Unidade 5 - Memória Principal

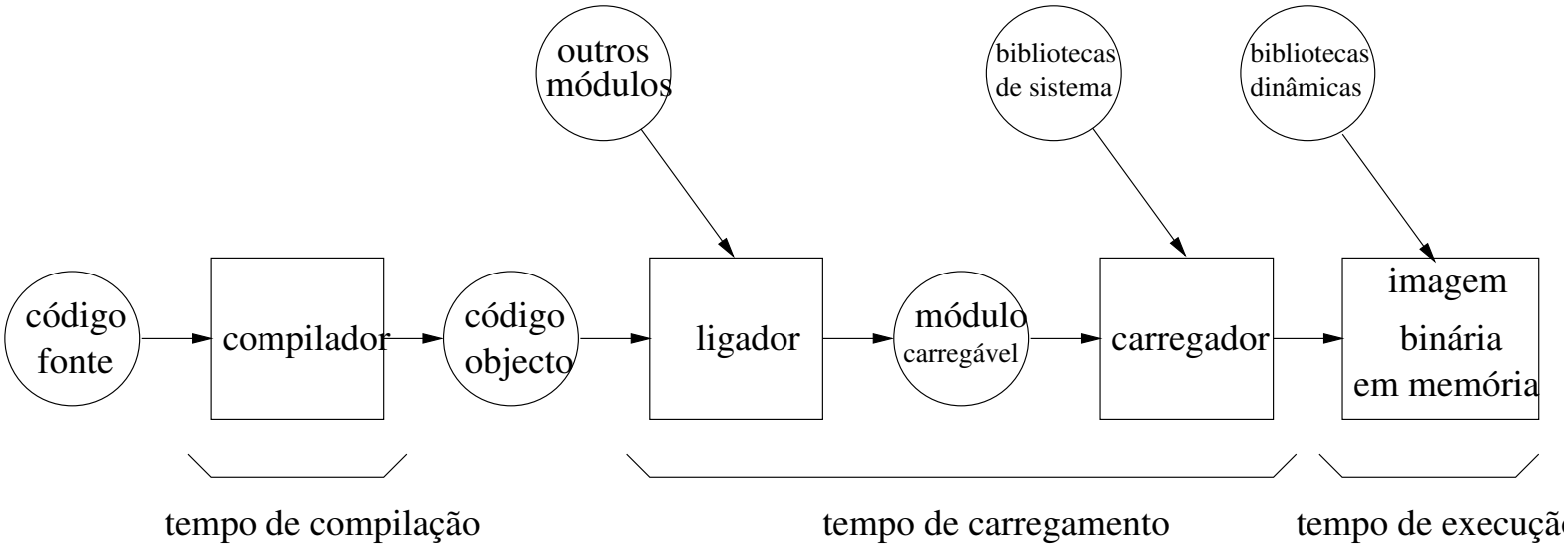
5.1 Conceitos Básicos

Conceitos Básicos

- **Registos** e **RAM** são as unidades de armazenamento diretamente acessíveis à CPU; o acesso aos registos consome tipicamente um ciclo de relógio; o acesso à RAM pode consumir vários ciclos de relógio; como memórias intermédias que são, as **caches** mascaram o diferencial de tempos de acesso entre registos e RAM
- Um **programa** reside em disco, sob a forma de um ficheiro binário executável; para o executar, é necessário carregá-lo na memória principal e criar um **processo**; a co-existência de vários programas em memória requer mecanismos de proteção
- Dependendo do sistema de gestão de memória do SO, um processo pode ser movido, no todo ou em parte, entre a memória e o disco, ao longo da sua vida
- As instruções dos programas em execução, e os dados associados, são acedidos através dos respetivos endereços em RAM (ciclo *fetch–decode–execute*):
  - *Carregamento* da próxima instrução: a partir do endereço dado pelo *PC*
  - *Descodificação* das instruções: pode causar o carregamento de operandos
  - *Execução* das instruções: pode produzir resultados que é necessário guardar
- Necessidade de mecanismos de associação de instruções e dados a endereços

5.2 Associação de Endereços

Associação de Endereços (1/7)



- Tipicamente, os programas passam por várias fases antes da execução; em cada fase, os endereços podem ser representados de diversas formas:
  - Ao nível do código fonte, os endereços são geralmente **simbólicos**
  - Cabe ao compilador associá-los a endereços **relativos** ou **relocáveis** (i.e., deslocamentos); exemplo: 14 bytes a partir do início do módulo
  - Por sua vez, o editor de ligações (*linker*) ou o carregador (*loader*) irão mapear os endereços relativos em endereços **absolutos**
- Cada associação é um mapeamento entre "espaços de endereçamento" diferentes

Associação de Endereços (2/7)

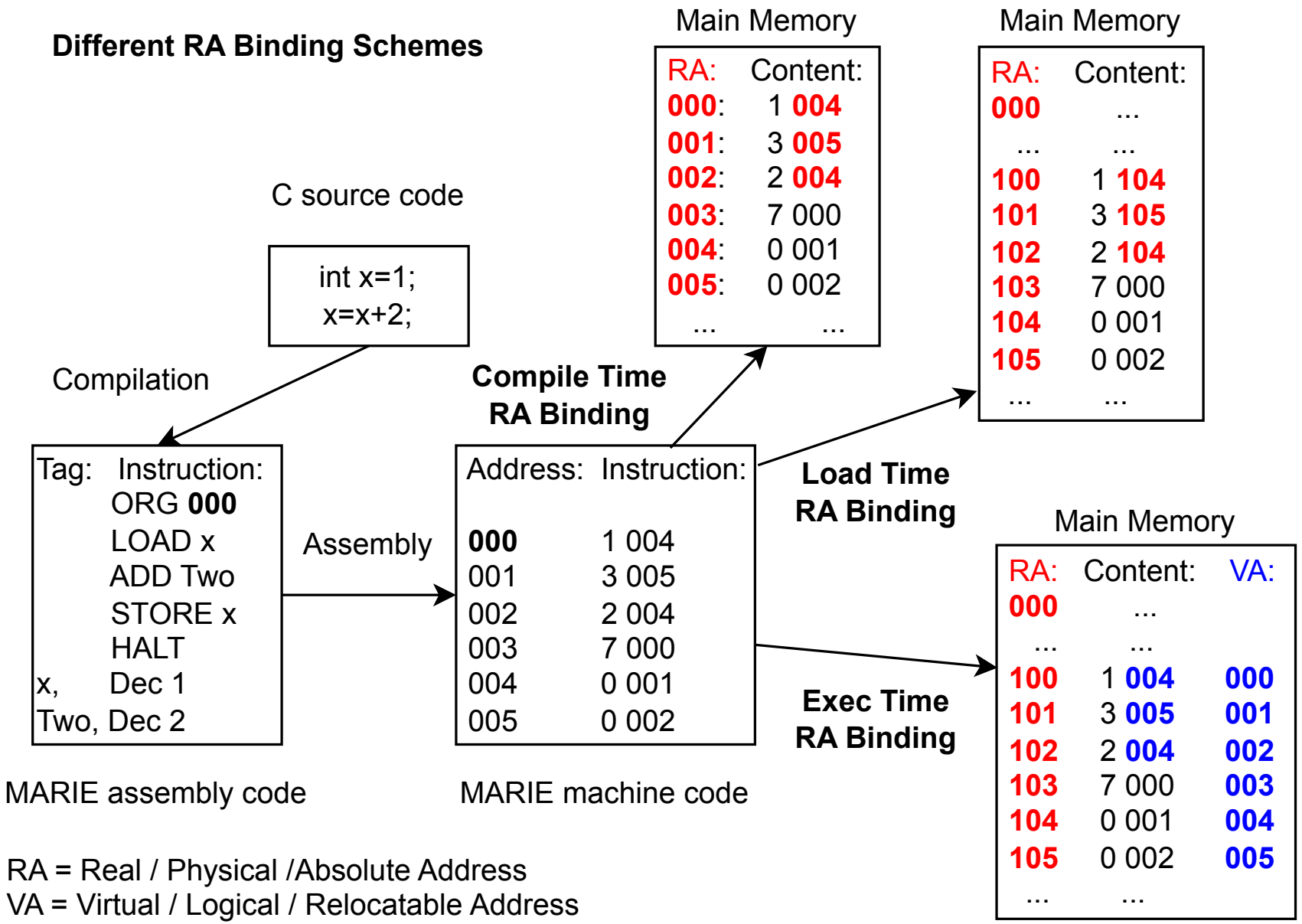
- A associação entre instruções/dados e os respetivos endereços **absolutos** de memória pode ser feita em três etapas distintas:

  - **tempo de compilação**:
    - se o espaço de memória do programa for conhecido à *priori*, então pode ser gerado código com referências absolutas, mas será necessário recompilar o código sempre que o programa for executado numa zona de memória distinta; exemplo: ficheiros MS-DOS de extensão .COM

  - **tempo de carregamento**:
    - se o espaço de memória não for conhecido no momento da compilação, então é necessário gerar código *relocável* (*relocatable*); as associações finais são portanto adiadas até ao tempo de carregamento

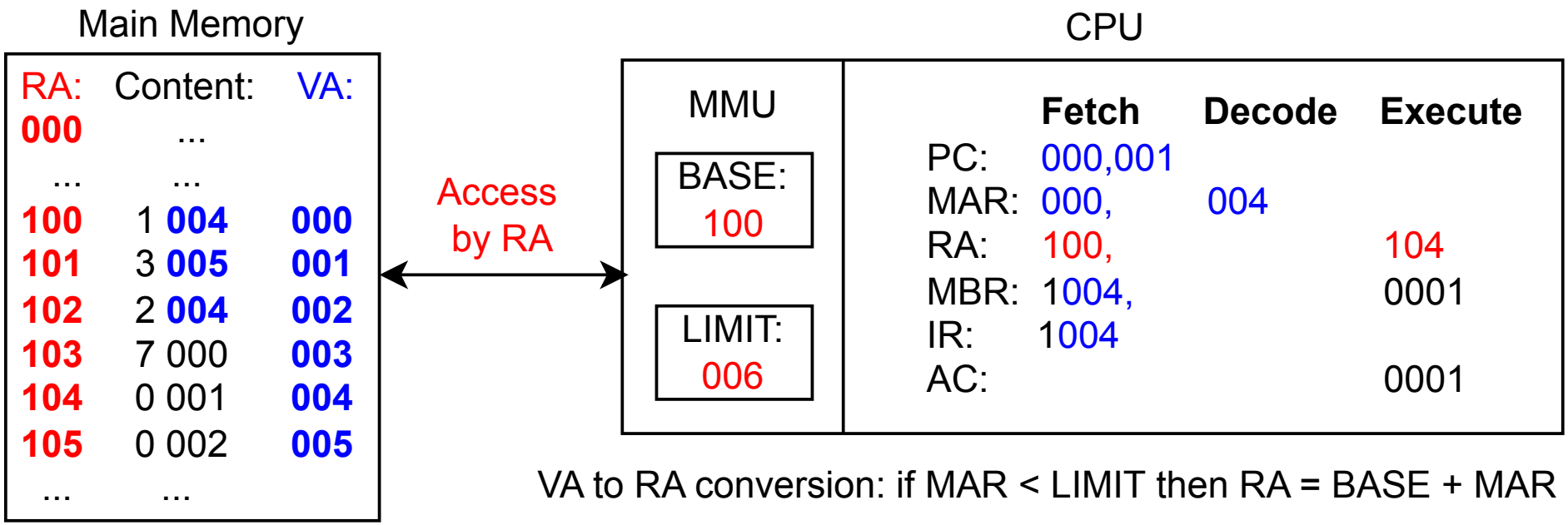
  - **tempo de execução**:
    - se a localização de um programa puder mudar durante a sua execução, a associação terá que ser adiada para tempo de execução; necessita de *hardware* auxiliar (par <registo base, registo limite>, MMU)

Associação de Endereços (3/7): Exemplo



Associação de Endereços: Exemplo (4/7)

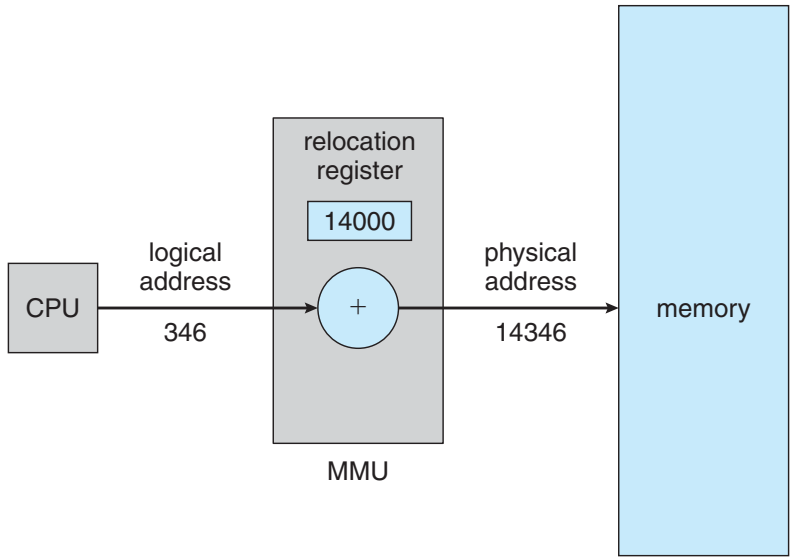
Execution of LOAD x (1 004) with Exec Time RA Binding



VA to RA conversion: if MAR < LIMIT then RA = BASE + MAR

Associação de Endereços (5/7): Endereços Lógicos e Físicos

- Endereço **lógico/virtual**: gerado pela CPU, durante a execução dos programas
- Endereço **físico/real**: endereço de uma posição real da memória principal (RAM)
- Endereços lógicos e físicos são iguais para associações de endereços em tempo de compilação e carregamento, e $\neq$s em associações efetuadas em tempo de exec.
- Em tempo de execução, o mapeamento entre endereços virtuais e endereços reais é feito pela **Unidade de Gestão de Memória** (MMU - Memory Management Unit)
- Os programas lidam com endereços lógicos; nunca conhecem os verdadeiros endereços físicos (exemplo: registos DS, CS, SS e ES nos processadores 80x86)



//Associação de Endereços (6/7): Carregamento Dinâmico

- Uma rotina não será carregada para memória enquanto não for necessária
- As rotinas em disco estão num formato relocável, pronto a ser carregado
- Quando a rotina é necessária, carrega–se se ainda não estiver em memória (convertem-se os endereços relativos em endereços absolutos, e atualizam--se as tabelas de endereços do programa) e inicia–se a execução da rotina
- As rotinas que nunca são invocadas não chegam a ser carregadas (o que permite uma utilização mais eficiente da memória principal)
- Especialmente útil para rotinas muito extensas e raramente invocadas

//Associação de Endereços (7/7): Ligação Dinâmica

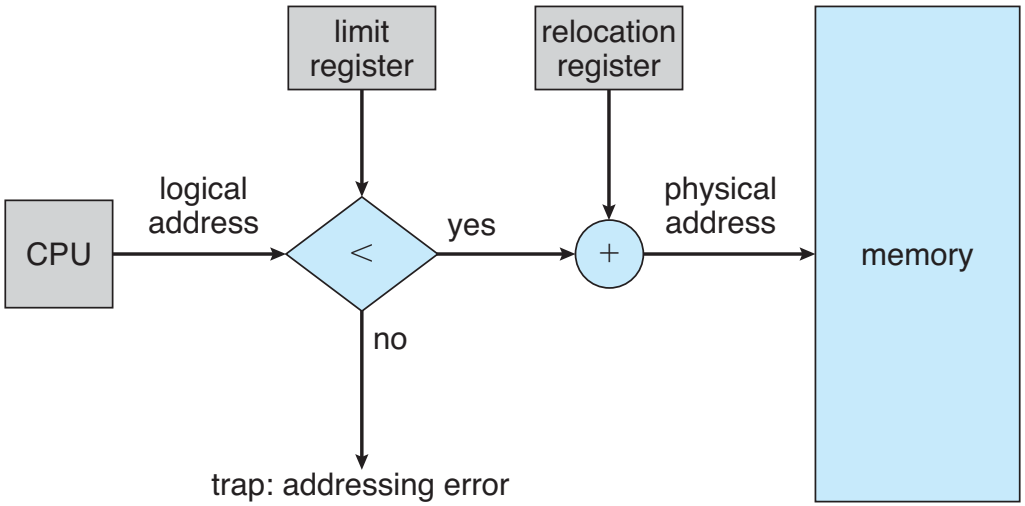
- Ligação Estática: as bibliotecas do sistema são tratadas como módulos de código objeto e combinadas pelo carregador na (grande) imagem binária do programa

- **Ligação Dinâmica**: ligação às bibliotecas de código feita em tempo de execução
  - estratégia comummente usada em bibliotecas de linguagens de programação
  - **stub**: código no lugar de cada invocação a uma rotina da biblioteca
  - o *stub* localiza a rotina nas bibliotecas residentes em memória; se a biblioteca não está ainda em memória, é carregada do disco; por fim, o *stub* substitui-se a si próprio pelo endereço da rotina em causa (o que torna imediata a sua execução em próximas invocações) e inicia a sua execução
  - necessária apenas uma instância da biblioteca em memória, partilhada por todos os processos; vantagens: i) economia de memória; ii) correções às bibliotecas são automaticamente propagadas a todos os programas
  - a manutenção de informação de versão, no programa e na biblioteca, evita que os programas utilizem versões eventualmente incompatíveis

5.3 Alocação Contígua

Alocação Contígua (1/6): Conceitos Básicos

- A memória principal é normalmente dividida em duas grandes partições:
  - uma, em memória baixa, para o vetor de interrupções e SO residente
  - outra, em memória alta, para os processos dos vários utilizadores
- É necessário i) proteger o SO de interferências dos processos dos utilizadores, ii) proteger os processos dos utilizadores de interferências mútuas
- Solução: na CPU, um **registo limite** define o *máximo endereço lógico válido* e um **registo base** (ou de *relocação*) define o *mínimo endereço físico válido*



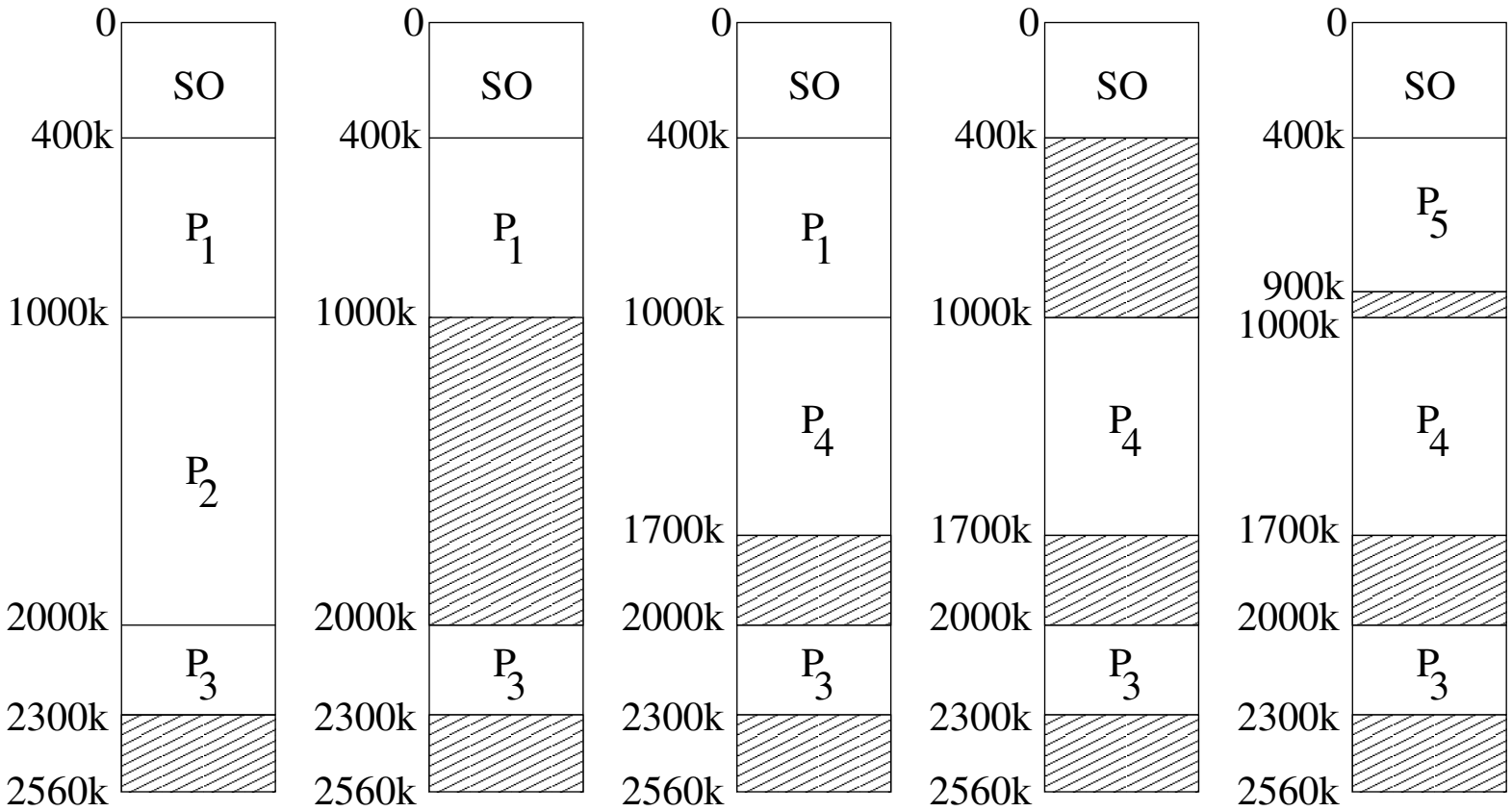
Alocação Contígua (2/6): Múltiplas Partições

- **Múltiplas Partições de tamanho FIXO**:
  - a memória principal é dividida em várias partições, de tamanho fixo
  - cada partição pode acomodar um processo; o número de partições disponíveis à partida define o *grau de multiprogramação* do sistema

- **Múltiplas Partições de tamanho VARIÁVEL**:
  - O SO mantém uma tabela que regista as partições livres e as ocupadas
  - inicialmente: uma única partição, correspondente a toda a memória livre
  - quando um processo é criado: aloca–se memória a partir de uma partição livre com dimensão suficiente; o resto da partição fica disponível
  - se um processo não pode ser satisfeito, fica em *standby*, e atribui–se memória a outro processo, em função do algoritmo de escalonamento
  - a memória fragmenta–se, progressivamente, em partições livres e ocupadas, de dimensão variável; partições livres e adjacentes são fusionadas numa só

Alocação Contígua (3/6): Múltiplas Partições (cont)

- exemplo: processos P_1, P_2, P_3, P_4 e P_5 com requisitos de memória de 600k, 1000k, 300k, 700k e 500k, respetivamente, escalonados segundo FCFS



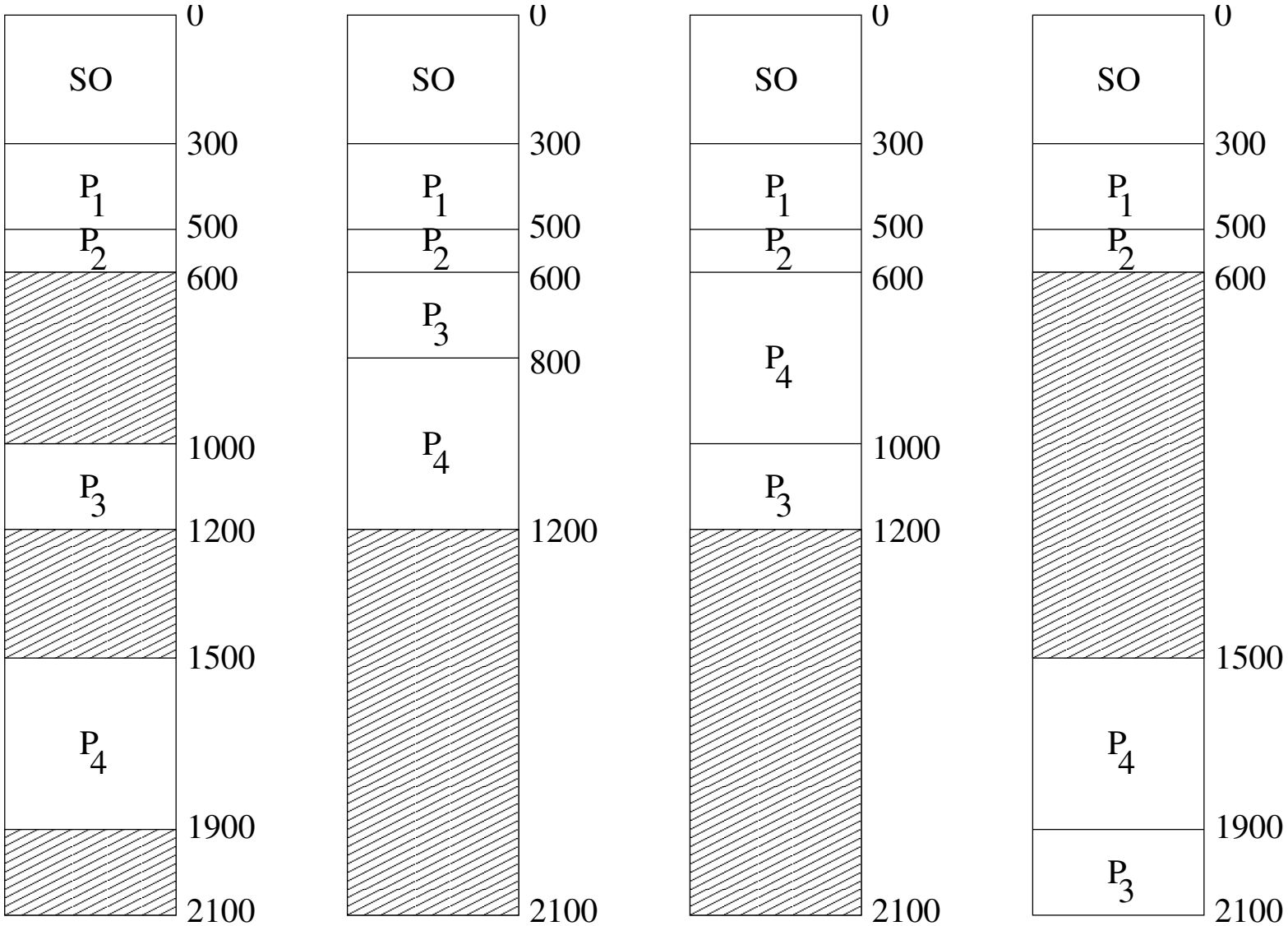
Alocação Contígua (4/6): Múltiplas Partições (cont)

- **Algoritmos de Armazenamento Dinâmico**: satisfazem um pedido de um bloco de tamanho *n* de entre uma lista de blocos livres de dimensões diversas
  - **First–Fit**: aloca o primeiro bloco livre suficiente que encontrar; a pesquisa começa do início da lista ou do bloco onde terminou a última pesquisa
  - **Best–Fit**: aloca o bloco mais pequeno de entre os que têm tamanho suficiente; obriga a percorrer a lista toda, a não ser que esteja ordenada por tamanho; acaba por produzir novos blocos livres (sobras) muito pequenos
  - **Worst–Fit**: escolhe o maior bloco livre; obriga a percorrer a lista toda, a não ser que esteja ordenada por tamanho; produz as maiores sobras, na esperança de que possam ser reutilizadas
- Simulações realizadas sugerem que o *First–Fit* e o *Best–Fit* são equiparáveis em termos de velocidade e utilização da memória, sendo o algoritmo *First–Fit* um pouco mais rápido; ambos algoritmos têm rendimento superior ao *Worst–Fit*

Alocação Contígua (5/6): Fragmentação e Compactação

- Os algoritmos de armazenamento dinâmico geram **fragmentação externa**
- **Fragmentação Externa**: existe memória suficiente para satisfazer um pedido, mas essa memória não é contígua, encontrando-se dispersa em pequenos blocos livres
- **Fragmentação Interna**: é alocado um bloco de memória ligeiramente maior que o necessário, criando um fragmento interno à partição, que não é usado
  - justifica–se se a dimensão do novo fragmento não compensar a sua gestão
- A fragmentação externa pode ser combatida através da **Compactação**:
  - juntar todos os blocos livres num único bloco contíguo
  - só é possível se a relocação for *dinâmica* e feita em *tempo de execução*
  - a compactação pelo algoritmo mais simples pode ser muito pesada ...
  - em geral, para uma determinada disposição dos blocos de memória livres, há esquemas de compactação alternativos, com menor impacto no desempenho
  - a compactação pode ser combinada com *swapping*: faz–se o *swap out* dos processos a mover e o *swap in* para as novas localizações

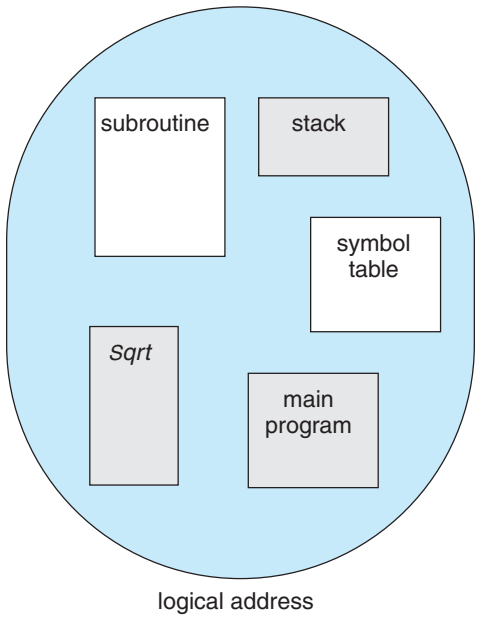
Alocação Contígua (6/6): Compactação



5.4 Segmentação

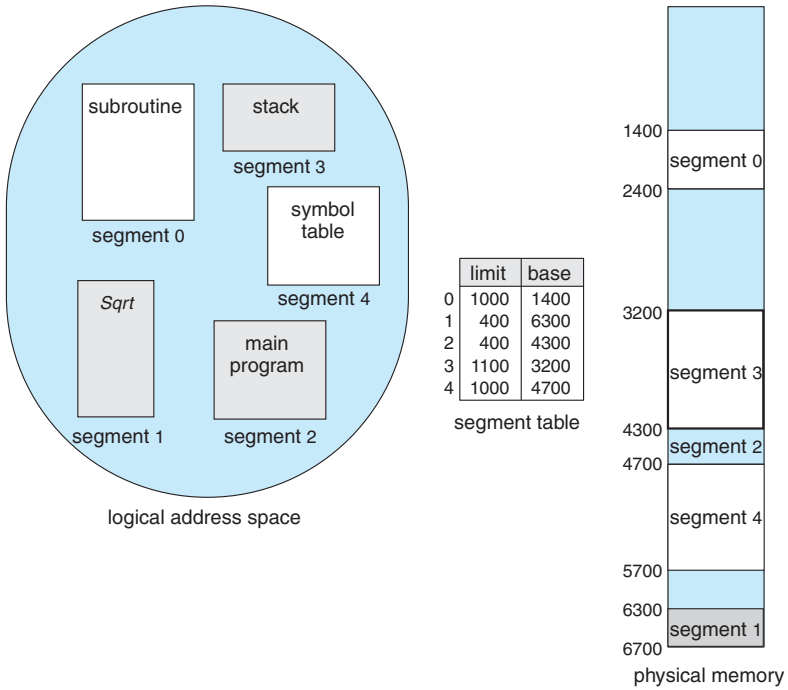
Segmentação (1/5) : Conceitos Básicos

- A **segmentação** suporta uma **visão funcional da memória**, como uma **coleção dispersa de segmentos**, de tamanhos e funções variáveis, compatível com a visão intuitiva que o utilizador / programador tem da memória (**visão conveniente**):
  - main e restantes funções/procedimentos; programa principal e módulos
  - estruturas de dados; variáveis locais e globais; memória estática e dinâmica



Segmentação (2/5) : Conceitos Básicos (cont.)

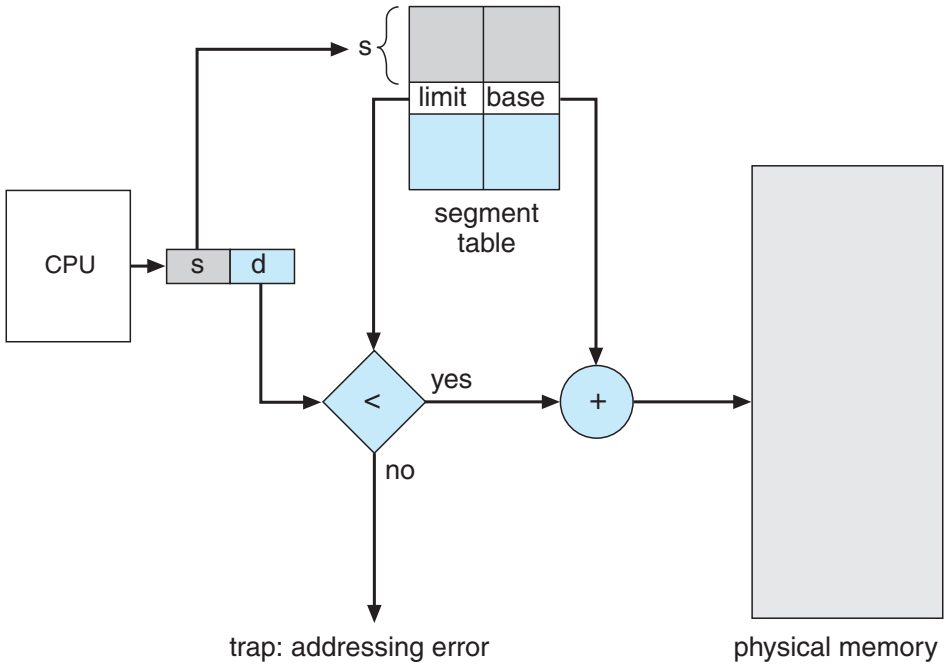
- A segmentação é um mecanismo de *relocação dinâmica* baseado numa **tabela de segmentos** - uma tabela de pares <base, limite>, com um par por cada segmento



- Em segmentação, a divisão do programa em segmentos é *explícita*, ocorrendo tipicamente ainda antes da execução (já na fase da compilação, por exemplo)
- Durante a execução do programa, é apenas necessário definir o segmento em uso num dado instante, programando para o efeito um registo específico da CPU

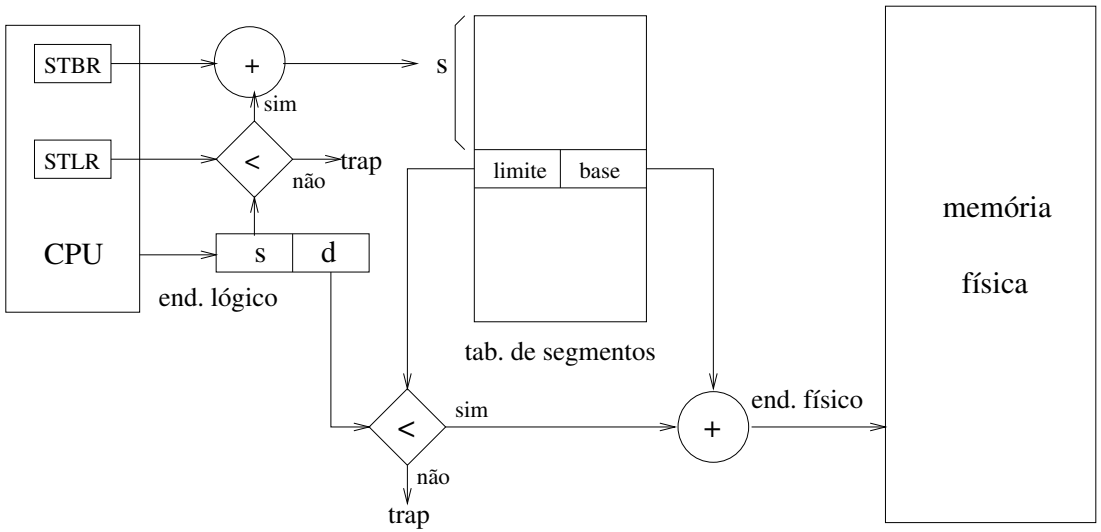
Segmentação (3/5): Tradução de Endereços

- Um programa em execução gera endereços lógicos/virtuais já divididos em:
  - **número do segmento** (s) - usado como índice numa **tabela de segmentos** em que cada entrada contém um endereço físico *base* e um *limite* (altura)
  - **deslocamento no segmento** (d) - somado ao endereço *base* do segmento para definir o endereço físico que é enviado para a unidade de memória



Segmentação (4/5) : Suporte do Hardware, Protecção

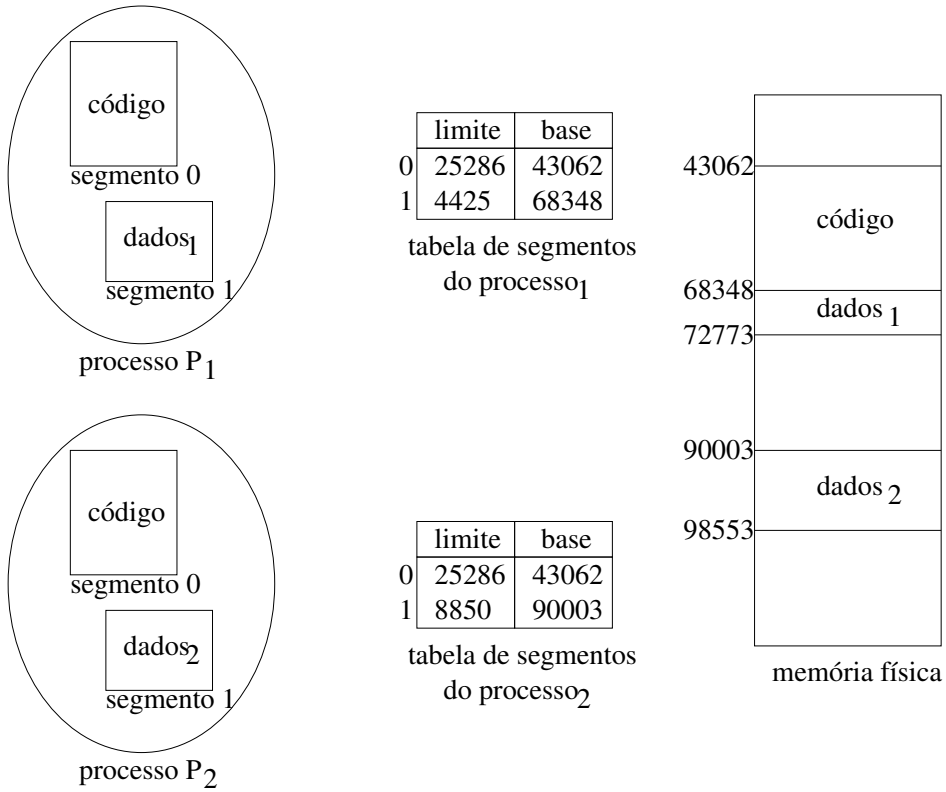
- **Suporte do Hardware**
  - a tab. de segm. (TS) será guardada em registos ou em memória principal
  - TS guardada em RAM: endereço em **registo base da tabela de segmentos** (STBR) e altura em **registo limite da tabela de segmentos** (STLR)
  - **caches associativas** guardam parte da TS para minimizar o acesso à TS



- **Protecção**:
  - "natural": o processo limita–se a aceder aos segmentos referenciados pela TS; por outro lado, o acesso à TS e aos registos STBR e STLR é privilegiado
  - mecanismos complementares: registo STLR, bits de validade e acesso (r,w,x)
  - unidade mínima de proteção: segmento (a semântica do acesso ao segmento deve ser a mesma, uma vez que representa uma partição lógica do programa)

Segmentação (5/5) : Partilha, Fragmentação

- **Partilha**:
  - unidade mínima de partilha de memória entre processos: segmento
  - em geral, segmentos partilhados não têm de ter o mesmo número ...



- **Fragmentação**:
  - a segmentação produz **fragmentação externa**
  - segmentos pequenos tendem a reduzir a fragmentação externa
  - alocação de segmentos em memória física: *best-fit* ou *first-fit*
  - a fragmentação externa da RAM pode ser combatida permitindo que o espaço de endereçamento de um processo seja *descontínuo*, como acontece na **paginação**

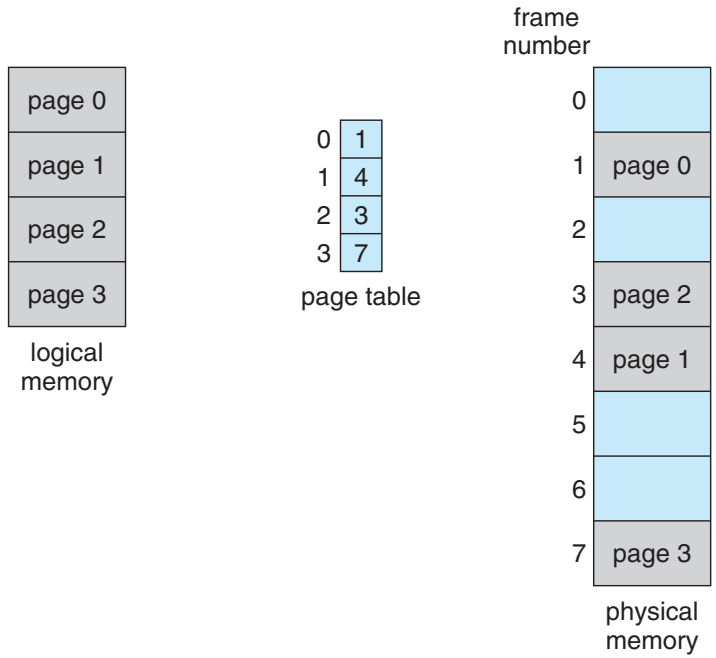
5.5 Paginação

Paginação (1/21) : Conceitos Básicos

- A **paginação** oferece uma **visão linear da memória**, como **sequência contígua de páginas**, de tamanho fixo e funções indiferenciadas, de gestão muito **eficiente**
  - Os processos assumem a disponibilidade de um Espaço de Endereçamento Virtual (EEV) próprio, de **grandes dimensões** e aparentemente **contíguo**
  - Na realidade, esse EEV é subdividido em **páginas** e estas são associadas, de forma transparente aos processos, a **frames em memória real** ou a **blocos em disco**, com base na consulta de **tabelas de páginas** (uma por processo)
    - a tradução entre endereços virtuais (EVs) e endereços reais (ERs) é feita, geralmente, por hardware (MMU); mas, por vezes, terá de ser o SO a realizar essa tradução por software (exemplo: as primitivas lidam com EVs mas o SO tem que colocar os seus resultados em ERs)
  - Os frames e blocos correspondentes às páginas de um processo estão, tipicamente, dispersos na memória real ou no disco (respetivamente), onde co-existem com frames e blocos pertencentes a outros processos
  - Assim, a paginação também resolve o problema da fragmentação da área de *swap* (*backing store*), onde recorrer à *compactação* seria muito ineficiente ...

Paginação (2/21) : Conceitos Básicos

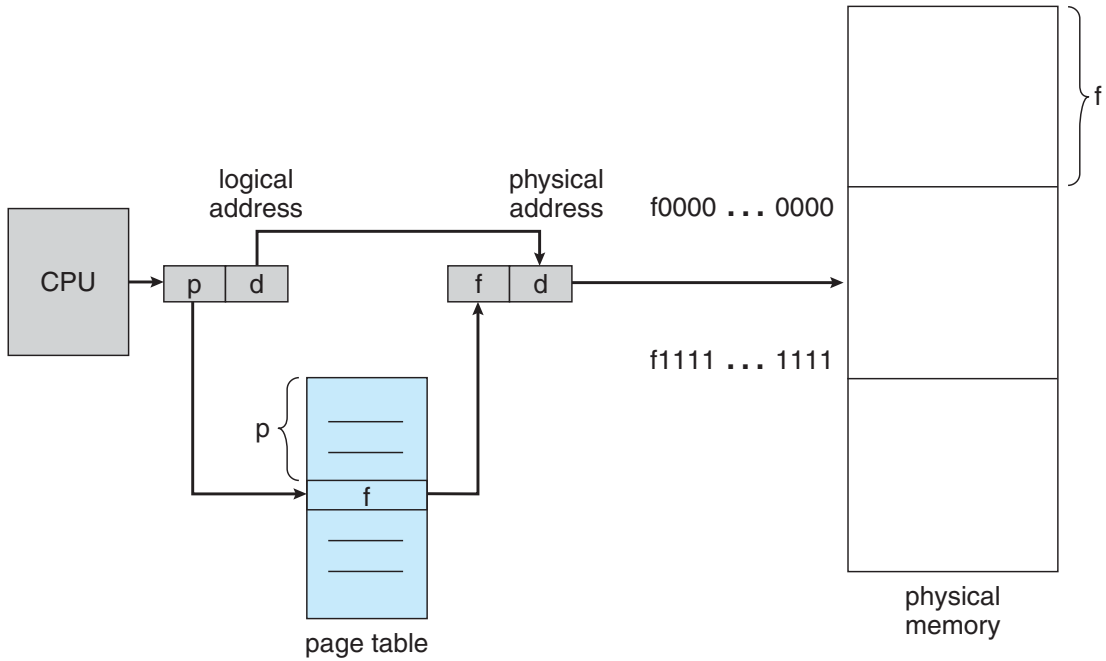
- A **paginação** é um mecanismo de *relocação dinâmica* baseado numa **tabela de páginas** (uma *tabela de registos de relocação*, com um registo por página):
  - a memória física é dividida em blocos iguais, de tamanho fixo, denominados de **frames**
  - em memória lógica (virtual), os blocos correspondentes aos frames são as **páginas**
  - uma **tabela de páginas**, por processo, permite associar páginas virtuais a frames reais
  - a área de swap é dividida em **blocos** do mesmo tamanho que as páginas/frames ...



- Em paginação, a divisão do processo em páginas é *implícita*, ocorrendo automaticamente à medida que o programa é carregado e executado.

Paginação (3/21) : Tradução de Endereços

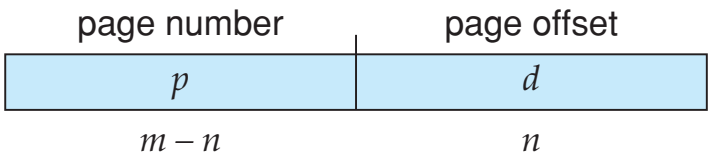
- Os programas em execução geram um endereço lógico (virtual) por inteiro, mas que será dividido *automaticamente* pela MMU em <página, deslocamento>
  - **número da página** (p) - usado como índice numa tabela de páginas na qual cada entrada contém o endereço físico base (f) de um frame
  - **deslocamento na página** (d) - somado ao endereço base (f) do frame para definir o endereço físico que é enviado para a unidade de memória



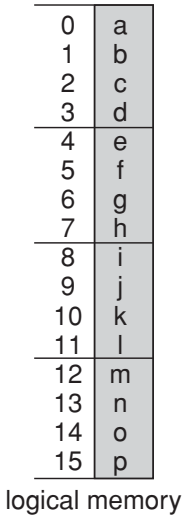
- a dimensão das páginas/frames é definida pelo HW sendo tipicamente uma potência de 2 (o que permite a tradução eficiente de endereços pelo HW)

Paginação (4/21) : Tradução de Endereços

- um Espaço de Endereçamento Virtual (EEV) de m bits tem 2^m endereços virtuais, distribuídos por 2^{m-n} páginas, com 2^n endereços por cada página
- **estrutura de um endereço virtual**:



- **exemplo**: EEV de m=5 bits ($\Rightarrow 2^5 = 32$ endereços virtuais <p,d>) e um EER de 4 bits ($\Rightarrow 2^4 = 16$ ender. reais <f,d>)
  - com páginas de 4 bytes, o deslocamento d usa n=$log_2(4) = 2$ bits
  - sobram m-n=3 bits para identificar uma página p ($\Rightarrow 2^3 = 8$ páginas)
  - sobram 4-2=2 bits para identificar uma frame f ($\Rightarrow 2^2 = 4$ frames)



Paginação (5/21) : Tradução de Endereços

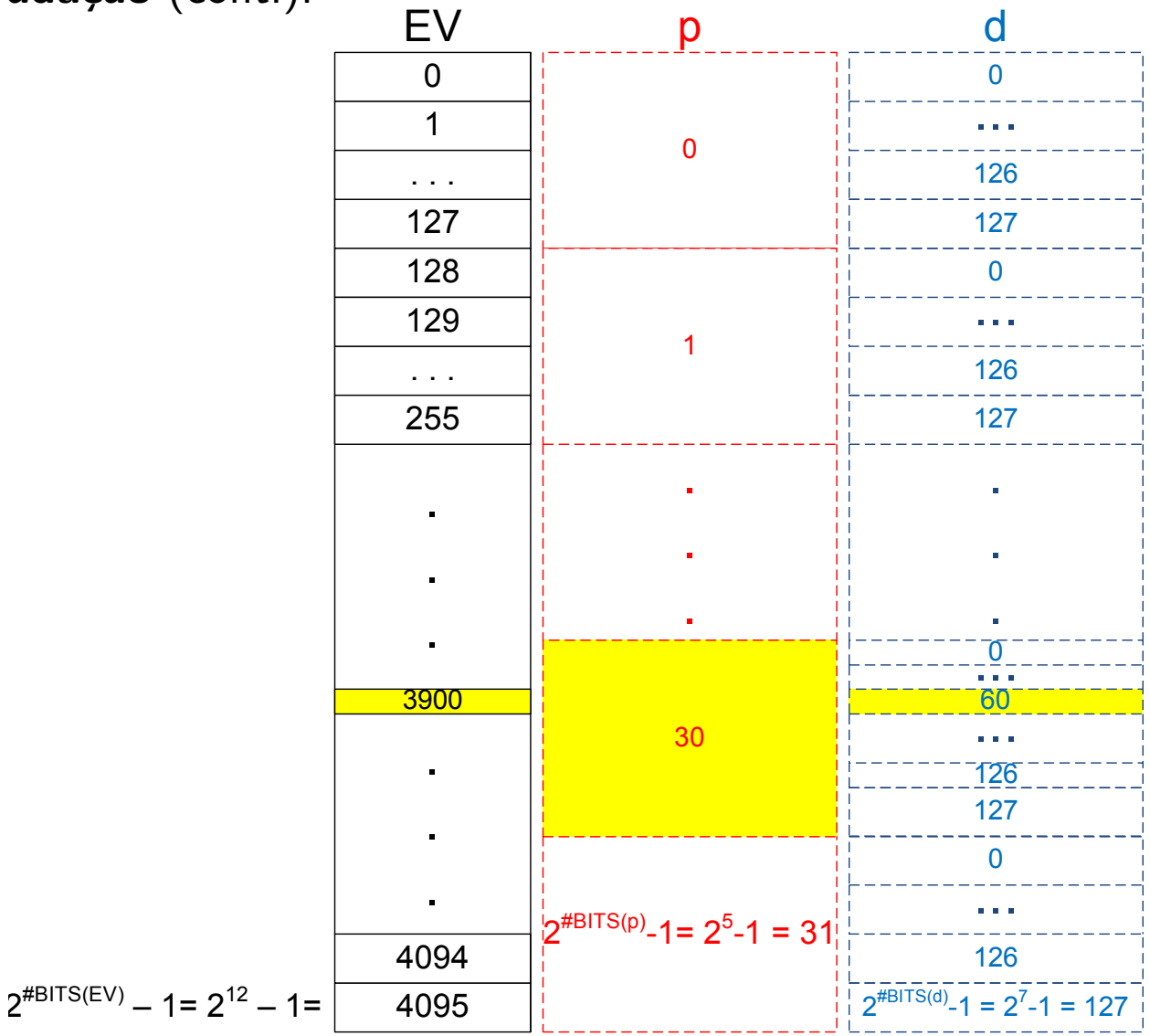
exemplo de tradução:

- EV $=< p, d >$; #BITS(EV) $= 12$ bits ; #BITS(d) $= 7$ bits
- ER $=< f, d >$; #BITS(ER) $= 10$ bits ; f $=$ p % 8
- $\text{EV}_{10} = 3900$; $\textbf{ER}_{10} =?$

- SIZEOF(p) $= 2^{\#\text{BITS(d)}} = 2^7 = 128$ endereços (ou deslocamentos)
- #BITS(p) $=$ #BITS(EV) $-$ #BITS(d) $= 12 - 7 = 5$ bits
- #PAGINAS $= 2^{\#\text{BITS(p)}} = 2^5 = 32$ paginas
- $\text{EV}_{10} = 3900 = 2^{11} + 2^{10} + 2^9 + 2^8 + 2^5 + 2^4 + 2^3 + 2^2$
- $\text{EV}_2 = 111100111100 =< 11110, 0111100 >=< p, d >$
- $\text{p}_2 = 11110 \Leftrightarrow \text{p}_{10} = 2^4 + 2^3 + 2^2 + 2^1 = 30$
- $\text{d}_2 = 0110010 \Leftrightarrow \text{d}_{10} = 2^5 + 2^4 + 2^3 + 2^2 = 32 + 16 + 8 + 4 = 60$
- verificação:
  $\text{EV}_{10} = \text{p}_{10} \times \text{SIZEOF(p)} + \text{d}_{10} \Leftrightarrow 3900_{10} = 30 \times 128 + 60$ (*VERDADE*)

Paginação (6/21) : Tradução de Endereços

exemplo de tradução (cont.):



Paginação (7/21) : Tradução de Endereços

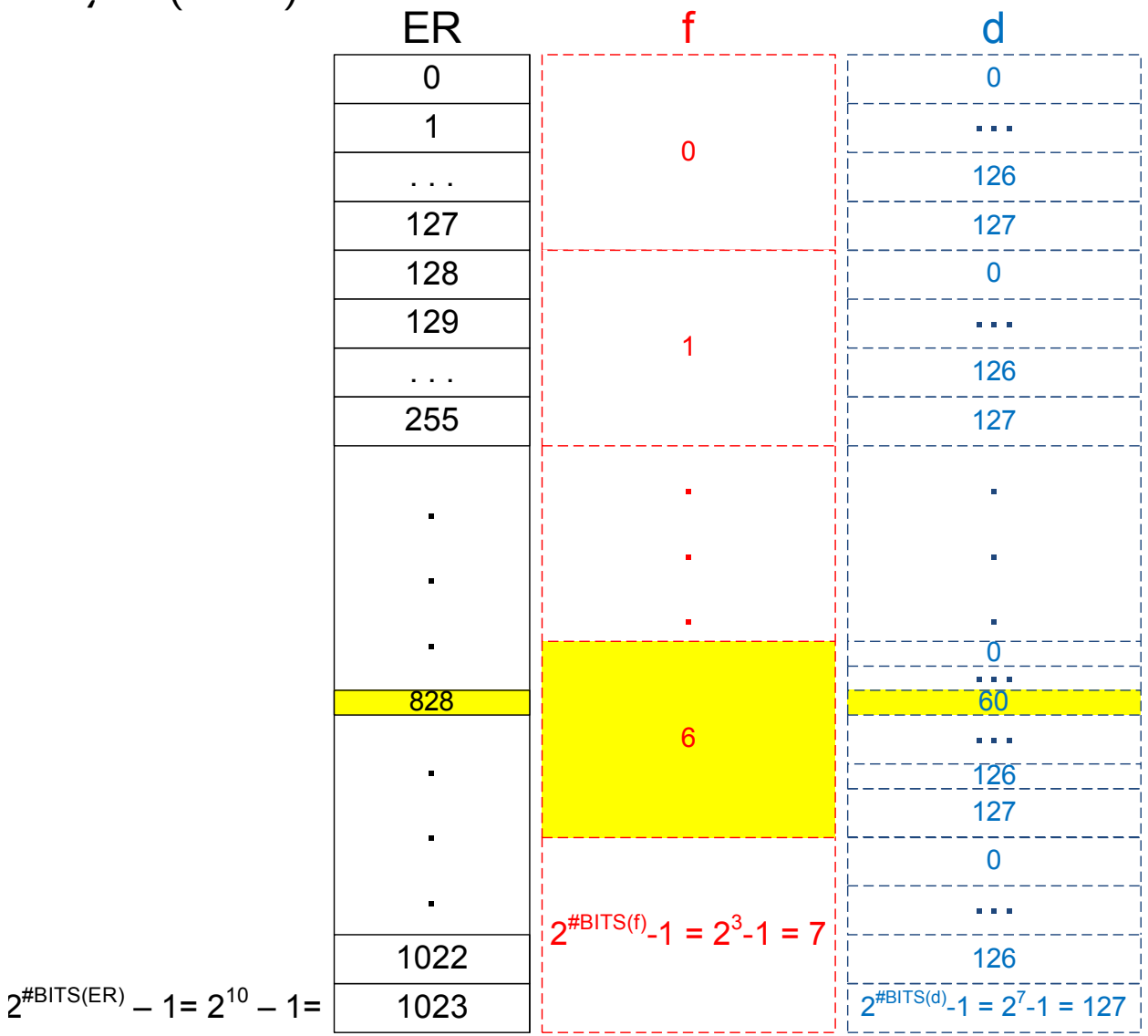
exemplo de tradução (cont.):

- EV $=< p, d >$; #BITS(EV) $= 12$ bits ; #BITS(d) $= 7$ bits
- ER $=< f, d >$; #BITS(ER) $= 10$ bits ; f $=$ p % 8
- $EV_{10} = 3900$; $\mathbf{ER}_{10} =?$

- SIZEOF(f) $=$ SIZEOF(p) $= 128$ endereços (ou deslocamentos)
- #BITS(f) $=$ #BITS(ER) $-$ #BITS(d) $= 10 - 7 = 3$ bits
- $\#FRAMES = 2^{\#BITS(f)} = 2^3 = 8$ frames
- $f_{10} = p_{10} \% \ 8 = 30 \ \% \ 8 = 6$
- $ER =< f, d >=< 6_{10}, 60_{10} >=< 110_2, 0111100_2 >= 1100111100_2$
- $ER_{10} = 2^9 + 2^8 + 2^5 + 2^4 + 2^3 + 2^2 = \mathbf{828}$
- verificação:
  $ER_{10} = f_{10} \times SIZEOF(f) + d_{10} \Leftrightarrow 828_{10} = 6 \times 128 + 60$ (*VERDADE*)

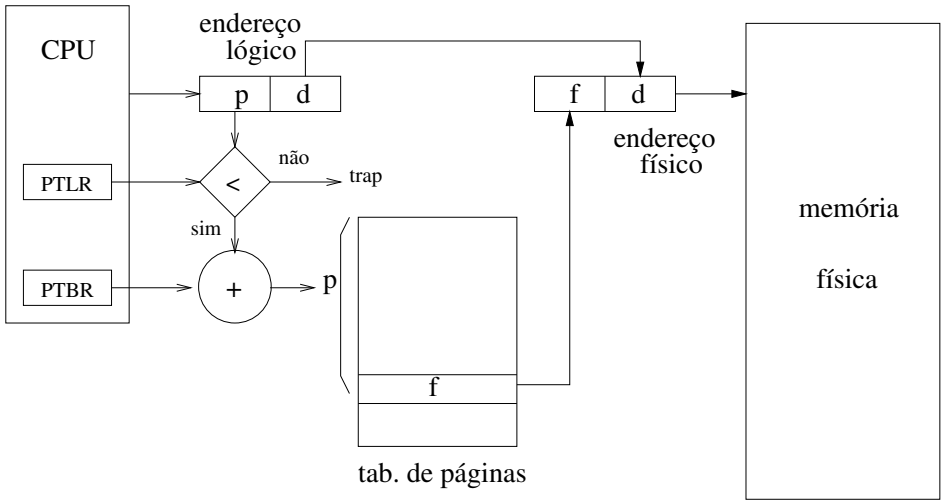
Paginação (8/21) : Tradução de Endereços

exemplo de tradução (cont.):



Paginação (9/21): Suporte do Hardware

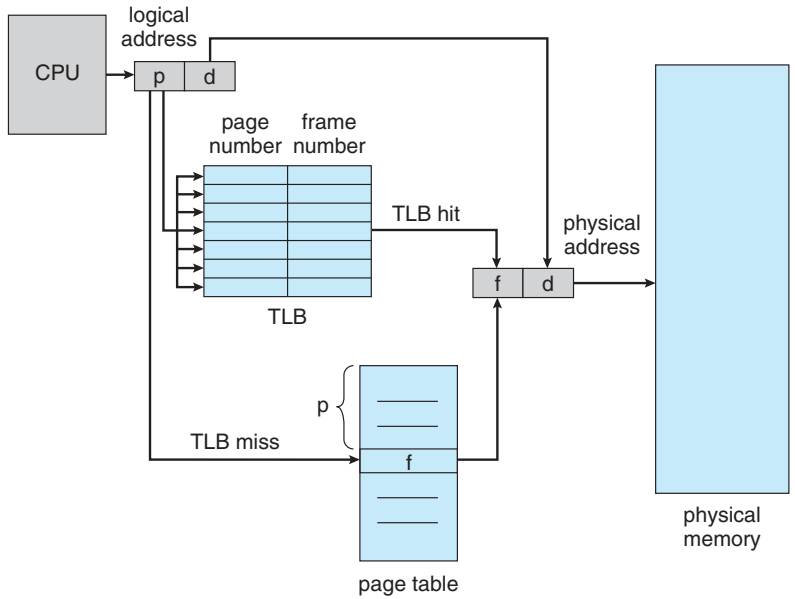
- Tabelas de Páginas (TPs) de pequenas dimensões podem ser copiadas integralmente para os registos da CPU e aí utilizadas durante a execução do processo
- TPs maiores terão de operar-se diretamente a partir da memória principal
  - o endereço da TP é carregado num **registo base da tabela de páginas** (PTBR) e a sua altura num **registo limite da tabela de páginas** (PTLR)



  - estes registos permitem uma comutação rápida de contexto ... mas operar TPs diretamente a partir da memória principal prejudica o desempenho
  - com a TP em memória, cada acesso à memória gerado por um processo resulta efetivamente em dois acessos: um à TP e outro ao endereço real

Paginação (10/21) : Suporte do Hardware (TLBs)

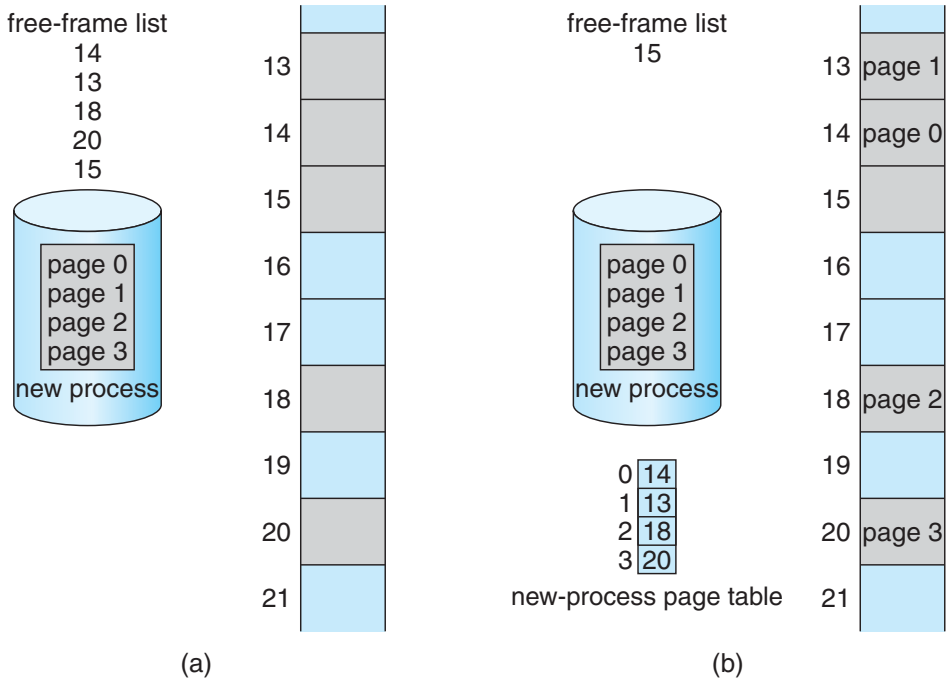
- A tradução entre EVs e ERs pode então ser acelerada através de uma cache especial de **registos associativos** ou **translation look-aside buffers** (TLBs)
  - memória associativa: permite pesquisar simultaneamente todas as entradas
- Os TLBs mantêm os pares <página, frame> da TP acedidos mais recentemente
- A pesquisa nos TLBs é efetuada simultaneamente com a pesquisa na TP



- A comutação de processos implica o esvaziamento prévio dos TLBs (exceto se suportarem *address-space identifiers* (ASIDs), diferentes entre processos; certos TLBs podem ser dedicados, permanentemente, a certas frames, e.g. do SO)

Paginação (11/21) : Atribuição de Frames

- um novo processo necessita de um certo número de páginas
- para cada página é necessário reservar uma frame
- o processo é carregado para RAM se houver frames livres suficientes
  - o SO mantém uma **tabela de frames** para gerir a memória física (estado livre/ocupado, página(s) e processo(s) que ocupam a frame)
  - *paginação por necessidade*: um mecanismo de memória virtual que permite a um processo ter apenas parte das suas páginas em frames
- **exemplo**:



Paginação (12/21) : Fragmentação

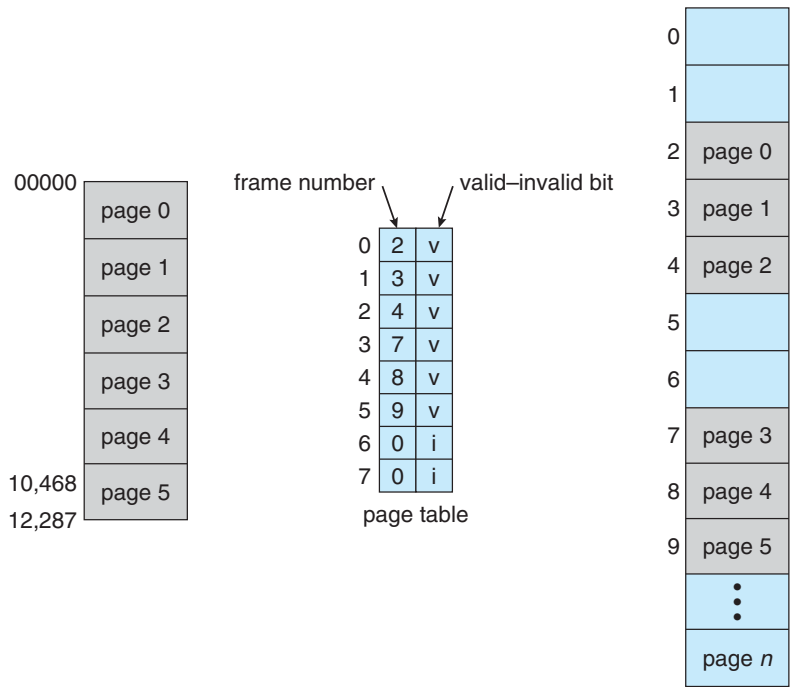
- A paginação **evita fragmentação externa**:
  - uma página pode ser mapeada em qualquer frame livre
- A paginação **gera fragmentação interna**, na última página
  - motivo: a unidade mínima de alocação é a página / frame
  - exemplo:
    - páginas de 2KB, processo de 72766 bytes; necessárias 36 páginas (35 completas mais 1086 bytes da última página; desperdício de 962 bytes)
  - em média, a fragmentação interna na última página será de 50% $\Rightarrow$ convém reduzir o tamanho da página $\Rightarrow$ tabela de páginas maior
  - mas páginas de dimensão equivalente ou múltipla dos blocos do disco permitem otimizar as transferências entre a memória e o disco ...

Paginação (13/21) : Proteção de Memória

- A paginação assegura, automaticamente, **proteção de memória**:
  - a cada processo é associada uma tabela de páginas, como parte dos seus atributos; o processo limita–se a aceder aos frames referenciados na TP
  - à partida, os frames referenciados na TP de um processo apenas podem ser acedidos por esse processo (salvo situações de *partilha* de memória)
  - o controlo do processo de mapeamento EVs $\leftrightarrow$ ERs é efetuado pelo SO (carregamento dos registos da MMU, manipulação das tabelas de páginas)
- Outro nível de proteção diz respeito ao **modo de acesso** (leitura, escrita ou execução) à página; uma entrada da TP reserva bits específicos para o efeito

Paginação (14/21) : Proteção de Memória

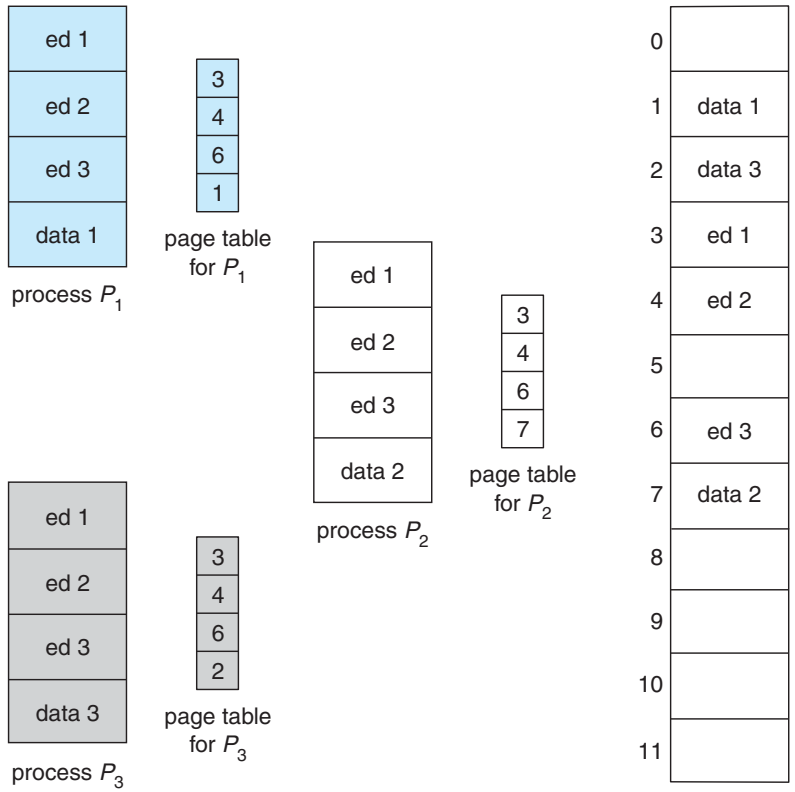
- Por outro lado, nem todas as entradas da TP referenciam frames válidos, ou seja, o EEV de um processo baseia–se, tipicamente, num pequeno número de páginas, quando comparado com a dimensão máxima possível desse EEV (tipica/ enorme)
- Assim, cada entrada da TP contém um **bit de validade**: indica se a página correspondente ao índice da entrada faz parte ou não do EEV do processo



- Complementarmente: a utilização do registo PTLR permite limitar a altura da TP, evitando uma TP desnecessariamente grande (útil nos casos em que o processo necessita apenas de um subconjunto inicial das páginas do seu EEV)

Paginação (15/21) : Partilha de Memória (Páginas)

- A paginação suporta a **partilha de memória** real (frames) entre processos
- Duas páginas virtuais p e q, pertencentes aos EEVs de dois processos distintos, de PIDs P_1 e P_2, serão partilhadas se a TP de P_1 e a TP de P_2 mapearem p e q, respetivamente, no mesmo frame real f (notar que p e q podem ou não ser iguais)



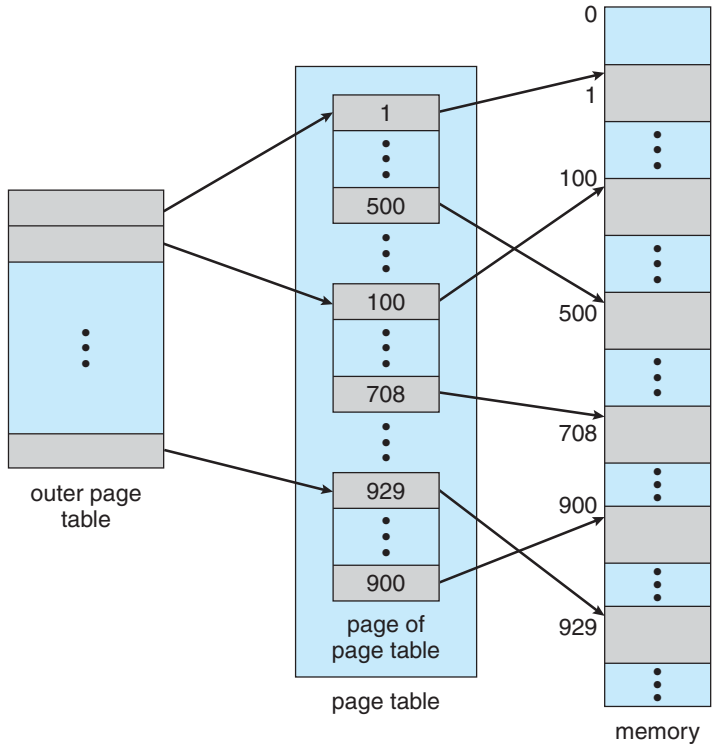
- Duas páginas virtuais p e q distintas, pertencentes ao EEV de um mesmo processo, podem ser mapeadas no mesmo frame real f; o processo poderá aceder aos endereços reais de f utilizando endereços virtuais da página p ou da página q

Paginação (16/21) : Estrutura da Tabela de Páginas

Paginação Hierárquica ou Multinível

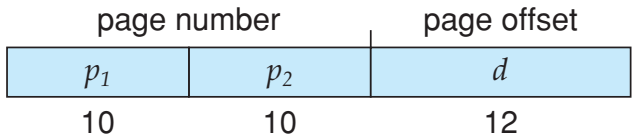
- Com EEVs de dimensão elevada (2^{32} ou 2^{64}) pode não ser viável manter em memória TPs completas; por exemplo: para um EEV de 32 bits e páginas de 4 Kbytes (2^{12} bytes), a TP terá 2^{20} entradas (mais de um milhão); se cada entrada consumir 4 bytes, então a TP de cada processo ocupará 4 Mbytes de memória; e como podem co-existir dezenas de processos (logo de TPs) na mesma máquina ...
- É também de evitar a necessidade de se usar memória real contígua para as TPs

- Uma solução: "paginar a própria tabela de páginas", i.e., as entradas de uma TP *exterior* apontam para TPs *interiores* até que as entradas da última TP apontam para frames
  - estratégia similar à usada no *esquema multinível* da alocação indexada nos Sist. de Ficheiros !

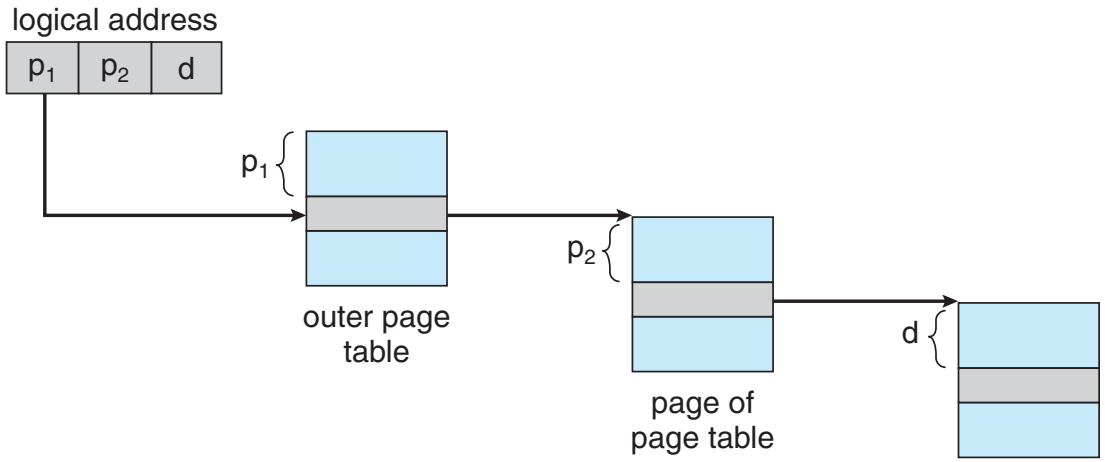


Paginação Hierárquica ou Multinível (cont.)

- Exemplo: **paginação a 2 níveis** de um EEV de 32 bits com páginas de 4 Kbytes
  - a componente deslocamento consome $log_2(4K) = 12$ bits
  - dividem–se os 32-12=20 bits da componente página em dois grupos
    - p_1 : 10 bits para indexar uma **tabela de 1º nível**
    - p_2 : 10 bits para indexar uma **tabela de 2º nível**



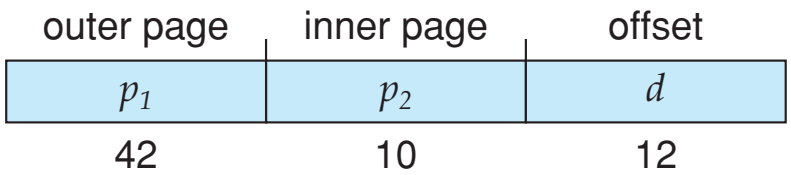
  - esquema de tradução de endereços:



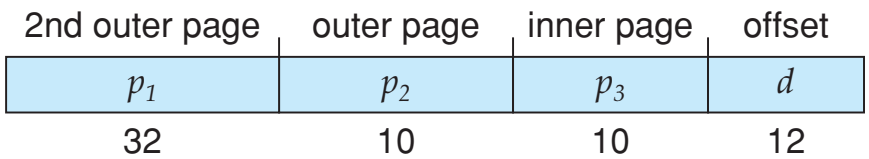
Paginação (18/21) : Estrutura da Tabela de Páginas

Paginação Hierárquica ou Multinível (cont.)

- Exemplo: **paginação a 2 níveis** de um EEV de **64** bits com páginas de 4 Kbytes
  - a componente deslocamento continua a consumir $log_2(4K) = 12$ bits
  - dividem–se os 64-12=52 bits da componente página em dois grupos
    - p_1 : 42 bits para indexar uma **tabela de 1º nível**
    - p_2 : 10 bits para indexar uma **tabela de 2º nível**



    - PROBLEMA: a tabela de 1º nível terá 2^{42} entradas (demasiado)
    - SOLUÇÃO: sub-dividir os 42 bits em vários grupos; por cada grupo / sub-divisão adicional está-se a criar mais um nível de paginação ...
    - exemplo: **paginação a 3 níveis**

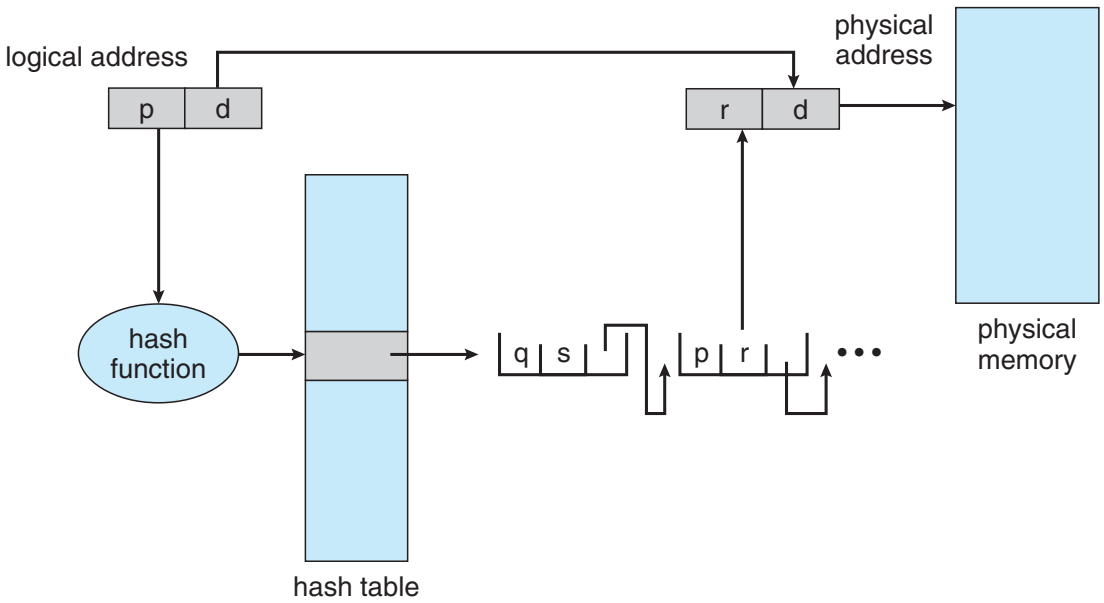


- A paginação multinível degrada o tempo de acesso à memória ...
- ... reforçando ainda mais necessidade de caches associativas
- Para EEVs com mais de 32 bits são convenientes outras soluções ...

Paginação (19/21) : Estrutura da Tabela de Páginas

Paginação baseada em Tabelas de Hashing (cont.)

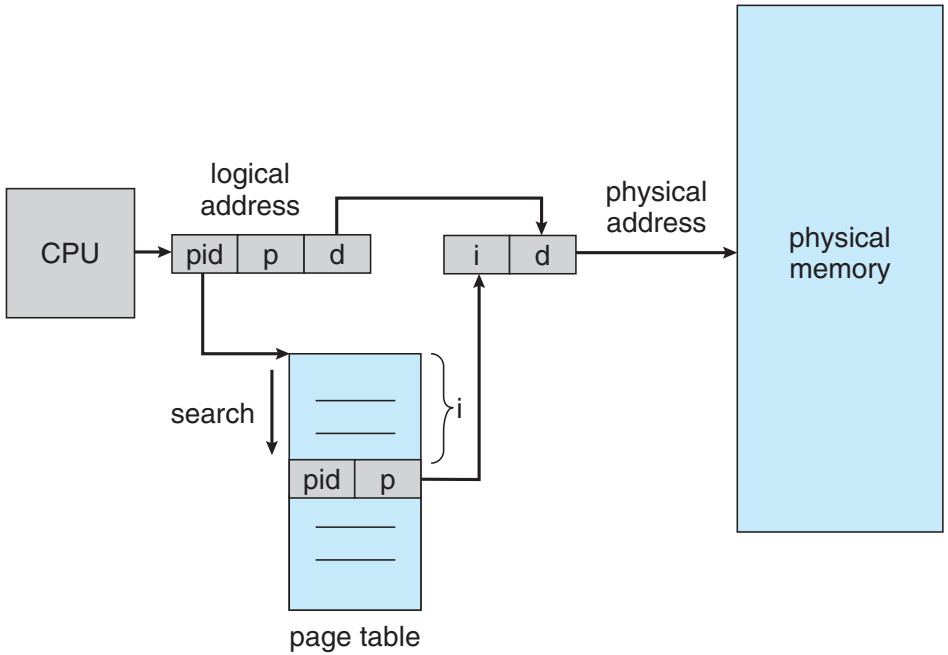
- é uma solução apropriada para EEVs de grandes dimensões (e.g., de 64 bits)
- a componente **página** de um EV é usada como argumento de uma função de hash
- a função de hash devolve um índice de uma entrada de uma tabela de hash
- nessa entrada encontram-se uma lista de pares <**página**,**frame**>
- exemplo: tradução da página **p** no frame **r**



Paginação (20/21) : Estrutura da Tabela de Páginas

Tabela de Páginas Invertida

- Problema: o EEV dos processos é tipicamente de grandes dimensões, exigindo TPs com muitas entradas, a maioria das quais nem sequer referenciam frames !
- Solução: em vez de manter uma TP por processo, o SO mantém uma única **tabela de páginas invertida**, global ao sistema, com uma entrada por cada frame:



- Filosofia: a TP invertida associa um frame (i) a um processo (pid / ASID) e a uma página (p), ou seja, associa o EV <pid, p, d> a um ER <i, d>

Paginação (21/21) : Estrutura da Tabela de Páginas

Tabela de Páginas Invertida (cont.)

- Vantagens:
  1: o nº máximo de frames é conhecido e limitado (porque a memória física é limitada), logo a TP invertida tem um tamanho fixo, bem definido

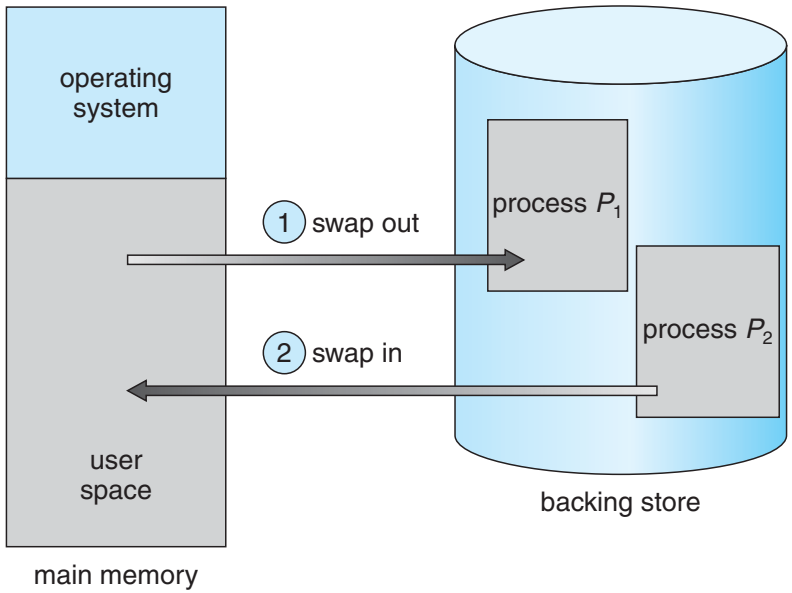
  2: EER $<<$ EEV $\Rightarrow$ a TP invertida é relativamente pequena ...

- Desvantagens:
  1: necessária pesquisa sequencial de todas as entradas da TP, em busca de uma entrada <i> com um par <pid, p> igual ao do EV (parcialmente resolúvel recorrendo a hashing e/ou a TLBs)

  2: esquema pouco apropriado à partilha de páginas entre processos

5.6 *Swapping*

Standard *Swapping* (1/3)

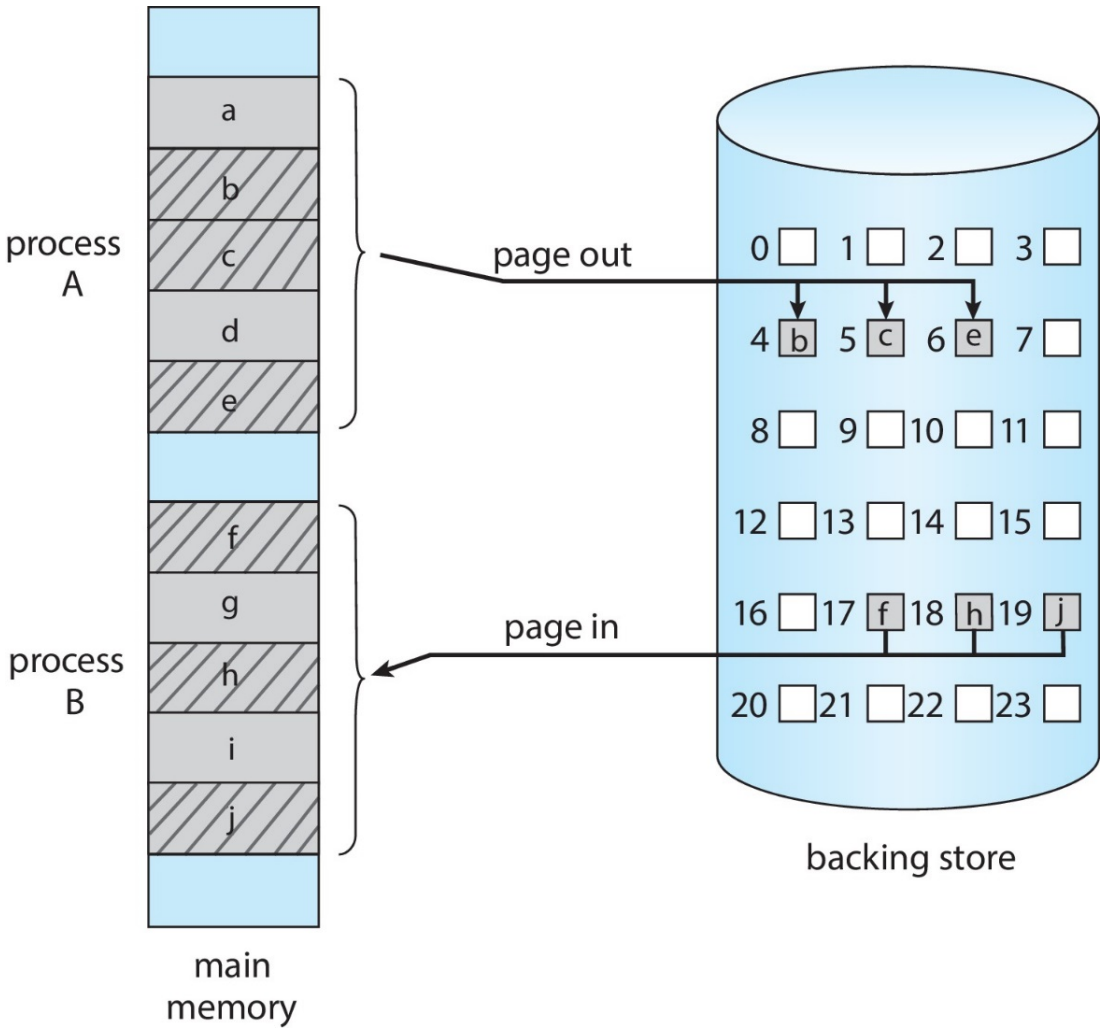


- Um processo pode ser retirado de memória para disco e depois recuperado
  - exemplo 1: ambientes multiprogramados baseado em RR – quando expira um quantum, faz–se *swap out* do processo e *swap in* de outro para a memória libertada; idealmente, a tx. de *swapping* deve ser suficiente para que existam sempre processos disponíveis em memória, prontos a serem executados
  - exemplo 2: esquema *roll out / roll in*, em ambientes multiprogramados com prioridades – processos de menor prioridade são retirados de memória (*roll out*) para que os mais prioritários possam ser carregados (*roll in*)

Standard *Swapping* (2/3)

- Para uma utilização eficiente da CPU, o tempo de execução do processo deve ser substancialmente superior ao tempo consumido em *swapping* (este tempo é apenas uma parte do tempo de comutação de contexto ...)
- A maior parte do tempo de *swapping* deve-se às transferências entre a memória e o disco; o tempo de transferência total é diretamente proporcional à imagem de memória a gravar no disco ou a recuperar do disco
- Para que se possa fazer *swap out* de um processo, o processo não deve estar em execução, nem deve ter operações E/S pendentes (problema: operação E/S assíncrona colocaria dados no espaço do processo substituto; solução: as operações E/S passam a trabalhar com buffers fornecidos pelo SO)
- A associação de endereços em tempo de execução permite fazer o *swap in* de um processo para uma zona diferente da que ocupava antes do *swap out*

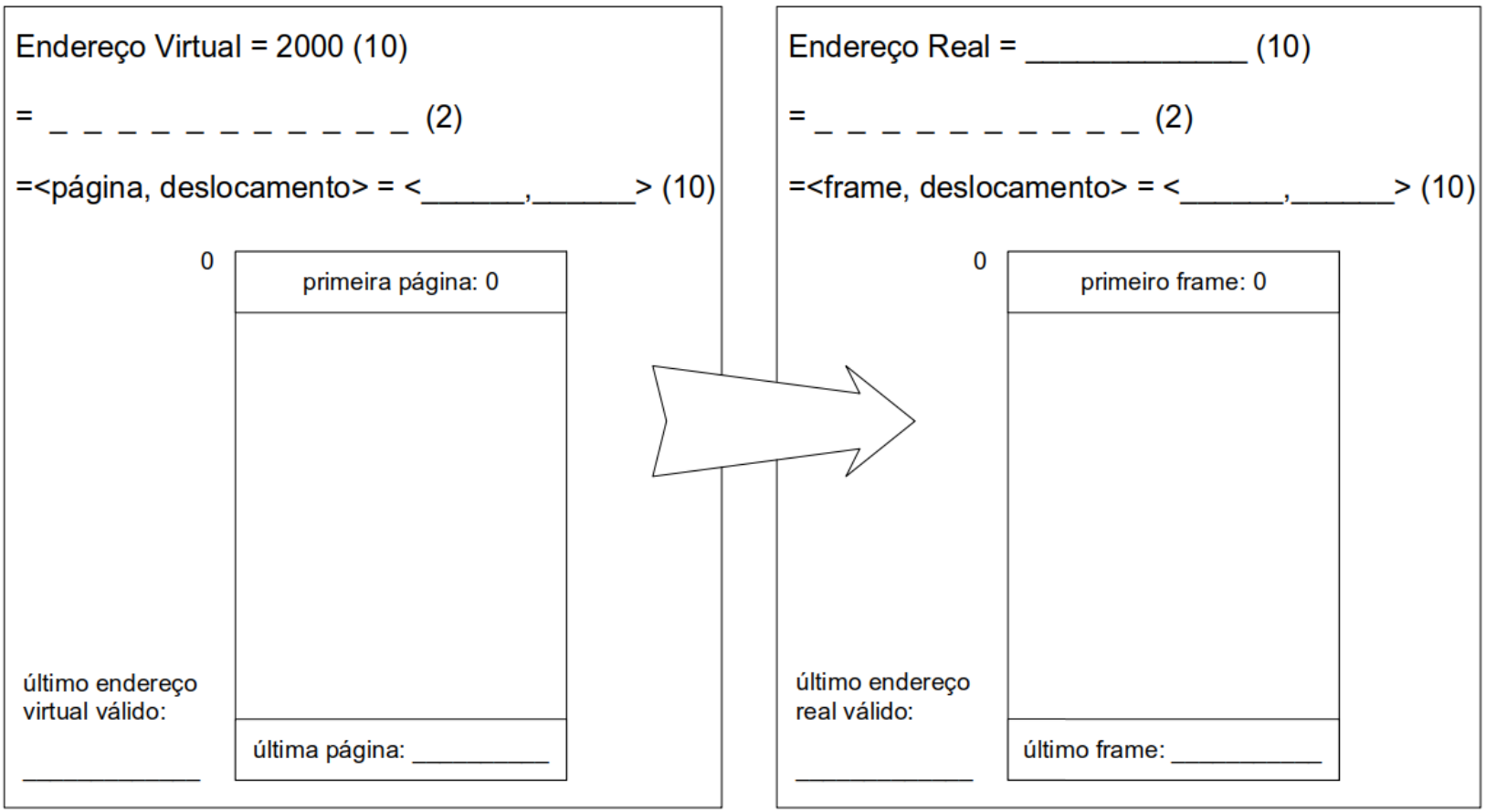
Swapping com Paginação (3/3)



5.7 Exercícios

Exercício 5.1: Paginação

Considere um Espaço de Endereçamento Virtual de 11 bits, paginado a um nível, em que 5 bits especificam a componente deslocamento. O Espaço de Endereçamento Real é de 10 bits. Todas as páginas são válidas e *frame = página % 32*. Complete o esquema abaixo, para a tradução do endereço virtual 2000 no correspondente endereço real.



Exercício 5.2: Paginação

Considere um Espaço de Endereçamento Virtual de 12 bits, paginado a um nível, com 3 bits para a componente página. A Tabela de Páginas referencia as *frames* 2, -1, 0, 1, -1, 3, -1, -1 (-1 significa que a página em causa ainda não tem *frame* atribuída). O Espaço de Endereçamento Real é de 11 bits.

a) Apresente:
- **i)** o número total de endereços virtuais e reais;
- **ii)** o número total de páginas e de *frames*.

b) Considerando a tradução do endereço virtual 2561 (base 10) num endereço real, apresente:
- **i)** o endereço virtual fornecido, expresso na base 2;
- **ii)** as componentes página e deslocamento do endereço virtual dado, na base 10;
- **iii)** a *frame* (expressa na base 10) correspondente ao endereço real pretendido;
- **iv)** o endereço real pretendido, expresso na base 2 e expresso na base 10.

Exercício 5.3: Paginação

Considere um Espaço de Endereçamento Virtual de 13 bits, paginado a um nível, em que 8 bits especificam a componente deslocamento. Todas as páginas são válidas e a correspondência entre uma página e uma *frame* é dada por *frame = página % 4*. O Espaço de Endereçamento Real é de 10 bits.

a) Apresente:

i) o número total de endereços virtuais e reais;

ii) o número total de páginas e de *frames*.

b) Considerando a tradução do endereço virtual 2050 (base 10) num endereço real, apresente:

i) o endereço virtual fornecido, expresso na base 2;

ii) as componentes página e deslocamento do endereço virtual dado, na base 10;

iii) a *frame* (expressa na base 10) correspondente ao endereço real pretendido;

iv) o endereço real pretendido, expresso na base 2 e expresso na base 10.

REFERÊNCIAS

1. "Operating System Concepts, 10th Ed.", Silberschatz & Galvin, Addison-Wesley, 2018: Capítulo 9
2. "Operating System Concepts, 9th Ed.", Silberschatz & Galvin, Addison-Wesley, 2013: Capítulo 8
3. "Fundamentos de Sistemas Operacionais, 6a Ed.", Silberschatz, Galvin & Gagne, LTC, 2004: Capítulo 9
4. Virtual Memory Explained (including Paging): `https://www.youtube.com/watch?v=fGP6VHxqkIM`
5. Page Tables and MMU: How Virtual Memory Actually Works Behind the Scenes: `https://www.youtube.com/watch?v=B6tJxvYBNrU`
6. MultiLevel Page Tables: How Virtual Memory is Optimized: `https://www.youtube.com/watch?v=zNPVmTGt7Ds`
7. What is a Page Fault: `https://www.youtube.com/watch?v=eoFcFqPiPjk`
8. Memory Translation and Segmentation: `https://web.archive.org/web/20180713151104/https://manybutfinite.com/post/memory-translation-and-segmentation/`