Sistemas Operacionais

Gerência de memória

2ª edição

Capítulo 6

Revisão: Fev/2003

Introdução

- Multiprogramação implica em manter-se vários processos em memória
- Memória necessita ser alocada de forma eficiente para permitir o máximo possível de processos
- Existem diferentes técnicas para gerência de memória
  - Dependem do hardware do processador

Considerações gerais

- Um sistema de memória possui pelo menos dois níveis:
  - Memória principal: acessada pela CPU
  - Memória secundária: discos
- Programas são armazenados em disco
  - Executar um programa se traduz em transferi-lo da memória secundária à memória primária
- Qualquer sistema operacional tem gerência de memória
  - Monotarefa: gerência é simples
  - Multitarefa: complexa
- Algoritmos de gerência de memória dependem de facilidades disponíveis pelo hardware da máquina

Memória lógica e memória física

- Memória lógica
  - É aquela que o processo “enxerga”
  - Endereços lógicos são aqueles manipulados por um processo
- Memória física
  - Implementada pelos circuitos integrados de memória
  - Endereços físicos são aqueles que correspondem a uma posição real de memória

Endereço lógico versus endereço físico

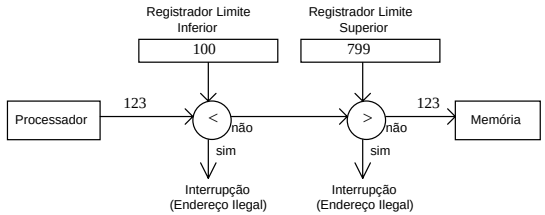
- Espaço lógico de um processo é diferente do espaço físico
  - Endereço lógico: gerado pela CPU (endereço virtual)
  - Endereço físico: endereços enviados para a memória RAM
- Programas de usuários "vêem" apenas endereços lógicos
- Endereços lógicos são transformados em endereços físicos no momento de execução dos processos

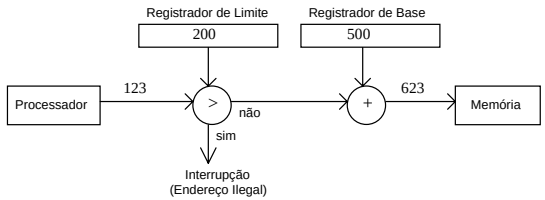
Unidade de gerência de memória

- *Memory Management Unit* (MMU)
- Hardware que faz o mapeamento entre endereço lógico e endereço físico

- Complexidade variável de acordo com a funcionalidade oferecida
  - Mecanismos de suporte para proteção, carga de programas, tradução de endereços lógicos a endereços físicos, etc...

Exemplos de MMU





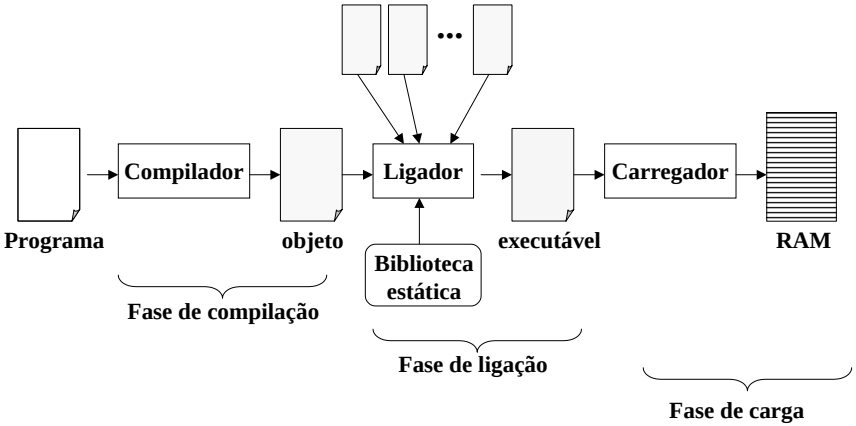
Execução de programas

- Um programa deve ser transformado em um processo para pode ser executado
  - Alocação de um descritor de processos
  - Alocação de áreas de memória para código, dados e pilha
- Transformação é feita através de uma série de passos, alguns com a ajuda do próprio programador
  - Compilação, diretivas de compilação e/ou montagem, ligação, etc...
- Amarração de endereços (*binding*)

Amarração de endereços (*binding*)

- Atribuição de endereços (posições de memória) para código e dados pode ser feita em três momentos diferentes:
  - Em tempo de compilação
  - Em tempo de carga
  - Em tempo de execução
- Diferenciação entre o endereço lógico e o endereço físico
  - Como traduzir endereço lógico em endereço físico
  - Código absoluto e código relocável

Transformação de programa em processo



Carregador absoluto versus carregador relocador

- Programador não tem conhecimento onde o programa será carregado na memória
- Endereço só é conhecido no momento da carga
  - Durante execução do programa sua localização física pode ser alterada
    - e.g.; procedimento de *swapping*
- Necessidade de traduzir endereços lógicos à endereços físicos
- Relocação é a técnica que fornece realiza essa tradução
  - Via software: carregador relocador
  - Via hardware: carregador absoluto

Código relocável

- Carregador relocador
  - Correção de todas as referências a memória de forma a corresponder ao endereço de carga do programa
- Necessidade de identificar quais endereços devem ser corrigidos
  - Código relocável
  - Mapeamento das posições a serem corrigidas é mantida através de tabelas
- Código executável mantém informações de relocação na forma de tabelas
  - No momento da carga o programa executável é interpretado e os endereços corrigidos

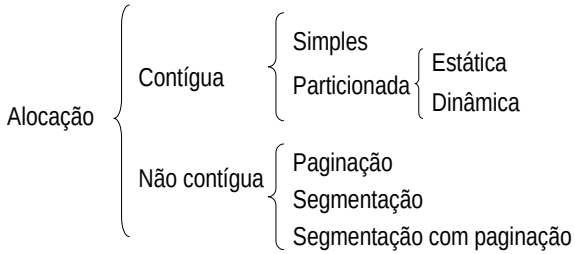
Código absoluto

- Carregador absoluto
  - Não realiza correção de endereços no momento da carga do programa em memória
- Código executável absoluto não necessita manter tabelas de endereços a serem corrigidos
- Endereço de carga
  - Fixo pelo programa (programador)
  - Qualquer
    - Correção pode ser feita automáticamente, de forma transparente, a partir de registradores de base

Mecanismos básicos de gerência de memória

- Um programa (processo) para ser executado deve estar na memória
  - Onde deve ser carregado?
    - Problema de alocação de memória
- A alocação de memória depende de:
  - Código absoluto versus código relocável
  - Necessidade de espaço contíguo ou não
- Necessidade de gerenciamento da memória
  - Determinação de áreas livres e ocupadas
  - Racionalizar a ocupação da memória

Mecanismos para alocação de memória



- Até estudarmos memória virtual supor que para um programa ser executada ele necessita estar carregado completamente em memória

Alocação contígua simples

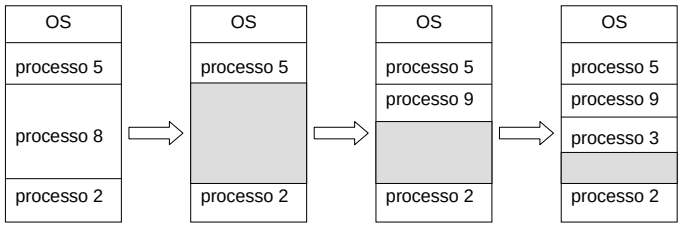
- Sistema mais simples
- Memória principal é dividida em duas partições:
  - Sistema operacional (parte baixa da memória)
  - Processo do usuário (restante da memória)
- Usuário tem controle total da memória podendo inclusive acessar a área do sistema operacional
  - e.g. DOS (não confiável)
- Evolução:
  - Inserir proteção através de mecanismos de *hardware* + *software*
    - Registradores de base e de limite
    - *Memory Management Unit* (MMU)

Alocação contígua particionada (1)

- Existência de múltiplas partições
- Imposta pela multiprogramação
- Filosofia:
  - Dividir a memória em blocos (partições)
  - Cada partição pode receber um processo (programa)
  - Grau de multiprogramação é fornecido pelo número de partições
    - Importante: não considerando a existência de *swapping*
- Duas formas básicas:
  - Alocação contígua com partições fixa (estática)
  - Alocação contígua com partições variáveis (dinâmica)

Alocação contígua particionada (2)

- O sistema operacional é responsável pelo controle das partições mantendo informações como:
  - Partições alocadas
  - Partições livres
  - Tamanho das partições



Alocação contígua particionada fixa

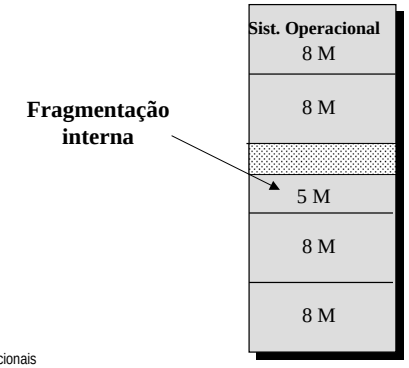
- Memória disponível é dividida em partições de tamanho fixo que podem ser do mesmo tamanho ou não
- Questões:
  - Processos podem ser carregados em qualquer partição?
    - Depende se código é absoluto ou relocável
  - Número de processos que podem estar em execução ao mesmo tempo
    - Sem *swapping* → igual ao número de partições (máximo)
    - Com *swapping* → maior que número de partições
  - Programa é maior que o tamanho da partição
    - Não executa a menos que se empregue um esquema de *overlay*

Gerenciamento de partições fixas

- Com código absoluto
  - Um processo só pode ser carregado na área de memória (partição) para a qual foi compilado
  - Pode haver disputa por uma partição mesmo tendo outras livres
    - Processo é mantido no escalonador de longo prazo (termo)
    - Empregar *swapping*
- Com código relocável
  - Um processo de tamanho menor ou igual ao tamanho da partição pode ser carregado em qualquer partição disponível
  - Se todas as partições estão ocupadas, duas soluções:
    - Processo é mantido no escalonador de longo prazo (termo)
    - Empregar *swapping* (escalonamento a médio prazo)

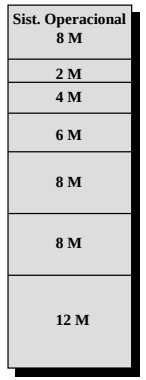
Fragmentação Interna

- Problema da alocação fixa é uso ineficiente da memória principal
- Um processo, não importando quão pequeno seja, ocupa uma partição inteira
  - Fragmentação interna



Paliativo para reduzir fragmentação interna

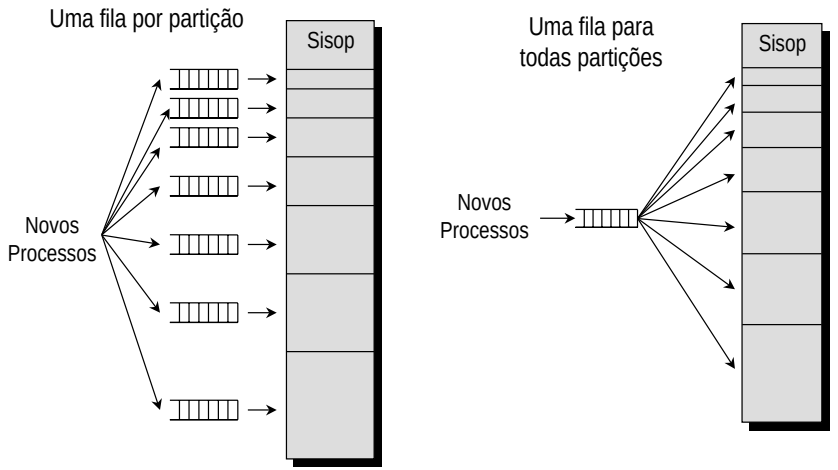
- Partições de tamanho diferentes



Algoritmos para alocação de partições fixas(1)

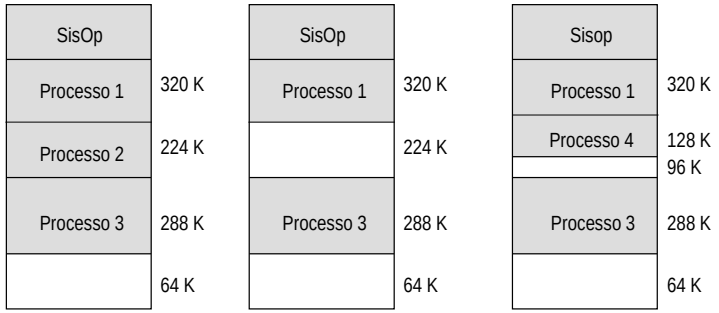
- Se código é absoluto a alocação é determinada na fase de montagem, compilação ou ligação
- Se código é relocável:
  - Partições de igual tamanho
    - Não importa qual partição é utilizada
  - Partições de diferentes tamanhos
    - Atribui o processo a menor partição livre capaz de armazená-lo
    - Processo são atribuídos a partições de forma a minimizar o desperdício de memória (fragmentação interna)

Algoritmos para alocação de partições fixas(2)



Alocação particionada dinâmica

- Objetivo é eliminar a fragmentação interna
- Processos alocam memória de acordo com suas necessidades
- Partições são em número e tamanho variáveis

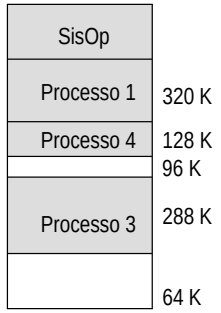


Fragmentação externa

- A execução de processos pode criar pedaços livres de memória
  - Pode haver memória disponível, mas não contígua
    - Fragmentação externa

Exemplo:

Criação processo 120K



Soluções possíveis fragmentação externa

- Reunir espaços adjacentes de memória
- Empregar compactação
  - Relocar as partições de forma a eliminar os espaços entre elas e criando uma área contígua
  - Desvantagem:
    - Consumo do processador
    - Acesso a disco
- Acionado somente quando ocorre fragmentação
- Necessidade de código relocável

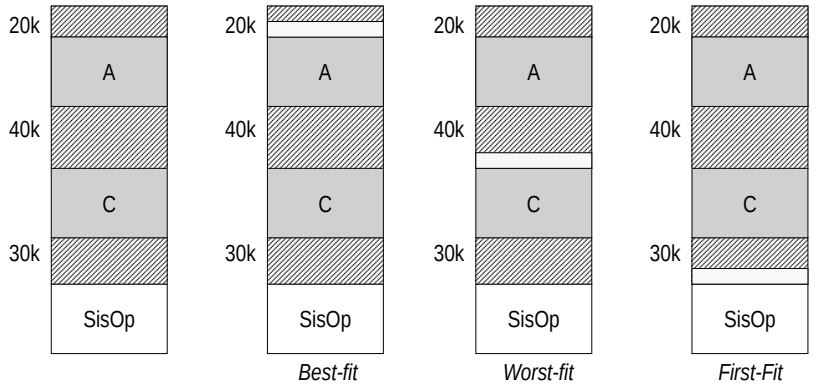
Gerenciamento de partições dinâmicas

- Determinar qual área de memória livre será alocada a um processo
- Sistema operacional mantém uma lista de lacunas
  - Pedaços de espaços livres em memória
- Necessidade de percorrer a lista de lacunas sempre que um processo é criado
  - Como percorrer essa lista??

Algoritmos para alocação contígua dinâmica

- *Best fit*
  - Minimizar tam_processo - tam_bloco
  - Deixar espaços livres os menores possíveis
- *Worst fit*
  - Maximizar tam_processo - tam_bloco
  - Deixar espaços livres os maiores possíveis
- *First fit*
  - tam_bloco > tam_processo
- *Circular fit*
  - Variação do *first-fit*

Exemplos: Algoritmos alocação contígua dinâmica

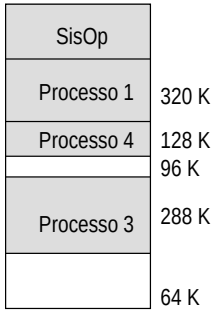


Desvantagem de partições variáveis

- Tende a criar lacunas de memória livres que individualmente podem não ser suficientes para acomodar um processo
  - Pode haver memória livre, mas não contígua
    - Fragmentação externa

Exemplo:

Criação processo 120k

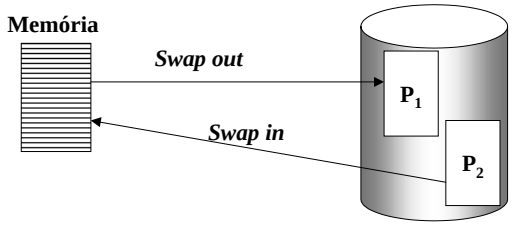


Soluções possíveis para fragmentação externa

- Reunir espaços adjacentes de memória
- Empregar compactação
  - Relocar as partições de forma a eliminar os espaços entre elas e criando uma área contígua
  - Desvantagem:
    - Consumo do processador
    - Acesso a disco
- Acionado somente quando ocorre fragmentação
- Necessidade de código relocável

Swapping (1)

- Processo necessita estar na memória para ser executado
  - Se não há mais espaço em memória é necessário fazer um rodizio de processos em memória



- Memória secundária suficientemente grande para armazenar cópias de todos os processos de usuários → *backing store*

Swapping (2)

- Tempo do *swap* é proporcional ao tamanho do processo
  - Possui influência na troca de contexto
    - Política de *swapping*
    - Escalonador de médio prazo (termo)
- Atenção!!! Processos que realizam E/S
  - Nunca realizar *swap* em processos que estão com E/S pendente
  - Utilizar buffers de E/S internos ao sistema
    - Evitar que o E/S seja transferido a endereços de memória inválidos
- Existem variantes do sistema de *swapping* utilizados em sistemas como UNIX ou Microsoft Windows

Leituras complementares

- R. Oliveira, A. Carissimi, S. Toscani *Sistemas Operacionais* Editora Sagra-Luzzato, 2001.
  - Capítulo 6, seções 6.2 e 6.3
- A. Silberchatz, P. Galvin *Operating System Concepts* Addison-Wesley. 4th edition.
  - Capítulo 8, Seção 8.4

Introdução

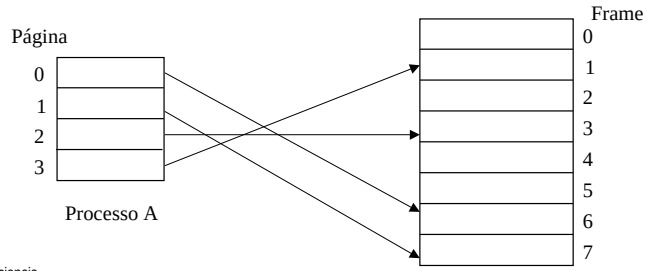
- Problemas com alocação particionada
  - Necessidade de uma área contígua de memória (tamanho do processo)
  - Fragmentação interna (partições fixas) ou externa (partições variáveis)
- Nova abordagem é considerar a existência de um espaço de endereçamento lógico e de um espaço de endereçamento físico
  - O espaço de endereçamento físico não precisa ser contíguo
  - Necessita "mapear" o espaço lógico no espaço físico
    - Dois métodos básicos:
      - Paginação
      - Segmentação
- Suposição: para ser executado o processo necessita estar completamente carregado em memória

Paginação (1)

- A memória física (sistema) e a memória lógica (processo) são dividos em blocos de tamanho fixo e idênticos
  - Memória física dividida em blocos de tamanho fixo denominados de frames
  - Memória lógica dividide em blocos de tamanho fixo denominados de páginas
- Elimina a fragmentação externa e reduz a fragmentação interna

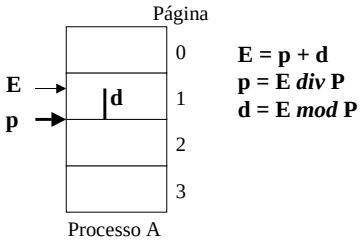
Paginação (2)

- Para executar um processo de *n* páginas, basta encontrar *n* frames livres na memória
  - Páginas são carregadas em qualquer frame livre
- Necessidade de traduzir endereços lógicos (páginas) em endereços físicos (frames)

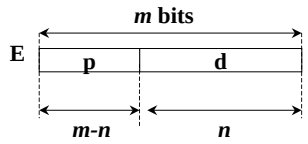


Paginação: Endereço lógico

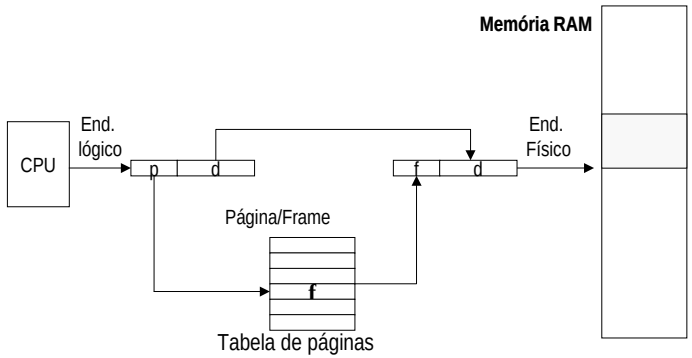
- Endereço lógico é dividido em duas componentes:
  - Número da página
  - Deslocamento dentro de uma página
- Tamanho da página (P) pode assumir qualquer tamanho porém emprega-se um tamanho potência de 2 para facilitar operações *div* e *mod*



E = p + d
p = E *div* P
d = E *mod* P

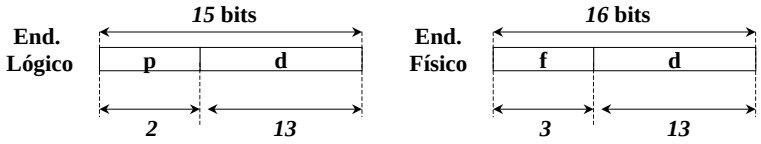


Tradução de endereço lógico em endereço físico

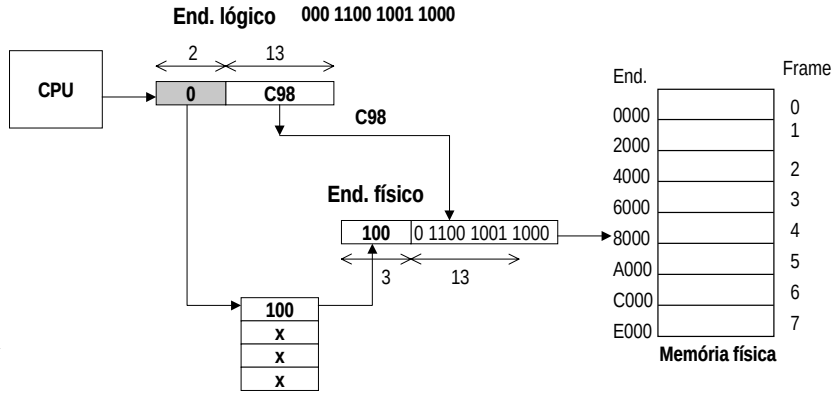


Exemplo de paginação (1)

- Características do sistema:
  - Memória física: 64 kbytes (16 bits)
  - Tamanho processo (máx): 32 kbytes (15 bits)
  - Páginas 8 kbytes
- Paginação:
  - Número de frames: 64/8 = 8 (0 a 7) → 3 bits
  - Número de páginas: 32/8 = 4 (0 a 3) → 2 bits
  - Deslocamento: 8 kbytes → 13 bits



Exemplo de paginação (2)



Características da paginação

- Paginação é um tipo de relocação (via hardware)
- Não gera fragmentação externa
- Fragmentação interna é restrita apenas a última página
- Importante:
  - Visão do usuário: espaço de endereçamento contíguo
  - Visão do sistema: processo é «esparramado » na memória física
- *n* páginas são alocadas a *n* frames implicando na criação de uma tabela de correspondência
  - Tabela de páginas
- Facilita implementação de proteção e compartilhamento

Tamanho da página

- Páginas grandes significam
  - Processos compostos por menos páginas (tabela de páginas menores)
  - Aumento da fragmentação interna na última página
- Páginas pequenas significam
  - Processos compostos por mais páginas (tabela de página maiores)
  - Diminuição da fragmentação interna na última página
- Objetivos conflitantes
- Tamanho da página é imposto pelo hardware (MMU)
  - Valores típicos variam entre 1 kbyte e 8 kbytes

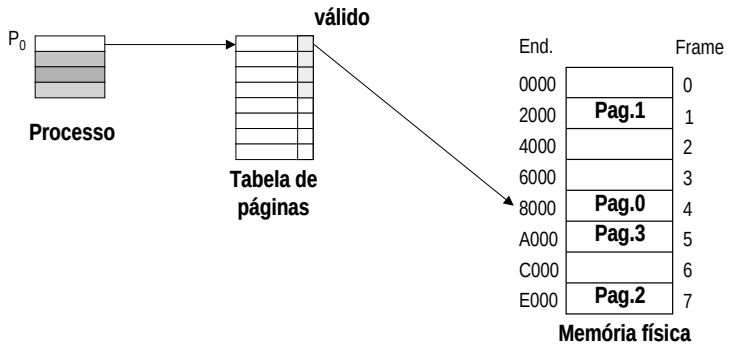
Questões relacionadas com a gerência de páginas

- A gerência de memória deve manter controle de áreas livres e ocupadas
- Inclusão de mecanismos de proteção
  - Evitar que um processo acesse área (páginas) de outros processos
  - Garantir que um processo acesse apenas endereços válidos
  - Garantir acessos autorizados a uma posição de memória
    - e.g.: páginas *read-only, read-write*, etc.
- Inclusão de mecanismos de compartilhamento
  - Permitir que dois ou mais processos dividam uma área comum
    - e.g.: páginas de código de um aplicativo do tipo editor de texto

Proteção

- Proteção de acesso é garantida por definição:
  - Processos acessam somente suas páginas → end. válidos
  - Endereço inválido apenas na última página
    - Se houver fragmentação interna
- Inclusão de bits de controle na tabela de página (por entrada)
  - Indicação se a página é de leitura, escrita ou executável
- Bit de validade:
  - Página pertence ou não ao end. lógico do processo

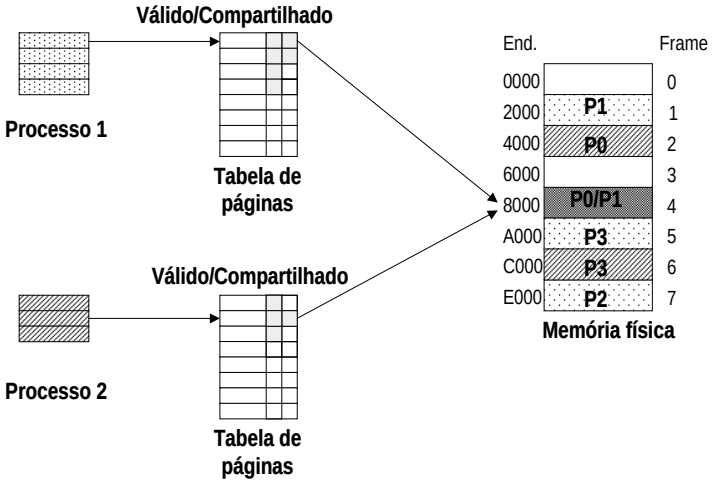
Exemplo de proteção



Compartilhamento de páginas

- Código compartilhado
  - Uma cópia do código (*read-only*, re-entrante) pode ser compartilhada entre vários processos (e.g.; editores de texto, compiladores, etc...)
  - O código compartilhado pertence ao espaço lógico de todos os processos
- Dados e código próprios
  - Cada processo possui sua própria área de código e seus dados

Exemplo de compartilhamento



Implementação da tabela de páginas

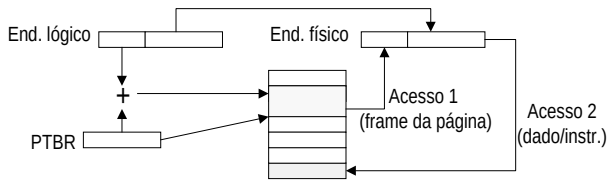
- Sistema operacional deve manter :
  - Frames livres/alocados
  - Número de frames e de páginas de um processo
  - Uma entrada para cada frame e para cada processo
- Como implementar a tabela de páginas?
  - Registradores
  - Memória

Implementação da tabela de páginas via registradores

- Tabela de páginas é mantida por um conjunto de registradores
  - Cada página um registrador
  - No descritor de processo devem ser mantidas cópias dos registradores
    - Troca de contexto: atualização dos registradores
- Desvantagem é o número de registradores

Implementação da tabela de páginas em memória

- Tabela de páginas é mantida em memória
  - *Page-table base register (*PTBR): início da tabela de páginas
  - *Page-table length register* (PTLR): tamanho em número de entradas.
- Cada acesso a dado/instrução necessita, no mínimo, dois acessos a memória
  - Número de acesso depende da largura da entrada da tabela de página e de como a memória é acessada (byte, word, etc...)



Translation look-aside buffers (TLBs)

- Uma espécie de meio termo entre implementação via registradores e via memória
- Baseada em uma memória cache especial (TLB) composta por um banco de registradores (memória associativa)
- Idéia é manter a tabela de páginas em memória com uma cópia parcial da tabela em um banco de registradores (TLB)
  - Página acessada está na TLB (*hit*): similar a solução de registradores
  - Página acessada não está na TLB (*miss*): similar a solução via memória

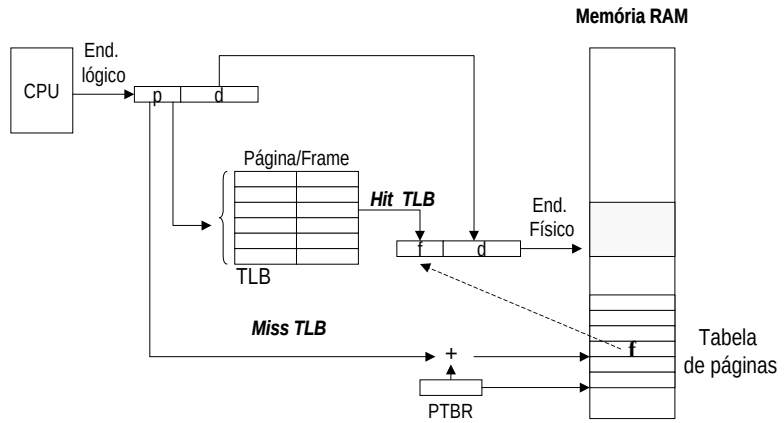
Registradores associativos

- Registradores associativos permitem a busca de valores por conteúdo, não por endereços
  - Pesquisa paralela

Key	value

- Funcionamento:
  - Se valor "*key*" está na memória associativa, se obtém valor (*value*).

Implementação da tabela de páginas via TLB

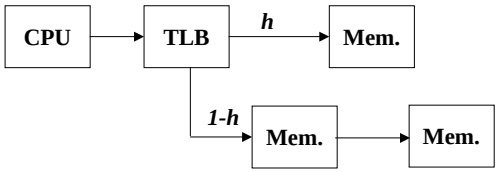


Aspectos relacionados com o uso de TLB

- Melhora o desempenho no acesso a tabela de páginas
  - Tempo de acesso 10 vezes menor que uma memória RAM
- Desvantagem é o seu custo
  - Tamanho limitado (de 8 a 2048 entradas)
  - Uma única TLB (pertence a MMU) que é compartilhada entre todos os processos
    - Apenas as páginas em uso por um processo necessitam estar na TLB
- Um acesso é feito em duas partes:
  - Se página está presente na TLB (*hit*) a tradução é feita
  - Se página não está presente na TLB (*miss*), consulta a tabela em memória e atualiza entrada na TLB

Hit ratio (h)

- Probabilidade de qualquer dado referenciado estar na memória, no caso, na TLB
  - Taxa de acerto: *hit ratio*
  - Seu complemento é a taxa de erro: *miss ratio*



$$t_{acesso} = t_{a_{tlb}} + t_{a_{mem}}$$

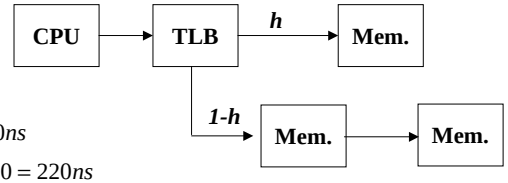
$$t_{acesso} = t_{a_{tlb}} + t_{a_{mem}} + t_{a_{mem}}$$

$$t_{medio} = h \times \left(t_{a_{tlb}} + t_{a_{mem}}\right) + (1-h) \times \left[t_{a_{tlb}} + t_{a_{mem}} + t_{a_{mem}}\right]$$

Exemplo: influência do *hit ratio* no desempenho

$t_{acesso\_tlb} = 20ns$

$t_{acesso\_mem} = 100ns$



$t_{acesso(hit)} = 20 + 100 = 120ns$

$t_{acesso(miss)} = 20 + 100 + 100 = 220ns$

$hit\_ratio = 0.80$

$t_{medio} = 0.80 \times (120) + 0.20 \times (220) = 140ns$

$hit\_ratio = 0.98$

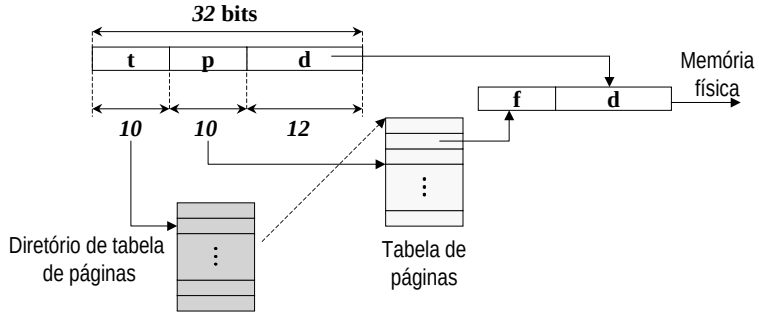
$t_{medio} = 0.98 \times (120) + 0.02 \times (220) = 122ns$

Paginação multinível

- Na prática as tabelas de página possuem tamanho variável
  - Como dimensionar o tamanho da tabela de páginas?
    - Fixo ou variável conforme a necessidade?
  - Como armazenar a tabela de páginas?
    - Contígüo em memória → fragmentação externa
    - Paginando a própria tabela
- A paginação multinível surge como solução a esses problemas
  - Diretórios de tabela de páginas (*n* níveis)
  - Tabelas de páginas

Exemplo: Paginação a dois níveis

- Processadores 80x86
  - End. Lógico: 4 Gbytes (32 bits)
  - Páginas: 4 kbytes
  - Tamanho da tabela de páginas: 4 Gbytes / 4 kbytes = 1048576 entradas



Paginação a três níveis

- Típico de arquiteturas de processadores de 64 bits

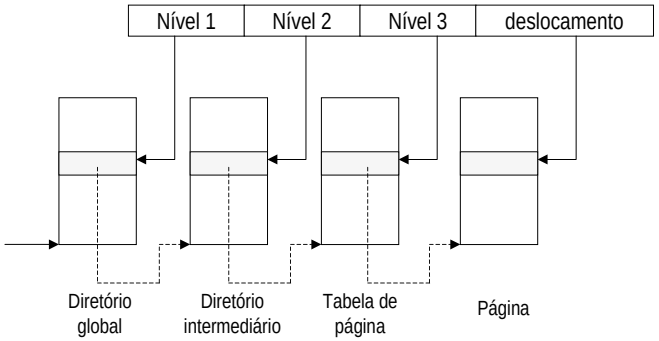
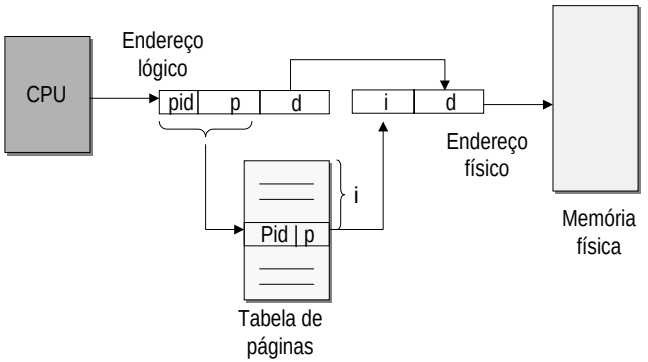


Tabela de páginas invertida

- Problema com tabela de páginas é o seu tamanho
- Tabela de páginas invertida surge como uma solução
  - Uma tabela de páginas para todo o sistema (não mais por processo)
  - Uma entrada para cada frame
  - Endereço lógico da página e a qual processo pertence
- Endereço lógico é formado por <*process_id*, página, deslocamento>
- Cada entrada da tabela possui <*process_id*; página>
- Tabela é pesquisada e retorna, se presente, o indice *i* associado a entrada
  - Cada índice corresponde a um frame

Esquema tabela de páginas invertida



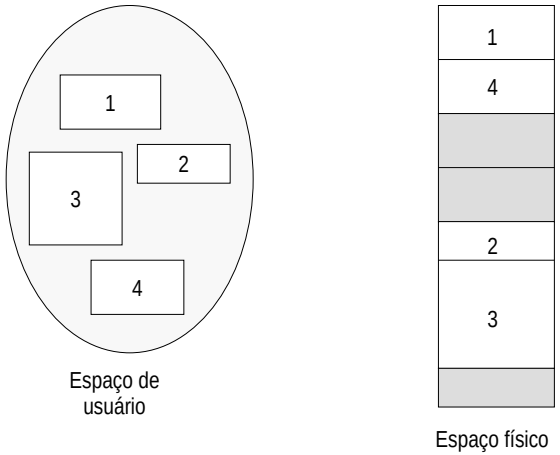
Leituras complementares

- R. Oliveira, A. Carissimi, S. Toscani *Sistemas Operacionais* Editora Sagra-Luzzato, 2001.
  - Capítulo 6, seção 6.5
- A. Silberchatz, P. Galvin *Operating System Concepts*. 4rd edition Addison-Wesley.
  - Seção 8.5

Segmentação

- Leva em consideração a visão de programadores e compiladores
- Um programa é uma coleção de segmentos, tipicamente:
  - Código
  - Dados alocados estaticamente
  - Dados alocados dinamicamente
  - Pilha
- Um segmento pode ser uma únidade lógica
  - e.g.: procedimentos (funções), bibliotecas
- Gerência de memória pode dar suporte diretamente ao conceito de segmentos

Esquema lógico da segmentação



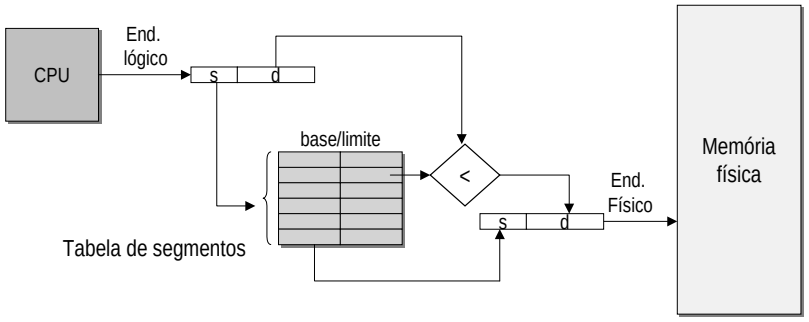
Endereço lógico em segmentação

- Endereço lógico é composto por duas partes:
  - Número de segmento
  - Deslocamento dentro do segmento
- Os segmentos não necessitam ter o mesmo tamanho
- Existe um tamanho máximo de segmento
- Segmentação é similar a alocação particionada dinâmica

Tradução de endereço lógico em endereço físico

- Tradução é feita de forma similar a paginação (via tabela)
  - Tabela de segmentos
- Entrada na tabela de segmento:
  - base: endereço inicial (físico) do segmento na memória
  - limite: tamanho do segmento
- Necessidade de verificar a cada acesso se ele é válido
  - Hardware (comparador)

Esquema de tradução da segmentação



Exemplo de tradução de endereço lógico em endereço físico

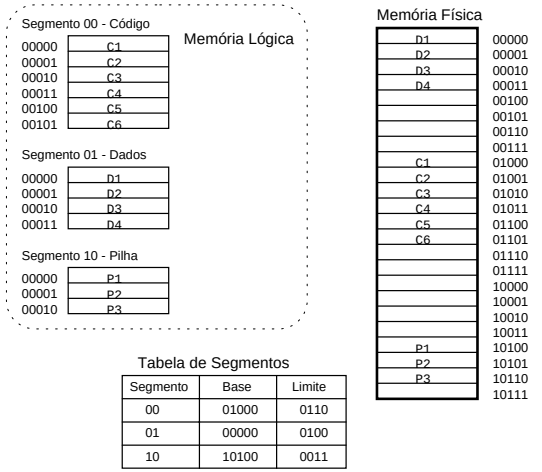


Tabela de Segmentos

Segmento	Base	Limite
00	01000	0110
01	00000	0100
10	10100	0011

Implementação da tabela de segmentos

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

Implementação da tabela de segmentos via registradores

- Tabela de segmentos é mantida por um conjunto de registradores
  - Cada segmento dois registradores (base e limite)
  - No descritor do processo devem ser mantidas cópias dos registradores
    - Troca de contexto: atualização dos registradores
- Número de registradores impõem uma limitação prática ao tamanho da tabela de segmentos (como na paginação)

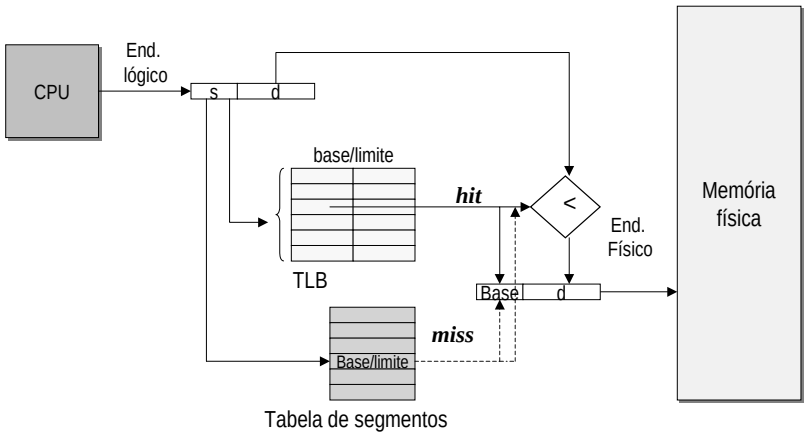
Implementação da tabela de segmentos em memória

- Tabela de segmentos armazenada em memória
- *Segment-table base register* (STBR): localização do início da tabela de segmentos na memória
- *Segment-table length register* (STLR): indica o número de segmentos de um processo
  - Segmento é válido apenas se: s < STLR.

Problemas com implementação da tabela em memória

- Problemas similares ao da paginação:
  - Tabela pode ser muito grande
  - Dois acessos a memória para acessar um dado/instrução
- Solução:
  - Empregar uma TLB
- Observação (válida também para a paginação)
  - A consulta a tabela em memória provoca no mínimo 2 acessos a memória, pois uma entrada na tabela pode representar mais de um acesso.

Implementação da tabela de segmentos via TLB



Aspectos de proteção e compartilhamento

- Os príncipios já estudados para paginação continuam válidos para a segmentação
  - e.g.; bits de proteção (*rwx*), bit de validade, bits de compartilhamento, etc..
- Segmentação adiciona a possibilidade de compartilhar apenas trechos da área de código
- Problema associado:
  - Segmentos compartilhados devem ter a mesma identificação (entrada) na tabela de segmentos

Desvantagem da segmentação

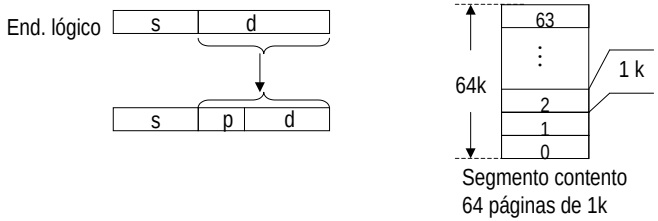
- A segmentação provoca fragmentação externa quando segmentos começam a ser liberar memória
- Mesmo problema de alocação partições variáveis com as mesmas soluções:
  - Concatenação de segmentos adjacentes
  - Compactação da memória

Solução para fragmentação externa

- A paginação é a solução natural para a fragmentação
- Analisar o problema sob dois pontos extremos:
  - Um processo é um único segmento
  - Cada byte é um segmento
    - Sem fragmentação externa, nem interna
    - Não viável pelos overheads envolvidos
    - Similar a página de 1 byte
- Solução: meio termo entre os extremos
  - Fazer um segmento ser composto por um número fixo (e reduzido) de bytes
  - Equivale a ter o segmento dividido em internamente em blocos

Segmentação com paginação

- Recuperar as vantagens dos dois métodos em relação a fragmentação:
  - Fragmentação interna: paginação apresenta, segmentação não
  - Fragmentação externa: segmentação apresenta, paginação não
- Solução se traduz em paginar segmentos



Leituras complementares

- R. Oliveira, A. Carissimi, S. Toscani; *Sistemas Operacionais*. Editora Sagra-Luzzato, 2001.
  - Capítulo 6
- A. Silberchatz, P. Galvin, G. Gane; *Applied Operating System Concepts*. Addison-Wesley, 2000.
  - Capítulo 9
- W. Stallings; *Operating Systems*. (4th edition). Prentice Hall, 2001.
  - Capítulo 7