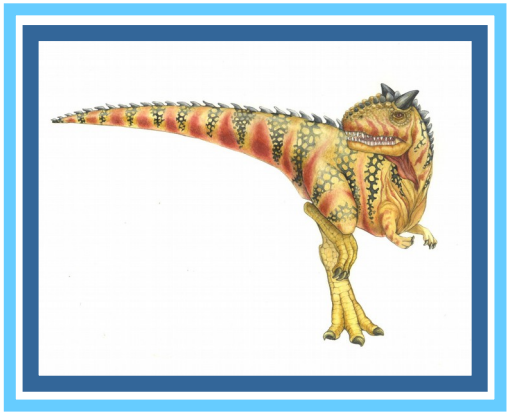
Chapter 8: Memória principal

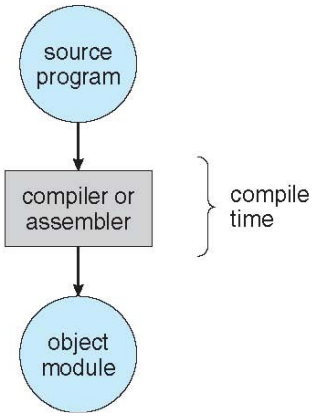


Definição dos endereços de memória

Os endereços das variáveis e trechos de código usados por um programa devem ser definidos em algum momento entre a escrita do código e sua execução pelo processador, que pode ser:

- **Durante a edição**: o programador escolhe a posição de cada uma das variáveis e do código do programa na memória. Esta abordagem normalmente só é usada na programação de sistemas embarcados simples, programados diretamente em linguagem de máquina.

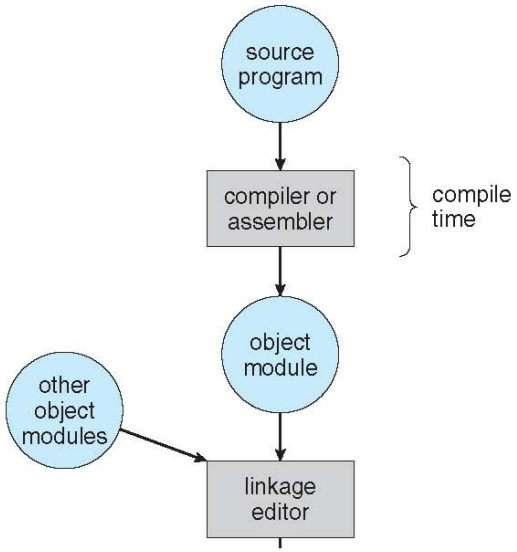
Definição dos endereços de memória



- **Durante a compilação**: *o compilador escolhe as posições das variáveis na memória.*

A escolha pode ser feita usando endereços relativos como, por exemplo, 3.471 bytes após o início do código (nesse caso, o início do código – endereço de referência – deve ser conhecido a priori).

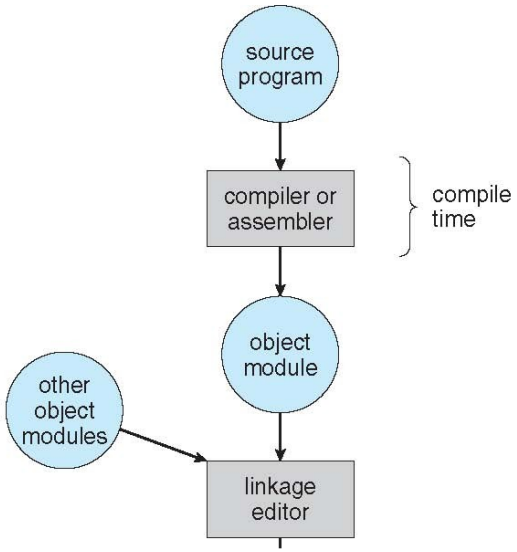
Definição dos endereços de memória



- **Durante a ligação**: o compilador gera símbolos que representam as variáveis mas não define seus endereços finais, gerando um arquivo que contém as instruções em linguagem de máquina e as definições das variáveis utilizadas, denominado arquivo objeto. *O ligador (ou link-editor) então lê todos os arquivos-objeto e as bibliotecas e gera um arquivo-objeto executável, no qual os endereços de todas as variáveis estão corretamente definidos.*

***"Compile time"**: If memory location is known a priori, **absolute code** can be generated; must recompile code if starting location changes.*

Definição dos endereços de memória

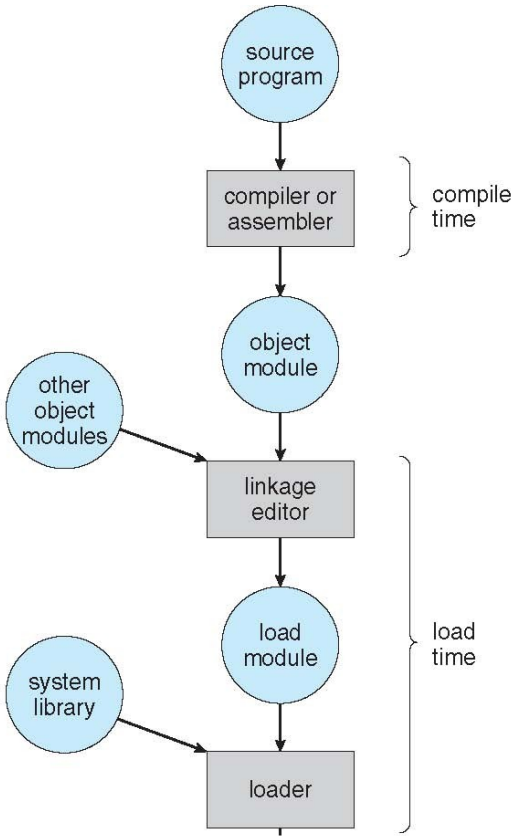


Quando o endereço é definido em tempo de compilação, o Sistema Operacional poderá definir o local de memória onde o programa será carregado?

Durante a execução do programa, o Sistema Operacional pode mudar a posição do programa na memória?

O programa deve ser recompilado caso seja necessário mudar o endereço no qual ele será carregado?

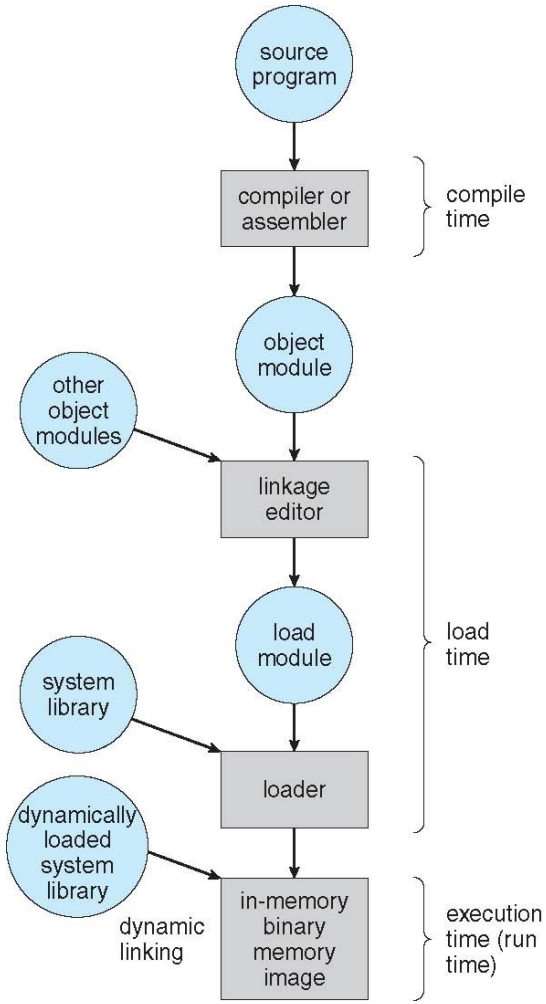
Definição dos endereços de memória



- **Durante a carga** : também é possível definir os endereços de variáveis e de funções durante o carregamento do código em memória. Nesse caso, o carregador (*loader*) é responsável por carregar o código do processo na memória e definir os endereços de memória que devem ser utilizados.

Nesse caso, o compilador deve gerar um código que seja realocável (endereçamento relativo).

Definição dos endereços de memória

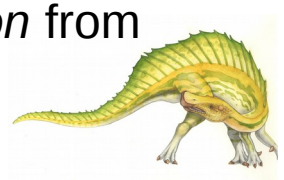


- **Durante a execução**: *os endereços emitidos pelo processador durante a execução do processo são analisados e convertidos nos endereços efetivos a serem acessados na memória real.*

Por exigir a análise e a conversão de cada endereço gerado pelo processador, este método só é viável com o uso de hardware dedicado para esse tratamento.

Esta é a abordagem usada na maioria dos sistemas computacionais atuais.

Execution time: Binding delayed until run time if the *process can be moved during its execution* from one memory segment to another.



Definição dos endereços de memória

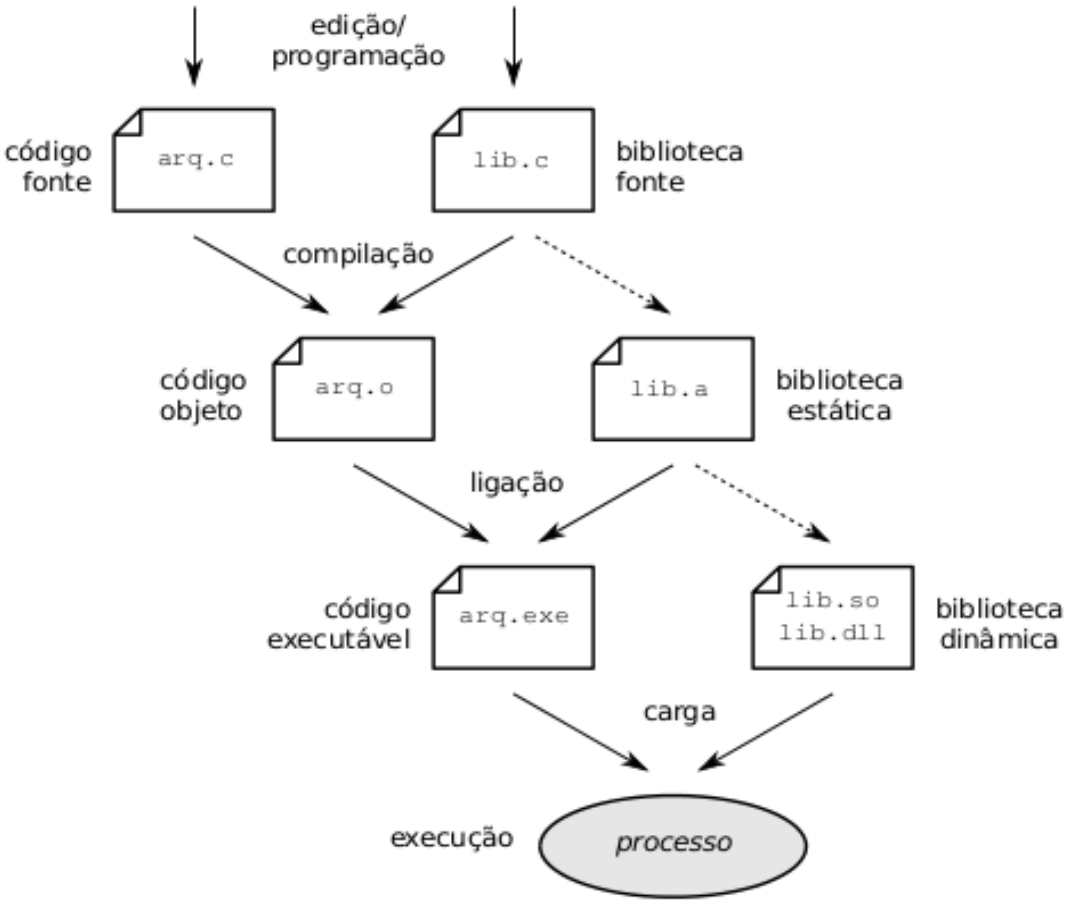


Figura 5.2: Momentos de atribuição de endereços.

Endereços lógicos e físicos – MMU

Os endereços de memória gerados pelo processador durante a execução de um processo são chamados de ***endereços lógicos***. Esses endereços não são necessariamente iguais aos endereços reais das instruções e variáveis na memória real do computador, que são chamados de ***endereços físicos.***

Os endereços lógicos emitidos pelo processador são interceptados por um hardware especial denominado ***Unidade de Gerência de Memória (MMU - Memory Management Unit)***.

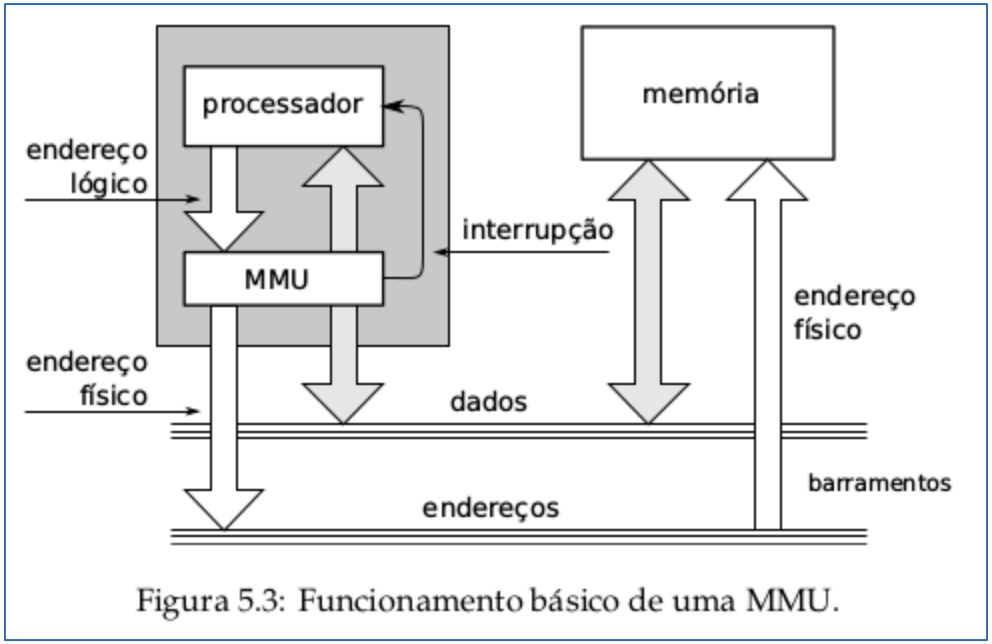


Figura 5.3: Funcionamento básico de uma MMU.

The user program deals with *logical* addresses; ***it never sees the real physical addresses.***

Modelo de memória dos processos

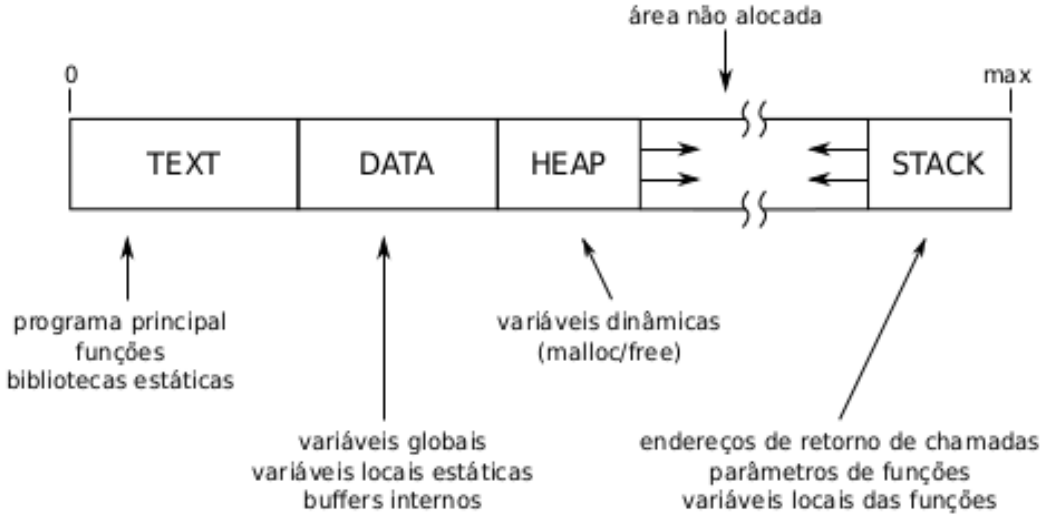


Figura 5.4: Organização da memória de um processo.

Como os processos podem ser alocados na memória?

Estratégias de alocação

Sistemas mono-processos

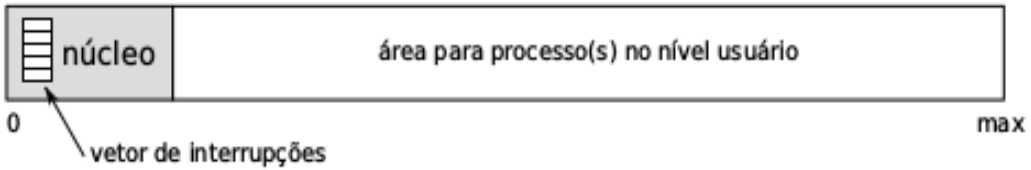
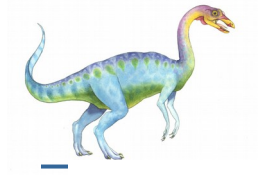


Figura 5.5: Organização da memória do sistema.

Vários processos podem ser carregados na memória para execução simultânea. Nesse caso, o espaço de memória destinado aos processos deve ser dividido entre eles usando uma estratégia que permita eficiência e flexibilidade de uso.



Estratégias de alocação: partições fixas

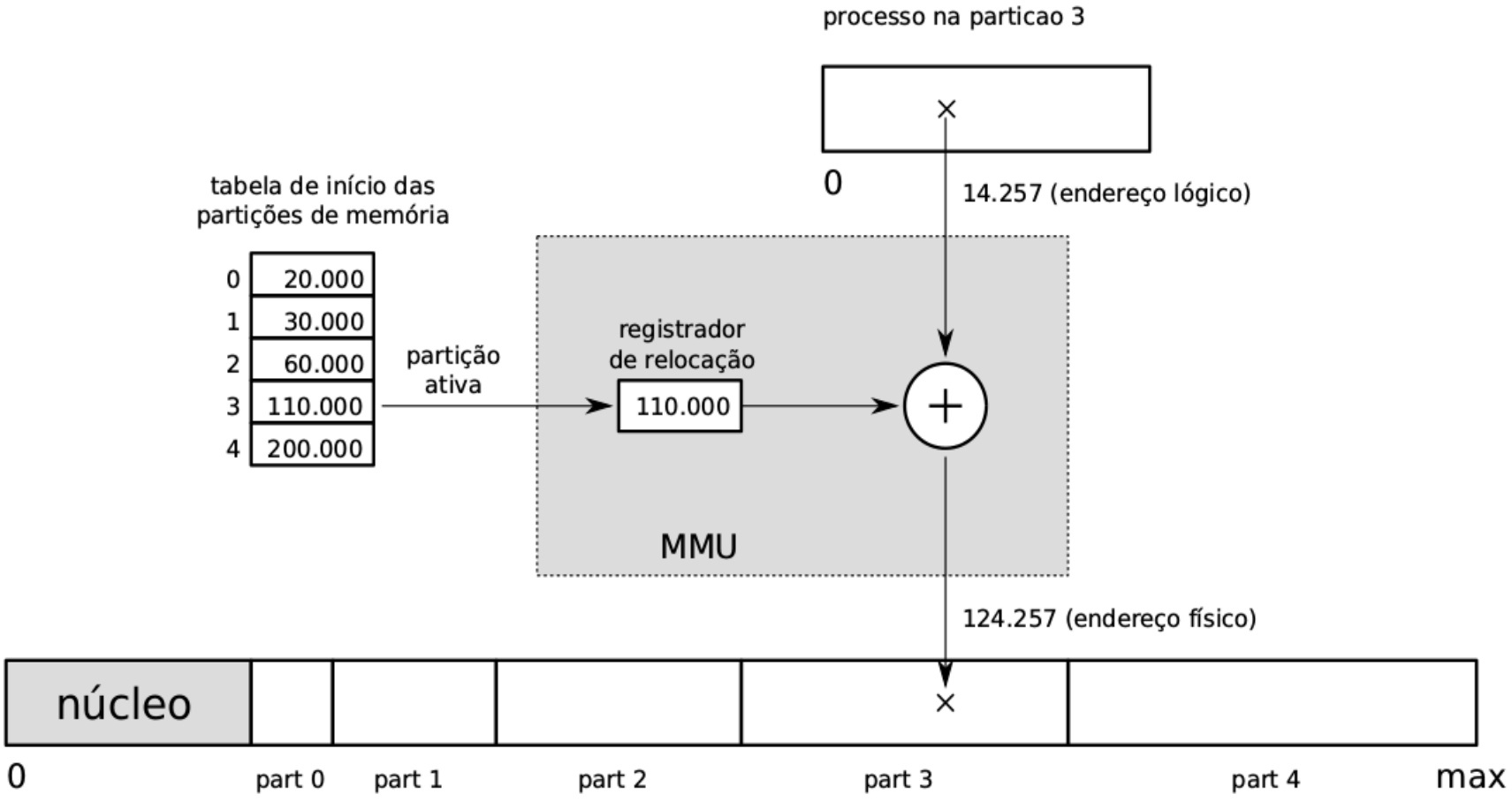


Figura 5.6: Alocação em partições fixas.

Estratégias de alocação: partições fixas

Essa abordagem é *extremamente simples*, todavia sua simplicidade não compensa suas várias *desvantagens*:

• Os processos podem ter tamanhos distintos dos tamanhos das partições, o que implica em áreas de memória sem uso no final de cada partição.

• O número máximo de processos na memória é limitado ao número de partições, mesmo que os processos sejam pequenos.

• Processos maiores que o tamanho da maior partição não poderão ser carregados na memória, mesmo se todas as partições estiverem livres.

Por essas razões, esta estratégia de alocação é pouco usada atualmente;

Estratégias de alocação: alocação contígua

O tamanho da partição é ajustado para se adequar à demanda específica de cada processo. Nesse caso, a MMU deve ser projetada para trabalhar com dois registradores próprios: um *registrador base*, que define o endereço inicial da partição ativa, e um *registrador limite*, que define o tamanho em bytes dessa partição.

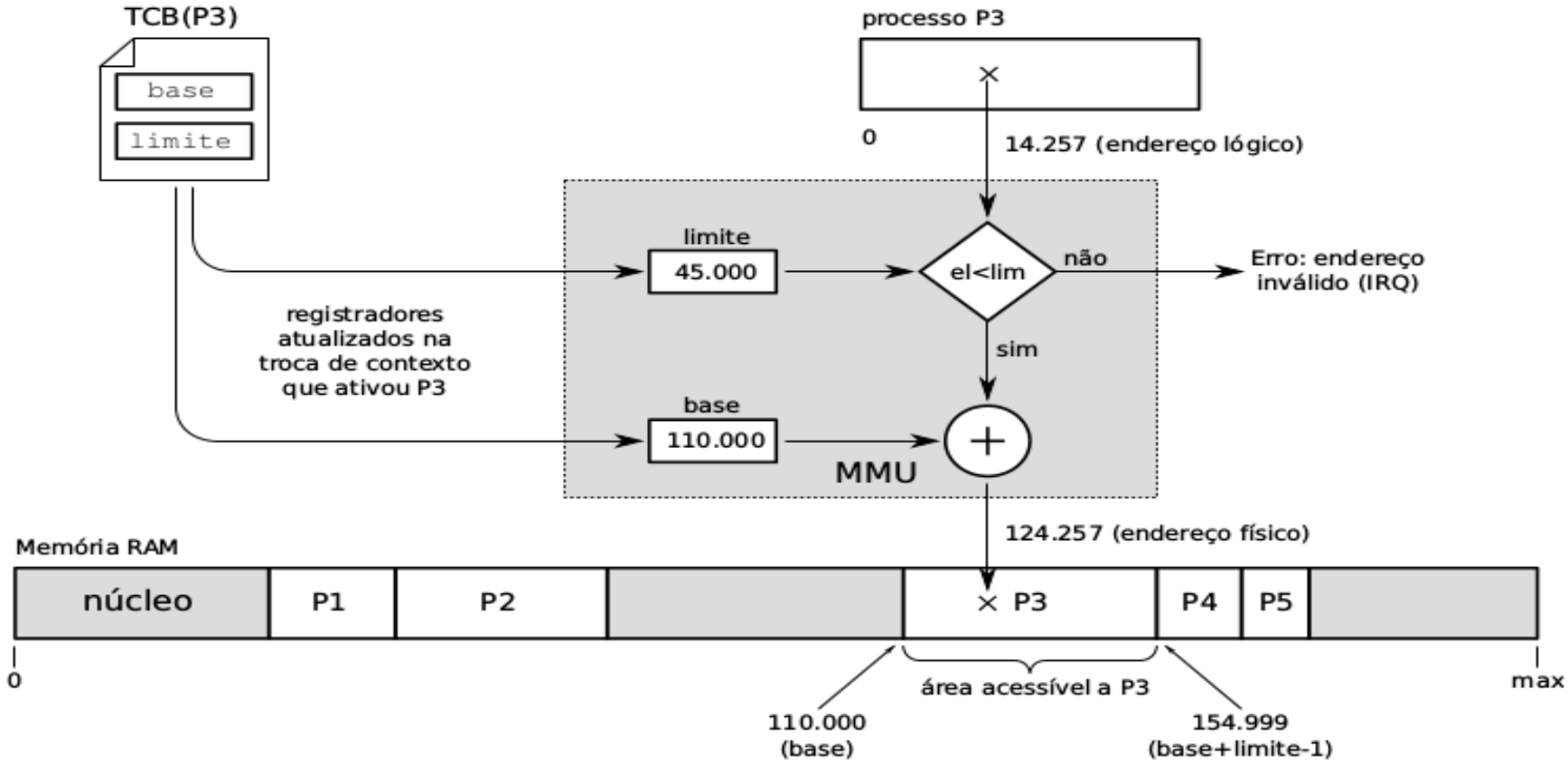


Figura 5.7: Alocação contígua de memória.

Estratégias de alocação: alocação contígua

Os valores dos registradores base e limite da MMU devem ser ajustados pelo despachante (dispatcher) a cada troca de contexto, ou seja, cada vez que o processo ativo é substituído.

Os valores de base e limite para cada processo do sistema devem estar armazenados no respectivo TCB (Task Control Block).

Além de traduzir endereços lógicos nos endereços físicos correspondentes, a ação da MMU propicia a proteção de memória entre os processos.

A maior vantagem da estratégia de alocação contígua é sua simplicidade pois depende apenas de dois registradores e de uma lógica simples para a tradução de endereços.

Todavia, é uma estratégia pouco flexível e está muito sujeita à *fragmentação externa*.

Estratégias de alocação: alocação contígua

Melhor encaixe (best-fit): consiste em escolher a menor área possível que possa atender à solicitação de alocação. Dessa forma, as áreas livres são usadas de forma otimizada, mas eventuais resíduos (sobras) podem ser pequenos demais para ter alguma utilidade.

Pior encaixe (worst-fit): consiste em escolher sempre a maior área livre possível, de forma que os resíduos sejam grandes e possam ser usados em outras alocações.

Primeiro encaixe (first-fit): consiste em escolher a primeira área livre que satisfaça o pedido de alocação; tem como vantagem a rapidez, sobretudo se a lista de áreas livres for muito longa.

Próximo encaixe (next-fit): variante da anterior (first-fit) que consiste em percorrer a lista a partir da última área alocada ou liberada, para que o uso das áreas livres seja distribuído de forma mais homogênea no espaço de memória.

Estratégias de alocação: alocação contígua

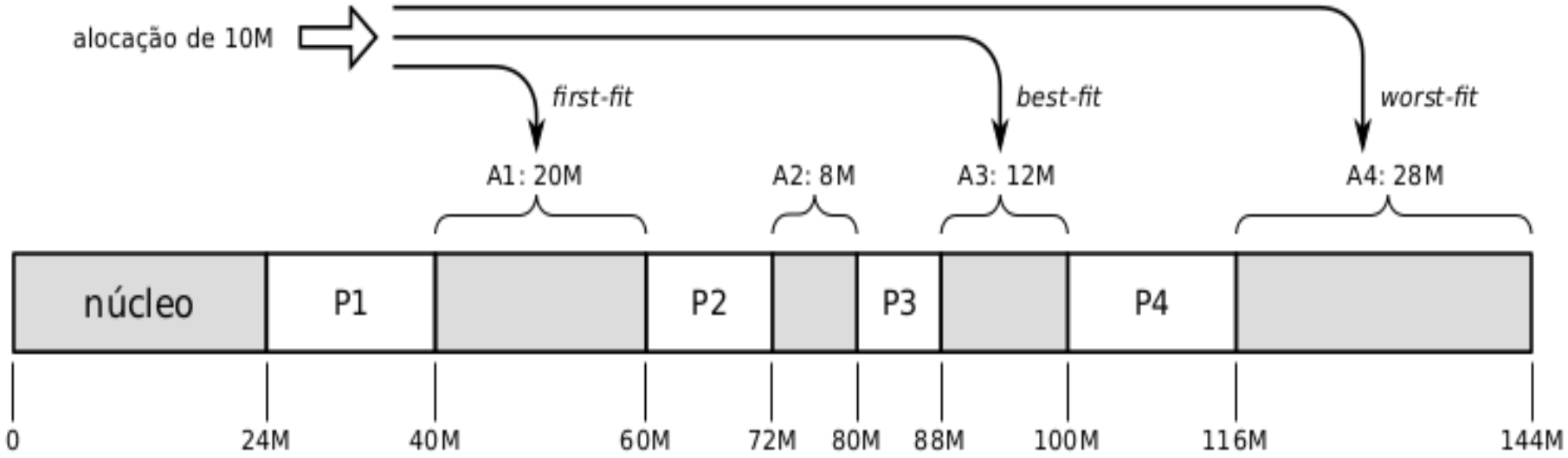


Figura 5.21: Estratégias para minimizar a fragmentação externa.

Fragmentação externa pode ocorrer independente da estratégia de alocação.

Estudos demonstram que o First fit gera 50% de fragmentação (a cada N blocos alocados, 0.5 N são perdidos por causa da fragmentação).

Fragmentação externa

Um processo de 50M não pode ser alocado mesmo havendo espaço disponível.

Solução: *desfragmentação*.

Como as possibilidades de movimentação de processos podem ser muitas, a desfragmentação deve ser tratada como um problema de otimização combinatória, cuja solução ótima pode ser difícil de calcular.

Possibilidades de desframentação

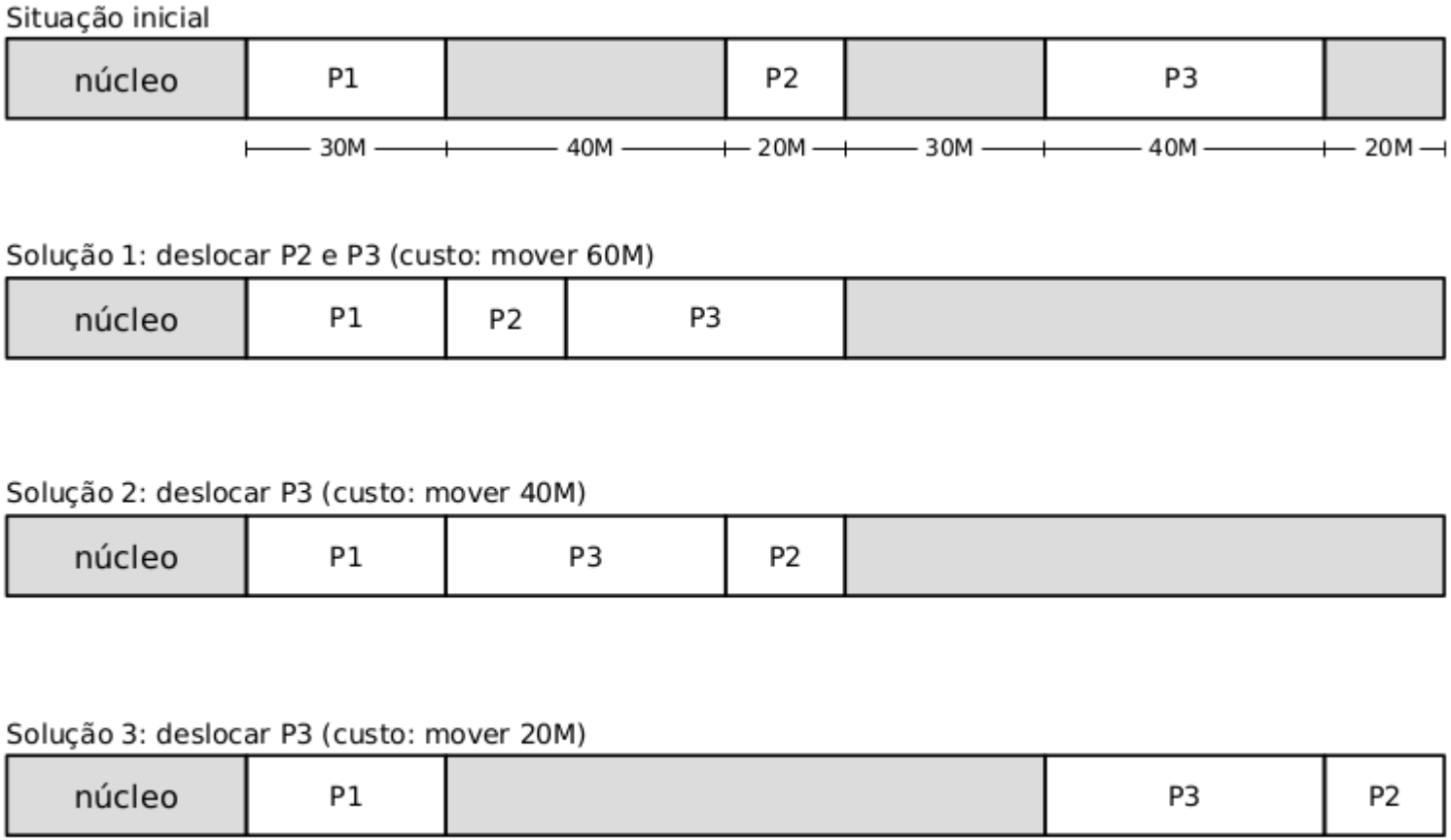


Figura 5.22: Possibilidades de desfragmentação.

A compactação é possível apenas se o código puder ser realocado em tempo de execução.

Fragmentação interna

A área residual de 100 Kbytes deve ser incluída na lista de áreas livres, o que representa um custo de gerência desnecessário.

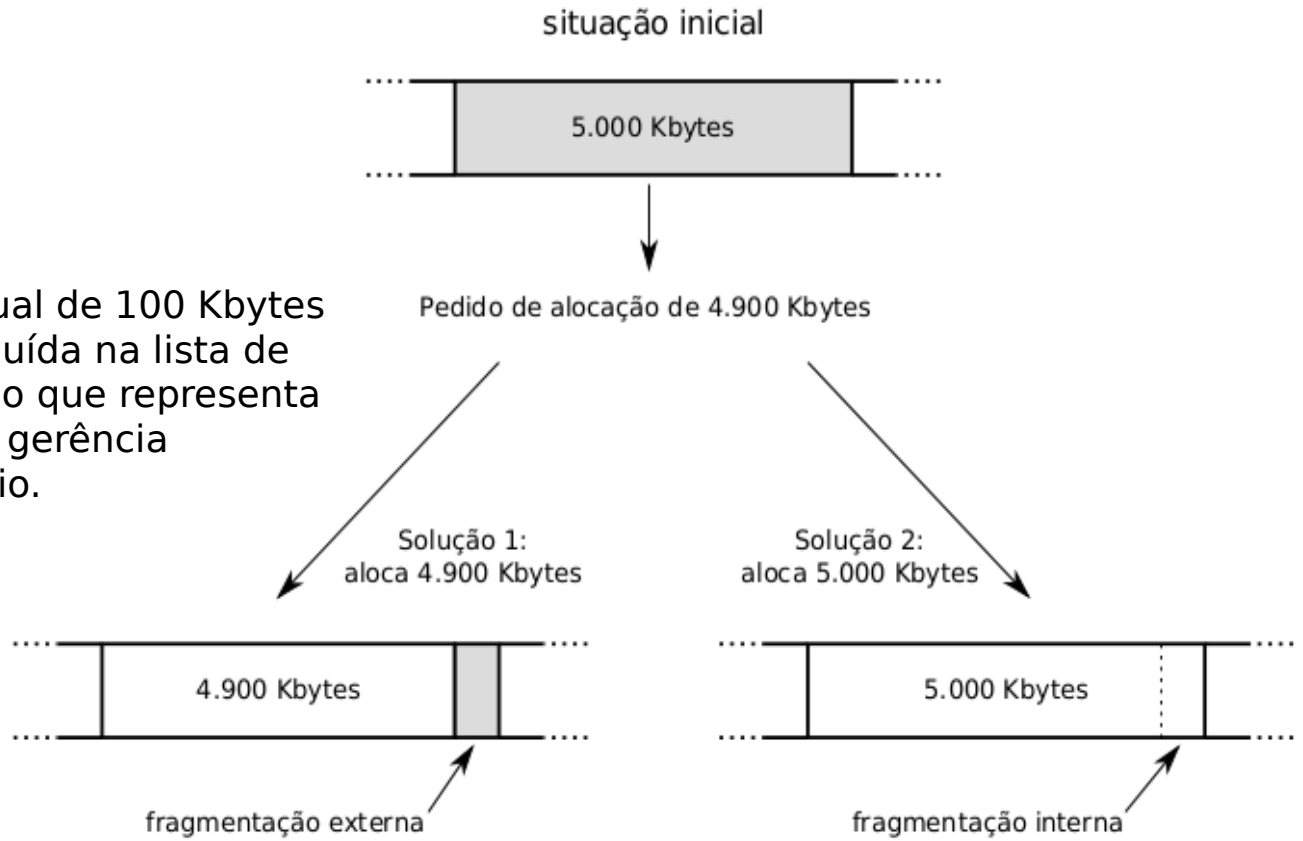
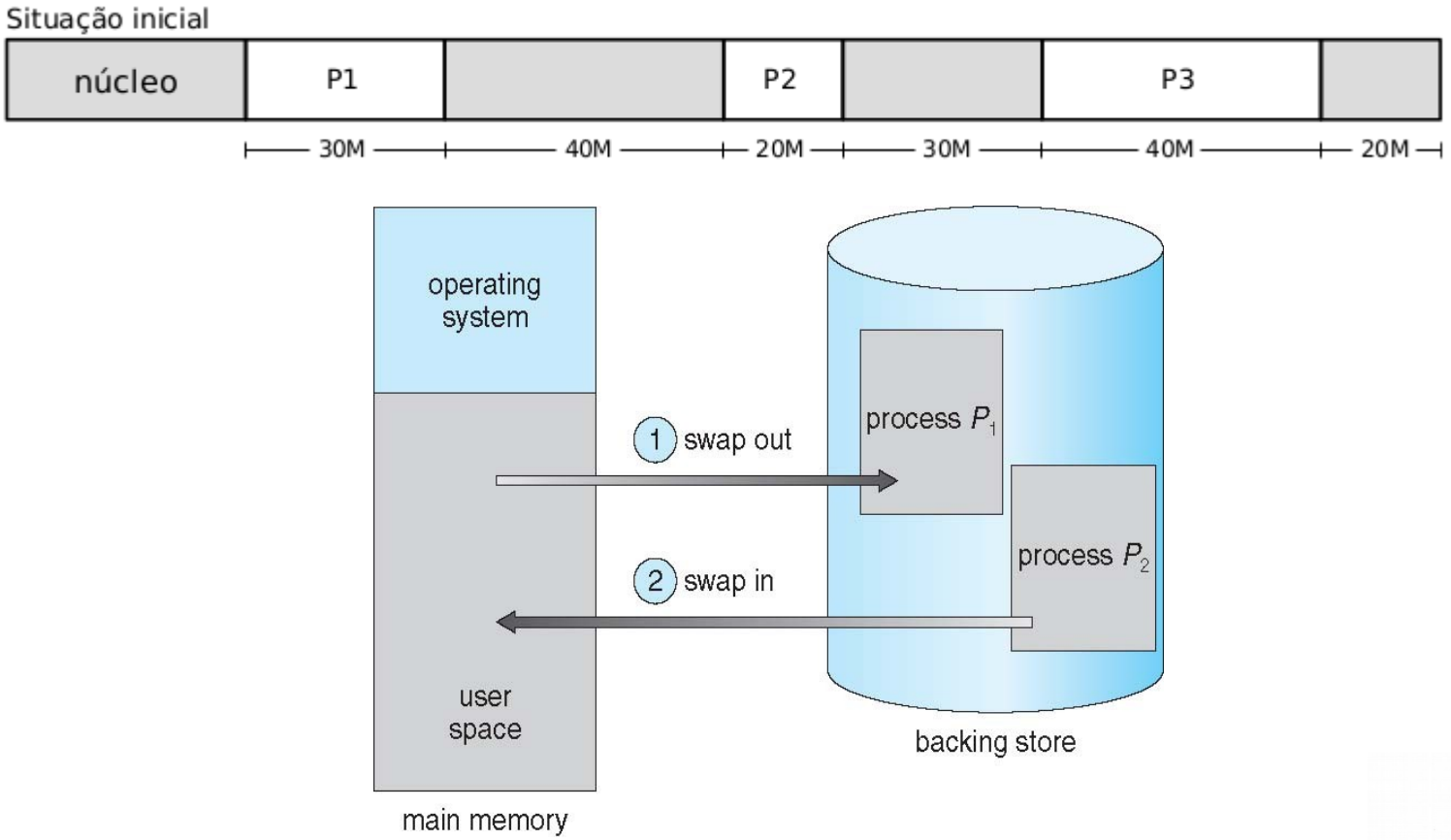


Figura 5.23: Fragmentação interna.

Swapping

- Um processo pode ser transferido (**swapped)** temporariamente da memória para outro dispositivo de armazenamento (HD) e mais tarde voltar para continuar a execução.



Swapping (Cont.)

- ***O processo que foi transferido para outro dispositivo de armazenamento precisa voltar para o mesmo endereço físico de memória?***

Swapping (Cont.)

- ***O processo que foi transferido para outro dispositivo de armazenamento precisa voltar para o mesmo endereço físico de memória?***
  - Depende do método de definição de endereços (address binding method).
  - Além disso, deve ser levado em conta operações de I/O pendentes.
    - Uma solução seria realizar todo I/O utilizando o espaço de endereçamento do Kernel. Este processo é conhecido como *double buffering* (adiciona overhead).
- ***Swapping*** pode ser habilitado e desabilidado de acordo com a necessidade.

Swapping em sistemas móveis

- Tipicamente não suportado
- Ao invés do swap, métodos para liberar memória são utilizados.
  - IOS pede que as aplicações volutariamente liberem memória alocada.
    - “Read-only data” são liberados e recarregados da memória flash se forem solicitados novamente.
  - Android encerra a aplicação se a memória disponível estiver baixa, mas primeiro escreve seu estado na memória flash, caso seja necessário reiniciá-la.
- Ambos SOs suportam paginação (discutido a seguir).

Segmentação

Na **alocação por segmentos** o espaço de memória de um processo é fracionado em *segmentos que podem ser alocados separadamente na memória física*.

Exemplo de segmentos: áreas funcionais básicas do processo (*text, data, stack e heap*) e segmentos específicos como *bibliotecas compartilhadas, vetores, matrizes, pilhas de threads, buffers de entrada/saída*, etc.

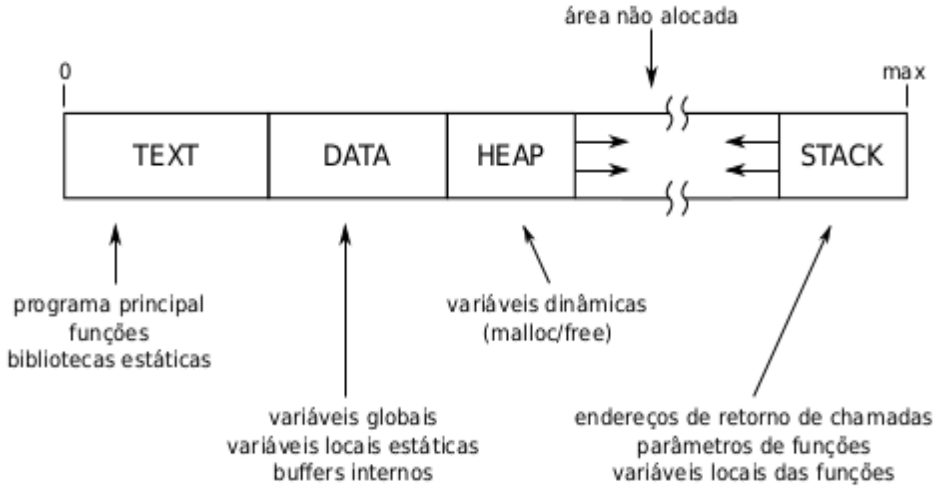
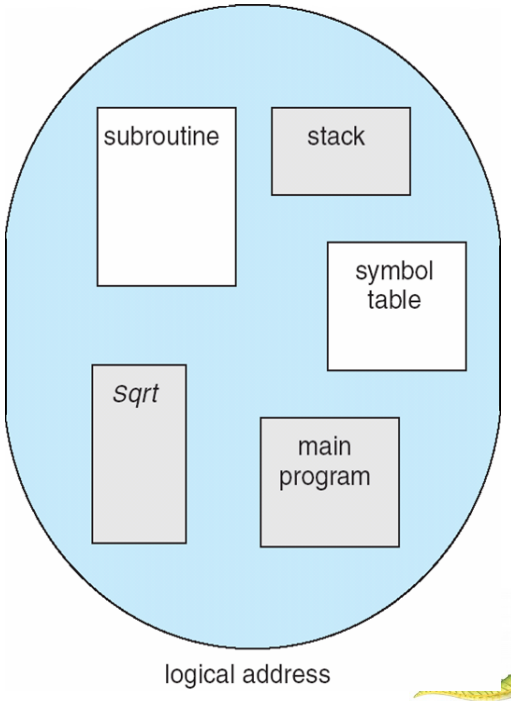
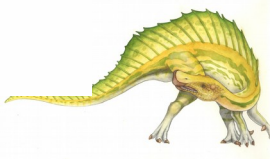


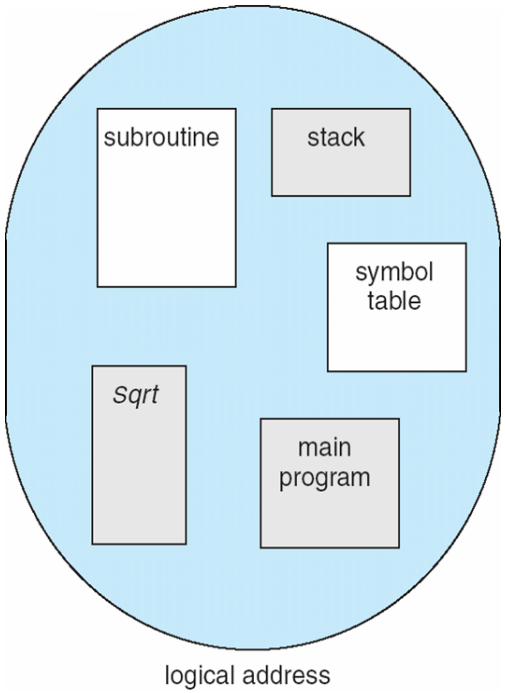
Figura 5.4: Organização da memória de um processo.

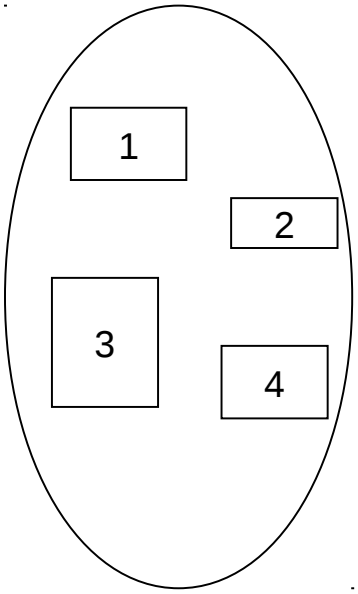


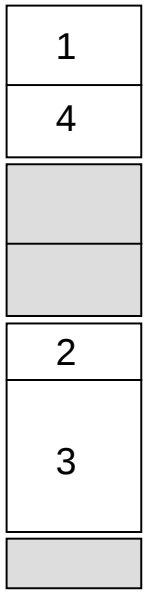


Segmentação

Os segmentos podem estar espalhados pela memória







Segmentação

No modelo de memória alocada por segmentos, os endereços gerados pelos processos devem indicar as posições de memória e os segmentos onde elas se encontram.

Os endereços lógicos são bidimensionais, compostos por pares [segmento:offset]. Os valores de offset variam de 0 (zero) ao tamanho do segmento.

Cada segmento terá seus próprios valores de **base** e **limite**, o que leva à necessidade de definir uma tabela de segmentos para cada processo do sistema.

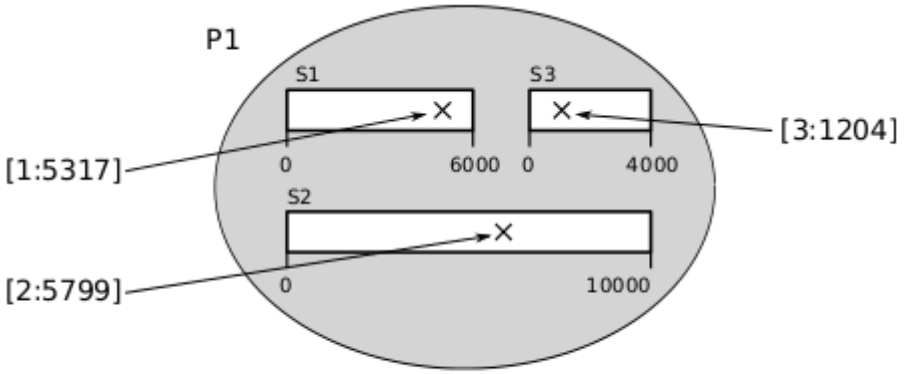


Figura 5.9: Endereços lógicos em segmentos.

Segmentação

"ST reg" indica o registrador que aponta para a tabela de segmentos ativa.

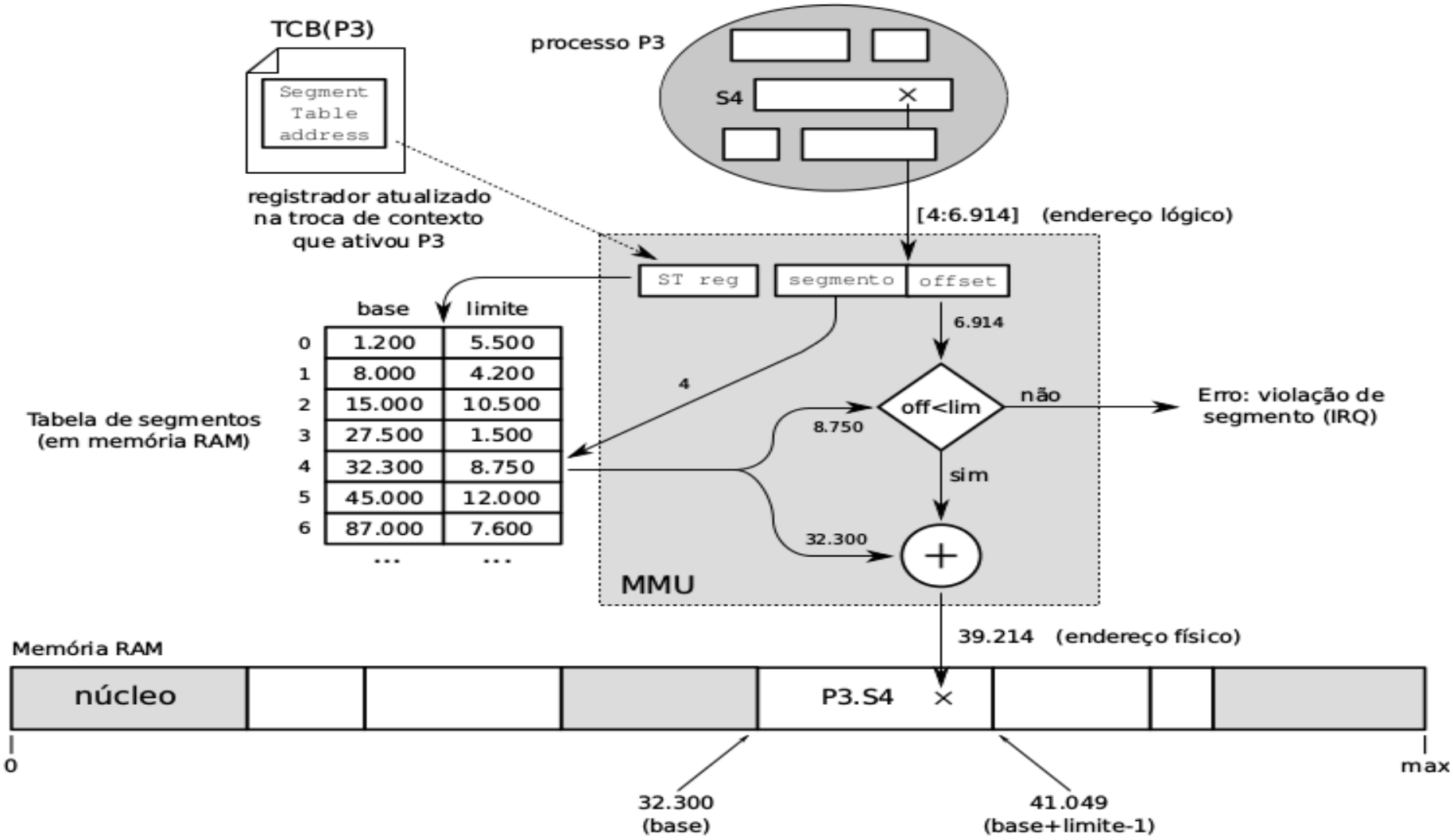


Figura 5.10: Tradução de endereços em memória alocada por segmentos.

Segmentação

Cabe ao ***compilador*** *colocar os diversos trechos do código-fonte de cada programa em segmentos separados*. Ele pode, por exemplo, colocar cada vetor ou matriz em um segmento próprio.

Caso o número de segmentos usados por cada processo seja pequeno, a tabela pode residir em registradores especializados do processador.

Por outro lado, caso o número de segmentos por processo seja elevado, será necessário alocar as tabelas na memória RAM.

O processador 80.386 usa **duas tabelas em RAM**: a LDT (**Local Descriptor Table**), que define os segmentos locais (exclusivos) de cada processo, e a GDT (**Global Descriptor Table**), usada para descrever segmentos globais que podem ser compartilhados entre processos distintos.

Segmentação

Como as tabelas de segmentos normalmente se encontram na memória principal, esses acessos têm um **custo significativo**.

Para cada acesso à memória são necessárias pelo menos **duas leituras adicionais** na memória para ler os valores de base e limite, o que torna cada acesso à memória **três vezes mais lento**.

Segmentação

Para contornar esse problema, os processadores definem alguns **registradores de segmentos**, que permitem armazenar os valores de base e limite dos segmentos mais usados pelo processo ativo.

- **CS: Code Segment**, indica o segmento onde se encontra o código atualmente em execução

- **SS: Stack Segment**, indica o segmento onde se encontra a pilha em uso pelo processo atual; caso o processo tenha várias threads, este registrador deve ser ajustado a cada troca de contexto entre threads.

- **DS, ES, FS e GS**: Data Segments, indicam quatro segmentos com dados usados pelo processo atual.

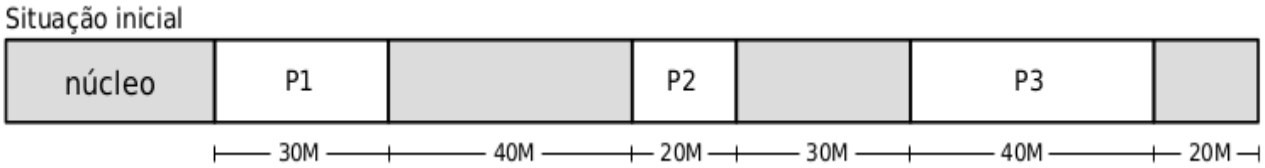
O conteúdo desses registradores é preservado no **TCB (Task Control Block)** de cada processo a cada troca de contexto.

Segmentação

Existe problema de fragmentação externa quando a memória é alocada por segmentos?

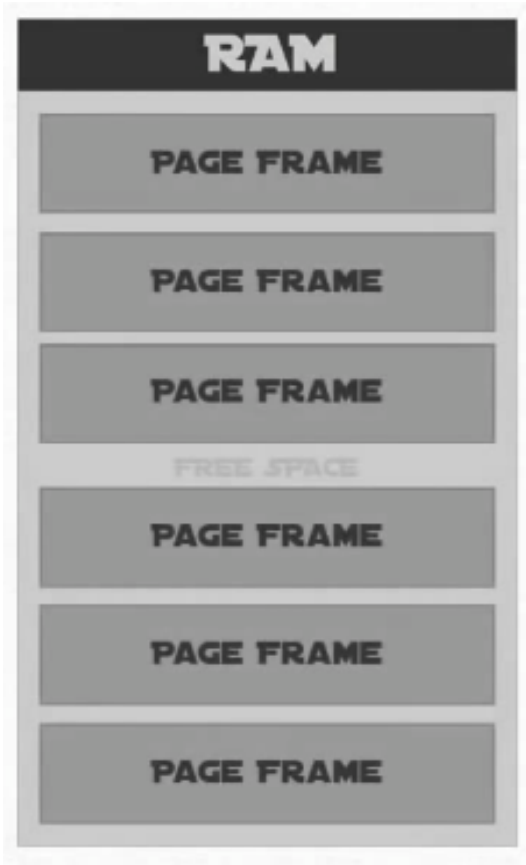
Faz sentido utilizar as estratégias de alocação *best fit, worst fit, first fit e next fit* quando se utiliza segmentação?

Qual técnica de alocação pode resolver o problema de fragmentação externa?



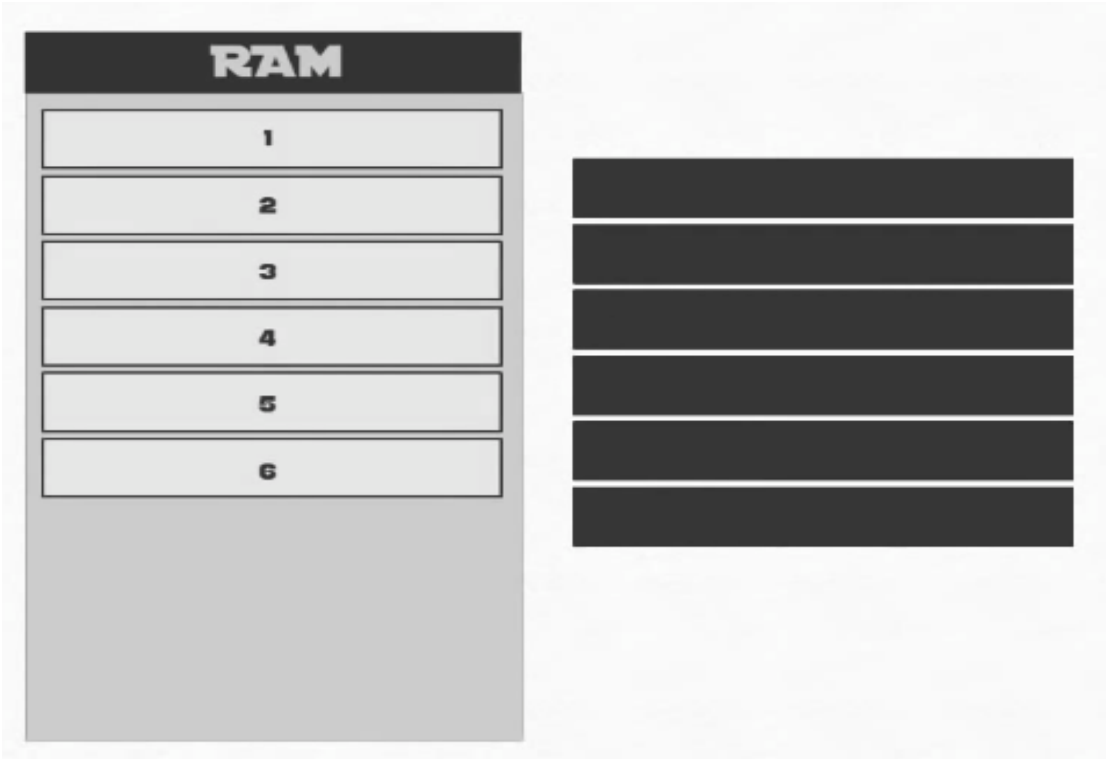
Paginação

Divide a memória em seções (partes) denomidas quadros de página (**page frames**).



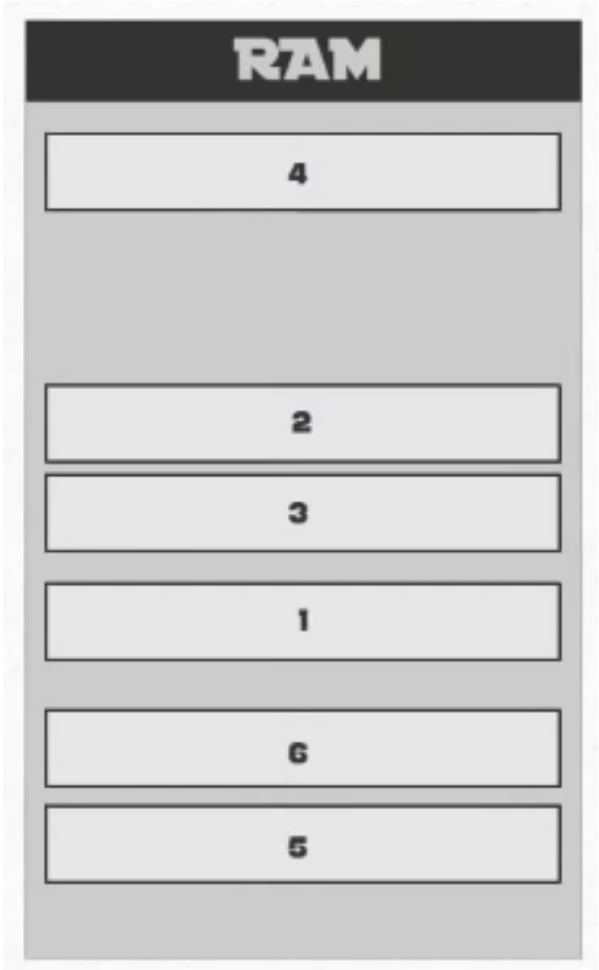
Paginação

Divide o processo e aloca as partes nos quadros de memória (quadros de páginas).



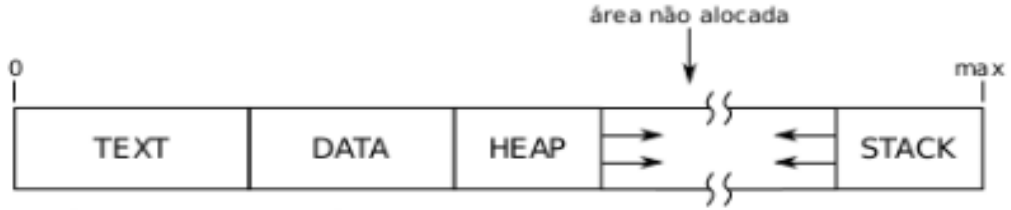
Paginação

A alocação não precisa ser feita em *frames* consecutivos.



Paginação: gerência da memória

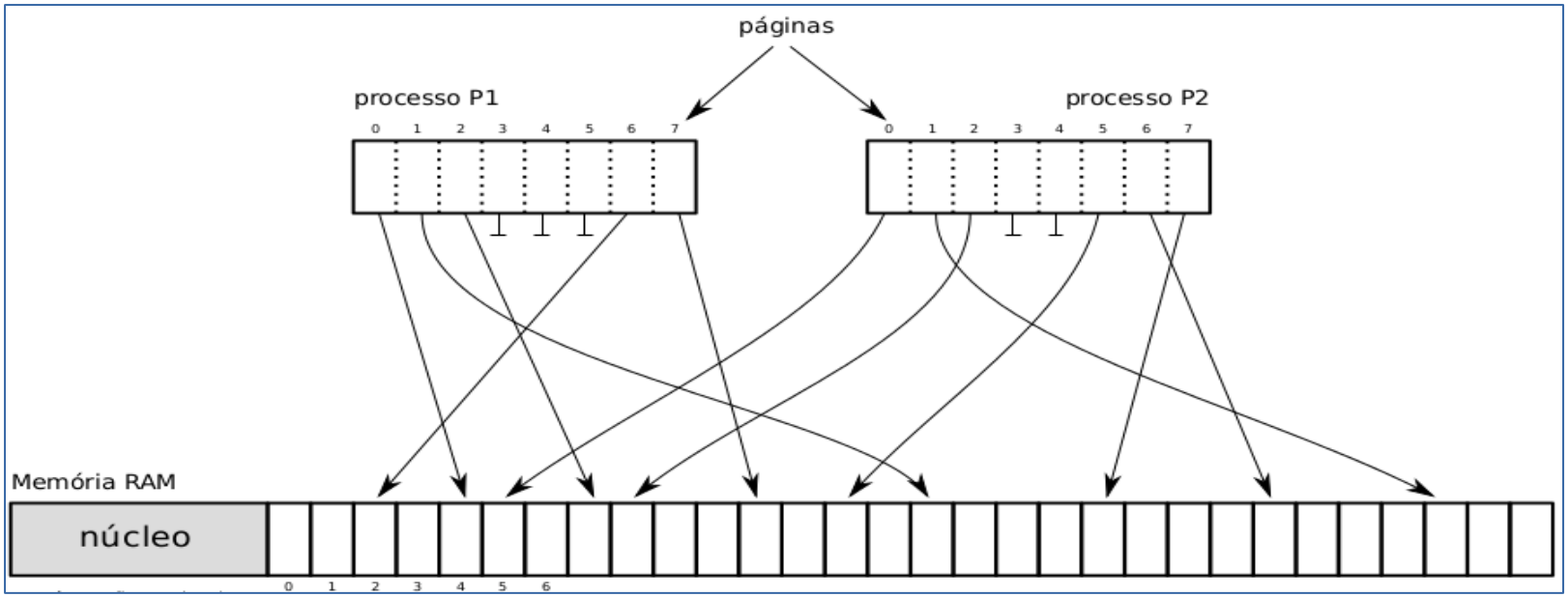
- O compilador gera um código com endereçamento lógico linear.



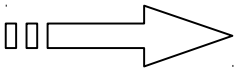
No momento do carregamento, o processo é dividido em páginas. A divisão é feita de acordo com o tamanho do frame (page frame) (Ex. 4KB).

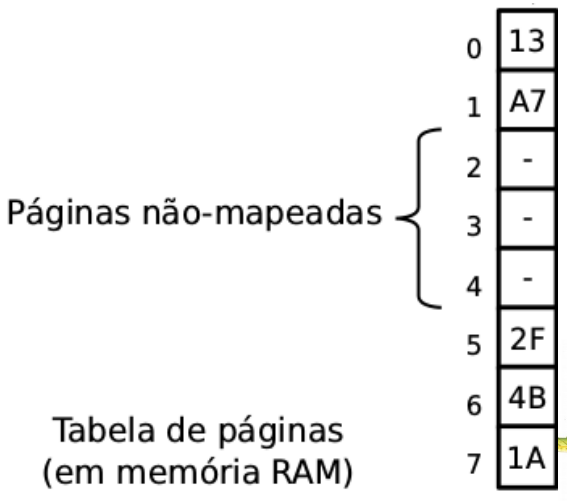
Paginação: gerência da memória

- As páginas são carregadas nos frames de memória.



- O SO gerencia o mapeamento página → frame.





Paginação: gerência da memória

- O SO então mantém a tabela de páginas para cada processo.

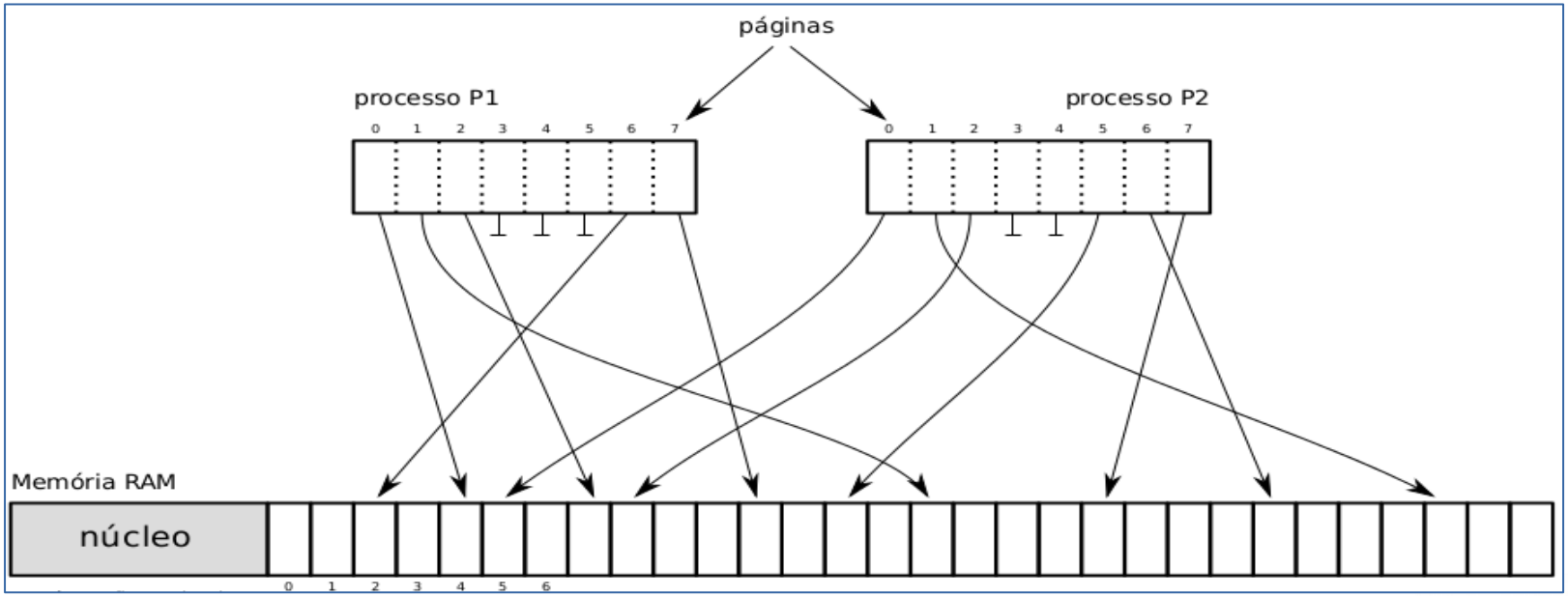
- Cada processo possui o endereço de memória onde está a sua tabela de páginas.

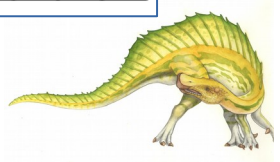
```
struct TCB{
    …
    PageTableAddress
    ...
}
```

- A cada troca de contexto, o **`PageTableAddress`** pode ser carregado para um registrador do processador.

Paginação: gerência da memória

Além de manter a tabela de páginas de cada processo, o SO precisa gerenciar a ocupação dos quadros na memória (livre ou ocupado).





Paginação: gerência da memória

- De uma maneira geral, o Kernel do SO gerencia:
  - a própria tabela de páginas de cada processo.
  - o endereço da tabela de páginas de cada processo, o qual pode ser gravado no TCB.
  - os quadros livres e ocupados da memória.

JOB TABLE	
JOB SIZE	PMT Location
400	3096
200	3100
500	3150

PAGE MAP TABLE	
PAGE NUMBER	PAGE FRAME NUMBER
0	5
2	4
1	3

MEMORY MAP TABLE	
PAGE FRAME	STATE
4	FREE
3	FREE
5	BUSY

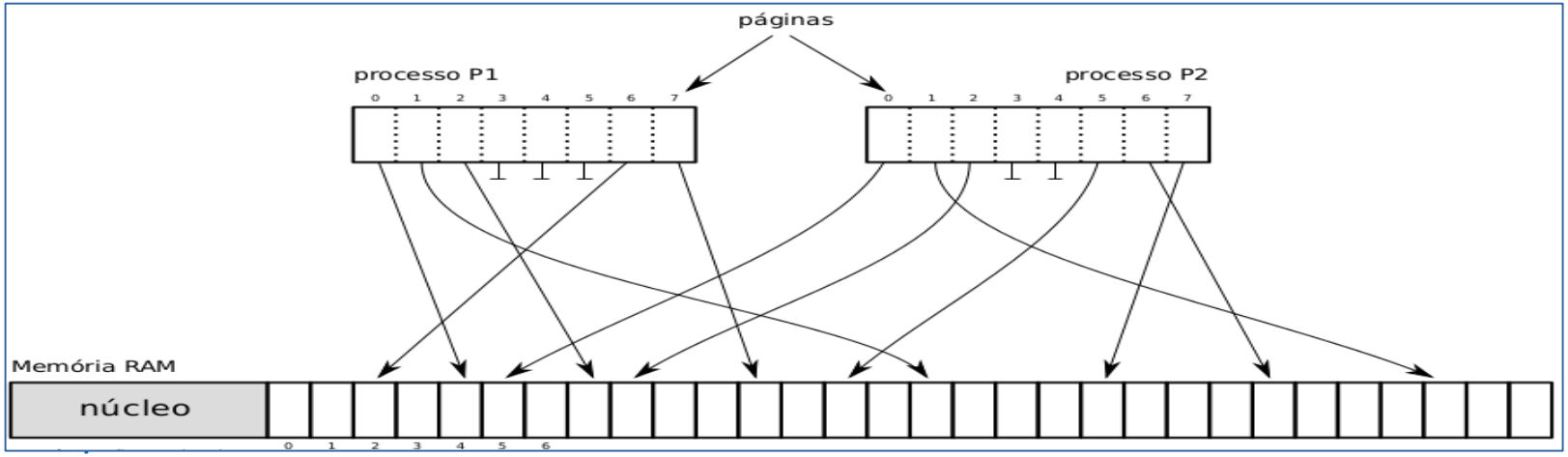
Paginação: gerência da memória

Resumindo

Na alocação de memória por páginas, ou alocação paginada, o espaço de **endereçamento lógico** dos processos é mantido linear e **unidimensional** (ao contrário da alocação por segmentos, que usa endereços bidimensionais).

Internamente, e de forma transparente aos processos, **o espaço de endereços lógicos é dividido em pequenos blocos de mesmo tamanho, denominados páginas.**

O espaço de memória física destinado aos processos também é dividido em blocos de mesmo tamanho que as páginas, denominados **quadros**.



Paginação: gerência da memória

Resumindo

O mapeamento entre as páginas de um processo e os quadros correspondentes na memória física é feita através de uma **tabela de páginas (page table)**, na qual cada entrada corresponde a uma página e contém o número do quadro onde ela se encontra.

Cada processo possui sua **própria tabela de páginas**.

A tabela de páginas ativa, que corresponde ao processo em execução no momento, **é referenciada por um registrador do processador denominado PTBR** – Page Table Base Register. A cada troca de contexto, esse registrador deve ser atualizado com o endereço da tabela de páginas do novo processo ativo.

Paginação: gerência da memória

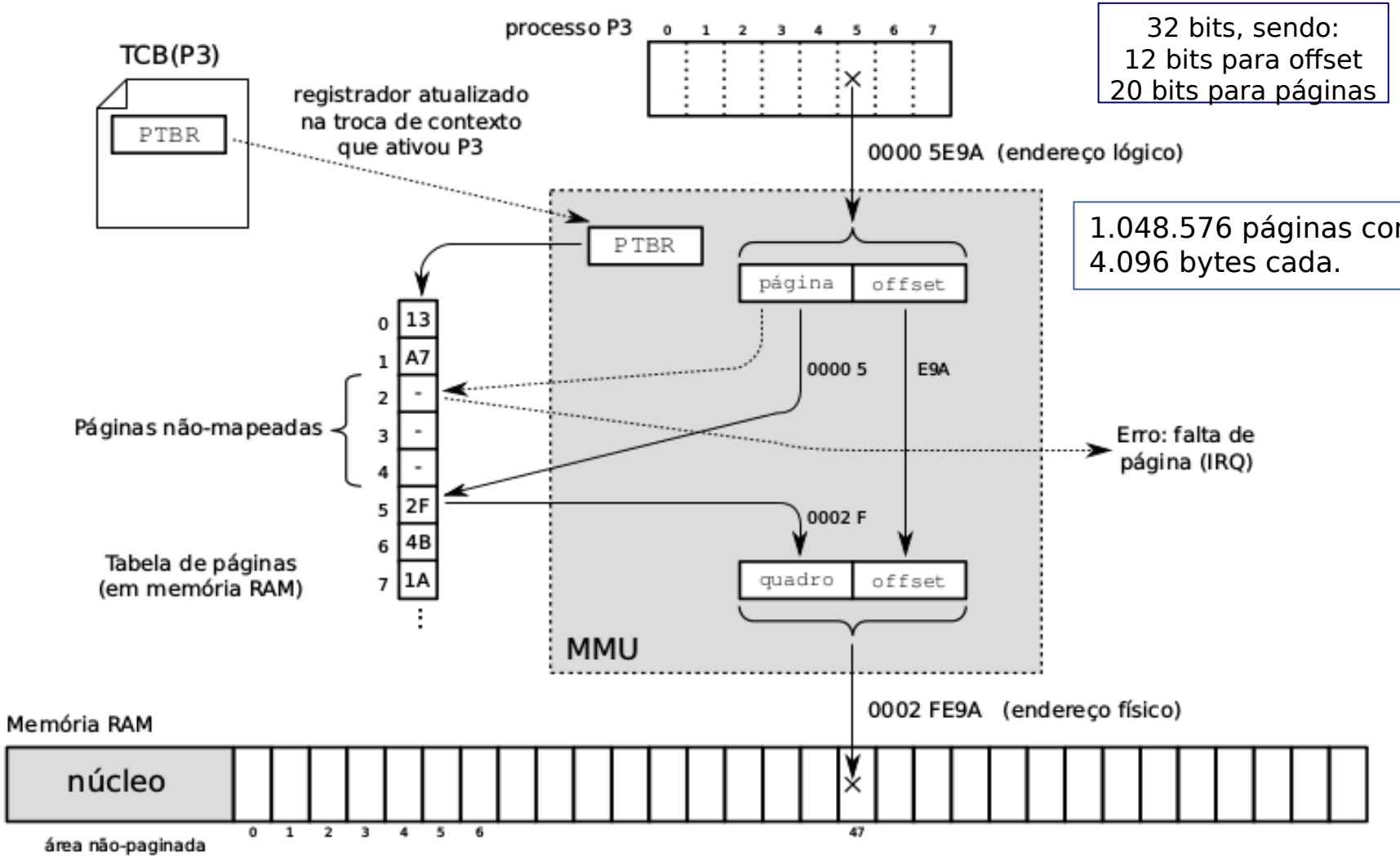
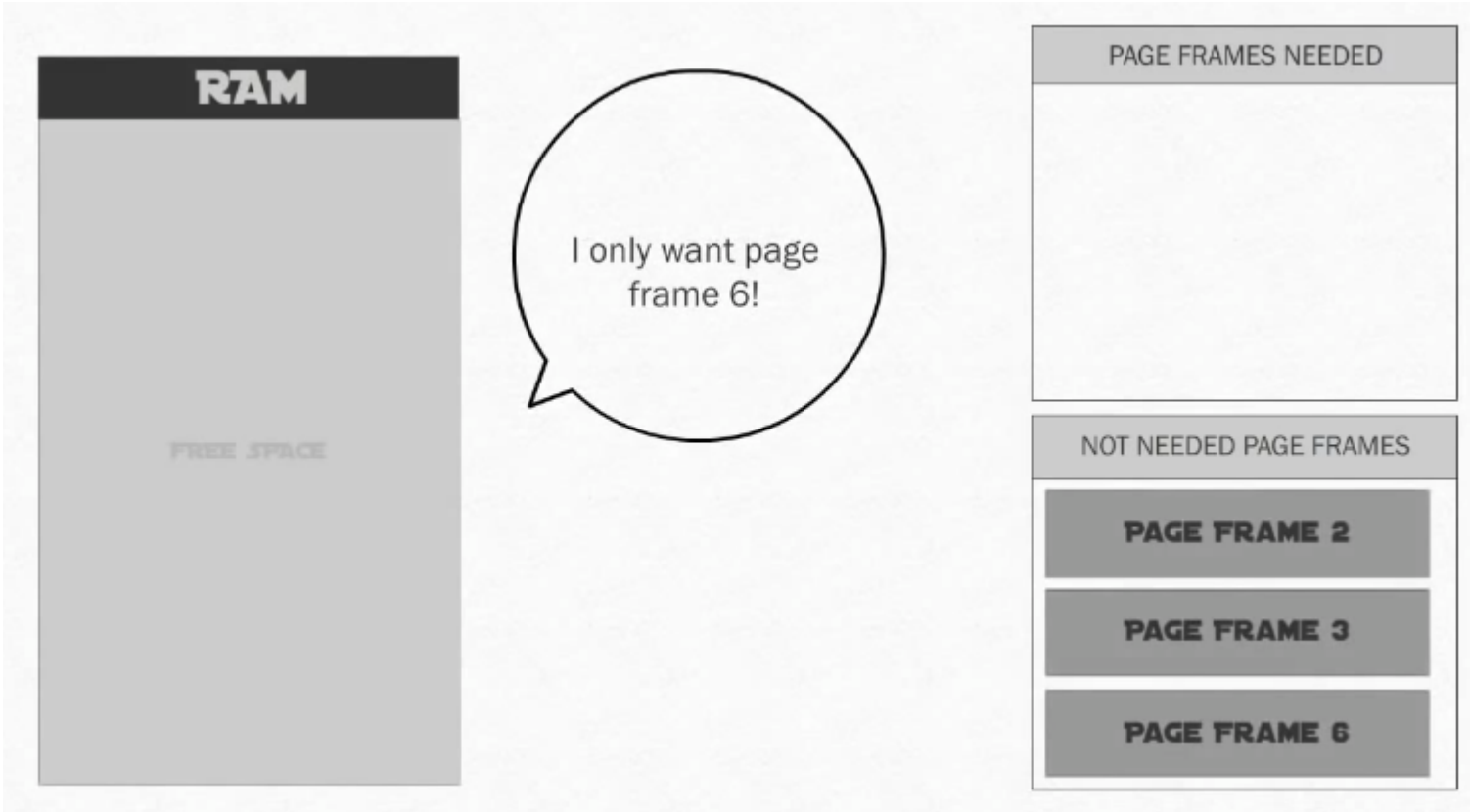
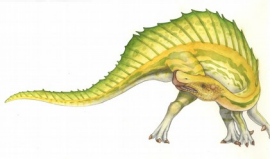


Figura 5.12: Tradução de endereços usando paginação.

Paginação: entradas da tabela de páginas

Paginação por demanda: somente carrega a página quando necessário (**bit de presença**).





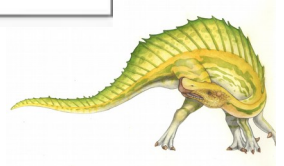
Paginação: entradas da tabela de páginas

Presença: indica se a página está presente na memória.

Proteção: bits indicando os direitos de acesso do processo à página (basicamente **leitura, escrita e/ou execução**);

Referência: indica se a página foi referenciada (acessada) recentemente. Usado pelos algoritmos de memória virtual.

PAGE MAP TABLE				
PAGE NUMBER	STATUS	MODIFIED	REFERENCED	PAGE FRAME NUMBER
0	Y	0	1	5
2	Y	0	0	4
1	N	1	1	3

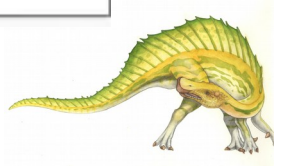


Paginação: entradas da tabela de páginas

Modificação: também chamado de dirty bit. Indica se a página foi modificada após ser carregada na memória. Usado pelos algoritmos de memória virtual.

Outras informações: se é uma página de usuário ou de sistema, se a página pode ser movida para disco, o tamanho da página, etc.

PAGE MAP TABLE				
PAGE NUMBER	STATUS	MODIFIED	REFERENCED	PAGE FRAME NUMBER
0	Y	0	1	5
2	Y	0	0	4
1	N	1	1	3



Paginação

- A paginação evita fragmentação externa?
- Divide a memória físca em blocos de tamanho fixo chamados **frames**
  - O tamanho de um frame é uma potência de 2, normalmente entre 512 bytes e 16 Mbytes
  - **`$ getconf PAGESIZE`**

    `4096`
- O gerente de memória deve manter controle de todos frames livres e alocados.
- *Caso não haja paginação por demanda, o SO precisa encontrar N frames livres para alocar as páginas do processo.*
- A paginação pode apresentar fragmentação interna?

Paginação: fragmentação interna

- Calculando a fragmentação interna:
  - Tamanho da página = 2,048 bytes
  - Tamanho do processo = 72,766 bytes
  - 35 páginas + 1,086 bytes
  - Fragmentação interna: 2,048 - 1,086 = 962 bytes
  - Pior caso de fragmentação interna: 1 frame – 1 byte
  - Em média, fragmentação interna = 0.5 do tamanho do frame.

Então é desejável que os frames sejam pequenos?

O problema de frames pequenos é que o número de entrada na tabela de páginas aumenta.

A tendência é que o tamanho do frame aumente como tempo. Alguns sistemas suportam dois tamanhos de páginas/frames (Solaris – 8 KB and 4 MB)

Páginas compartilhadas

- **Páginas compartilhadas**
  - As instâncias de um programa (N processos) podem compartilhar as páginas de apenas leitura que contêm a área de código (semelhante a múltiplas threads compartilhando a mesma área de código).
  - O compartilhamento de páginas também é útil na comunicação entre processos (memória compartilhada). Nestes casos, as páginas são “read-write”

Tabelas multiníveis

Considerando: 32 bits → 12 bits offset (4 KB)
20 bits páginas (1.048.576 páginas)

Espaço de endereçamento lógico de um processo de 1.048.576 páginas. Se considerarmos que cada entrada da tabela de página ocupa 4 bytes, então a tabela de páginas completa ocupa (2^20 * 4) = 4MB.

No caso de **processos pequenos**, com muitas páginas não mapeadas, uma tabela de páginas linear **ocupará mais espaço na memória que o próprio processo**!!

Tabelas de páginas multiníveis: uma tabela de páginas de primeiro nível contém ponteiros para tabelas de páginas de segundo nível, e assim por diante, até chegar à tabela que contém os números dos quadros desejados.

Tabelas multiníveis

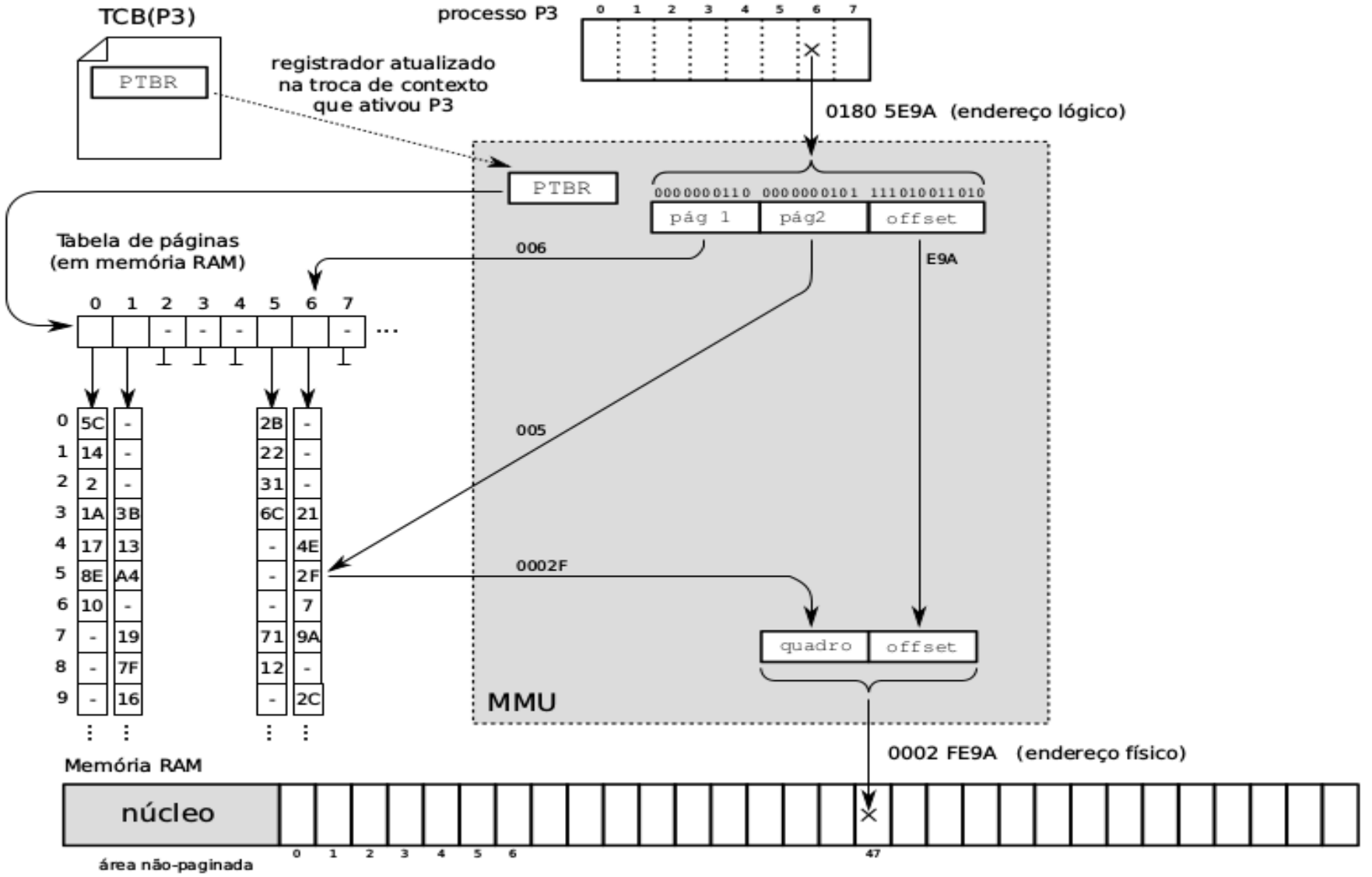


Figura 5.14: Tabela de página multi-nível.

Tabelas multiníveis

Linear:
32 bits → 12 bits offset (4 KB)
20 bits páginas (1.048.576 páginas)
1.048.576 * 4 bytes = 4MB

Multinível:

2^10 * 4= 4KB primeiro nível.

+

2^10 * 4= 4KB segundo nível (para cada alocação no segundo nível).

Tabelas de segundo nível são alocadas quando necessário.

Na situação limite, onde um processo ocupa toda a memória possível, seriam necessárias uma tabela de primeiro nível e 1.024 tabelas de segundo nível.

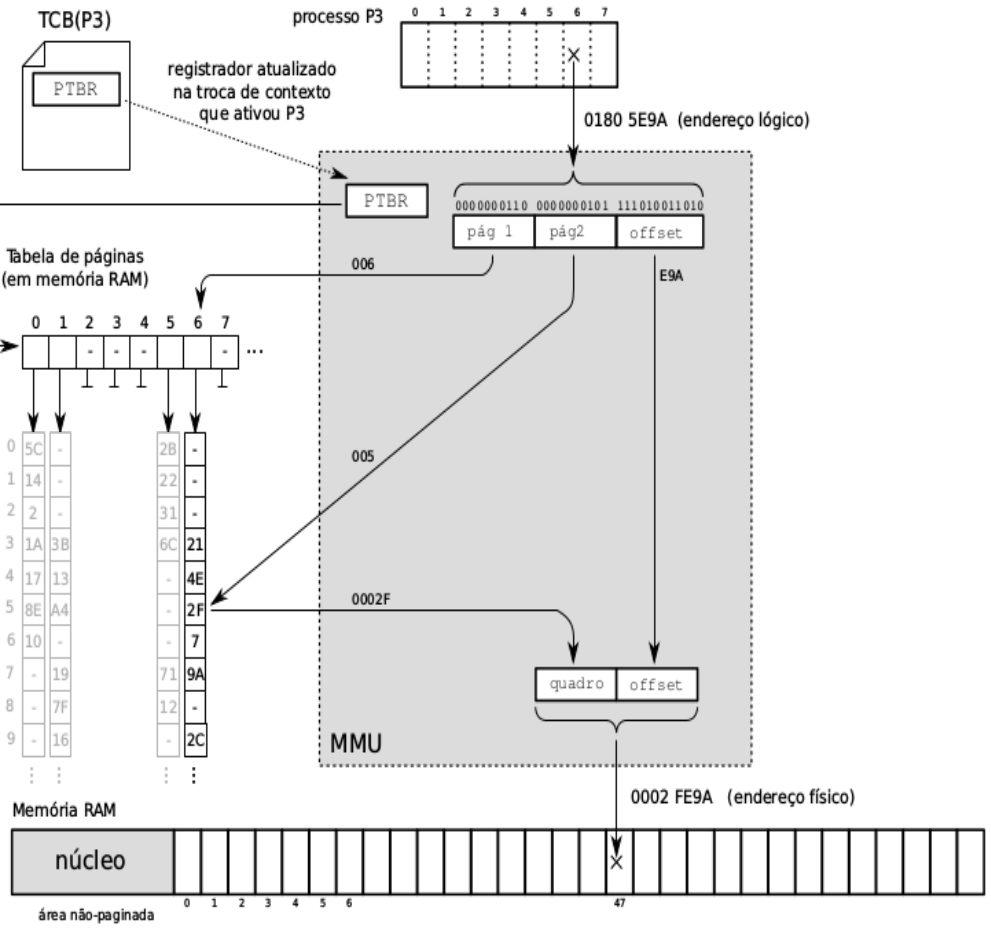
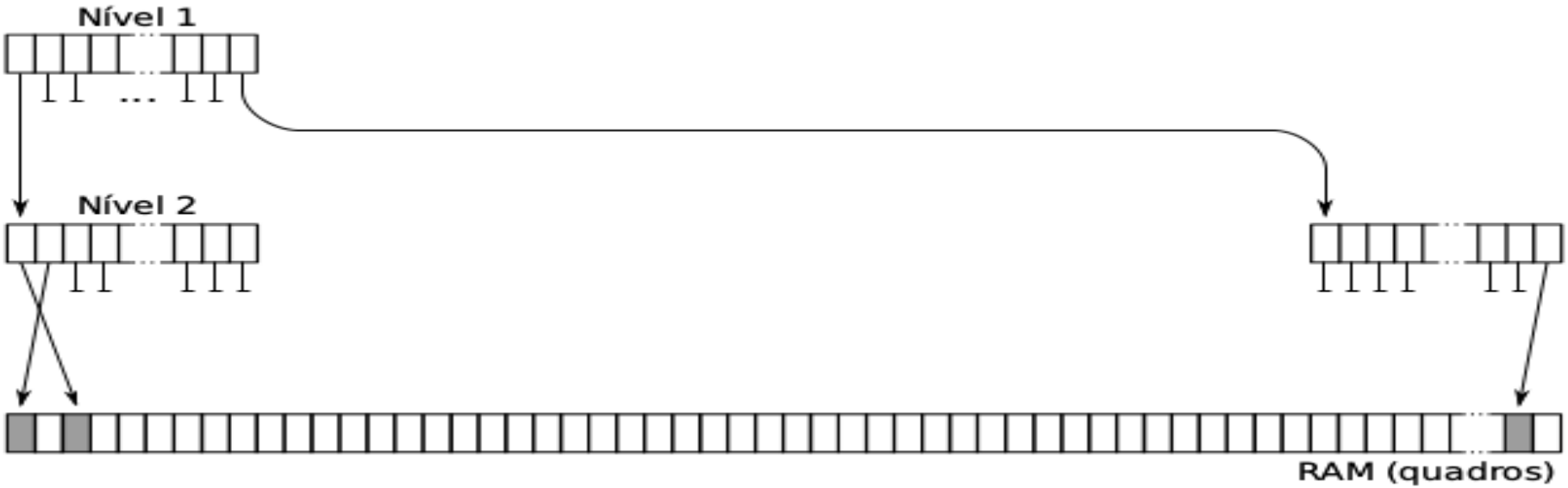
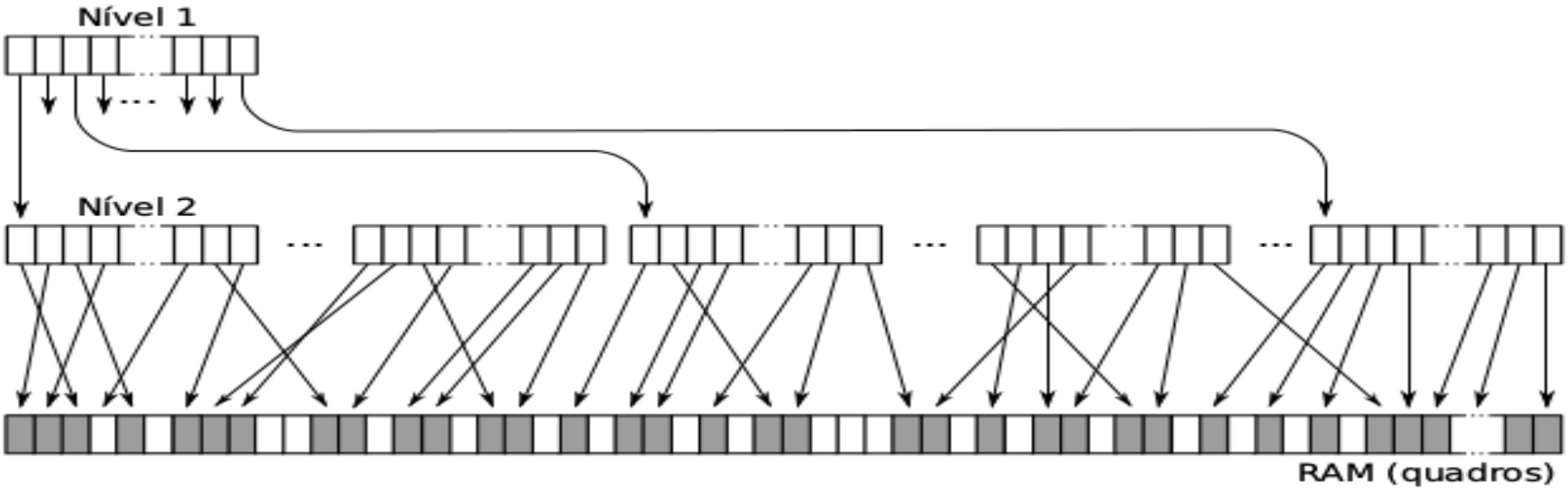


Figura 5.14: Tabela de página multi-nível.

Tabelas multiníveis





Tabelas multiníveis

Em um sistema com tabelas de dois níveis, cada acesso à memória solicitado pelo processador implica em mais dois acessos à memória para percorrer os dois níveis de tabelas. Com isso, **o tempo efetivo de acesso à memória se torna três vezes maior**.

Consultas recentes à tabela de páginas são armazenadas em um cache dentro da própria MMU. Em cada instante, os acessos tendem a se concentrar em poucas páginas.

O cache de tabela de páginas na MMU, denominado TLB (Translation Lookaside Buffer) ou cache associativo, armazena pares [página, quadro] obtidos em consultas recentes.

Tabelas multiníveis

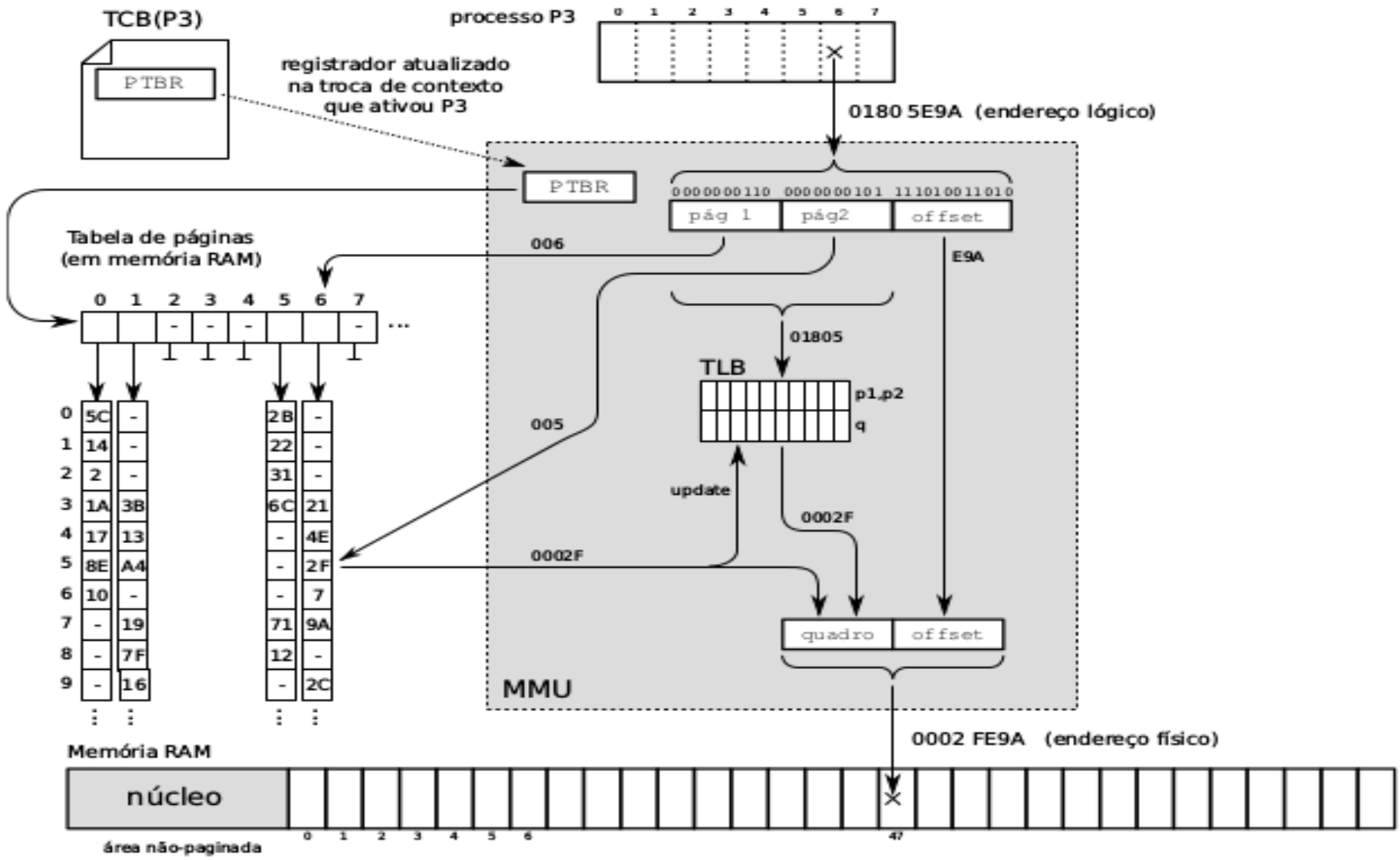


Figura 5.16: Uso da TLB.

Tempo médio de acesso à memória

Tempo médio de acesso

- Tempo de acesso à RAM é de 50 ns.
- Tabelas de páginas com 3 níveis.
- TLB: custo de acerto de 0,5 ns
  custo de erro de 10 ns
  taxa de acerto de 95%

O tempo médio de acesso à memória pode ser estimado como segue:

$$
\begin{aligned}
t_{médio} &= 95\% \times 0,5ns && \text{// em caso de acerto} \\
&+ 5\% \times (10ns + 3 \times 50ns) && \text{// em caso de erro, consultar as tabelas} \\
&+ 50ns && \text{// acesso ao quadro desejado}
\end{aligned}
$$

$$
t_{médio} = 58,475ns
$$

Tempo médio de acesso à memória

Tempo médio de acesso
- Tempo de acesso à RAM é de 50 ns.
- Tabelas de páginas com 3 níveis.
- TLB: custo de acerto de 0,5 ns
  custo de erro de 10 ns
  taxa de acerto de 95%

- Enquanto um sistema de paginação multinível sem cache tem um custo de 50ns x 3 = 150 ns + 50ns (dado), com cache esse custo cai para 58.475 ns.

- Em um sistema que não possui paginação, cada acesso a memória ocorre em 50 ns. Por outro lado, ao adicionar paginação com uso de cache, esse tempo aumenta em média 8,475 ns (58,475 → 16,9%).

- O custo é muito dependente da taxa de acerto do TLB: no cálculo anterior, caso a taxa de acerto fosse de 90%, o custo adicional seria de 32,9%; caso a taxa subisse a 99%, o custo adicional cairia para 4,2%.

TLB

Por exemplo, o Intel i386 tem um TLB com 64 entradas para páginas de dados e 32 entradas para páginas de código; por sua vez, o Intel Itanium tem 128 entradas para páginas de dados e 96 entradas para páginas de código.

$ x86info -c

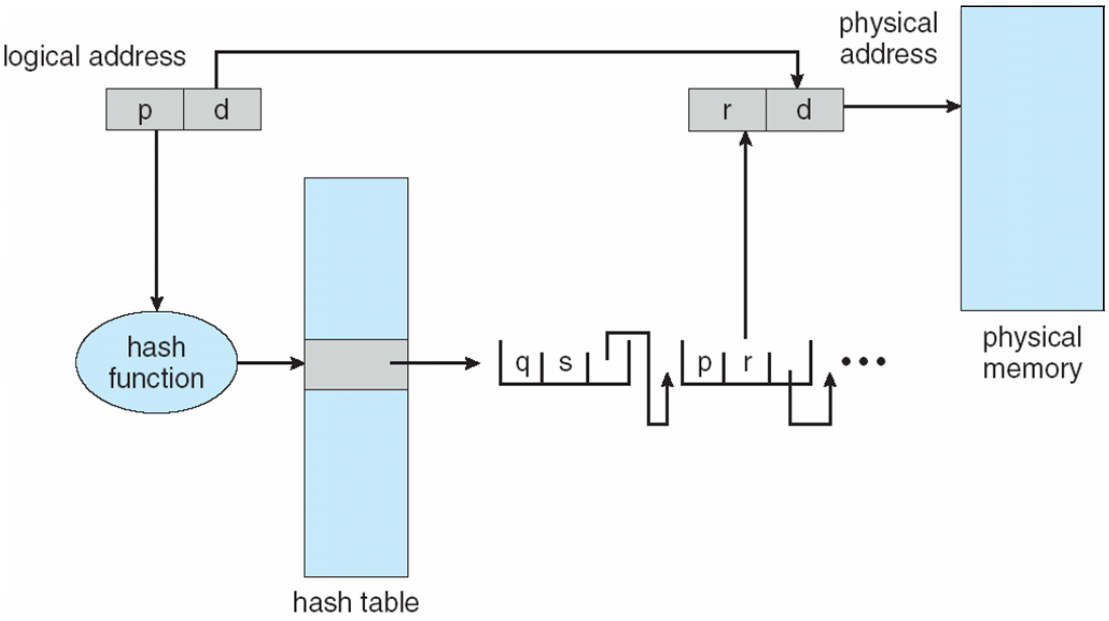
```
Instruction TLB: 2MB or 4MB pages – 7 entries
Instruction TLB: 4K pages, – 64 entries.
Data TLB: 4KB or 4MB pages, fully associative, 32 entries.
Data TLB: 4K pages, 4-way associative, 512 entries.
```

TLB

- Taxa de acertos também é influenciada pela **política de substituição das entradas** do TLB.
- Outro aspecto que influencia significativamente a taxa de acerto do TLB é a **forma como cada processo acessa a memória** (localidade de referência).
- **A cada troca de contexto**, a tabela de páginas é substituída e portanto **o cache TLB deve ser esvaziado**, pois seu conteúdo não é mais válido.
  - Para não ter que limpar o cache a cada troca de contexto, cada entrada pode conter um identificador de espaço de endereçamento **address-space identifiers** (**ASIDs**).
  - Algumas entradas do TLB podem ser fixas (ex. frames do Sistema Operacional).

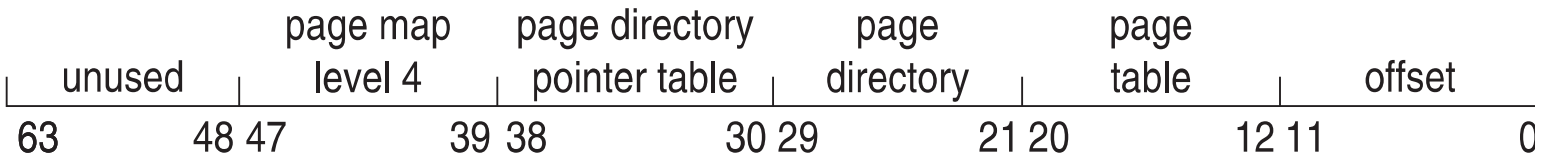
Hashed Page Tables

- A tabela de página multinível é apenas uma opção de implementação. Outras estruturas também podem ser utilizadas, tais como Hashed Page Tables.
- Uma função hash é aplicada ao endereço de página.
- Cada entrada da tabela pode conter uma lista encadeada com os mapeamentos: página → frame



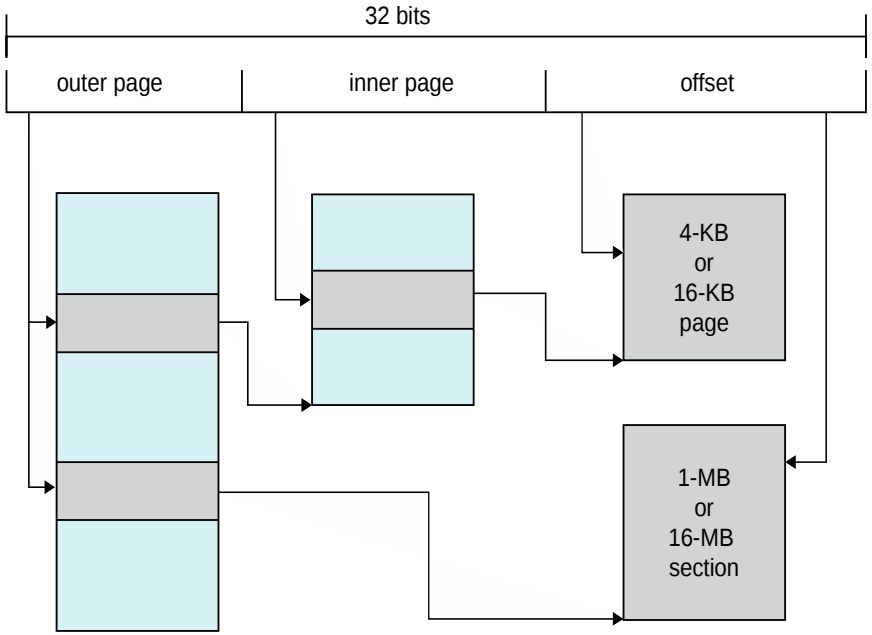
Intel x86-64

- 64 bits (> 16 exabytes)
- Na prática utiliza apenas 48 bits de endereçamento
  - Páginas de tamanho: 4 KB, 2 MB, 1 GB.
  - Endereço lógico linear possui 4 níveis de hierarquia de páginas.



Arquitetura ARM

- Páginas de 4 KB, 16 KB, 1MB e 16MB.
- 1MB e 16 MB são chamadas de seções.



Fim do capítulo 8