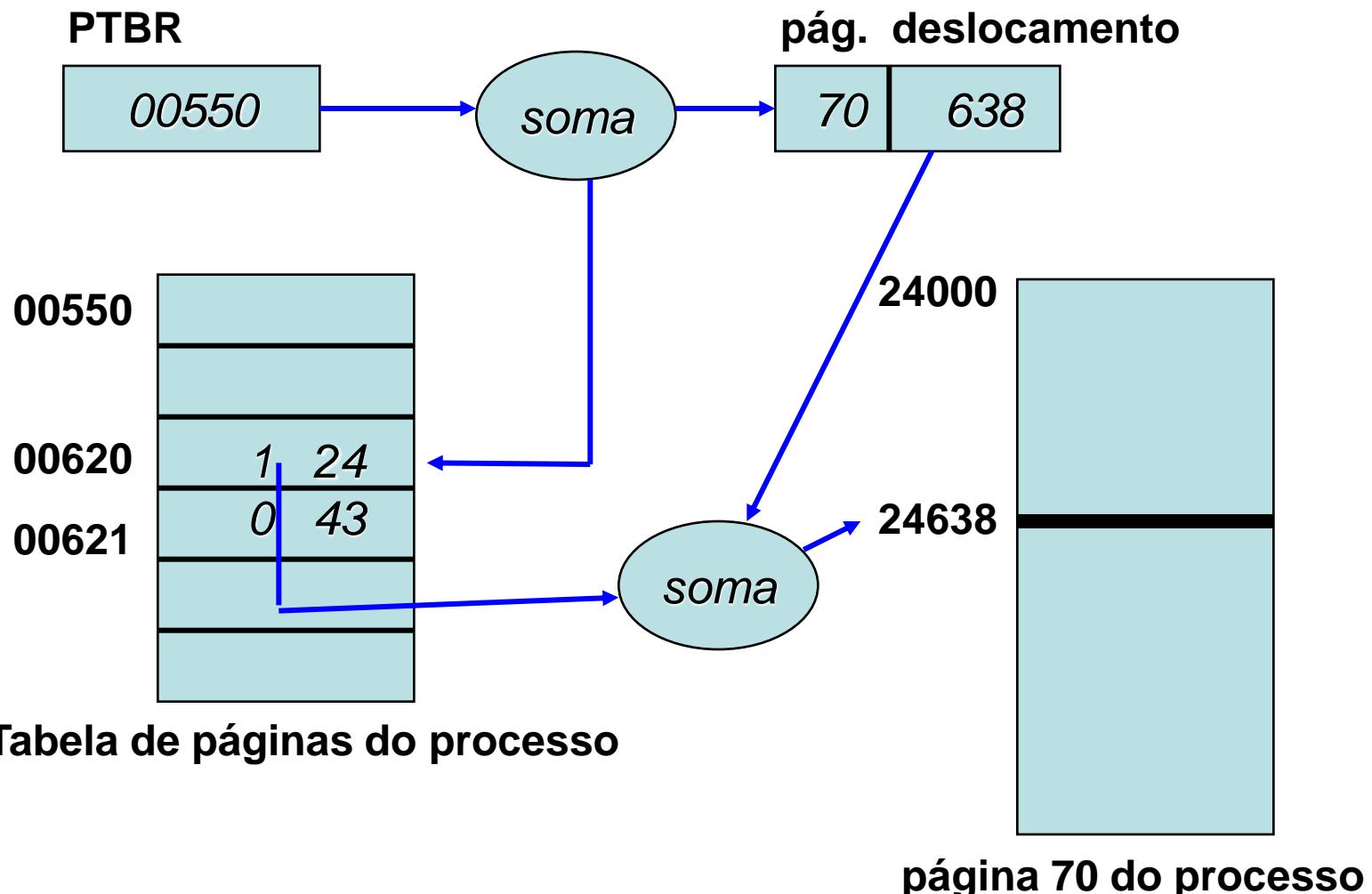
Necessidades para implementação

- ❑ Hardware deve suportar paginação ou segmentação
- ❑ Sistema operacional deve controlar o fluxo de páginas / segmentos entre a memória secundária (disco) e a memória principal
- ❑ Necessidade de gerenciar
 - Áreas livres e ocupadas
 - Mapeamento da memória lógica em memória física
 - Substituição de páginas / segmentos

Paginação por demanda

- ❑ Forma mais comum de implementar memória virtual
- ❑ Similar à paginação com *swapping*
 - Realiza-se o *swap-in/out* de uma página do processo (*page-in/out*)
- ❑ Carrega uma página à memória principal somente quando ela é necessária
 - Para saber se a página está ou não na memória existe um *bit* de referência (*bit* de presença)
 - *Bit=1*: página presente na memória
 - *Bit=0*: página não está na memória (*page-fault*)

Paginação por demanda



- Passos realizados pelo sistema
 - 1. Trap para sistema operacional
 - 2. Salvamento de contexto
 - 3. Detecção que a interrupção é por *page-fault*
 - 4. Verifica se referência é válida e determina localização da página no disco
 - 5. Solicita operação de leitura da página “faltante” do disco para o *frame*
 - 6. Passa processo da página “faltante” para estado suspenso e escalona outro processo
 - 7. Interrupção do disco (final de transferência da página)
 - 8. Atualiza tabela de páginas
 - 9. Passa processo do estado suspenso para estado pronto

Implementação da memória virtual

- ❑ A memória virtual pode ser implementada por:
 - Paginação sob demanda
 - Segmentação sob demanda
 - Segmentação com paginação sob demanda
- ❑ O princípio de funcionamento é o mesmo, porém é mais simples de gerenciar paginação por demanda
- ❑ Memória física é limitada, logo necessita-se
 - Política de carga de página
 - Política de localização da página
 - Política de substituição de página
- ❑ Partição de swap
 - Área do disco para armazenar páginas / segmentos
 - Organizada de forma diferente do sistema de arquivos para otimizar o acesso

Política de carga de páginas

- Carrega uma página para um *frame*
- Duas situações
 - *Frame* livre: carrega página no *frame*
 - Não há *frame* disponível
 - Libera espaço transferindo páginas da memória (*pager-out*) para o disco (área de *swap*)
 - Política de substituição para seleção da página “vítima”
- Otimizações
 - Carregar mais de uma página na memória
 - Nem toda página necessita *pager-out*
 - Páginas não modificadas
 - Páginas *read-only* (código)

Política de localização de páginas

- ❑ Determina na memória real a localização das páginas de um processo
- ❑ Realizado pelo hardware de paginação / segmentação do processador
 - Transparente sob o ponto de vista do sistema operacional

Substituição de páginas

- As páginas físicas (*frames*) podem ficar totalmente ocupadas à medida que páginas lógicas do processo são carregadas
- Necessidade de liberar uma página física (*frame*) para atender à falta de página
- O algoritmo de substituição de páginas é responsável pela escolha de uma página “vítima”

Bits auxiliares

- ❑ O objetivo é auxiliar a implementação do mecanismo de substituição de páginas
 - Não são absolutamente necessários
- ❑ *Bit de sujeira (dirty bit)*
 - Indica quando uma página foi alterada durante a execução do processo
 - Se a página não foi alterada, não é necessário salvar seu conteúdo no disco
- ❑ *Bit de referência (reference bit)*
 - Indica se uma página foi acessada dentro de um intervalo de tempo
- ❑ *Bit de tranca (lock bit)*
 - Evita que uma página seja selecionada como “vítima”

Política de substituição de páginas

- Utilizada quando não há mais *frames* livres
- Seleciona na memória uma página a ser substituída quando outra página necessita ser carregada na memória
 - O problema é determinar a página menos necessária
- Certas páginas que não devem ser substituídas podem ser “grampeadas” a *frames* (*frame locking*)
 - Código e estruturas de dados do sistema operacional
 - *Buffers* de E/S

Política de substituição de páginas

- Selecionar para substituição uma página que será referenciada dentro do maior intervalo de tempo
 - Algoritmo ótimo
 - Impossível conhecer o futuro
- Algoritmos usados para substituição de páginas
 - *First-come-first-served* (FCFS)
 - *Least Recently Used* (LRU)
 - Baseado em contadores

First-come-first-served (FCFS)

- Também denominado *First-in, First-out* (FIFO)
- Frames na memória principal são alocados a páginas de um buffer circular
- Implementação simples
 - Substitui a página que está a mais tempo na memória
 - Desvantagem é que a página substituída pode ser necessária logo a seguir

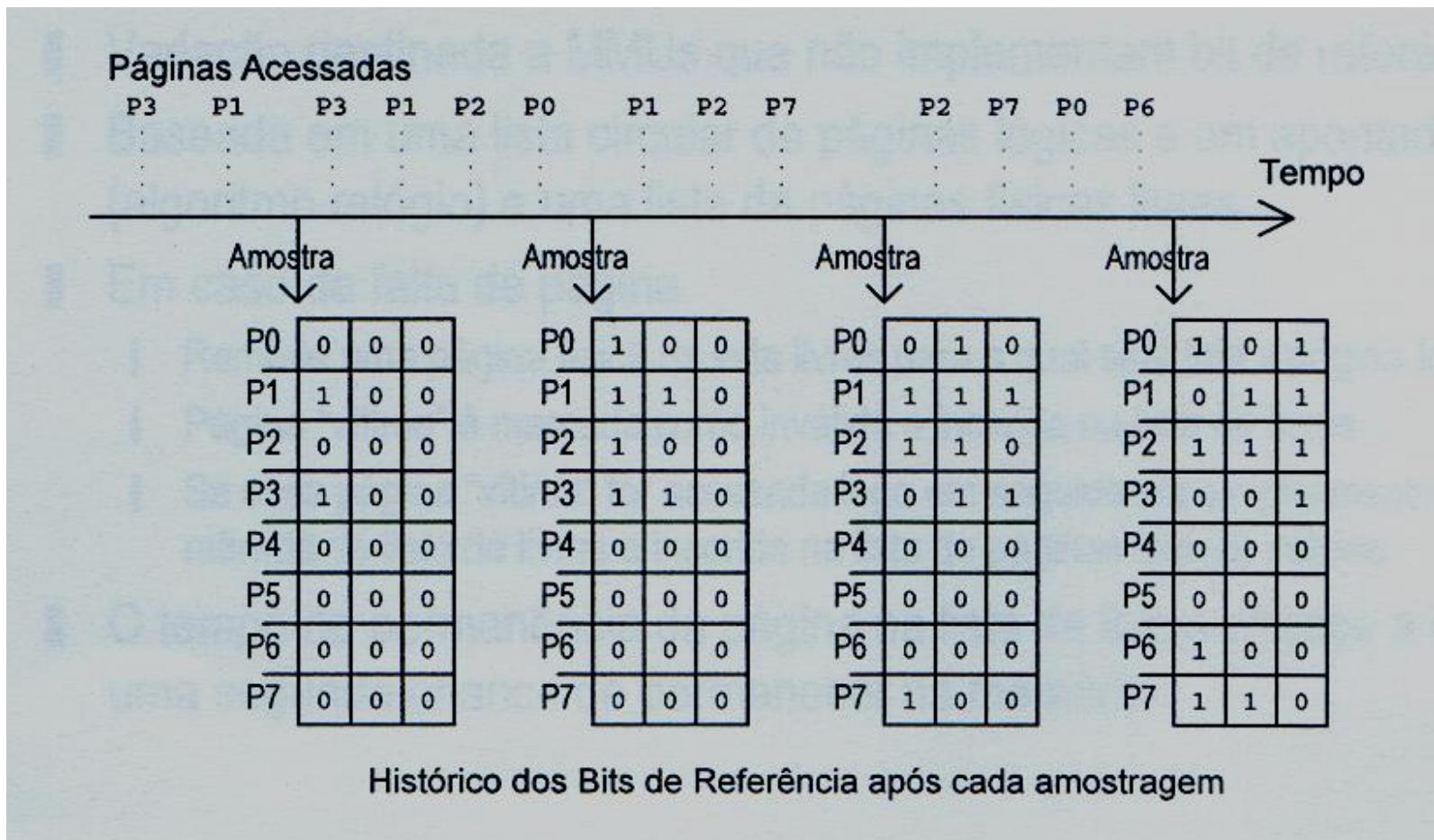
Least Recently Used (LRU)

- Premissa básica é que páginas acessadas recentemente por um processo serão novamente acessadas em um futuro próximo
- Página a ser substituída é a página referenciada a mais tempo
 - Pelo princípio da localidade, esta página deve ser a de menor probabilidade de ser referenciada em um futuro próximo
 - Desvantagem
 - Cada página deve possuir a “data” da última referência e esta é uma característica raramente suportada pelas MMUs

□ Implementação

- Contador na tabela de páginas
- Tempo da última referência (*time stamp*)
- Desvantagem é o *overhead*
 - Atualização da entrada na tabela de páginas
- Seleção da página “vítima”
- Pilha
 - Página referenciada é inserida na pilha
 - No topo da pilha está a página mais recentemente referenciada
 - Na base da pilha está a página menos recentemente referenciada
 - Lista duplamente encadeada

Construção de histórico de bits de referência



Algoritmo de segunda chance

- Denominado também de algoritmo do relógio
- Baseado em *bit* de referência
- Considera que páginas lógicas formam uma lista circular
 - Apontador percorre a lista circular informando qual será a próxima “vítima”
 - Se página apontada (sentido horário) tem o *bit* de referência = 1, então
 - Posiciona *bit* de referência em zero e mantém página na memória
 - Substitui próxima página que tem *bit* de referência = 0

Algoritmo de substituição baseado em contadores

- A cada página é associado um contador de número de referências
- Duas políticas básicas
 - *Least Frequently Used* (LFU)
 - Substitui a página que possui o menor valor
 - *Most Frequently Used* (MFU)
 - Não substitui a página que possui o menor valor
- São algoritmos não utilizados
 - *Overhead*
 - Comportamento muito distante do algoritmo ótimo

Alocação de páginas físicas (*frames*)

- ❑ Como se deve alocar *frames* para “n”processos?
 - Usando um algoritmo de alocação
 - Alocação igualitária
 - Alocação proporcional
- ❑ Qual o mínimo necessário de *frames* por processo?
 - Quanto menor o número de *frames* alocados a um processo, maior será a taxa de *page faults*
 - Queda de desempenho do sistema
 - Tentar manter o máximo de páginas na memória
- ❑ De onde virão os *frames* a serem alocados?
 - Alocação local
 - Alocação global

Alocação igualitária

- Princípio é dividir os m frames da memória física entre os n processos aptos a executar
 - Cada processo recebe m/n frames
 - A sobra pode ser um *pool* de frames livres
 - O número de frames é ajustado dinamicamente em função do grau de multiprogramação
- Desvantagem
 - Provoca distorções, já que os processos possuem diferentes necessidades de memória

Alocação proporcional

- ❑ O princípio é alocar *frames* em função do tamanho do processo
- ❑ A alocação deve ser reajustada dinamicamente em função do grau de multiprogramação
- ❑ Pode-se empregar a prioridade de um processo, ao invés do seu tamanho

Alocação local

- ❑ A gerência de memória define quantas páginas físicas cada processo pode dispor
- ❑ Em caso de falta de páginas, a substituição ocorre entre as próprias páginas do processo que gerou a falta
- ❑ Desvantagens
 - Definição do número de páginas físicas para cada processo impede que um processo utilize páginas físicas disponíveis pertencentes a outros processos

Alocação global

- Uma lista única de gerência de páginas físicas compartilhada por todos os processos
- Um processo pode receber uma página física de outro processo
- Desvantagens
 - O conjunto de páginas físicas ocupado por um processo depende do comportamento dos outros processos
 - Um processo de maior prioridade pode recuperar as páginas físicas de um processo de menor prioridade
- É o método mais empregado

Thrashing

- ❑ Um processo está em *thrashing* quando passa a maior parte do tempo de execução fazendo paginação
- ❑ Consequências do *thrashing*
 - Baixa taxa de uso da CPU para execução de processos dos usuários
 - Adição de processos implica em maior necessidade de *frames*
 - Conclusão:
 - Existe um ponto onde a multiprogramação compromete o desempenho do sistema

Thrashing

- ❑ Para retirar um sistema do estado de *thrashing* é necessário suspender alguns processos temporariamente
- ❑ O mecanismo natural é o *swapping*
 - Não é desejável porque aumenta o tempo de resposta dos processos
- ❑ Deve-se então tentar prevenir o *thrashing*

- Pode-se usar as abordagens
 - Método frequência de falta de página
 - Objetivo é controlar a taxa de faltas de páginas para manter dentro de um limite
 - Taxa maior que o máximo aceitável
 - Há processos que estão necessitando de páginas físicas
 - Realiza o swap-out de alguns processos
 - Libera páginas físicas para processos que necessitam

Método do *working-set*

- Método do *working-set*
 - Baseado no histórico de *bits* de referência
 - Tamanho do histórico e o intervalo de tempo entre as amostragens determina se páginas são consideradas em uso ou não
 - Baseado no princípio da localidade
 - Examina-se as últimas n referências a páginas
 - Se página está em uso: pertence ao *working-set*
 - Se página não está em uso: eliminada do *working-set* em n unidades

Pré-paginação

- Consiste em trazer para a memória todo o *working-set* de um processo
 - Possível para casos em que processos que estão realizando transições dos estados bloqueado / suspenso para apto
- Custo da pré-paginação deve ser menor que custo de tratamento de falta de páginas

Tamanho da página

□ Fatores a serem considerados

- Fragmentação
- Tamanho de estruturas internas do sistema operacional (tabelas de páginas)
- Overhead das operações de E/S
- Localidade
- Menor o tamanho da página
 - Menor a quantidade de fragmentação interna
 - Maior a quantidade de páginas por processo
 - Maior a quantidade de páginas na memória
- Mais páginas por processo, maior tabela de páginas
 - Maior a necessidade de uso de memória
 - Mais desperdício de memória
 - Maior ocorrência de *page-faults*

Referências Bibliográficas

DEITEL, H. M.; DEITEL P. J. e CHOFFNES, D. R. Sistemas Operacionais, 2 ed. Pearson Prentice Hall, 2005.

OLIVEIRA, R. S.; CARÍSSIMI, A. S. e TOSCANI, S. S. Sistemas Operacionais, 1 ed. Ed. Sagra-Luzzatto. 2001.

TANENBAUM, A. S. Sistemas Operacionais Modernos. 2 ed. Pearson Prentice Hall, 2003.