Memória Principal

MCTA026-13 - Sistemas Operacionais

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Sistemas Operacionais na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Este material foi baseado nas ilustrações e texto preparados por Silberschatz, Galvin e Gagne [SGG] e Tanenbaum e Bos [TB] detentores do copyright.
- Estes slides contém alguns textos e figuras preparados por Dror Bereznitsky.

Introdução

Roteiro

UFABC

- Fundamentos
- Troca (*swapping*)
- Alocação contígua de memória
- Segmentação
- Paginação
- Estrutura da tabela de páginas
- Exemplo: As arquiteturas Intel 32 e 64 bits
- Exemplo: A arquitetura ARM

Objetivos

UFABC

- Fornecer uma descrição detalhada das diversas maneiras que podemos organizar o hardware de memória
- Discutir diversas maneiras de fazer o gerenciamento de memória incluindo paginação e segmentação
- Detalhar o processador Intel Pentium que tem suporte para segmentação pura e para segmentação e paginação conjuntas

Fundamentos

UFABC

- Um programa precisa ser trazido do disco para a memória e ”colocado em um processo”para ser executado
- Um programa pode ser escrito em linguagens de alto ou baixo nível
- A memória principal e os registradores são as únicas posições de memória que um processador pode acessar diretamente
- A CPU busca as instruções da memória principal de acordo com o valor que está no program counter
- Ciclo típico para a execução de uma instrução: busca da memória, decodifica a instrução, busca os operandos, possível armazenamento dos resultados na memória

UFABC

- A unidade de memória apenas ”enxerga”um fluxo como abaixo:
  - Endereço + requisições de leitura (ex. carregue a memória na posição x para o registrador 8)
  - Endereço + requisições de escrita (ex. armazene o conteúdo do registrador 6 na posição de memória 1090)
- A unidade de memória não sabe como esses endereços foram gerados
- Acesso aos registradores tipicamente leva um ciclo (ou menos) da CPU

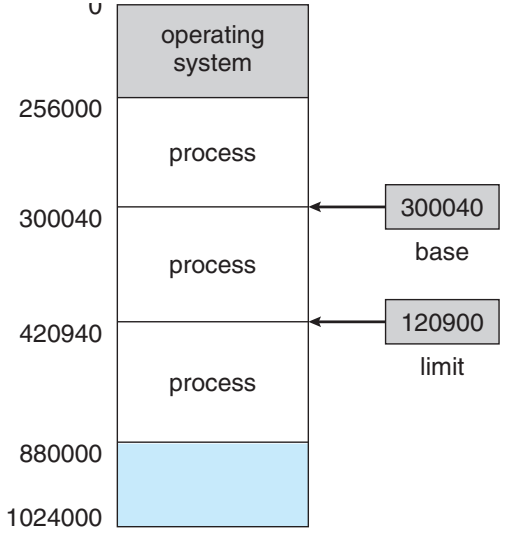
UFABC

- Do início ao fim de um acesso à memória, podem ter se passado inúmeros ciclos da CPU. Neste caso, enquanto espera, dizemos que o processador fez um *stall*, ou ficou esperando, já que ele não tinha o dado com o qual trabalhar.
- A memória *cache* fica entre a memória principal e a os registradores da CPU para amenizar o problema de stalls
- Proteção de memória é necessária para garantir o funcionamento correto do sistema
  - Processos do usuário não tem acesso à memória do SO
  - Um processo do usuário não tem acesso à memória de outros processos

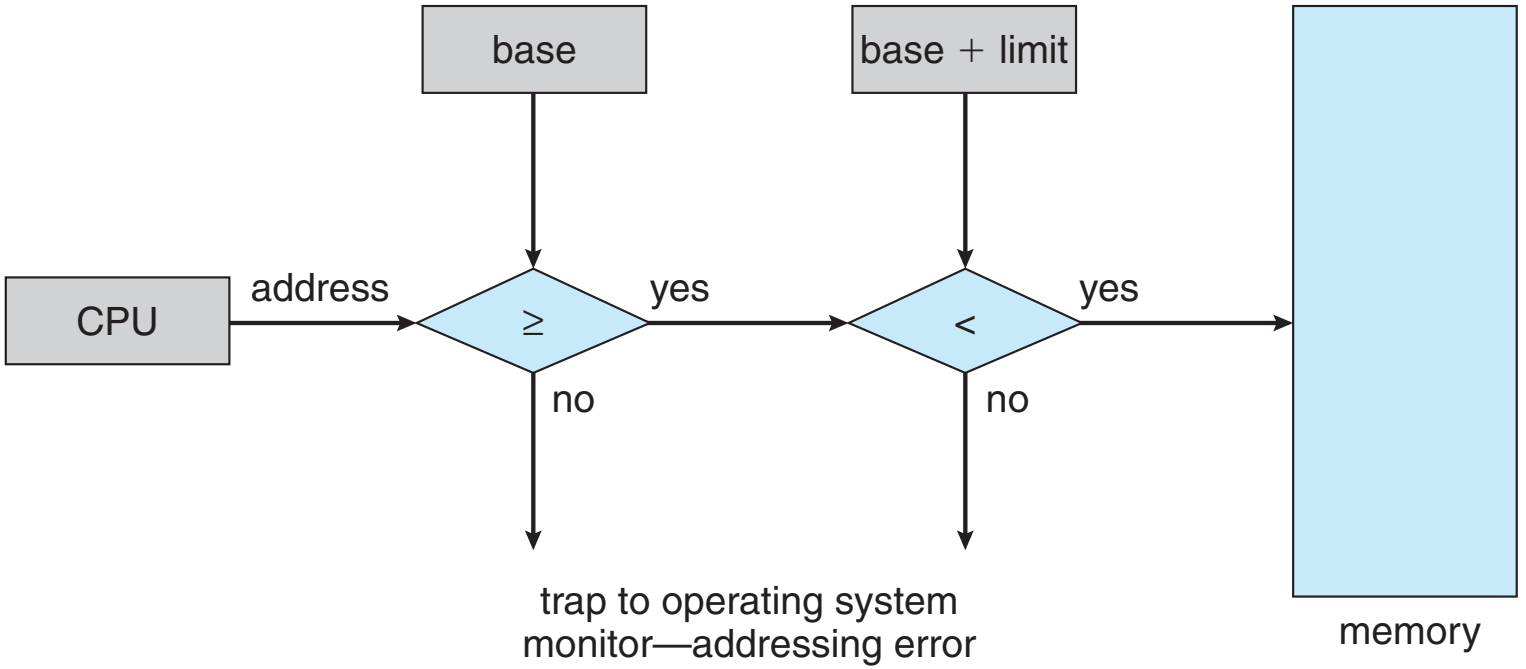
Proteção de memória

UFABC

- Um registrador base (*base register*), que armazena o menor endereço físico de um programa na memória, e um registrador limite (*limit register*), que especifica o tamanho do programa, definem os limites de um programa na memória
- A CPU verifica a cada acesso que for gerado em modo de usuário se o endereço requisitado está dentro dos intervalos (base - base+limite) para aquele usuário



Proteção de acesso por hardware



Vinculação de endereços

UFABC

- Um programa no disco precisa ser trazido à memória para executar. Normalmente os programas são armazenados em um arquivo em um formato binário e mantido em uma fila de entradas (*input queue*)
- Em geral, não sabemos ao certo onde um programa residirá na memória, logo é conveniente assumir que o primeiro endereço físico de um programa sempre inicia no endereço 0000.
- Sem suporte do hardware ou do software, os programas precisariam ser carregados na posição 0000.
- Não é nada prático assumir que os programas sempre iniciarão na posição 0000
- Por isto quase todas as CPUs têm um hardware para tratar do gerenciamento de memória

Vinculação de endereços

UFABC

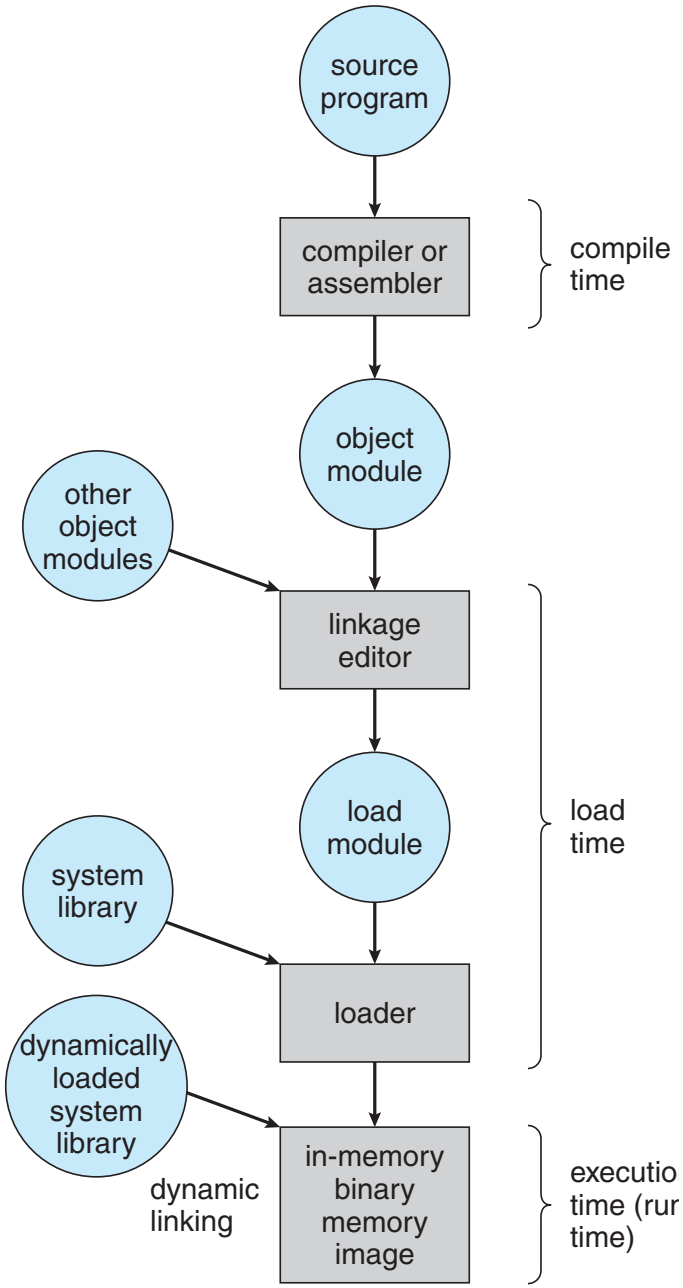
- Em geral, endereços são representados de diferentes maneiras dependendo do estágio do ciclo da vida de um programa
  - Endereços em um código fonte são normalmente simbólicos (por exemplo, a variável `contador`)
  - Um compilador tipicamente vincula (*bind*) estes endereços simbólicos para endereços reposicionáveis/ *relocatable* (ex. 14 bytes do início deste módulo)
  - Linker ou carregador (loader) vincula os endereços reposicionáveis para endereços absolutos (ex. 17014)
  - Cada vínculo mapeia um endereço de um espaço de endereço para outro

Vínculos de instruções e dados à memória

UFABC

- O vínculo de instruções e dados aos endereços de memória podem ocorrer em 3 diferentes momentos:
  - **Compilação** - Se a localização da memória é sabida a priori, código com posicionamento absoluto pode ser gerado. É preciso contudo recompilar o código caso as posições sejam alteradas.
  - **Carregamento** - Se a localização da memória não for sabida em tempo de compilação e não há suporte de hardware, então código reposicionável (*relocatable code*)precisa ser gerado
  - **Execução** - Atrasar o vínculo até o tempo de execução se o processo puder ser movido durante a sua execução de um segmento de memória para outro
    - É preciso suporte de hardware para mapear os endereços (ex. registradores de base e limite)

Passos do processamento de um programa de usuário



Endereços físicos e lógicos

UFABC

- O conceito de endereço lógico que é separado do conceito de endereço físico é comum a muitas CPUs
  - Endereço lógico - Gerado pela CPU
  - Endereço físico - Endereço visto pela unidade de memória
- Endereços lógicos e físicos:
  - São os mesmos nas técnicas de vínculo em tempo de compilação ou de carregamento
  - Eles são diferentes na técnica de vínculo em tempo de execução. Neste caso o endereço lógico é chamado de endereço virtual
  - Acabamos usando o termo endereço virtual como um sinônimo de endereço lógico

Endereços físicos e lógicos

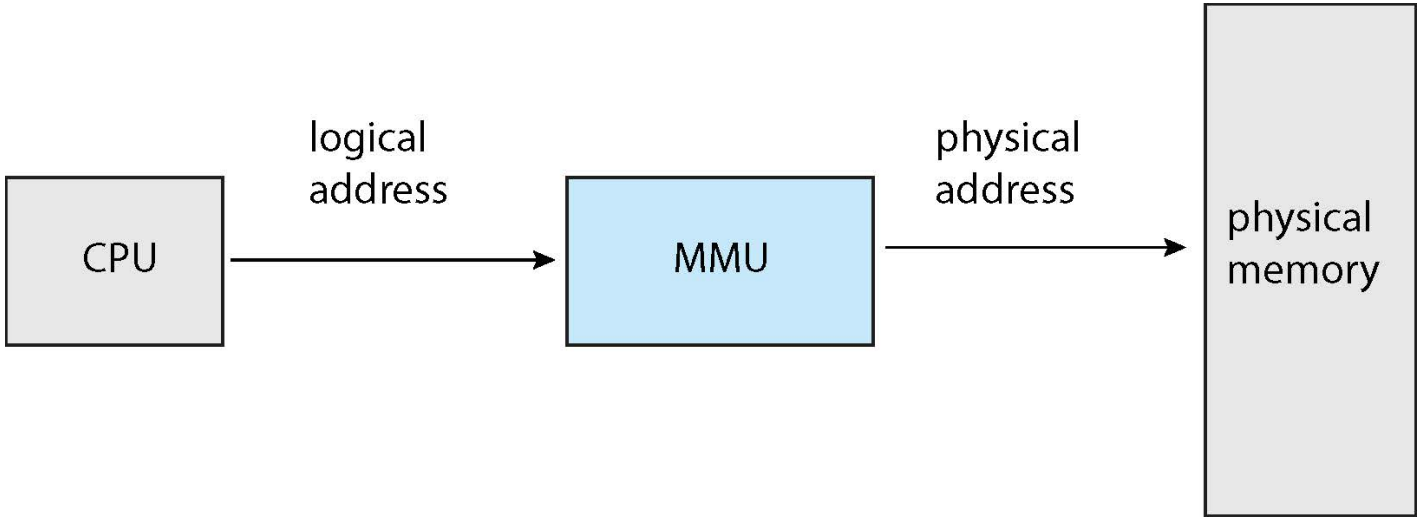
UFABC

- Espaço de endereçamento lógico é o conjunto de todos os endereços lógicos gerados por um programa
- Espaço de endereçamento físico é o conjunto de todos os endereços que correspondem à um espaço de endereçamento lógico

Unidade de gerenciamento de memória (MMU)

UFABC

- MMU - Dispositivo de hardware que em tempo de execução mapeia endereços virtuais aos endereços reais

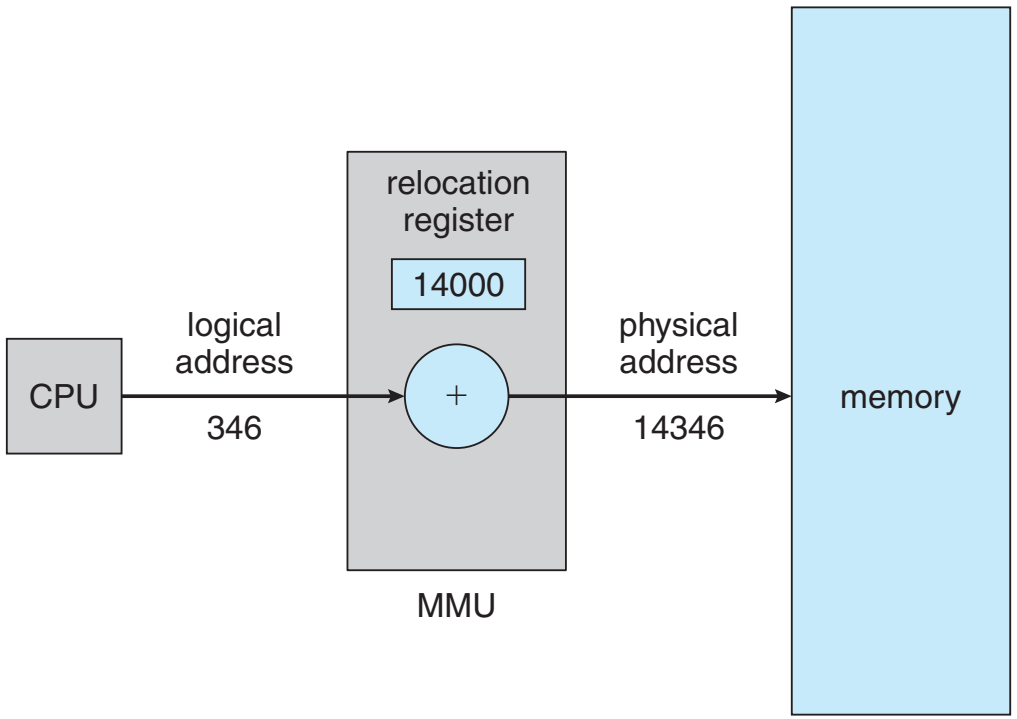


- Várias estratégias podem ser adotadas (veremos algumas)
- O programa de usuário sempre lida com endereços virtuais, ele nunca enxerga os endereços físicos
  - O vínculo em tempo de execução ocorre quando uma referência é feita para uma posição de memória
  - Endereço lógico é vinculado ao endereço físico

Reposicionamento dinâmico com um registrador de deslocamento

UFABC

- Simplesmente soma um deslocamento a todos os endereços
  - Registrador base agora chamado de registrador de deslocamento
  - MS-DOS em processadores Intel usava 4 registradores de deslocamento



Carregamento dinâmico

UFABC

- Até agora nós tínhamos assumido que o programa inteiro e seus dados precisam estar na memória principal para executar
- Carregamento dinâmico permite que uma rotina (módulo) seja carregada em memória apenas quando for necessária
- Gera um melhor aproveitamento do espaço de memória já que rotinas não utilizadas não são carregadas
- Todas as rotinas são mantidas em disco em um formato reposicionável
- É bem útil quando diversos trechos de código são raramente utilizados (ex. tratamento de erros)
- Não é necessário nenhum suporte especial do SO
  - É responsabilidade dos usuários desenvolverem seus programas para que tirem vantagem deste método
  - Mas o SO pode ajudar fornecendo bibliotecas que facilitem a tarefa

Vínculos dinâmicos

UFABC

- Bibliotecas de vínculo dinâmico (*dynamic linked libraries*) são bibliotecas do sistema que são vinculadas (linked/linkadas) aos programas de usuário quando durante a execução
  - Semelhantes às bibliotecas de carregamento dinâmico, mas o linking é feito apenas em tempo de execução e não de carregamento
- Pequeno pedaço de código stub é utilizado para localizar a rotina apropriada em memória
- Após a primeira execução o stub é substituído pelo endereço da rotina correta

Vínculos dinâmicos

UFABC

- SO verifica se a rotina está no espaço de memória do processo
  - Se não estiver, adiciona
- Vínculos dinâmicos são especialmente interessantes para bibliotecas
- Alguns sistemas também dão o nome de bibliotecas compartilhadas (*shared libraries*)

Alocação contígua

UFABC

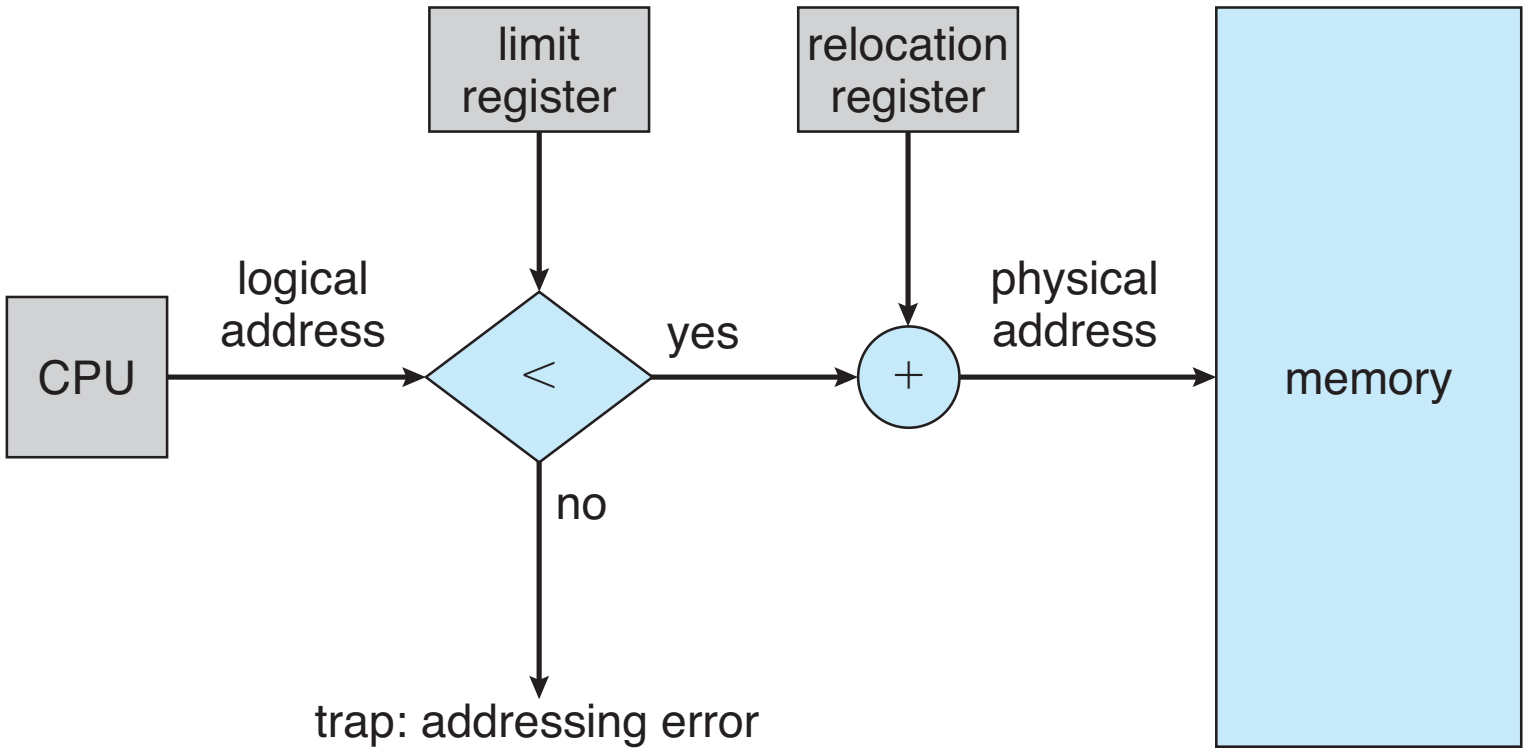
- A memória principal precisa dar suporte tanto para processo de usuário quanto do SO
- A memória é limitada - precisa ser usada com eficiência
- Alocação contígua é um dos primeiros métodos
- A memória é dividida em duas partições
  - O sistema operacional é colocado em memória baixa, com um vetor de controle de interrupções
  - Processos do usuário colocados em memória alta
  - Cada processo recebe um intervalo contíguo de endereços de memória

Alocação contígua

UFABC

- Registradores de deslocamento são usados para proteger os processos dos usuários uns dos outros e também evitam que os processos mudem dados pertencentes ao SO
  - Registrador base contem o valor do menor endereço físico possível
  - Registrador de limite contém o intervalo dos endereços lógicos - cada endereço lógico precisa ser menor que o limite
  - MMU mapeia os endereços dinamicamente
  - Podem então permitir ações como, por exemplo, o código do kernel ser transiente - é carregado e descarregado conforme necessário. Logo o kernel pode mudar o seu tamanho dinamicamente.

Suporte de hardware para reposicionamento e registradores de limite

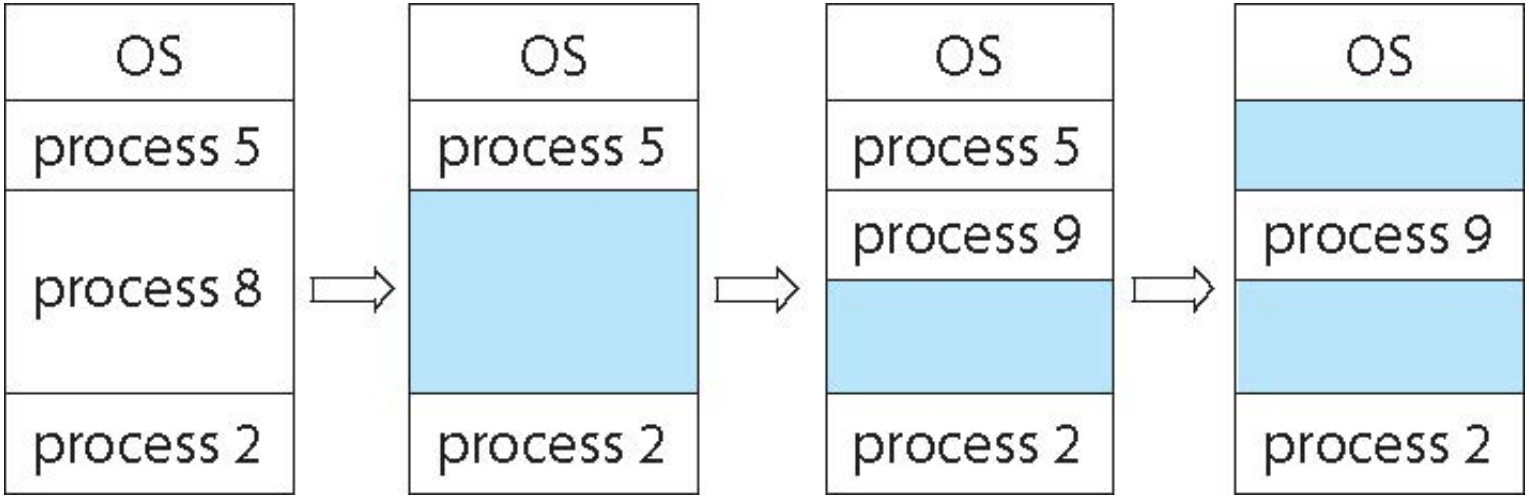


Alocação com múltiplas partições

UFABC

- Partição variável - Dimensionada para as necessidades de um processo específico
- Buraco (*hole*) - bloco de memória disponível; buracos de diferentes tamanhos são espalhados pela memória
- Quando um programa chega, ele recebe memória alocada de um buraco grande o suficiente para suas necessidades
- Processo que finaliza a execução libera a sua partição (partições adjacentes são combinadas)
- O SO mantém informações sobre partições alocadas e partições livres (buracos)

Alocação com múltiplas partições



Problema da alocação dinâmica de espaço

UFABC

- Como satisfazer uma requisição de tamanho **n** de uma lista de buracos?
  - First-fit - Aloca o **primeiro** buraco que for grande o suficiente
  - Best-fit - Aloca o **menor** buraco que for grande o suficiente
    - Precisa varrer a lista inteira a menos que esteja ordenada por tamanho
    - Produz o menor buraco de resto
  - Worst-fit - Aloca o **maior** buraco; também precisa procurar a lista inteira a menos que ela seja mantida em ordem
    - Produz o maior buraco de resto
  - First-fit e best-fit são em geral melhores que worst-fit em termos de espaço e tempo de execução

Fragmentação

UFABC

- Fragmentação externa - espaço total de memória existe para satisfazer uma requisição, mas não é contíguo e portanto não pode ser usado
  - Uma análise do firt-fit revela que dados N blocos alocados, outros $0.5N$ blocos serão perdidos devido à fragmentação
    - 1/3 da memória pode se tornar inutilizável – regra dos 50%
- Fragmentação interna - a memória alocada pode ser ligeiramente maior que a memória requisitada
  - Pode ocorrer quando o buraco é de 15.000 bytes mas o processo precisa de 14.900 bytes. Manter um buraco de 100 bytes não vale o esforço então o SO aloca 15.000.
  - A diferença de 100 bytes é interna à memória alocada, à uma partição, e não está sendo usada

UFABC

- A fragmentação externa pode ser reduzida através do uso de compactação
  - Movimenta todos os buracos na memória para que sejam todos contíguos
  - Compactação é possível apenas se reposicionamento é dinâmico e for feito em tempo de execução
  - Problema de E/S – Não é possível fazer a compactação enquanto E/S estiver ativa e envolver uma posição de memória sendo compactada
    - Solução 1: travar o processo na memória enquanto a E/S estiver sendo feita
    - Solução 2: fazer E/S apenas em buffers do próprio SO

Alocação não contígua

UFABC

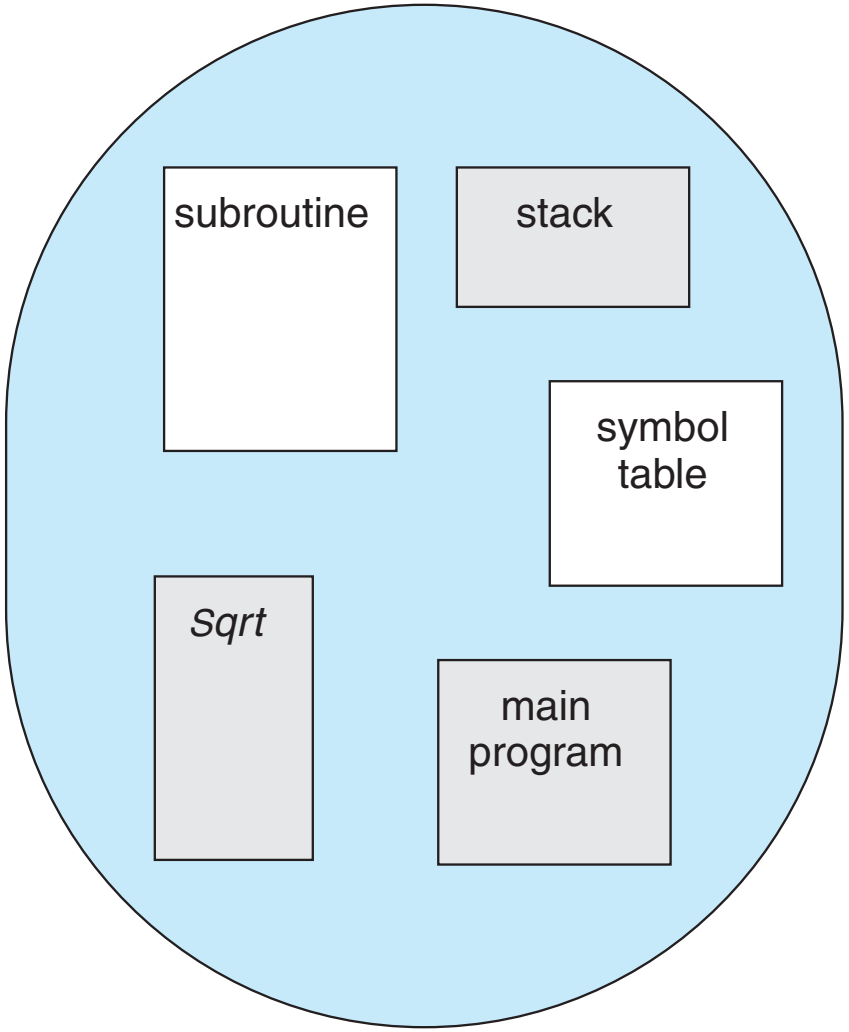
- Particionar o programa em um número de unidades pequenas, cada uma delas residindo em uma parte diferente da memória
- É preciso suporte do hardware
- Há vários métodos
  - Segmentação
  - Paginação
  - Segmentação paginada

Segmentação

UFABC

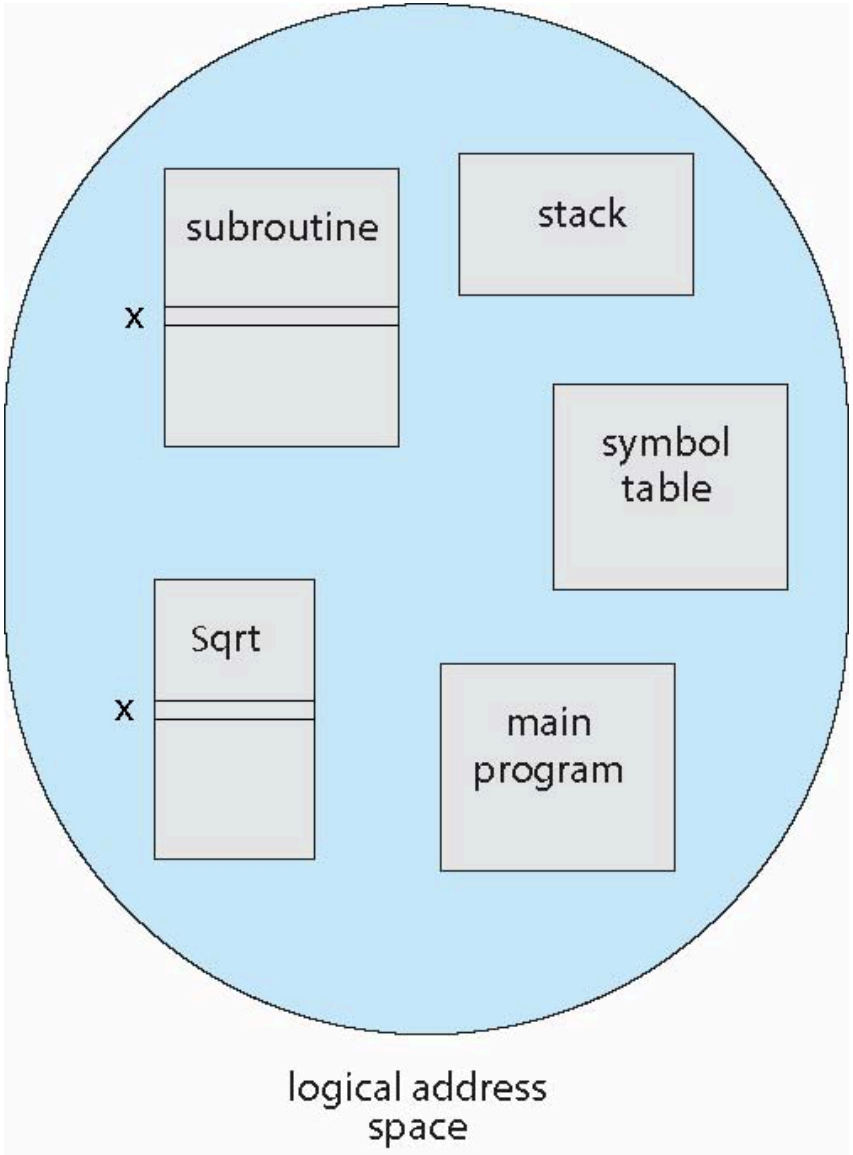
- A segmentação é um mecanismo de gerenciamento de memória que da suporte à visão do usuário da memória
- Um programa é dividido em uma coleção de segmentos, ou unidades lógicas, tais como:
  - programa principal
  - procedimentos
  - funções
  - métodos
  - objetos
  - variáveis locais e globais
  - pilha
  - tabela de símbolos
  - ...
- Cada segmento pode residir em uma parte diferente da memória como uma maneira de contornar o problema da alocação contígua

Segmentação

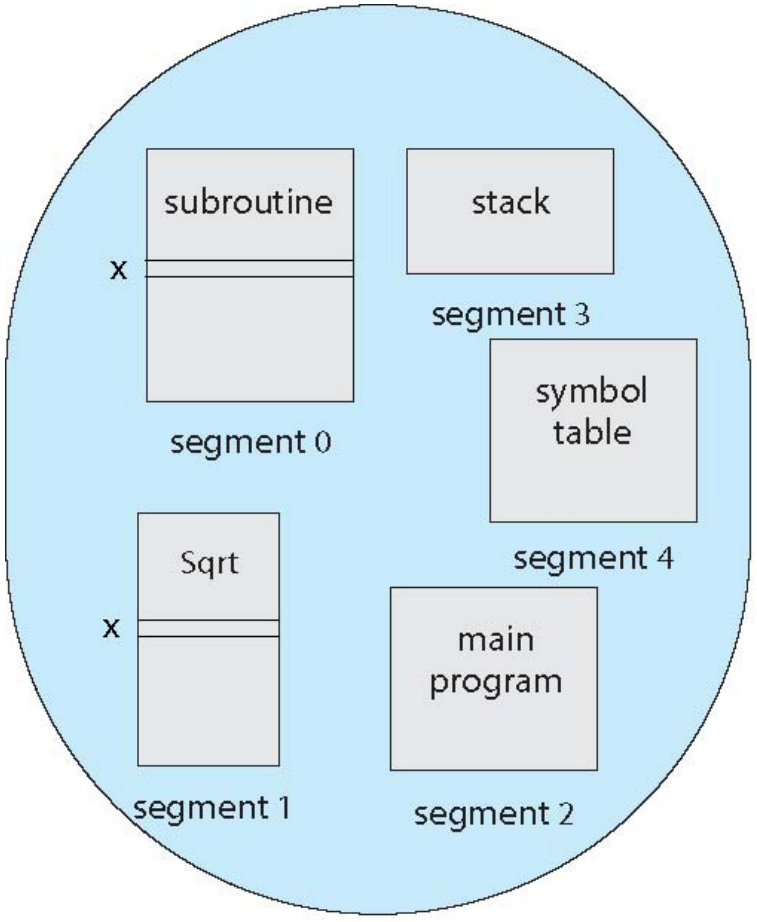


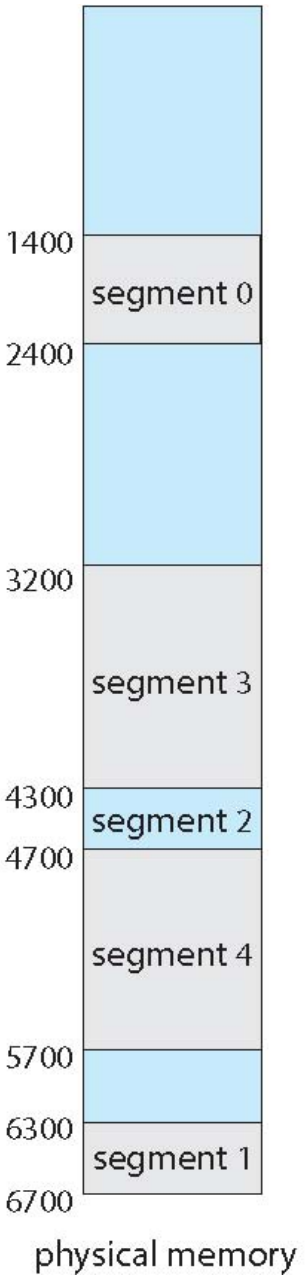
logical address

Segmentação bi-dimensional



Visão lógica e física da memória





Arquitetura da segmentação

UFABC

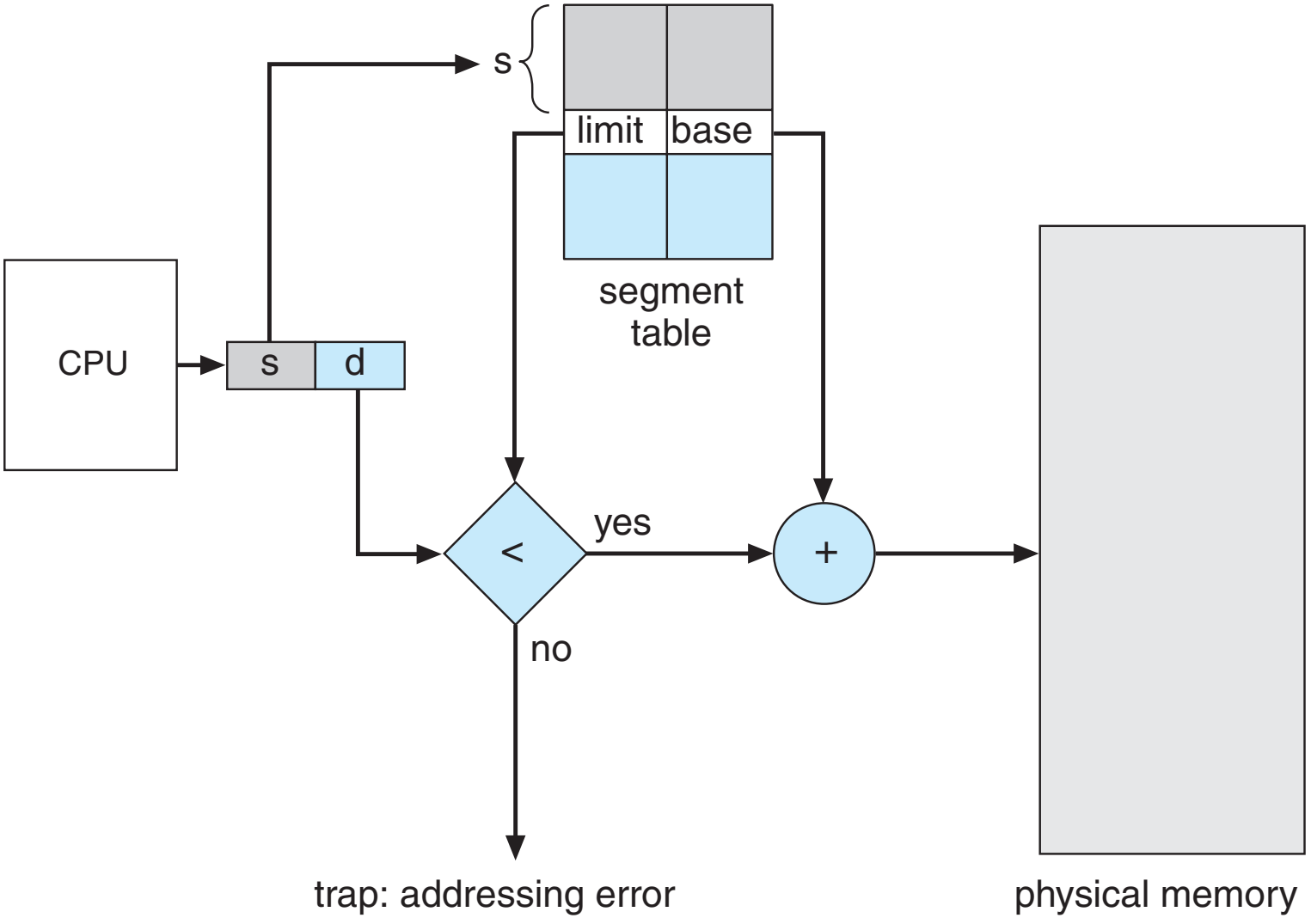
- O endereço lógico consiste de uma tupla:
  - `<número do segmento, deslocamento>`
  - Tipicamente usamos a palavra *offset* do inglês para nos referirmos ao deslocamento
- É necessário mapear um endereço lógico bidimensional para um endereço físico unidimensional. Isso é feito através da tabela de segmentos
  - Base - endereço de início do segmento
  - Limite - tamanho do segmento

Arquitetura da segmentação

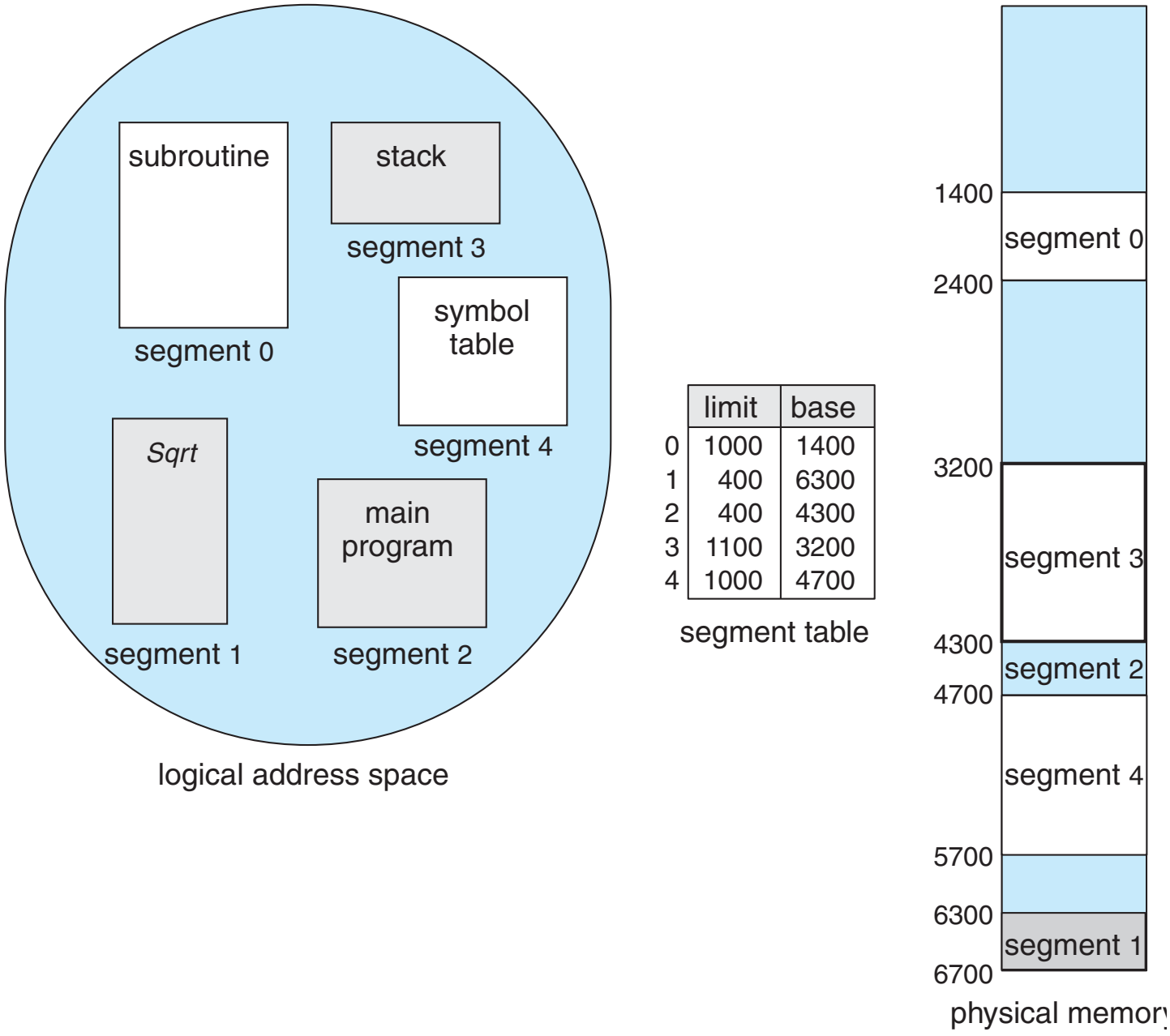
UFABC

- A tabela de segmentos é mantida em memória
  - Registrador da base da tabela de segmentos (*Segment-table base register* - STBR) - Aponta para a localização da tabela em memória
  - Registrador do tamanho da tabela de segmentos (*Segment-table length register* - STLR) - indica o número de segmentos usados por um programa
    - Um segmento `s` é legal se `s < STLR`

Hardware para segmentação



Exemplo de segmentação



Paginação

- O espaço de endereçamento físico de um processo pode não ser contíguo
- Dividimos o processo em blocos de tamanho fixo, cada qual pode residir em uma parte diferente da memória física
- Dividimos a memória física em blocos de tamanho físico chamados de frames
  - Frames têm, em geral, um tamanho que varia de 512 bytes até 16 MBytes

UFABC

- Dividimos a memória lógica em blocos dos mesmos tamanhos dos frames. Cada um destes blocos é chamado de página (*page*)
- A memória onde o programa é mantido também é dividida em unidades de armazenamento chamadas de blocos (que tipicamente tem o mesmo tamanho das páginas e dos frames)
- A memória física é alocada sempre que blocos estão disponíveis
  - Evita fragmentação externa
  - Mas ainda tem fragmentação interna

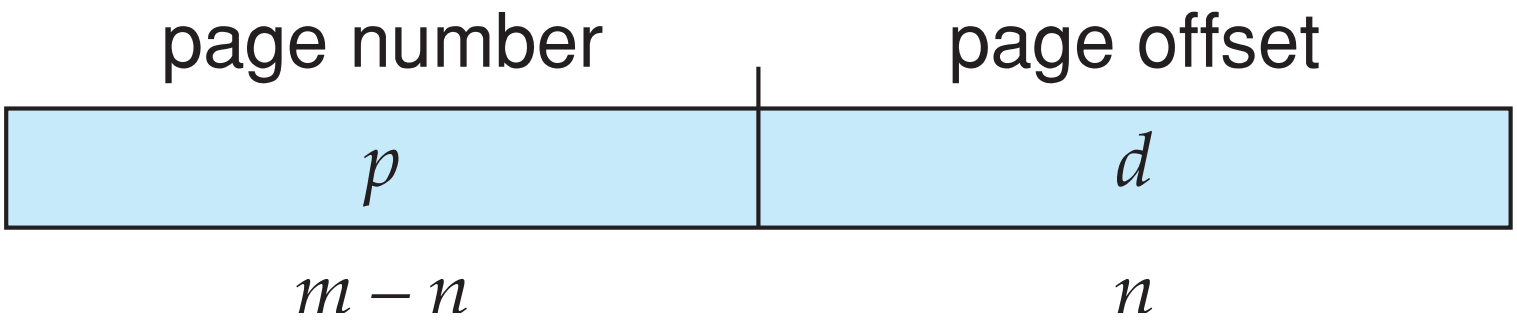
UFABC

- É necessário manter um controle de todos os frames livres
- Para executar um programa com tamanho de N páginas, é preciso achar N frames livres e carregar o programa
- A correspondência entre os frames e as páginas é feito pela tabela de páginas (*page table*)
- A tabela de páginas é mantida em memória
  - Registrador da base da tabela de páginas (*Page-table base register* - PTBR) - aponta para a tabela de páginas
  - Registrador do tamanho da tabela de páginas (*Page-table length register* - PTLR)
- Ainda tem fragmentação interna...

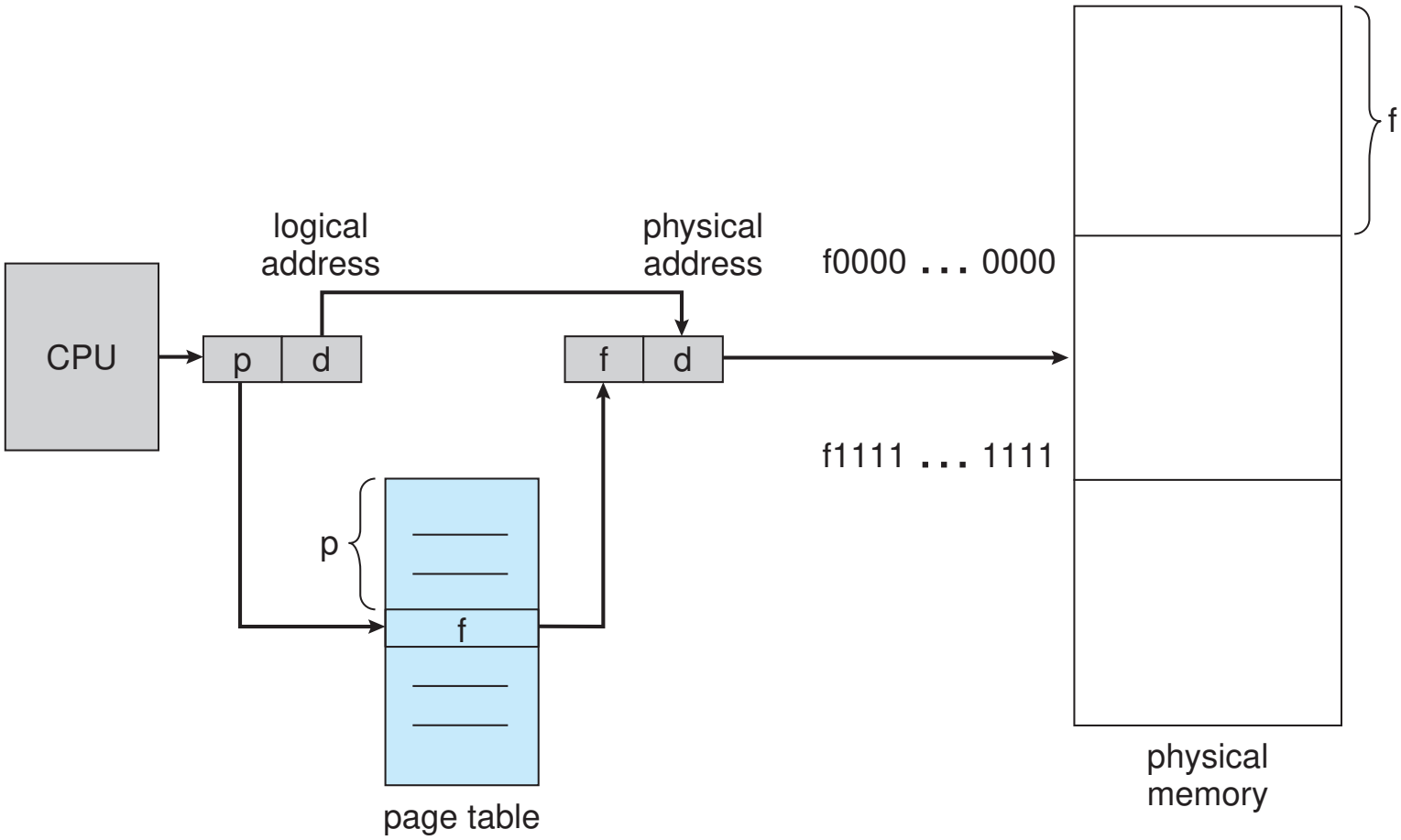
Mecanismo de tradução de endereços

UFABC

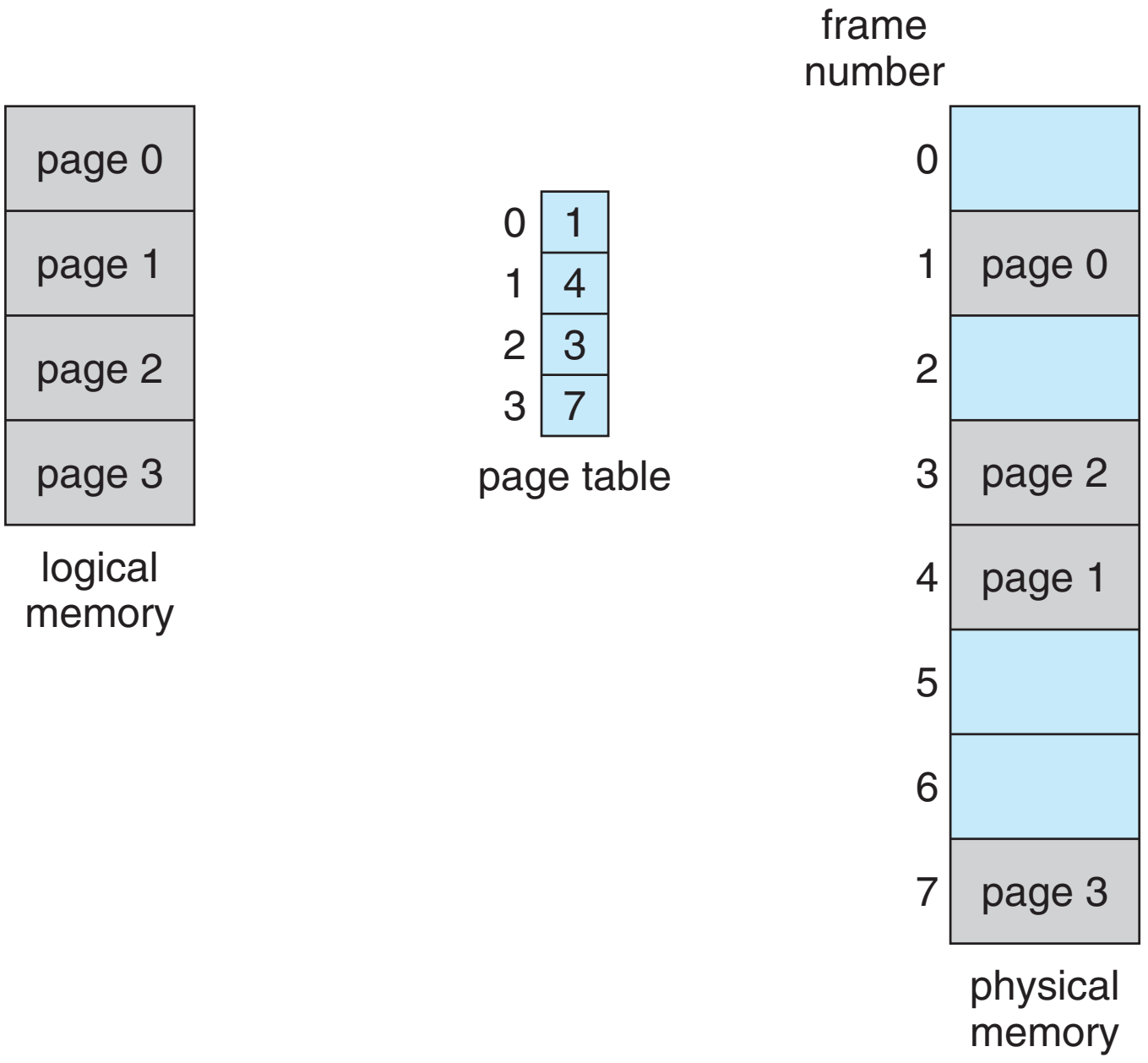
- Assuma que o espaço de endereços seja 2^m. (Como determinamos m?)
- Assuma que o tamanho da página é 2^n
- O endereço gerado pela CPU é dividido em:
  - Número da página (p) - usado para indexar a tabela de páginas que contém o endereço base de cada página na memória física. Tamanho de $p = m - n$
  - Deslocamento na pagina (d) - combinado com a base para definir o endereço físico que é enviado à memória. Tamanho de $d = n$



Hardware de paginação

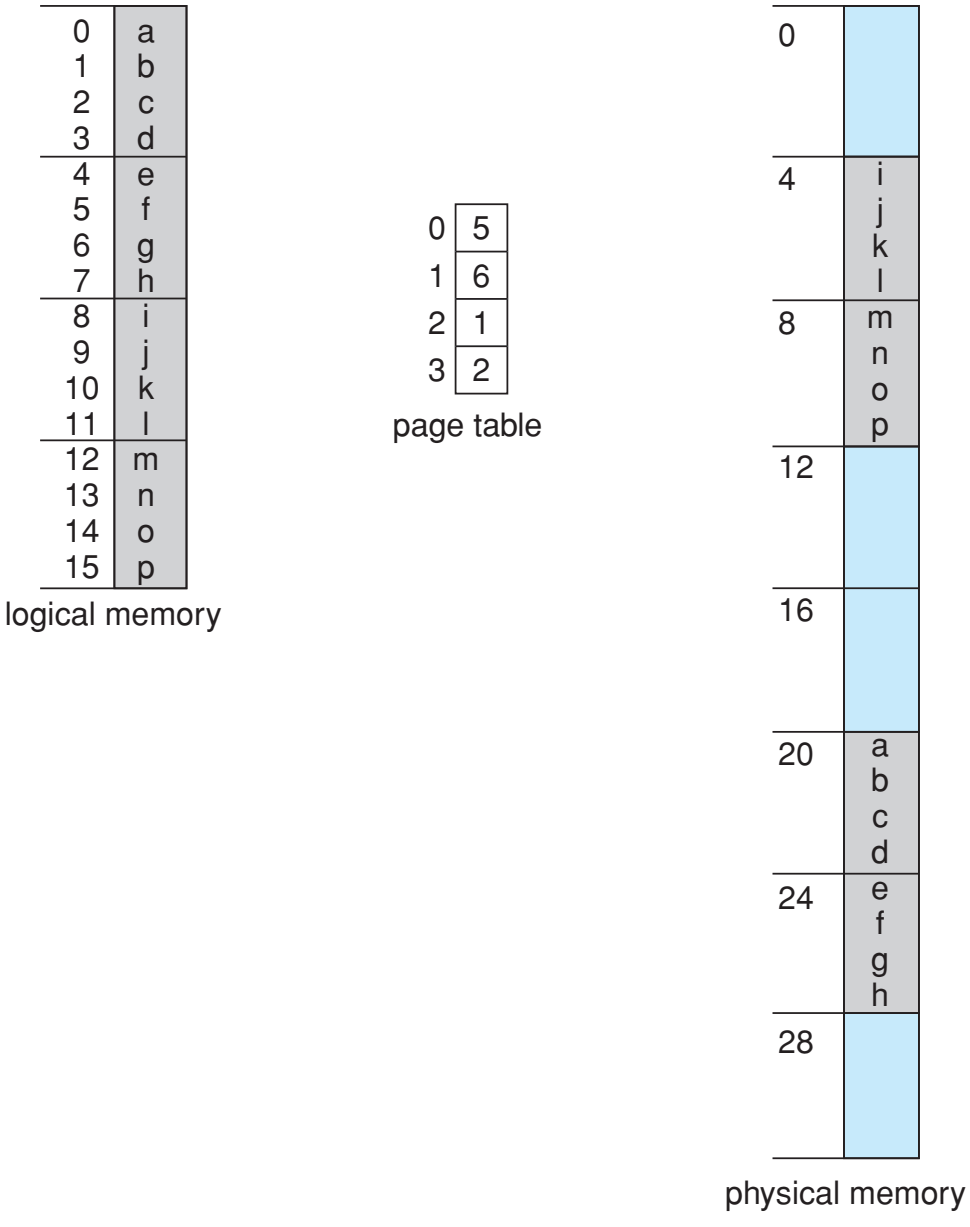


Modelo de paginação - Visão física e lógica



Exemplo de paginação

Assuma $m = 4$, $n = 2$, memória de 32 bytes e páginas de 4 bytes

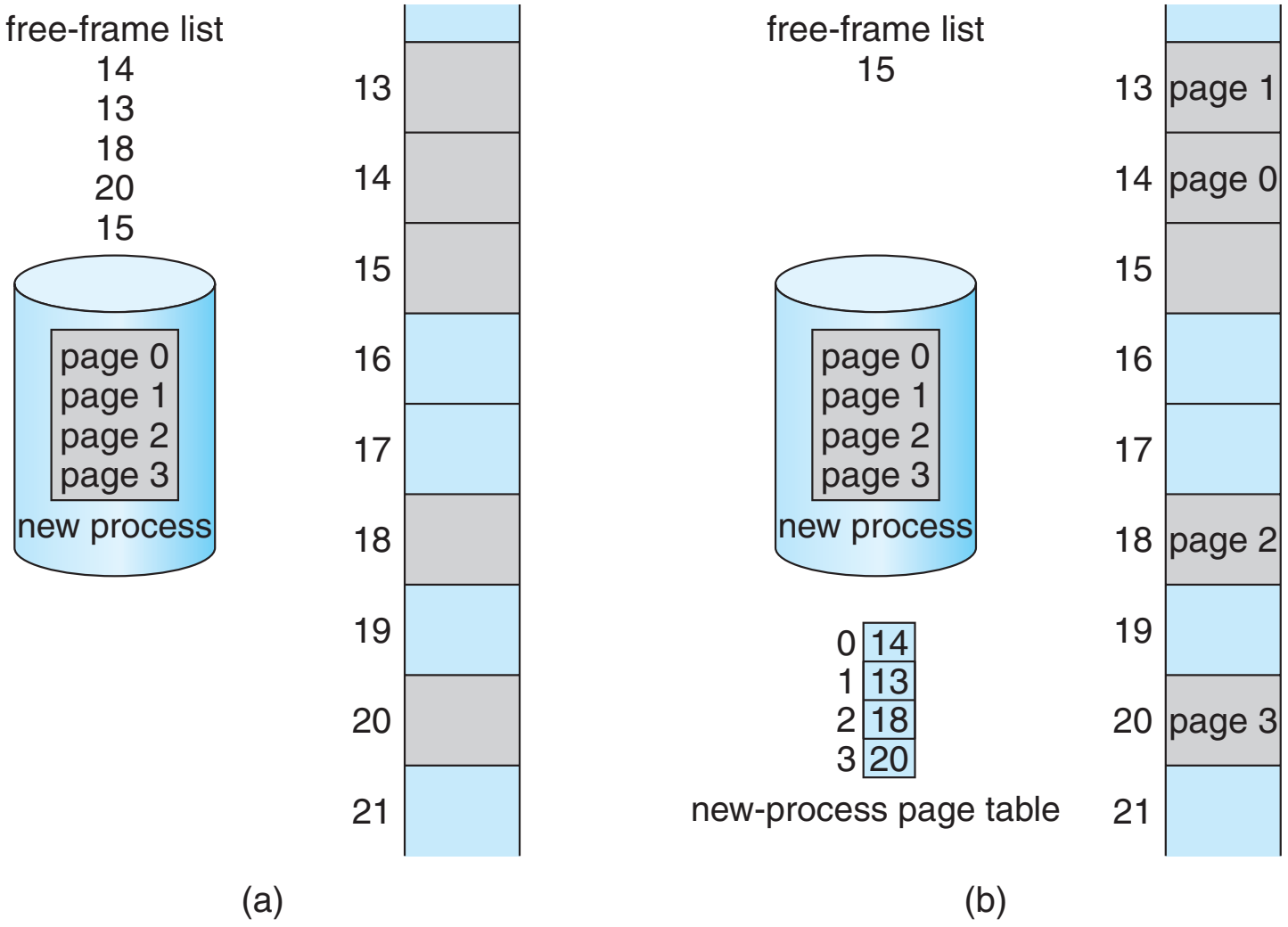


Fragmentação interna

UFABC

- Calculando fragmentação interna
  - Tamanho da página = 2.048 bytes
  - Tamanho do processo = 72.766 bytes
  - 35 páginas = 1.086 bytes
  - Fragmentação interna de 2.048 - 1.086 = 962 bytes
  - Pior caso = 1 frame - 1 byte
  - Na média a fragmentação é de frame / 2
  - Pequenos tamanhos de frames são desejáveis?
  - Cada entrada na tabela de páginas consome memória
  - Páginas vêm crescendo com o tempo
    - Solaris dá suporte a dois tipos de páginas: 8KB e 4MB
    - Linux Huge Pages
  - Por implementação, o processo só pode acessar a sua própria memória

Alocando frames para um novo processo



(a) antes da alocação

(b) após a alocação

TLB - Memória associativa

UFABC

- Se a tabela de memória for mantida em memória, cada instrução e dado precisaria de dois acessos à memória
  - 1 para a tabela de páginas e 1 para o dado/instrução propriamente dito
- Esse problema pode ser resolvido mantendo um cache da tabela no próprio processador

TLB - Memória associativa

UFABC

- TLB - Translation look-aside buffer é um hardware específico para guardar as correspondências de páginas e frames
- É uma memória associativa que faz a busca em paralelo em suas entradas

Nº Página	Nº Frame
p_x	F_i
p_y	F_k
P_z	F_j

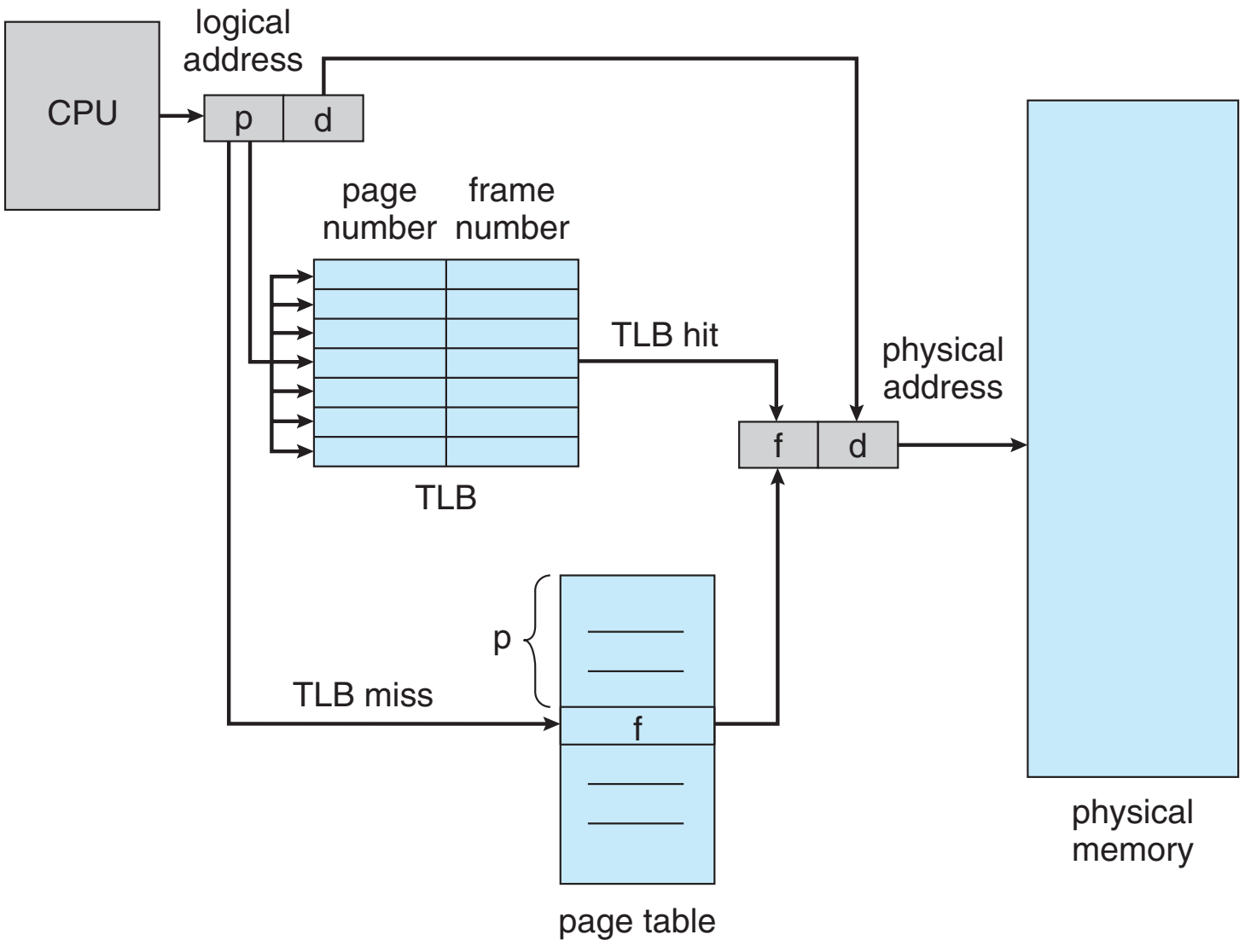
- Tradução de endereços (p, d)
  - Se p estiver na TLB pega número do frame
  - Senão, pega o frame da memória principal e coloca na TLB

TLB - Limitações

UFABC

- A TLB é tipicamente pequena (64 a 1024 entradas)
- Em um miss, o valor buscado é mantido na tabela para uma eventual subsequente consulta
  - E se não houver entradas livres? Políticas de substituição precisam ser consideradas
  - Algumas entradas podem ser fixadas (*wired down*) para acesso rápido permanente
- Algumas TLBs também armazenam identificadores de espaços de endereçamento (*address-space identifiers* - ASIDs) que identificam a qual processo aquela entrada diz respeito
  - Caso contrário a TLB deveria ser esvaziada a cada troca de contexto

Hardware de paginação com TLB



Tempo de acesso efetivo

UFABC

- Tempo de acesso à memória = τ unidades de tempo
- Tempo de busca na TLB = ϵ unidades de tempo
  - Tipicamente é < 10% do tempo de acesso e não é incomum que seja 0 (é feito em paralelo)
- Taxa de acertos = α
  - Taxa de acertos - % das vezes que a página é encontrada na TLB. A razão é relacionada ao número de entradas na tabela
- Tempo Efetivo de Acesso (EAT)

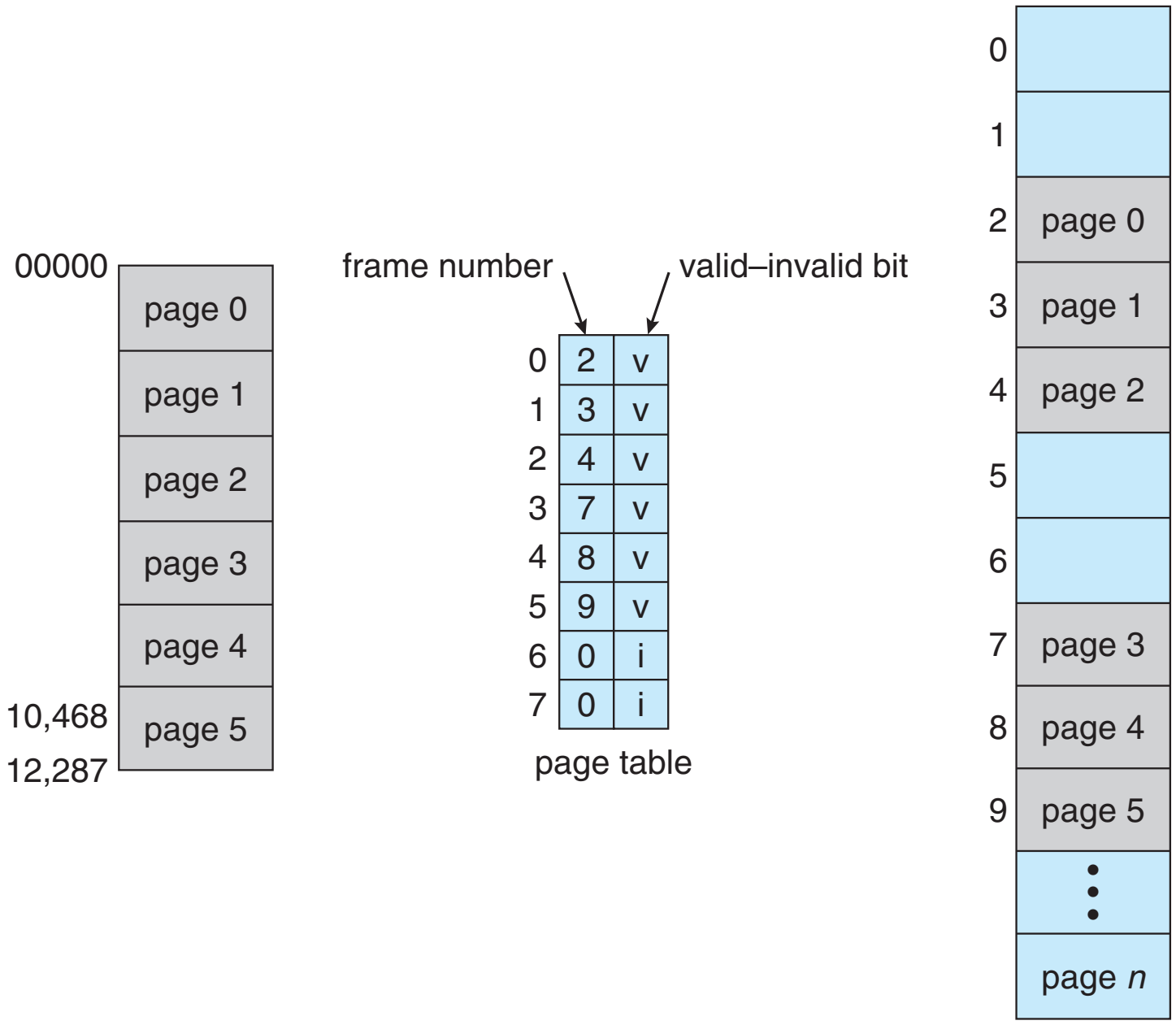
  $(EAT) = (\tau + \epsilon)\alpha + (2\tau + \epsilon)(1 - \alpha) = \tau(2 - \alpha) + \epsilon$
- Considere $\epsilon = 2$ ns para buscas na TLB e $\tau = 100$ ns para cada acesso à memória
  - Se α = 80% $\rightarrow$ EAT $= 100(2 - 0.8) + 2 = 122$ ns
  - Considere agora uma taxa de acerto mais realista, de $\alpha = 99$ %
    - EAT $= 100(2 - 0.99) + 2 = 103$ ns

Proteção de memória

UFABC

- A proteção de memória é feita através da associação de bits de proteção a cada um dos frames. Esses bits indicam se o frame é apenas leitura, leitura e escrita ou ainda se pode ser executada
- Válido-Inválido é um bit que fica associado a cada entrada da tabela de páginas
  - Válido indica que a página associada está no espaço de endereçamento lógico e é portanto uma página legal
  - Inválido indica que a página não está no espaço de endereçamento do processo
  - Pode-se usar também o registrador de comprimento da tabela de páginas (PTLR)
  - Quaisquer violações serão comunicadas pelo processador ao kernel

Bits válidos e inválidos na tabela de páginas

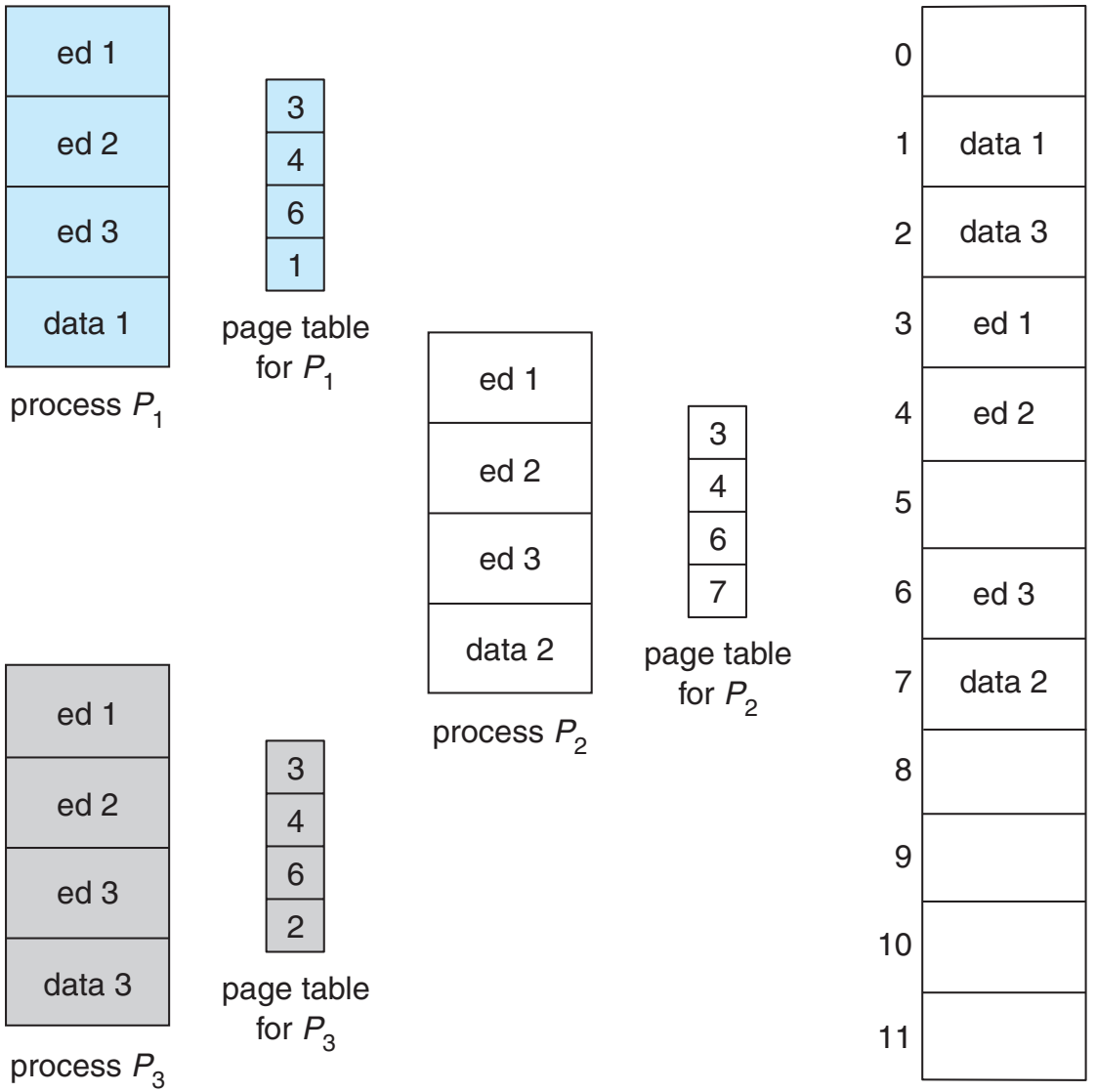


Páginas compartilhadas

UFABC

- Código compartilhado
  - Uma cópia apenas para leitura (reentrante) do código compartilhada entre todos os processos
  - Similar à múltiplos processos compartilhando o mesmo espaço de endereçamento
  - Também é útil para comunicação inter-processos se o compartilhamento de páginas em modo de leitura e escrita for permitida
- Código privado e dados
  - Cada processo mantém uma cópia separada do código e dos dados
  - As páginas para o código privado e para os dados pode aparecer em qualquer lugar no espaço de endereçamento lógico

Páginas compartilhadas - Exemplo



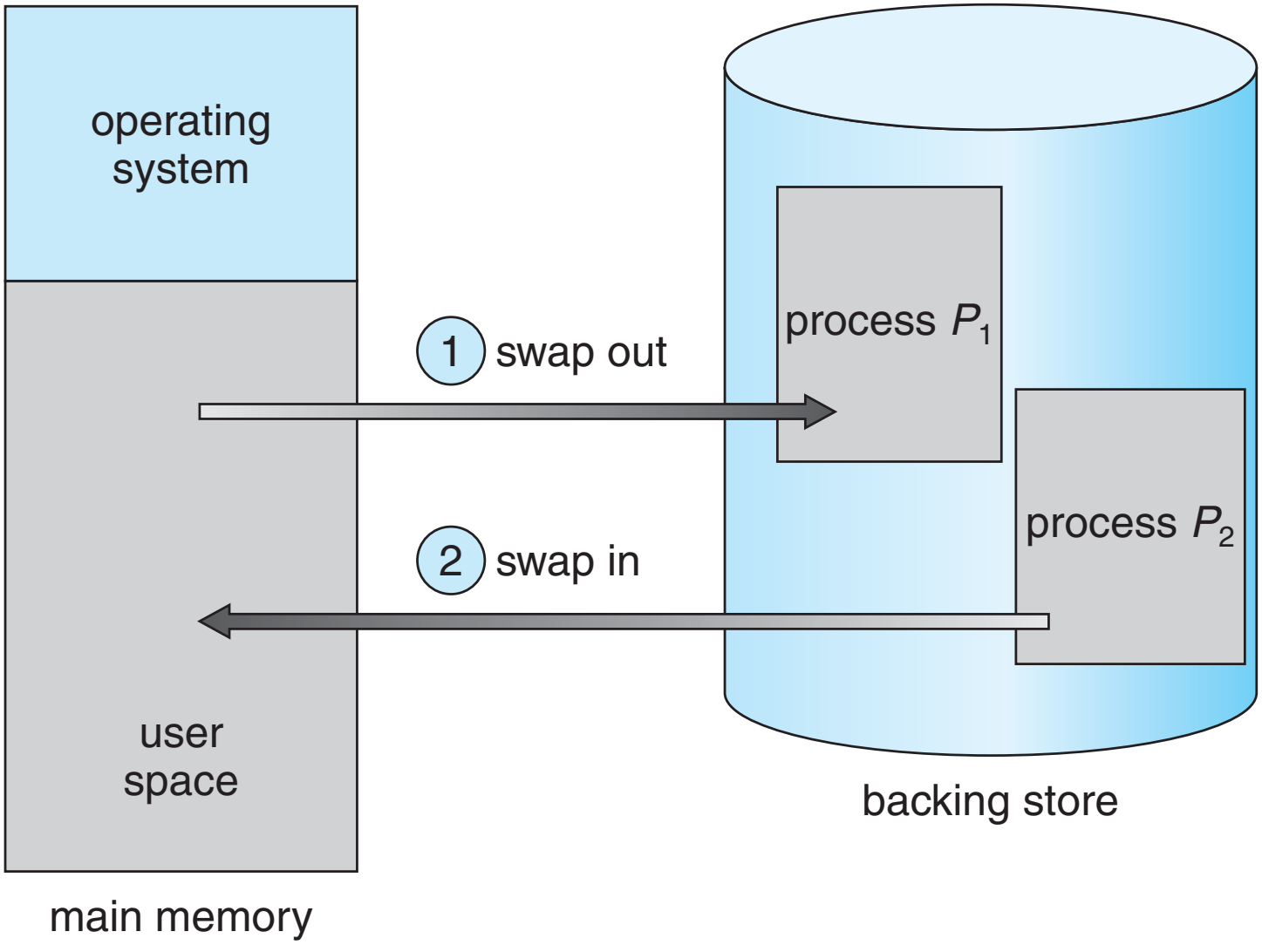
Swapping

Swapping ou Troca

UFABC

- Um processo pode ser *swapped*, ou retirado, temporariamente da memória para um sistema secundário de armazenamento e mais tarde ser trazido de volta à memória para continuar a execução
  - Dessa maneira o total de memória disponível para a execução de processos pode ser estendido além da quantidade de memória física da máquina
- Sistema secundário de armazenamento - disco ou SSD grande o bastante para acomodar cópias das imagens de todos os processos; é preciso que haja uma maneira direta de acesso a essas imagens
- Sistema mantém uma fila de prontos de processos que podem estar tanto na memória quando em disco
- A maior parte do swapping é devido à transferência de dados: o tempo total é diretamente proporcional à quantidade de memória trocada

Swapping


operating system
user space
main memory
1 swap out
2 swap in
process P_1
process P_2
backing store

Swapping

UFABC

- Quando um processo for trazido de volta do disco (swap in), ele precisa ser colocado na mesma posição de memória física?
  - Depende do mecanismo de vinculação em uso
  - É preciso considerar também operações de E/S pendentes
- Diversas versões de swapping podem ser encontradas na maioria dos SOs modernos. A variações mais comuns:
  - Swapping está normalmente desabilitada
  - Swapping é iniciado quando a quantidade de memória livre no sistema cai abaixo de um certo limiar
  - Swapping é desabilitado novamente quando a quantidade de memória livre passar de um mínimo
- Outra variação é fazer o swap de apenas porções do processo (e não o processo todo) para economizar tempo
- Em geral esse tipo de otimização é feita em conjunto com memória virtual (próxima aula!)

Tempo para troca de contexto incluindo o swapping

UFABC

- Se o processo a ser colocado para execução não estiver em memória e não houver espaço suficiente para acomodá-lo, então precisamos fazer o swap out de um processo em memória para disco
- O tempo para a troca de contexto pode se tornar muito alto
- Considere um disco com taxa média de escrita de 50MB/s e um processo de 100MB
  - Tempo para swap out: 2000ms
  - Adiciona-se o tempo para swap in de um processo (que podemos supor tenha aproximadamente o mesmo tamanho)
  - Tempo total aproximado: 4000ms
- Pode-se diminuir o tempo informando ao SO qual parte da memória está efetivamente sendo usada
  - `request_memory()` e `release_memory()`

Outros problemas com swapping

UFABC

- Requisições pendentes de E/S - Não se pode fazer o swap out
  - Alternativa seria fazer E/S em um buffer do SO (double buffering). Contudo diminui o desempenho do sistema
- Swapping padrão não é usado em sistemas operacionais modernos
  - A versão modificada é bem comum

Swapping em sistemas móveis

UFABC

- Não é tipicamente suportado. Sistemas móveis tipicamente usam memórias Flash que
  - Tem um número limitado de ciclos de escrita (Opa! E o SSD de desktops/notebooks?)
  - Baixa taxa de transferência entre a CPU e a memória Flash
- Tipicamente os sistemas móveis empregam outros sistemas para liberar memória em caso de necessidade
  - iOS **pede** para que os apps liberem memória voluntariamente
    - Dados apenas de leitura são descartados e lidos novamente da Flash se necessário
    - Caso o app não libere memória suficiente ele pode ser morto
  - Android mata as apps se estiver com pouca memória, mas antes grava o estado da aplicação para memória para reinício rápido
  - Ambos os sistemas têm suporte à paginação