Sistemas Operacionais: Conceitos e Mecanismos

V - Gerência de Memória

Prof. Carlos Alberto Maziero
DInf UFPR
http://www.inf.ufpr.br/maziero

4 de agosto de 2017

Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *GNU/Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto LaTeX 2$_\varepsilon$, gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criadores de gráficos *GNUPlot* e *GraphViz* e processador PS/PDF *GhostScript*, entre outros.

Sumário

Resumo

A memória principal é um componente fundamental em qualquer sistema de computação. Ela constitui o "espaço de trabalho" do sistema, no qual são mantidos os processos, threads, bibliotecas compartilhadas e canais de comunicação, além do próprio núcleo do sistema operacional, com seu código e suas estruturas de dados. O hardware de memória pode ser bastante complexo, envolvendo diversas estruturas, como caches, unidade de gerência, etc, o que exige um esforço de gerência significativo por parte do sistema operacional.

Uma gerência adequada da memória é essencial para o bom desempenho de um computador. Neste capítulo serão estudados os elementos de hardware que compõe a memória de um sistema computacional e os mecanismos implementados ou controlados pelo sistema operacional para a gerência da memória.

1 Estruturas de memória

Existem diversos tipos de memória em um sistema de computação, cada um com suas próprias características e particularidades, mas todos com um mesmo objetivo: armazenar dados. Observando um sistema computacional típico, pode-se identificar vários locais onde dados são armazenados: os registradores e o cache interno do processador (denominado *cache L1*), o cache externo da placa mãe (*cache L2*) e a memória principal (RAM). Além disso, discos rígidos e unidades de armazenamento externas (*pendrives*, CD-ROMs, DVD-ROMs, fitas magnéticas, etc.) também podem ser considerados memória em um sentido mais amplo, pois também têm como função o armazenamento de dados.

Esses componentes de hardware são construídos usando diversas tecnologias e por isso têm características distintas, como a capacidade de armazenamento, a velocidade de operação, o consumo de energia e o custo por byte armazenado. Essas características permitem definir uma *hierarquia de memória*, representada na forma de uma pirâmide (Figura 1).

Nessa pirâmide, observa-se que memórias mais rápidas, como os registradores da CPU e os caches, são menores (têm menor capacidade de armazenamento), mais caras e consomem mais energia que memórias mais lentas, como a memória principal (RAM) e os discos rígidos. Além disso, as memórias mais rápidas são *voláteis*, ou seja, perdem seu conteúdo ao ficarem sem energia. Memórias que preservam seu conteúdo mesmo quando não tiverem energia são denominadas *não-voláteis*.

Outra característica importante das memórias é a rapidez de seu funcionamento, que pode ser detalhada em duas dimensões: *tempo de acesso* (ou *latência*) e *taxa de transferência*. O tempo de acesso caracteriza o tempo necessário para iniciar uma transferência de dados de/para um determinado meio de armazenamento. Por sua vez, a taxa de transferência indica quantos bytes por segundo podem ser lidos/escritos naquele meio, uma vez iniciada a transferência de dados. Para ilustrar esses dois conceitos complementares, a Tabela 1 traz valores de tempo de acesso e taxa de transferência de alguns meios de armazenamento usuais.

Neste capítulo serão estudados os mecanismos envolvidos na gerência da memória principal do computador, que geralmente é constituída por um grande espaço de

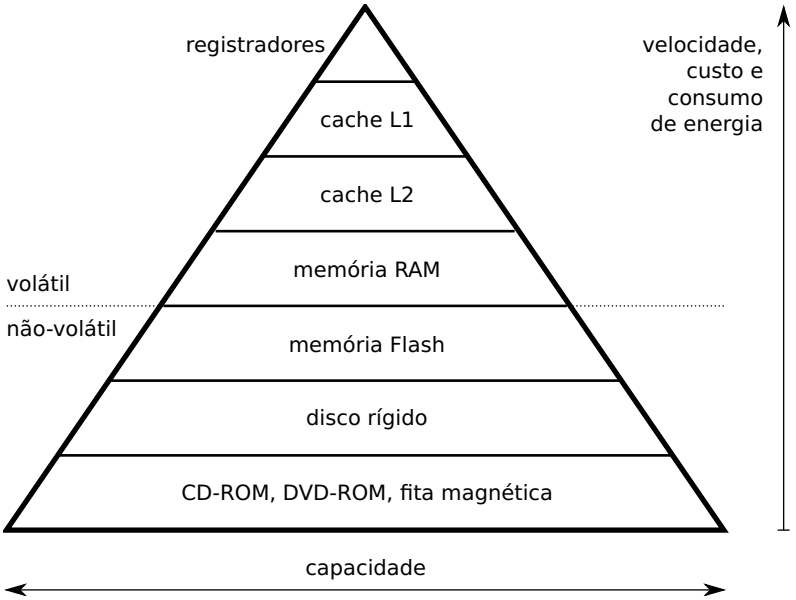

Figura 1: Hierarquia de memória.

Meio	**Tempo de acesso**	**Taxa de transferência**
Cache L2	1 ns	1 GB/s (1 ns/byte)
Memória RAM	60 ns	1 GB/s (1 ns/byte)
Memória *flash* (NAND)	2 ms	10 MB/s (100 ns/byte)
Disco rígido IDE	10 ms (tempo necessário para o deslocamento da cabeça de leitura e rotação do disco até o setor desejado)	80 MB/s (12 ns/byte)
DVD-ROM	de 100 ms a vários minutos (caso a gaveta do leitor esteja aberta ou o disco não esteja no leitor)	10 MB/s (100 ns/byte)

Tabela 1: Tempos de acesso e taxas de transferência típicas [Patterson and Henessy, 2005].

memória do tipo RAM (*Random Access Memory* ou memória de leitura/escrita). Também será estudado o uso do disco rígido como extensão da memória principal, através de mecanismos de memória virtual (Seção 7). A gerência dos espaços de armazenamento em disco rígido é abordada no Capítulo **??**. Os mecanismos de gerência dos caches L1 e L2 geralmente são implementados em hardware e são independentes do sistema operacional. Detalhes sobre seu funcionamento podem ser obtidos em [Patterson and Henessy, 2005].

2 Endereços, variáveis e funções

Ao escrever um programa usando uma linguagem de alto nível, como C, C++ ou Java, o programador usa apenas referências a entidades abstratas, como variáveis, funções, parâmetros e valores de retorno. Não há necessidade do programador definir

ou manipular endereços de memória explicitamente. O trecho de código em C a seguir (`soma.c`) ilustra esse conceito; nele, são usados símbolos para referenciar posições de dados (`i` e `soma`) ou de trechos de código (`main`, `printf` e `exit`):

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main ()
5  {
6     int i, soma = 0 ;
7
8     for (i=0; i< 5; i++)
9     {
10       soma += i ;
11       printf ("i vale %d e soma vale %d\n", i, soma) ;
12    }
13    exit(0) ;
14 }
```

Todavia, o processador do computador acessa endereços de memória para buscar as instruções a executar e seus operandos; acessa também outros endereços de memória para escrever os resultados do processamento das instruções. Por isso, quando programa `soma.c` for compilado, ligado a bibliotecas, carregado na memória e executado pelo processador, cada variável ou trecho de código definido pelo programador deverá ocupar um espaço específico e exclusivo na memória, com seus próprios endereços. A listagem a seguir apresenta o código *Assembly* correspondente à compilação do programa `soma.c`. Nele, pode-se observar que não há mais referências a nomes simbólicos, apenas a endereços:

```
00000000 <main>:
   0: 8d 4c 24 04           lea    0x4(%esp),%ecx
   4: 83 e4 f0              and    $0xfffffff0,%esp
   7: ff 71 fc              pushl  -0x4(%ecx)
   a: 55                    push   %ebp
   b: 89 e5                 mov    %esp,%ebp
   d: 51                    push   %ecx
   e: 83 ec 14              sub    $0x14,%esp
  11: c7 45 f4 00 00 00 00  movl   $0x0,-0xc(%ebp)
  18: c7 45 f8 00 00 00 00  movl   $0x0,-0x8(%ebp)
  1f: eb 1f                 jmp    40 <main+0x40>
  21: 8b 45 f8              mov    -0x8(%ebp),%eax
  24: 01 45 f4              add    %eax,-0xc(%ebp)
  27: 83 ec 04              sub    $0x4,%esp
  2a: ff 75 f4              pushl  -0xc(%ebp)
  2d: ff 75 f8              pushl  -0x8(%ebp)
  30: 68 00 00 00 00        push   $0x0
  35: e8 fc ff ff ff        call   36 <main+0x36>
  3a: 83 c4 10              add    $0x10,%esp
  3d: ff 45 f8              incl   -0x8(%ebp)
  40: 83 7d f8 04           cmpl   $0x4,-0x8(%ebp)
  44: 7e db                 jle    21 <main+0x21>
  46: 83 ec 0c              sub    $0xc,%esp
  49: 6a 00                 push   $0x0
  4b: e8 fc ff ff ff        call   4c <main+0x4c>
```

Dessa forma, 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.

Durante a compilação : o compilador escolhe as posições das variáveis na memória. Para isso, todos os códigos fontes que fazem parte do programa devem ser conhecidos no momento da compilação, para evitar conflitos de endereços entre variáveis. Uma outra técnica bastante usada é a geração de código independente de posição (PIC - *Position-Independent Code*), no qual todas as referências a variáveis são feitas usando endereços relativos (como "3.471 bytes após o início do módulo", ou "15 bytes após o *program counter*", por exemplo).

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[1]. Os arquivos com extensão `.o` em UNIX ou `.obj` em Windows são exemplos de arquivos-objeto obtidos da compilação de arquivos em C ou outra linguagem de alto nível. 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.

Durante a carga : também é possível definir os endereços de variáveis e de funções durante a carga do código em memória para o lançamento de um novo processo. Nesse caso, um *carregador* (*loader*) é responsável por carregar o código do processo na memória e definir os endereços de memória que devem ser utilizados. O carregador pode ser parte do núcleo do sistema operacional ou uma biblioteca ligada ao executável, ou ambos. Esse mecanismo normalmente é usado na carga das bibliotecas dinâmicas (DLL - *Dynamic Linking Libraries*).

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 (como os computadores pessoais), e será descrita nas próximas seções.

A Figura 2 ilustra os diferentes momentos da vida de um processo em que pode ocorrer a resolução dos endereços de variáveis e de código.

2.1 Endereços lógicos e físicos

Ao executar uma sequência de instruções, o processador escreve endereços no barramento de endereços do computador, que servem para buscar instruções e operandos, mas também para ler e escrever valores em posições de memória e portas de entrada/saída. Os endereços de memória gerados pelo processador à medida em que executa algum código são chamados de *endereços lógicos*, porque correspondem à lógica do programa, mas 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*), que pode fazer parte do próprio processador (como ocorre nos sistemas atuais) ou constituir um dispositivo separado (como ocorria nas máquinas mais antigas). A MMU faz a análise dos endereços lógicos emitidos pelo processador e determina os endereços físicos correspondentes na memória da máquina, permitindo então seu acesso pelo processador. Caso o acesso a um determinado endereço solicitado pelo processador não

[1] *Arquivos-objeto* são formatos de arquivo projetados para conter código binário e dados provenientes de uma compilação de código fonte. Existem diversos formatos de arquivos-objeto; os mais simples, como os arquivos `.com` do DOS, apenas definem uma sequência de bytes a carregar em uma posição fixa da memória; os mais complexos, como os formatos UNIX ELF (*Executable and Library Format*) e Microsoft PE (*Portable Executable Format*), permitem definir seções internas, tabelas de relocação, informação de depuração, etc. [Levine, 2000].

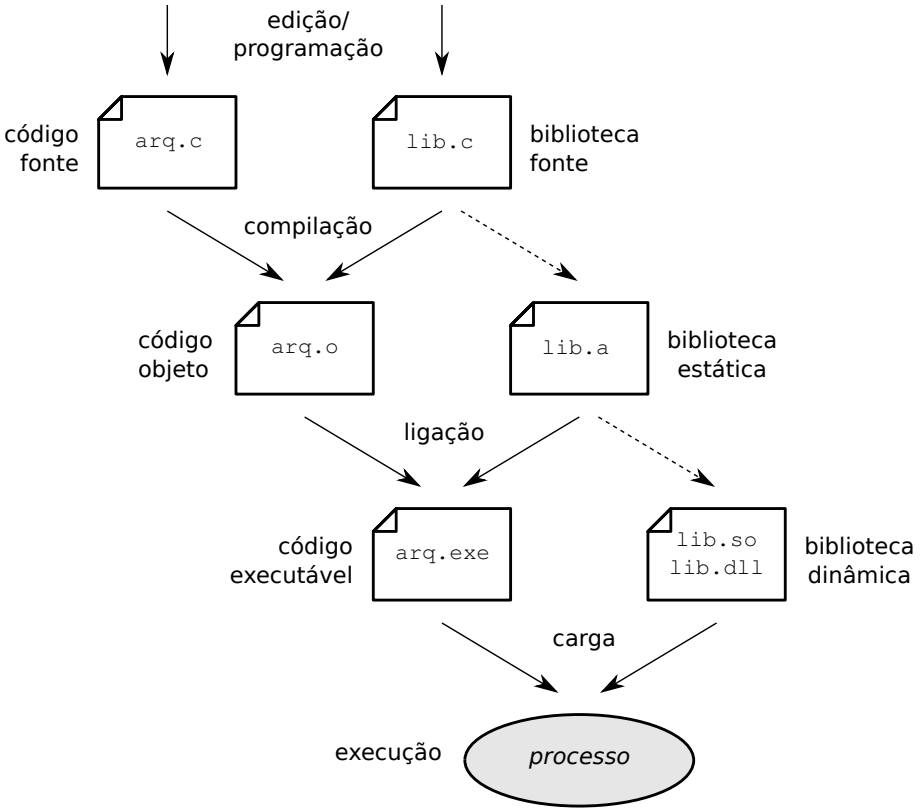

Figura 2: Momentos de atribuição de endereços.

seja possível, a MMU gera uma interrupção de hardware para notificar o processador sobre a tentativa de acesso indevido. O funcionamento básico da MMU está ilustrado na Figura 3.

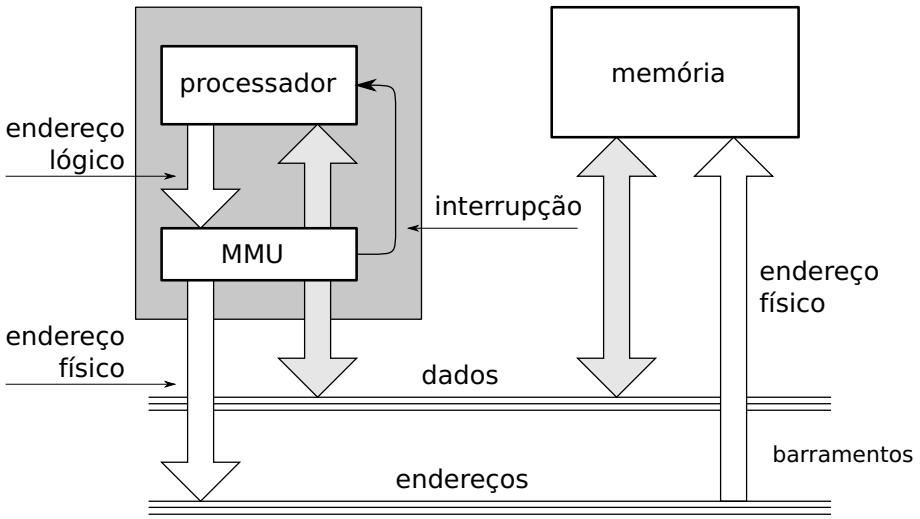

Figura 3: Funcionamento básico de uma MMU.

A proteção de memória entre processos é essencial para a segurança e estabilidade dos sistemas mais complexos, nos quais centenas ou milhares de processos podem estar

na memória simultaneamente. A MMU pode ser rapidamente ajustada para mudar a forma de conversão entre endereços lógicos e físicos, o que permite implementar uma área de memória exclusiva para cada processo do sistema. Assim, a cada troca de contexto entre processos, as regras de conversão da MMU devem ser ajustadas para somente permitir o acesso à área de memória definida para cada novo processo corrente.

2.2 Modelo de memória dos processos

Cada processo é visto pelo sistema operacional como uma cápsula isolada, ou seja, uma área de memória exclusiva que só ele e o núcleo do sistema podem acessar. Essa área de memória contém todas as informações necessárias à execução do processo, divididas nas seguintes seções:

TEXT : contém o código a ser executado pelo processo, gerado durante a compilação e a ligação com as bibliotecas. Esta área tem tamanho fixo, calculado durante a compilação, e normalmente só deve estar acessível para leitura e execução.

DATA : esta área contém os dados estáticos usados pelo programa, ou seja, suas variáveis globais e as variáveis locais estáticas (na linguagem C, são as variáveis definidas como `static` dentro das funções). Como o tamanho dessas variáveis pode ser determinado durante a compilação, esta área tem tamanho fixo; deve estar acessível para leituras e escritas, mas não para execução.

HEAP : área usada para armazenar dados através de alocação dinâmica, usando operadores como `malloc` e `free` ou similares. Esta área tem tamanho variável, podendo aumentar/diminuir conforme as alocações/liberações de memória feitas pelo processo. Ao longo do uso, esta área pode se tornar fragmentada, ou seja, pode conter lacunas entre os blocos de memória alocados. São necessários então algoritmos de alocação que minimizem sua fragmentação.

STACK : área usada para manter a pilha de execução do processo, ou seja, a estrutura responsável por gerenciar o fluxo de execução nas chamadas de função e também para armazenar os parâmetros, variáveis locais e o valor de retorno das funções. Geralmente a pilha cresce "para baixo", ou seja, inicia em endereços elevados e cresce em direção aos endereços menores da memória. No caso de programas com múltiplas *threads*, esta área contém somente a pilha do programa principal. Como *threads* podem ser criadas e destruídas dinamicamente, a pilha de cada *thread* é mantida em uma área própria, geralmente alocada no *heap*.

A Figura 4 apresenta a organização da memória de um processo. Nela, observa-se que as duas áreas de tamanho variável (*stack* e *heap*) estão dispostas em posições opostas e vizinhas à memória livre (não alocada). Dessa forma, a memória livre disponível ao processo pode ser aproveitada da melhor forma possível, tanto pelo *heap* quanto pelo *stack*, ou por ambos.

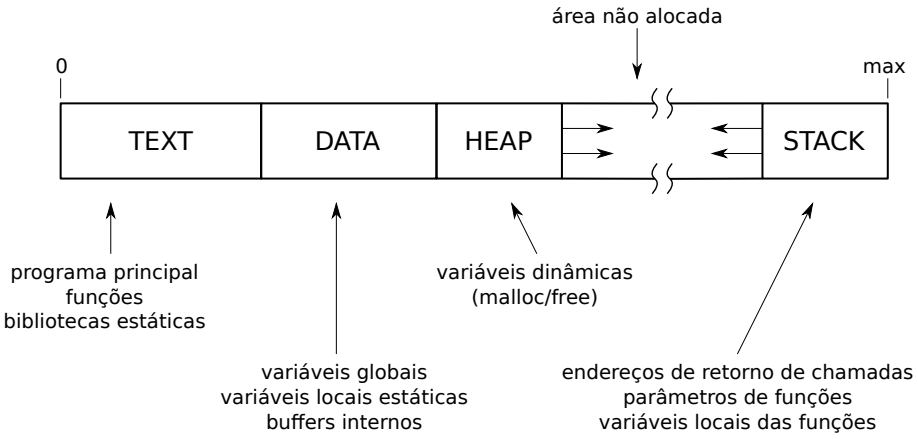

Figura 4: Organização da memória de um processo.

3 Estratégias de alocação

Em um sistema monoprocesso, em que apenas um processo por vez é carregado em memória para execução, a alocação da memória principal é um problema simples de resolver: basta reservar uma área de memória para o núcleo do sistema operacional e alocar o processo na memória restante, respeitando a disposição de suas áreas internas, conforme apresentado na Figura 4.

A memória reservada para o núcleo do sistema operacional pode estar no início ou no final da área de memória física disponível. Como a maioria das arquiteturas de hardware define o vetor de interrupções (vide Seção **??**) nos endereços iniciais da memória (também chamados *endereços baixos*), geralmente o núcleo também é colocado na parte inicial da memória. Assim, toda a memória disponível após o núcleo do sistema é destinada aos processos no nível do usuário (*user-level*). A Figura 5 ilustra essa organização da memória.

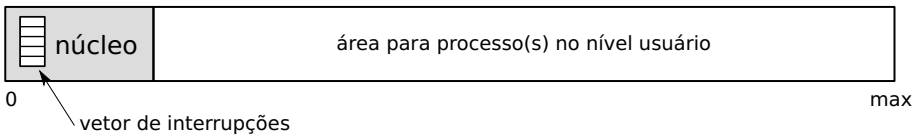

Figura 5: Organização da memória do sistema.

Nos sistemas multiprocessos, 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. As principais estratégias de alocação da memória física serão estudadas nas próximas seções.

3.1 Partições fixas

A forma mais simples de alocação de memória consiste em dividir a memória destinada aos processos em N partições fixas, de tamanhos iguais ou distintos. Em

cada partição pode ser carregado um processo. Nesse esquema, a tradução entre os endereços lógicos vistos pelos processos e os endereços físicos é feita através de um simples registrador de relocação, cujo valor é somado ao endereço lógico gerado pelo processador, a fim de obter o endereço físico correspondente. Endereços lógicos maiores que o tamanho da partição em uso são simplesmente rejeitados pela MMU. O exemplo da Figura 6 ilustra essa estratégia. No exemplo, o processo da partição 3 está executando e deseja acessar o endereço lógico 14.257. A MMU recebe esse endereço e o soma ao valor do registrador de relocação (110.000) para obter o endereço físico 124.257, que então é acessado. Deve-se observar que o valor contido no registrador de relocação é o endereço de início da partição ativa (partição 3); esse registrador deve ser atualizado a cada troca de processo ativo.

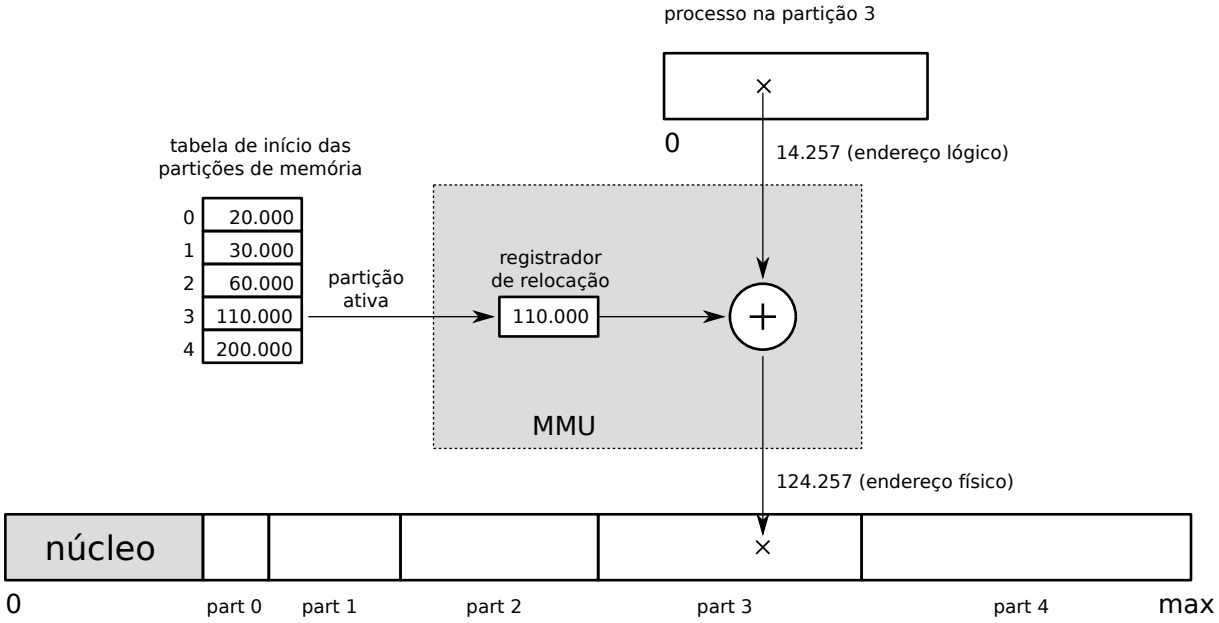

Figura 6: Alocação em partições fixas.

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; ela foi muito usada no OS/360, um sistema operacional da IBM usado nas décadas de 1960-70 [Tanenbaum, 2003].

3.2 Alocação contígua

A estratégia anterior, com partições fixas, pode ser tornar bem mais flexível caso o tamanho de cada partição possa ser 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. O algoritmo de tradução de endereços lógicos em físicos é bem simples: cada endereço lógico gerado pelo processo em execução é comparado ao valor do registrador limite; caso seja maior ou igual a este, uma interrupção é gerada pela MMU de volta para o processador, indicando um endereço inválido. Caso contrário, o endereço lógico é somado ao valor do registrador base, para a obtenção do endereço físico correspondente. A Figura 7 apresenta uma visão geral dessa estratégia. Na Figura, o processo p_3 tenta acessar o endereço lógico 14.257.

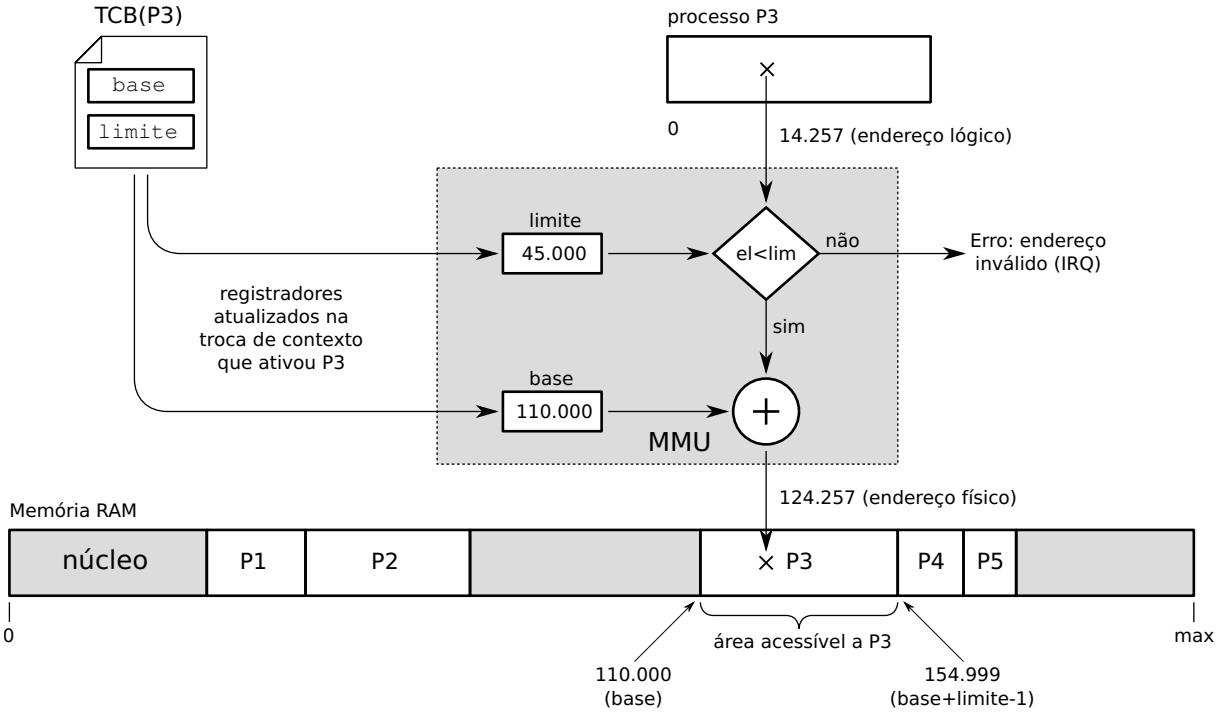

Figura 7: Alocação contígua de memória.

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*, vide Seção **??**). Obviamente, quando o núcleo estiver executando, os valores de base e limite devem ser ajustados respectivamente para 0 e ∞, para permitir o acesso direto a toda a memória física.

Além de traduzir endereços lógicos nos endereços físicos correspondentes, a ação da MMU propicia a proteção de memória entre os processos: quando um processo p_i estiver executando, ele só pode acessar endereços lógicos no intervalo $[0, limite(p_i) - 1]$, que correspondem a endereços físicos no intervalo $[base(p_i), base(p_i) + limite(p_i) - 1]$.

Ao detectar uma tentativa de acesso a um endereço fora desse intervalo, a MMU irá gerar uma solicitação de interrupção (IRQ - *Interrupt ReQuest*, vide Seção **??**) para o processador, indicando o endereço inválido. Ao receber a interrupção, o processador interrompe o fluxo de execução do processo p_i, retorna ao núcleo e ativa a rotina de tratamento da interrupção, que poderá abortar o processo ou tomar outras providências.

A maior vantagem da estratégia de alocação contígua é sua simplicidade: por depender apenas de dois registradores e de uma lógica simples para a tradução de endereços, pode ser implementada em hardware de baixo custo, ou mesmo incorporada a processadores mais simples. Todavia, é uma estratégia pouco flexível e está muito sujeita à fragmentação externa, conforme será discutido na Seção 5.

3.3 Alocação por segmentos

A alocação por segmentos, ou *alocação segmentada*, é uma extensão da alocação contígua, na qual o espaço de memória de um processo é fracionado em áreas, ou *segmentos*, que podem ser alocados separadamente na memória física. Além das quatro áreas funcionais básicas da memória do processo discutidas na Seção 2.2 (*text*, *data*, *stack* e *heap*), também podem ser definidos segmentos para itens específicos, como bibliotecas compartilhadas, vetores, matrizes, pilhas de *threads*, buffers de entrada/saída, etc.

Ao estruturar a memória em segmentos, o espaço de memória de cada processo não é mais visto como uma sequência linear de endereços lógicos, mas como uma coleção de segmentos de tamanhos diversos e políticas de acesso distintas. A Figura 8 apresenta a visão lógica da memória de um processo e a sua forma de mapeamento para a memória física.

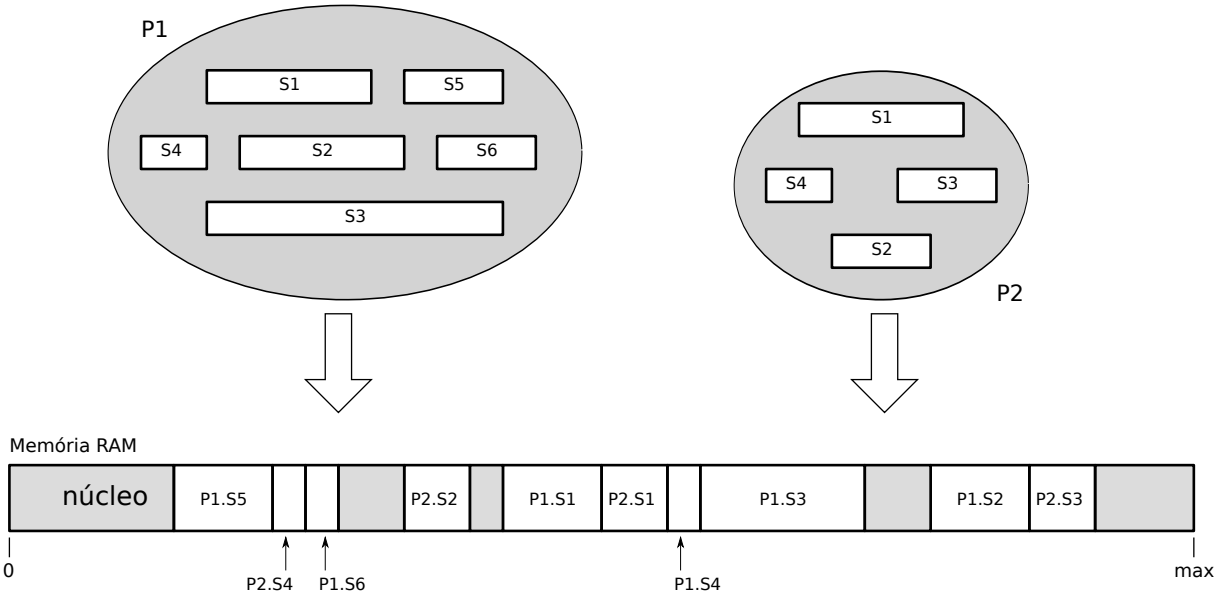

Figura 8: Alocação de memória por segmentos.

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. Em

outras palavras, este modelo usa endereços lógicos *bidimensionais*, compostos por pares [*segmento*:*offset*], onde *segmento* indica o número do segmento desejado e *offset* indica a posição desejada dentro do segmento. Os valores de *offset* variam de 0 (zero) ao tamanho do segmento. A Figura 9 mostra alguns exemplos de endereços lógicos usando alocação por segmentos.

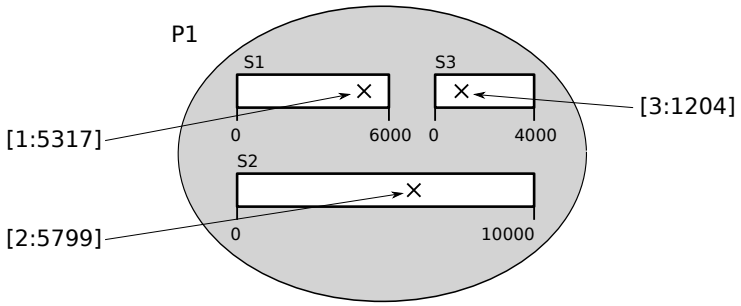

Figura 9: Endereços lógicos em segmentos.

Na alocação de memória por segmentos, a forma de tradução de endereços lógicos em físicos é similar à da alocação contígua. Contudo, como os segmentos podem ter tamanhos distintos e ser alocados separadamente na memória física, 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. Essa tabela contém os valores de base e limite para cada segmento usado pelo processo, além de *flags* com informações sobre cada segmento, como permissões de acesso, etc. (vide Seção 3.4.1). A Figura 10 apresenta os principais elementos envolvidos na tradução de endereços lógicos em físicos usando memória alocada por segmentos. Nessa figura, "`ST reg`" indica o registrador que aponta para a tabela de segmentos ativa.

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. Dessa forma, erros frequentes como acessos a índices além do tamanho de um vetor irão gerar endereços fora do respectivo segmento, que serão detectados pelo hardware de gerência de memória e notificados ao sistema operacional.

A implementação da tabela de segmentos varia conforme a arquitetura de hardware considerada. 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 (vide Seção 6). Cada uma dessas duas tabelas comporta até 8.192 segmentos. As tabelas em uso pelo processo em execução são indicadas por registradores específicos do processador. A cada troca de contexto, os registradores que indicam a tabela de segmentos ativa devem ser atualizados para refletir as áreas de memória usadas pelo processo que será ativado.

Para cada endereço de memória acessado pelo processo em execução, é necessário acessar a tabela de segmentos para obter os valores de base e limite correspondentes

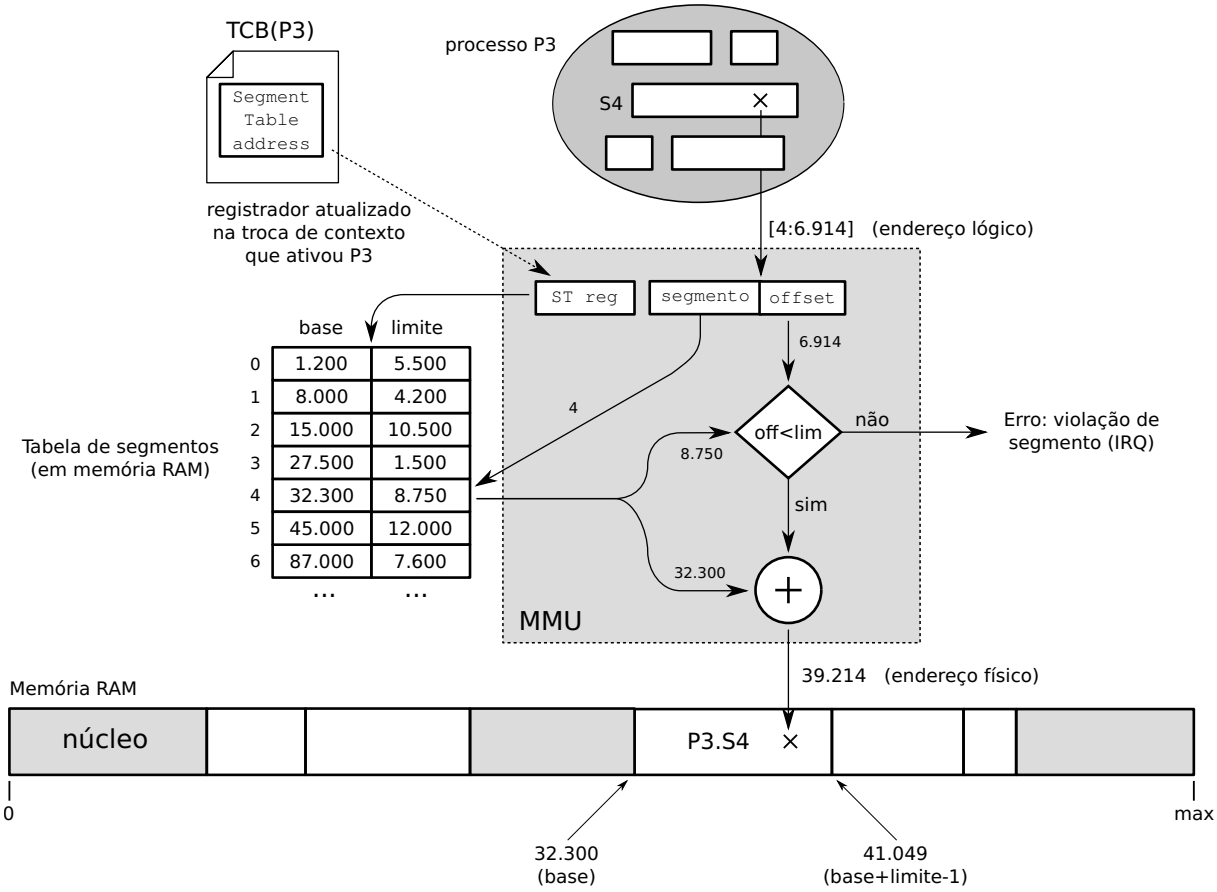

Figura 10: Tradução de endereços em memória alocada por segmentos.

ao endereço lógico acessado. Todavia, como as tabelas de segmentos normalmente se encontram na memória principal, esses acessos têm um custo significativo: considerando um sistema de 32 bits, para cada acesso à memória seriam necessárias pelo menos duas leituras adicionais na memória, para ler os valores de base e limite, o que tornaria cada acesso à memória três vezes mais lento. 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. Assim, caso o número de segmentos em uso simultâneo seja pequeno, não há necessidade de consultar a tabela de segmentos com excessiva frequência, o que mantém o desempenho de acesso à memória em um nível satisfatório. O processador 80.386 define os seguintes registradores de segmentos:

- **CS**: *Code Segment*, indica o segmento onde se encontra o código atualmente em execução; este valor é automaticamente ajustado no caso de chamadas de funções de bibliotecas, chamadas de sistema, interrupções ou operações similares.
- **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, que podem conter variáveis globais, vetores ou áreas alocadas dinamicamente. Esses registradores podem ser ajustados em caso de necessidade, para acessar outros segmentos de dados.

O conteúdo desses registradores é preservado no TCB (*Task Control Block*) de cada processo a cada troca de contexto, tornando o acesso à memória bastante eficiente caso poucos segmentos sejam usados simultaneamente. Portanto, o compilador tem uma grande responsabilidade na geração de código executável: minimizar o número de segmentos necessários à execução do processo a cada instante, para não prejudicar o desempenho de acesso à memória.

Exemplos de processadores que utilizam a alocação por segmentos incluem o 80.386 e seus sucessores (486, Pentium, Athlon e processadores correlatos).

3.4 Alocação paginada

Conforme visto na Seção anterior, a alocação de memória por segmentos exige o uso de endereços bidimensionais na forma [*segmento*:*offset*], o que é pouco intuitivo para o programador e torna mais complexa a construção de compiladores. Além disso, é uma forma de alocação bastante suscetível à fragmentação externa, conforme será discutido na Seção 5. Essas deficiências levaram os projetistas de hardware a desenvolver outras técnicas para a alocação da memória principal.

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 para os processos, o espaço de endereços lógicos é dividido em pequenos blocos de mesmo tamanho, denominados *páginas*. Nas arquiteturas atuais, as páginas geralmente têm 4 KBytes (4.096 bytes), mas podem ser encontradas arquiteturas com páginas de outros tamanhos[2]. O espaço de memória física destinado aos processos também é dividido em blocos de mesmo tamanho que as páginas, denominados *quadros* (do inglês *frames*). A alocação dos processos na memória física é então feita simplesmente indicando em que quadro da memória física se encontra cada página de cada processo, conforme ilustra a Figura 11. É importante observar que as páginas de um processo podem estar em qualquer posição da memória física disponível aos processos, ou seja, podem estar associadas a quaisquer quadros, o que permite uma grande flexibilidade de alocação. Além disso, as páginas não usadas pelo processo não precisam estar mapeadas na memória física, o que proporciona maior eficiência no uso da mesma.

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

[2] As arquiteturas de processador mais recentes suportam diversos tamanhos de páginas, inclusive páginas muito grandes, as chamadas *superpáginas* (*hugepages*, *superpages* ou *largepages*). Uma superpágina tem geralmente entre 1 e 16 MBytes, ou mesmo acima disso; seu uso em conjunto com as páginas normais permite obter mais desempenho no acesso à memória, mas torna os mecanismos de gerência de memória bem mais complexos. O artigo [Navarro et al., 2002] traz uma discussão mais detalhada sobre esse tema.

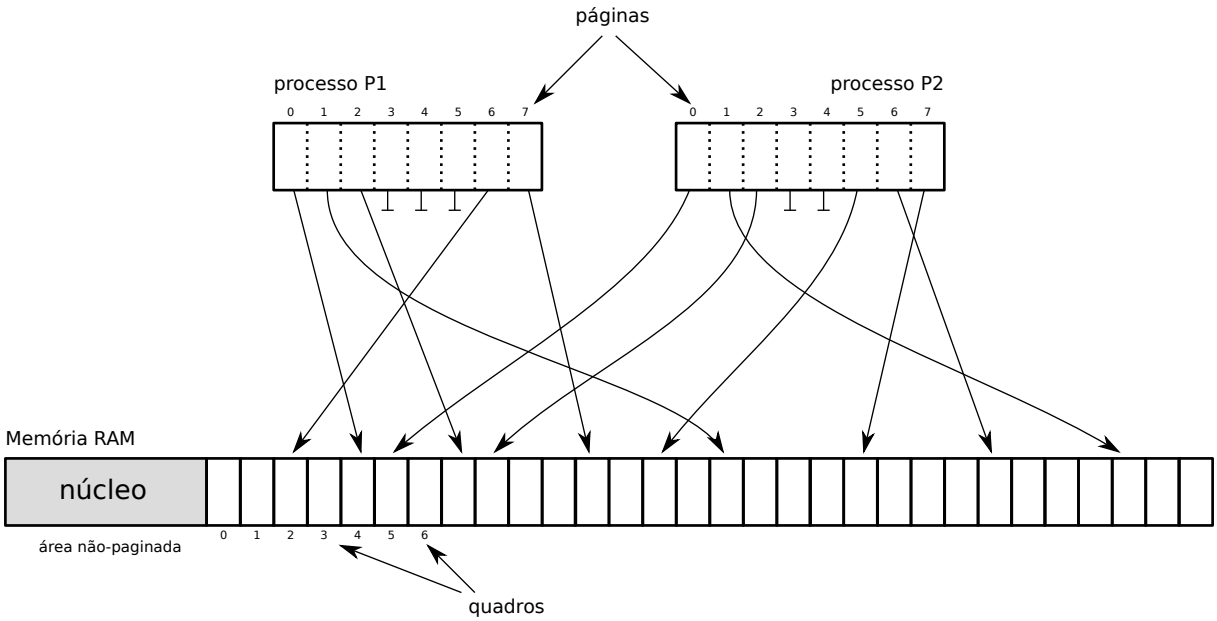

Figura 11: Alocação de memória por páginas.

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.

A divisão do espaço de endereçamento lógico de um processo em páginas pode ser feita de forma muito simples: como as páginas sempre têm 2^n bytes de tamanho (por exemplo, 2^{12} bytes para páginas de 4 KBytes) os n bits menos significativos de cada endereço lógico definem a posição daquele endereço dentro da página (deslocamento ou *offset*), enquanto os bits restantes (mais significativos) são usados para definir o número da página. Por exemplo, o processador Intel 80.386 usa endereços lógicos de 32 bits e páginas com 4 KBytes; um endereço lógico de 32 bits é decomposto em um *offset* de 12 bits, que representa uma posição entre 0 e 4.095 dentro da página, e um número de página com 20 bits. Dessa forma, podem ser endereçadas 2^{20} páginas com 2^{12} bytes cada (1.048.576 páginas com 4.096 bytes cada). Eis um exemplo de decomposição de um endereço lógico nesse sistema:

$$
\begin{aligned}
01805E9A_H &\rightarrow 0000\ 0001\ 1000\ 0000\ 0101\ 1110\ 1001\ 1010_2 \\
&\rightarrow \underbrace{0000\ 0001\ 1000\ 0000\ 0101_2}_{20\text{ bits}}\ \text{e}\ \underbrace{1110\ 1001\ 1010_2}_{12\text{ bits}} \\
&\rightarrow \underbrace{01805_H}_{\text{página}}\ \text{e}\ \underbrace{E9A_H}_{\text{offset}} \\
&\rightarrow \text{página } 01805_H \text{ e offset } E9A_H
\end{aligned}
$$

Para traduzir um endereço lógico no endereço físico correspondente, a MMU precisa efetuar os seguintes passos:

1. decompor o endereço lógico em número de página e *offset*;
2. obter o número do quadro onde se encontra a página desejada;
3. construir o endereço físico, compondo o número do quadro com o *offset*; como páginas e quadros têm o mesmo tamanho, o valor do *offset* é preservado na conversão;
4. caso a página solicitada não esteja mapeada em um quadro da memória física, a MMU deve gerar uma interrupção de *falta de página* (*page fault*) para o processador;
5. essa interrupção provoca o desvio da execução para o núcleo do sistema operacional, que deve então tratar a falta de página.

A Figura 12 apresenta os principais elementos que proporcionam a tradução de endereços em um sistema paginado com páginas de 4.096 bytes.

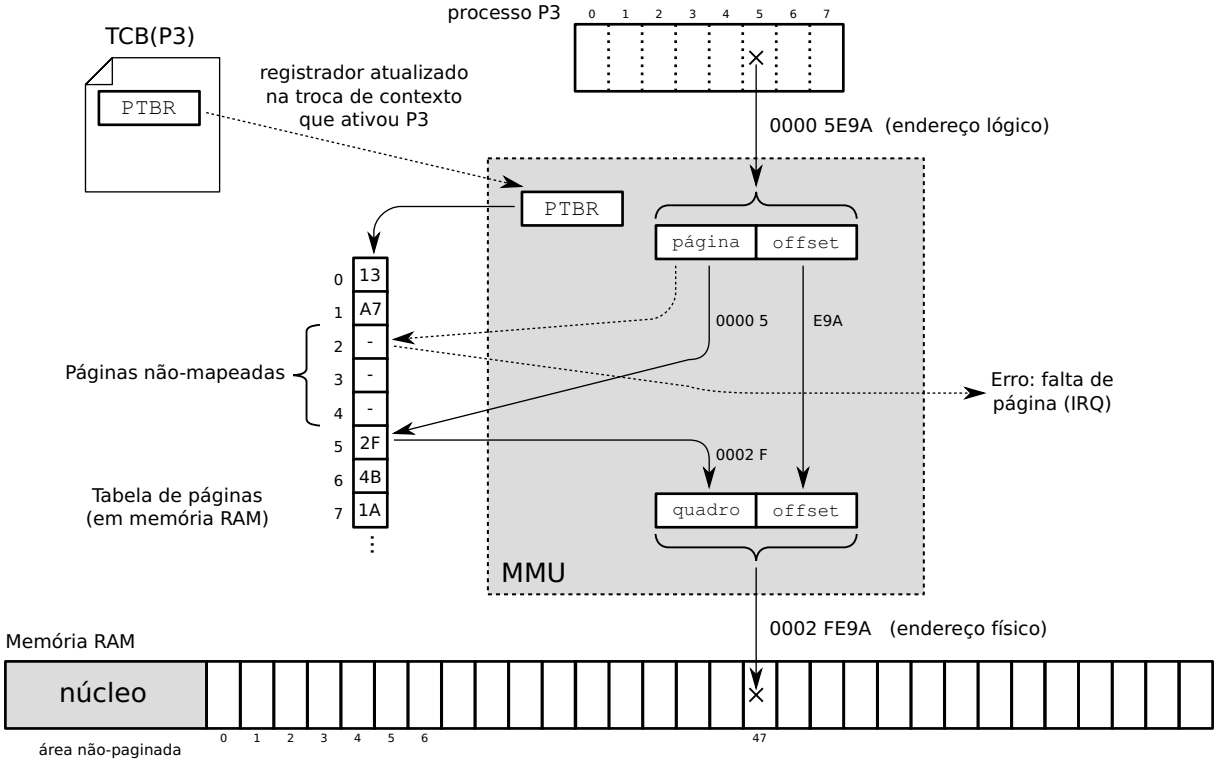

Figura 12: Tradução de endereços usando paginação.

3.4.1 Flags de controle

Como o espaço de endereçamento lógico de um processo pode ser extremamente grande (por exemplo, o espaço de endereços lógicos de cada processo em uma arquitetura

de 32 bits pode ir até 2^{32} bytes), uma parte significativa das páginas de um processo pode não estar mapeada em quadros de memória física. Áreas de memória não usadas por um processo não precisam estar mapeadas na memória física, o que permite um uso mais eficiente da memória. Assim, a tabela de páginas de cada processo indica as áreas não mapeadas com um flag adequado (*válido/inválido*). Cada entrada da tabela de páginas de um processo contém o número do quadro correspondente e um conjunto de flags (bits) de controle, com diversas finalidades:

- *Presença*: indica se a página está presente (mapeada) no espaço de endereçamento daquele processo;
- *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, sendo ajustado para 1 pelo próprio hardware a cada acesso à página. Este bit é usado pelos algoritmos de memória virtual (apresentados na Seção 7);
- *Modificação*: também chamado de *dirty bit*, é ajustado para 1 pelo hardware a cada escrita na página, indicando se a página foi modificada após ser carregada na memória; é usado pelos algoritmos de memória virtual.

Além destes, podem ser definidos outros bits, indicando a política de *caching* da página, se é uma página de usuário ou de sistema, se a página pode ser movida para disco, o tamanho da página (no caso de sistemas que permitam mais de um tamanho de página), etc. O conteúdo exato de cada entrada da tabela de páginas depende da arquitetura do hardware considerado.

3.4.2 Tabelas multi-níveis

Em uma arquitetura de 32 bits com páginas de 4 KBytes, cada entrada na tabela de páginas ocupa cerca de 32 bits, ou 4 bytes (20 bits para o número de quadro e os 12 bits restantes para flags). Considerando que cada tabela de páginas tem 2^{20} páginas, cada tabela ocupará 4 MBytes de memória (4×2^{20} bytes) se for armazenada de forma linear na memória. No caso de processos pequenos, com muitas páginas não mapeadas, uma tabela de páginas linear ocupará mais espaço na memória que o próprio processo, como mostra a Figura 13, o que torna seu uso pouco interessante.

Para resolver esse problema, são usadas *tabelas de páginas multinível*, estruturadas na forma de árvores: uma *tabela de páginas de primeiro nível* (ou *diretório de páginas*) contém ponteiros para *tabelas de páginas de segundo nível*, e assim por diante, até chegar à tabela que contém os números dos quadros desejados. Para percorrer essa árvore, o número de página é dividido em duas ou mais partes, que são usadas de forma sequencial, um para cada nível de tabela, até encontrar o número de quadro desejado. O número de níveis da tabela depende da arquitetura considerada: os processadores *Intel 80.386* usam tabelas com dois níveis, os processadores *Sun Sparc* e *DEC Alpha* usam tabelas com 3 níveis; processadores mais recentes, como *Intel Itanium*, podem usar tabelas com 3 ou 4 níveis.

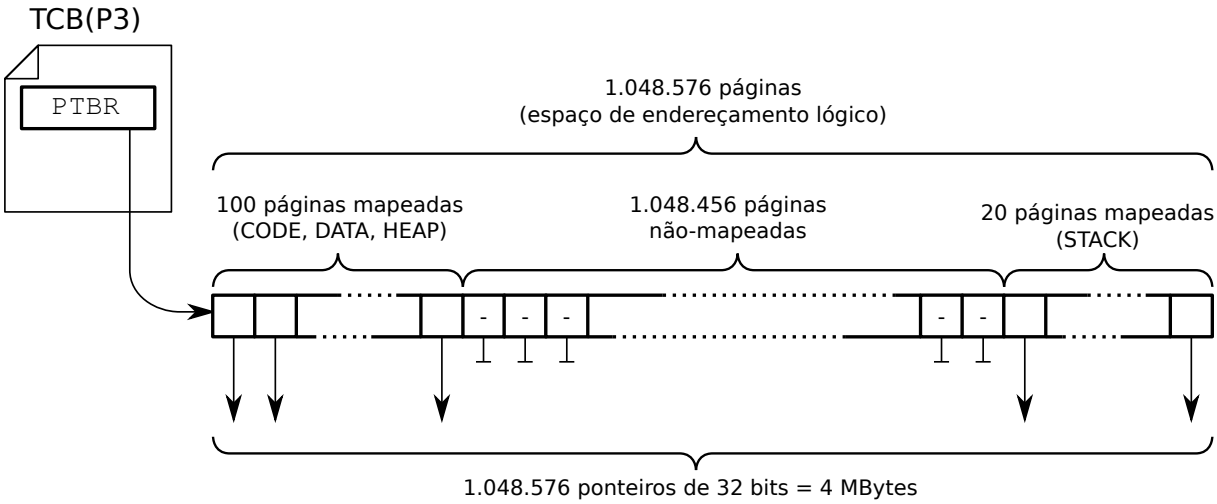

Figura 13: Inviabilidade de tabelas de página lineares.

Um exemplo permite explicar melhor esse conceito: considerando uma arquitetura de 32 bits com páginas de 4 KBytes, 20 bits são usados para acessar a tabela de páginas. Esses 20 bits podem ser divididos em dois grupos de 10 bits que são usados como índices em uma tabela de páginas com dois níveis:

$$
\begin{aligned}
01805E9A_H &\rightarrow 0000\ 0001\ 1000\ 0000\ 0101\ 1110\ 1001\ 1010_2 \\
&\rightarrow \underbrace{0000\ 0001\ 10_2}_{10\text{ bits}} \text{ e } \underbrace{00\ 0000\ 0101_2}_{10\text{ bits}} \text{ e } \underbrace{1110\ 1001\ 1010_2}_{12\text{ bits}} \\
&\rightarrow 0006_H \text{ e } 0005_H \text{ e } E9A_H \\
&\rightarrow p_1\ 0006_H,\ p_2\ 0005_H \text{ e offset } E9A_H
\end{aligned}
$$

A tradução de endereços lógicos em físicos usando uma tabela de páginas estruturada em dois níveis é efetuada através dos seguintes passos, que são ilustrados na Figura 14:

1. o endereço lógico *el* (0180 $5E9A_H$) é decomposto em um *offset* de 12 bits *o* ($E9A_H$) e dois números de página de 10 bits cada: o número de página de primeiro nível p_1 (006_H) e o número de página de segundo nível p_2 (005_H);

2. o número de página p_1 é usado como índice na tabela de página de primeiro nível, para encontrar o endereço de uma tabela de página de segundo nível;

3. o número de página p_2 é usado como índice na tabela de página de segundo nível, para encontrar o número de quadro *q* ($2F_H$) que corresponde a $[p_1 p_2]$;

4. o número de quadro *q* é combinado ao *offset o* para obter o endereço físico *ef* (0002 $FE9A_H$) correspondente ao endereço lógico solicitado *el*.

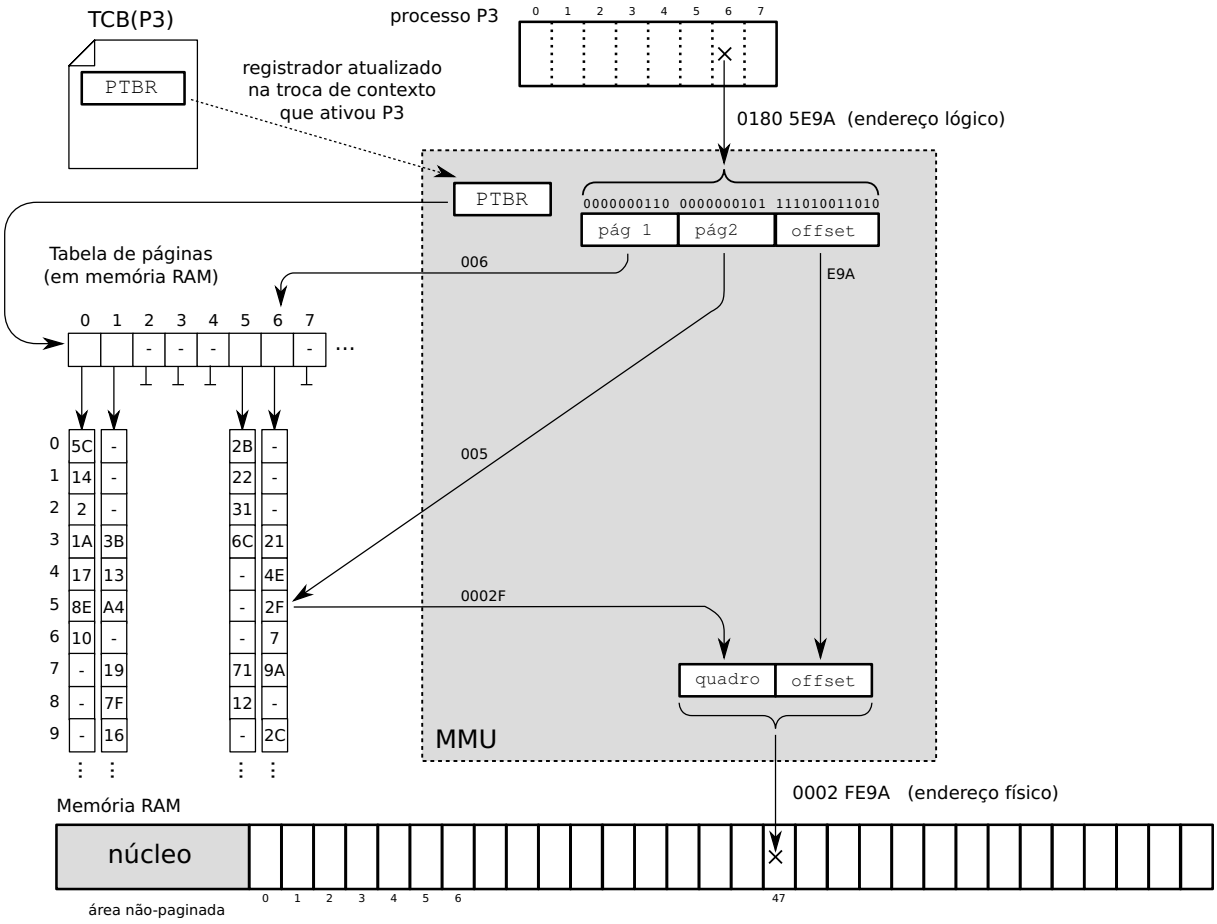

Figura 14: Tabela de página multinível.

Com a estruturação da tabela de páginas em níveis, a quantidade de memória necessária para armazená-la diminui significativamente, sobretudo no caso de processos pequenos. Considerando o processo apresentado como exemplo na Figura 13, ele faria uso de uma tabela de primeiro nível e somente duas tabelas de segundo nível (uma para mapear suas primeiras páginas e outra para mapear suas últimas páginas); todas as demais entradas da tabela de primeiro nível estariam vazias. Assumindo que cada entrada de tabela ocupa 4 bytes, serão necessários somente 12 KBytes para armazenar essas três tabelas ($4 \times 3 \times 2^{10}$ bytes). 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. Essas tabelas ocupariam de $4 \times (2^{10} \times 2^{10} + 2^{10})$ bytes, ou seja, 0,098% a mais que se a tabela de páginas fosse estruturada em um só nível (4×2^{20} bytes). Essas duas situações extremas de alocação estão ilustradas na Figura 15.

3.4.3 Cache da tabela de páginas

A estruturação das tabelas de páginas em vários níveis resolve o problema do espaço ocupado pelas tabelas de forma muito eficiente, mas tem um efeito colateral muito nocivo: aumenta drasticamente o tempo de acesso à memória. Como as tabelas de páginas são armazenadas na memória, cada acesso a um endereço de memória implica

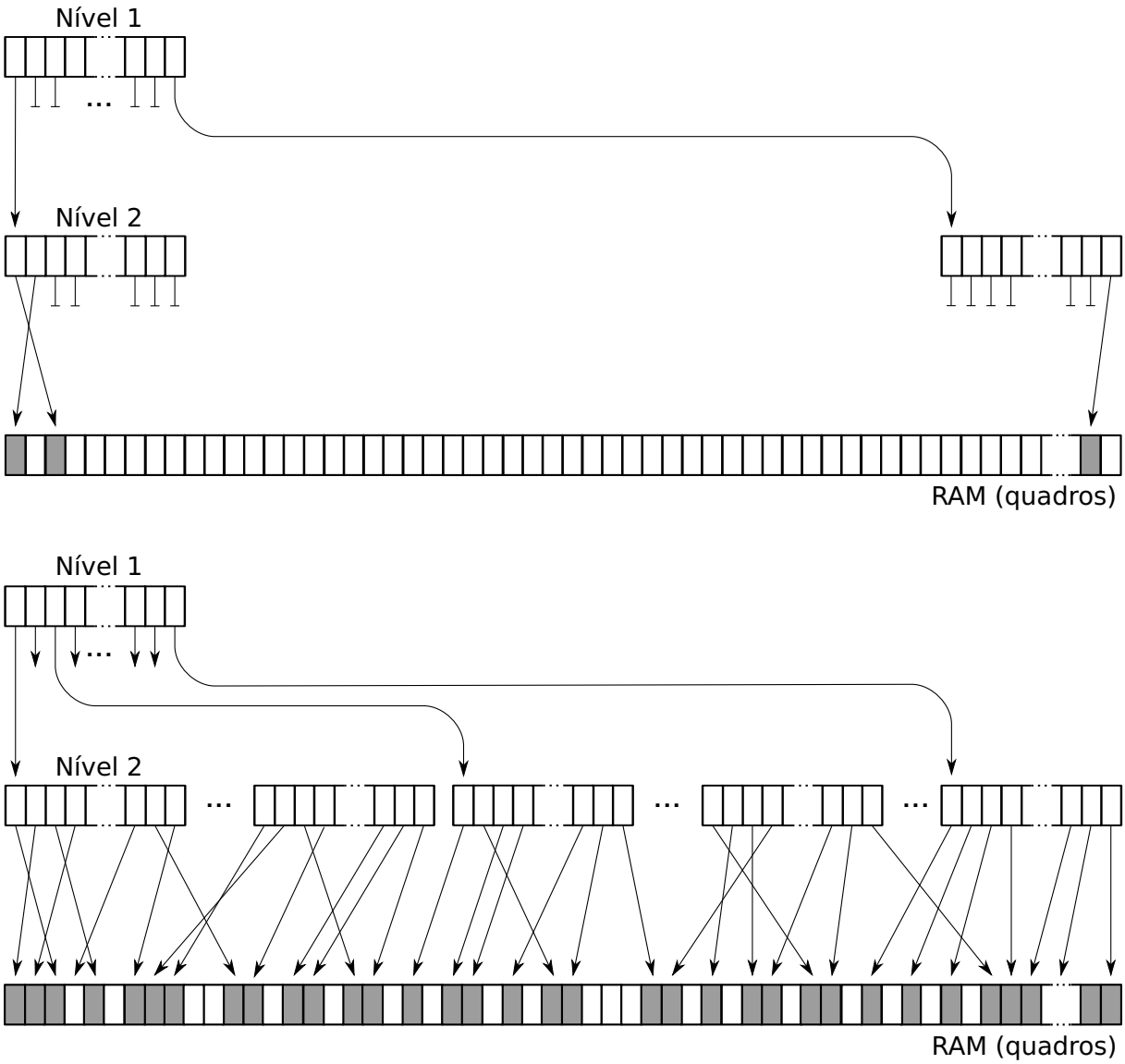

Figura 15: Tabela de páginas multinível vazia (no alto) e cheia (embaixo).

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

Quando um processo executa, ele acessa endereços de memória para buscar instruções e operandos e ler/escrever dados. Em cada instante, os acessos tendem a se concentrar em poucas páginas, que contém o código e as variáveis usadas naquele instante[3]. Dessa forma, a MMU terá de fazer muitas traduções consecutivas de endereços nas mesmas páginas, que irão resultar nos mesmos quadros de memória física. Por isso, consultas recentes à tabela de páginas são armazenadas em um *cache* dentro da própria MMU,

[3]Esse fenômeno é conhecido como *localidade de referências* e será detalhado na Seção 4.

evitando ter de repeti-las constantemente e assim diminuindo o tempo de acesso à memória física.

O cache de tabela de páginas na MMU, denominado TLB (*Translation Lookaside Buffer*) ou *cache associativo*, armazena pares [*página*, *quadro*] obtidos em consultas recentes às tabelas de páginas do processo ativo. Esse cache funciona como uma tabela de *hash*: dado um número de página *p* em sua entrada, ele apresenta em sua saída o número de quadro *q* correspondente, ou um flag de erro chamado *erro de cache* (*cache miss*). Por ser implementado em um hardware especial rápido e caro, geralmente esse cache é pequeno: TLBs de processadores típicos têm entre 16 e 256 entradas. Seu tempo de acesso é pequeno: um acerto custa cerca de 1 ciclo de relógio da CPU, enquanto um erro pode custar entre 10 e 30 ciclos.

A tradução de endereços lógicos em físicos usando TLBs se torna mais rápida, mas também mais complexa. Ao receber um endereço lógico, a MMU consulta o TLB; caso o número do quadro correspondente esteja em cache, ele é usado para compor o endereço físico e o acesso à memória é efetuado. Caso contrário, uma busca normal (completa) na tabela de páginas deve ser realizada. O quadro obtido nessa busca é usado para compor o endereço físico e também é adicionado ao TLB para agilizar as consultas futuras. A Figura 16 apresenta os detalhes desse procedimento.

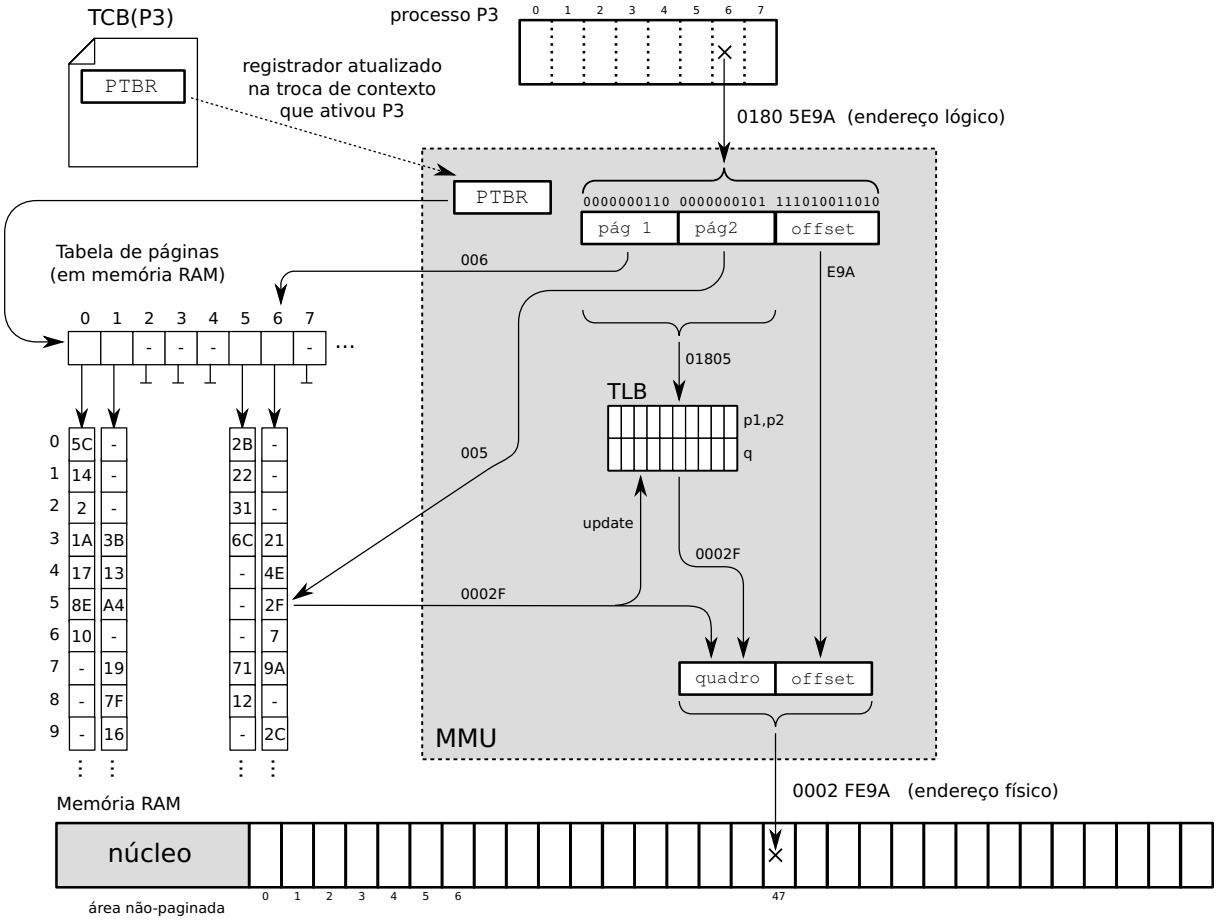

Figura 16: Uso da TLB.

É fácil perceber que, quanto maior a taxa de acertos do TLB (*cache hit ratio*), melhor é o desempenho dos acessos à memória física. O tempo médio de acesso à memória pode então ser determinado pela média ponderada entre o tempo de acesso com acerto de cache e o tempo de acesso no caso de erro. Por exemplo, considerando um sistema operando a 2 GHz (relógio de 0,5 ns) com tempo de acesso à RAM de 50 *ns*, tabelas de páginas com 3 níveis e um TLB com custo de acerto de 0,5 *ns* (um ciclo de relógio), custo de erro de 10 *ns* (20 ciclos de relógio) e taxa de acerto de 95%, o tempo médio de acesso à memória pode ser estimado como segue:

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

$$t_{médio} = 58,475ns$$

Este resultado indica que o sistema de paginação multinível aumenta em 8,475 *ns* (16,9%) o tempo de acesso à memória, o que é razoável considerando-se os benefícios e flexibilidade que esse sistema traz. Todavia, esse 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%.

Obviamente, quanto mais entradas houverem no TLB, melhor será sua taxa de acerto. Contudo, trata-se de um hardware caro e volumoso, por isso os processadores atuais geralmente têm TLBs com poucas entradas (geralmente entre 16 e 256 entradas). 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.

O tamanho do TLB é um fator que influencia a sua taxa de acertos, mas há outros fatores importantes a considerar, como a política de substituição das entradas do TLB. Essa política define o que ocorre quando há um erro de cache e não há entradas livres no TLB: em alguns processadores, a associação [*página*, *quadro*] que gerou o erro é adicionada ao cache, substituindo a entrada mais antiga; todavia, na maioria dos processadores mais recentes, cada erro de cache provoca uma interrupção, que transfere ao sistema operacional a tarefa de gerenciar o conteúdo do TLB [Patterson and Henessy, 2005].

Outro aspecto que influencia significativamente a taxa de acerto do TLB é a forma como cada processo acessa a memória. Processos que concentram seus acessos em poucas páginas de cada vez farão um uso eficiente desse cache, enquanto processos que acessam muitas páginas distintas em um curto período irão gerar frequentes erros de cache, prejudicando seu desempenho no acesso à memória. Essa propriedade é conhecida como *localidade de referência* (Seção 4).

Finalmente, é importante observar que o conteúdo do TLB reflete a tabela de páginas ativa, que indica as páginas de memória pertencentes ao processo em execução naquele momento. 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. Isso permite concluir que trocas de contexto muito frequentes prejudicam a eficiência de acesso à memória, tornando o sistema mais lento.

3.5 Alocação segmentada paginada

Cada uma das principais formas de alocação de memória vistas até agora tem suas vantagens: a alocação contígua prima pela simplicidade e rapidez; a alocação por segmentos oferece múltiplos espaços de endereçamento para cada processo, oferecendo flexibilidade ao programador; a alocação por páginas oferece um grande espaço de endereçamento linear, enquanto elimina a fragmentação externa. Alguns processadores oferecem mais de uma forma de alocação, deixando aos projetistas do sistema operacional a escolha da forma mais adequada de organizar a memória usada por seus processos.

Vários processadores permitem combinar mais de uma forma de alocação. Por exemplo, os processadores *Intel i386* permitem combinar a alocação com segmentos com a alocação por páginas, visando oferecer a flexibilidade da alocação por segmentos com a baixa fragmentação da alocação por páginas.

Nessa abordagem, os processos veem a memória estruturada em segmentos, conforme indicado na Figura 9. O hardware da MMU converte os endereços lógicos na forma [*segmento*:*offset*] para endereços lógicos lineares (unidimensionais), usando as tabelas de descritores de segmentos (Seção 3.3). Em seguida, esse endereços lógicos lineares são convertidos nos endereços físicos correspondentes através do hardware de paginação (tabelas de páginas e TLB), visando obter o endereço físico correspondente.

Apesar do processador *Intel i386* oferece as duas formas de alocação de memória, a maioria dos sistemas operacionais que o suportam não fazem uso de todas as suas possibilidades: os sistemas da família Windows NT (2000, XP, Vista) e também os da família UNIX (Linux, FreeBSD) usam somente a alocação por páginas. O antigo DOS e o Windows 3.* usavam somente a alocação por segmentos. O OS/2 da IBM foi um dos poucos sistemas operacionais comerciais a fazer uso pleno das possibilidades de alocação de memória nessa arquitetura, combinando segmentos e páginas.

4 Localidade de referências

A forma como os processos acessam a memória tem um impacto direto na eficiência dos mecanismos de gerência de memória, sobretudo o cache de páginas (TLB, Seção 3.4.3) e o mecanismo de memória virtual (Seção 7). Processos que concentram seus acessos em poucas páginas de cada vez farão um uso eficiente desses mecanismos, enquanto processos que acessam muitas páginas distintas em um curto período irão gerar frequentes erros de cache (TLB) e faltas de página, prejudicando seu desempenho no acesso à memória.

A propriedade de um processo ou sistema concentrar seus acessos em poucas áreas da memória a cada instante é chamada *localidade de referências* [Denning, 2006]. Existem ao menos três formas de localidade de referências:

Localidade temporal : um recurso usado há pouco tempo será provavelmente usado novamente em um futuro próximo (esta propriedade é usada pelos algoritmos de gerência de memória virtual);

Localidade espacial : um recurso será mais provavelmente acessado se outro recurso próximo a ele já foi acessado (é a propriedade verificada na primeira execução);

Localidade sequencial : é um caso particular da localidade espacial, no qual há uma predominância de acesso sequencial aos recursos (esta propriedade é útil na otimização de sistemas de arquivos).

Como exemplo prático da importância da localidade de referências, considere um programa para o preenchimento de uma matriz de 4.096×4.096 bytes, onde cada linha da matriz está alocada em uma página distinta (considerando páginas de 4.096 bytes). O trecho de código a seguir implementa essa operação, percorrendo a matriz linha por linha:

```
1  unsigned char buffer[4096][4096] ;
2
3  int main ()
4  {
5     int i, j ;
6
7     for (i=0; i<4096; i++) // percorre as linhas do buffer
8        for (j=0; j<4096; j++) // percorre as colunas do buffer
9           buffer[i][j]= (i+j) % 256 ;
10 }
```

Outra implementação possível seria percorrer a matriz coluna por coluna, conforme o código a seguir:

```
1  unsigned char buffer[4096][4096] ;
2
3  int main ()
4  {
5     int i, j ;
6
7     for (j=0; j<4096; j++) // percorre as colunas do buffer
8        for (i=0; i<4096; i++) // percorre as linhas do buffer
9           buffer[i][j]= (i+j) % 256 ;
10 }
```

Embora percorram a matriz de forma distinta, os dois programas geram o mesmo resultado e são conceitualmente equivalentes (a Figura 17 mostra o padrão de acesso à memória dos dois programas). Entretanto, eles não têm o mesmo desempenho. A primeira implementação (percurso linha por linha) usa de forma eficiente o cache da tabela de páginas, porque só gera um erro de cache a cada nova linha acessada. Por outro lado, a implementação com percurso por colunas gera um erro de cache TLB a cada célula acessada, pois o cache TLB não tem tamanho suficiente para armazenar as 4.096 entradas referentes às páginas usadas pela matriz.

A diferença de desempenho entre as duas implementações pode ser grande: em processadores *Intel* e AMD, versões 32 e 64 bits, o primeiro código executa cerca de 10 vezes mais rapidamente que o segundo! Além disso, caso o sistema não tenha memória suficiente para manter as 4.096 páginas em memória, o mecanismo de memória virtual será ativado, fazendo com que a diferença de desempenho seja muito maior.

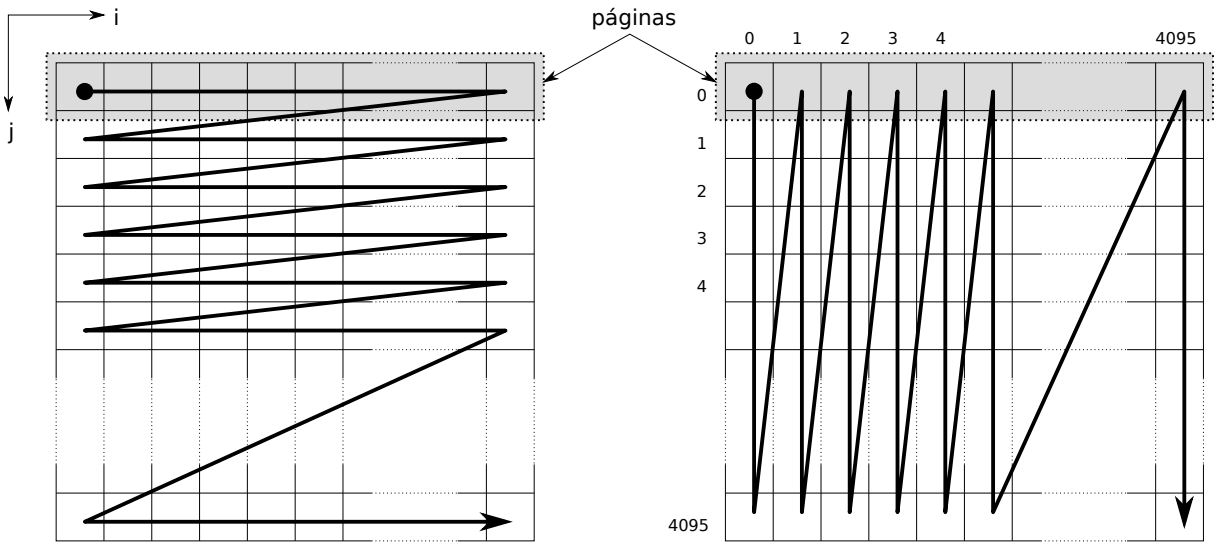

Figura 17: Comportamento dos programas no acesso à memória.

A diferença de comportamento das duas execuções pode ser observada na Figura 18, que mostra a distribuição dos endereços de memória acessados pelos dois códigos[4]. Nos gráficos, percebe-se claramente que a primeira implementação tem uma localidade de referências muito mais forte que a segunda: enquanto a primeira execução usa em média 5 páginas distintas em cada 100.000 acessos à memória, na segunda execução essa média sobe para 3.031 páginas distintas.

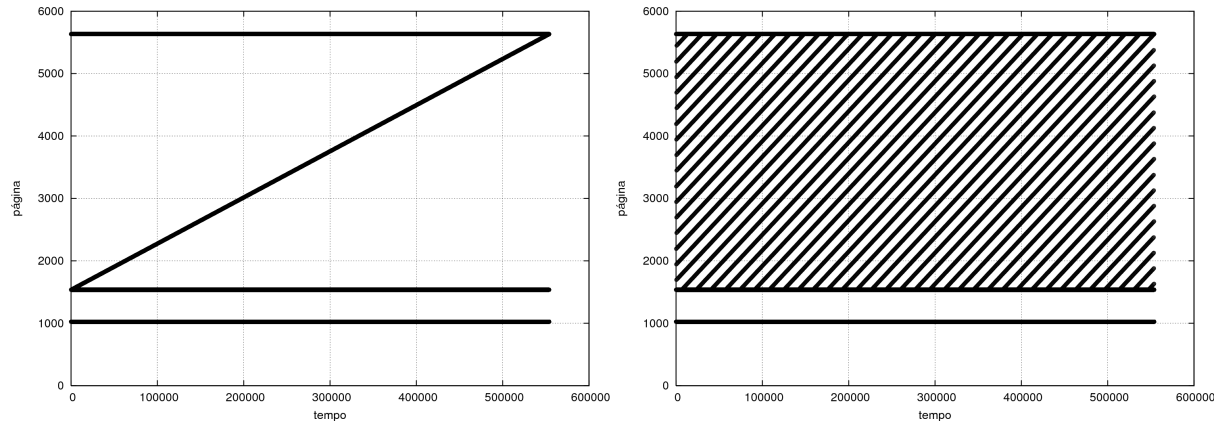

Figura 18: Localidade de referências nas duas execuções.

A Figura 19 traz outro exemplo de boa localidade de referências. Ela mostra as páginas acessadas durante uma execução do visualizador gráfico *gThumb*, ao abrir um arquivo de imagem. O gráfico da esquerda dá uma visão geral da distribuição dos acessos na memória, enquanto o gráfico da direita detalha os acessos da parte inferior, que corresponde às áreas de código, dados e *heap* do processo.

[4]Como a execução total de cada código gera mais de 500 milhões de referências à memória, foi feita uma amostragem da execução para construir os gráficos.

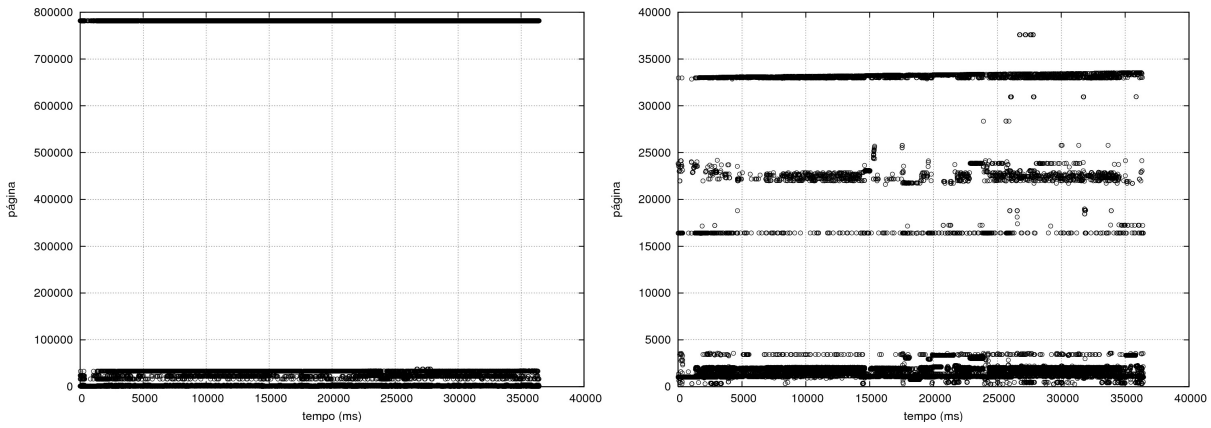

Figura 19: Distribuição dos acessos à memória do programa *gThumb*: visão geral (à esquerda) e detalhe da parte inferior (à direita).

A localidade de referência de uma implementação depende de um conjunto de fatores, que incluem:

- As estruturas de dados usadas pelo programa: estruturas como vetores e matrizes têm seus elementos alocados de forma contígua na memória, o que leva a uma localidade de referências maior que estruturas mais dispersas, como listas encadeadas e árvores;
- Os algoritmos usados pelo programa: o comportamento do programa no acesso à memória é definido pelos algoritmos que ele implementa;
- A qualidade do compilador: cabe ao compilador analisar quais variáveis e trechos de código são usadas com frequência juntos e colocá-los nas mesmas páginas de memória, para aumentar a localidade de referências do código gerado.

A localidade de referências é uma propriedade importante para a construção de programas eficientes. Ela também é útil em outras áreas da computação, como a gerência das páginas armazenadas nos caches de navegadores *web* e servidores *proxy*, nos mecanismos de otimização de leituras/escritas em sistemas de arquivos, na construção da lista "arquivos recentes" dos menus de muitas aplicações interativas, etc.

5 Fragmentação

Ao longo da vida de um sistema, áreas de memória são liberadas por processos que concluem sua execução e outras áreas são alocadas por novos processos, de forma contínua. Com isso, podem surgir áreas livres (vazios ou *buracos* na memória) entre os processos, o que constitui um problema conhecido como *fragmentação externa*. Esse problema somente afeta as estratégias de alocação que trabalham com blocos de tamanho variável, como a alocação contígua e a alocação segmentada. Por outro lado, a alocação paginada sempre trabalha com blocos de mesmo tamanho (os quadros e páginas), sendo por isso imune à fragmentação externa.

A fragmentação externa é prejudicial porque limita a capacidade de alocação de memória no sistema. A Figura 20 apresenta um sistema com alocação contígua de memória no qual ocorre fragmentação externa. Nessa Figura, observa-se que existem 68 MBytes de memória livre em quatro áreas separadas ($A_1 \ldots A_4$), mas somente processos com até 28 MBytes podem ser alocados (usando a maior área livre, A_4). Além disso, quanto mais fragmentada estiver a memória livre, maior o esforço necessário para gerenciá-la: as áreas livres são mantidas em uma lista encadeada de área de memória, que é manipulada a cada pedido de alocação ou liberação de memória.

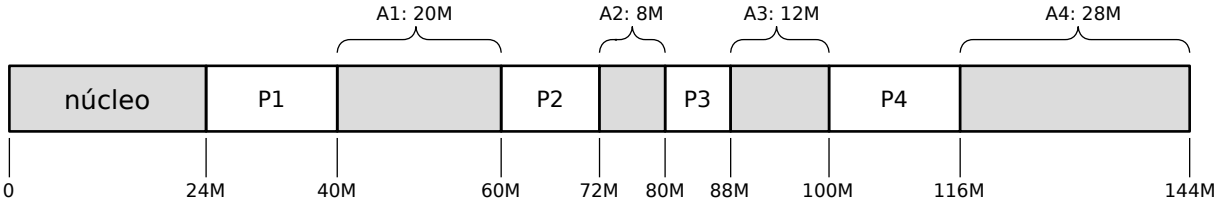

Figura 20: Memória com fragmentação externa.

Pode-se enfrentar o problema da fragmentação externa de duas formas: *minimizando* sua ocorrência, através de critérios de escolha das áreas a alocar, ou *desfragmentando* periodicamente a memória do sistema. Para minimizar a ocorrência de fragmentação externa, cada pedido de alocação deve ser analisado para encontrar a área de memória livre que melhor o atenda. Essa análise pode ser feita usando um dos seguintes critérios:

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.

Diversas pesquisas [Johnstone and Wilson, 1999] demonstraram que as abordagens mais eficientes são a de melhor encaixe e a de primeiro encaixe, sendo esta última bem mais rápida. A Figura 21 ilustra essas estratégias.

Outra forma de tratar a fragmentação externa consiste em *desfragmentar* a memória periodicamente. Para tal, as áreas de memória usadas pelos processos devem ser movidas na memória de forma a concatenar as áreas livres e assim diminuir a fragmentação. Ao mover um processo na memória, suas informações de alocação (registrador base ou tabela de segmentos) devem ser ajustadas para refletir a nova posição do processo.

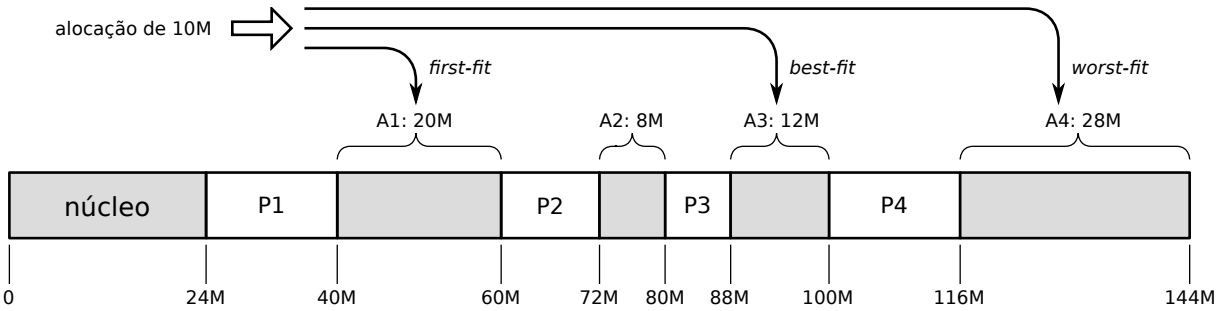

Figura 21: Estratégias para minimizar a fragmentação externa.

Obviamente, nenhum processo pode executar durante a desfragmentação. Portanto, é importante que esse procedimento seja executado rapidamente e com pouca frequência, para não interferir nas atividades normais do sistema. 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. A Figura 22 ilustra três possibilidades de desfragmentação de uma determinada situação de memória; as três alternativas produzem o mesmo resultado, mas apresentam custos distintos.

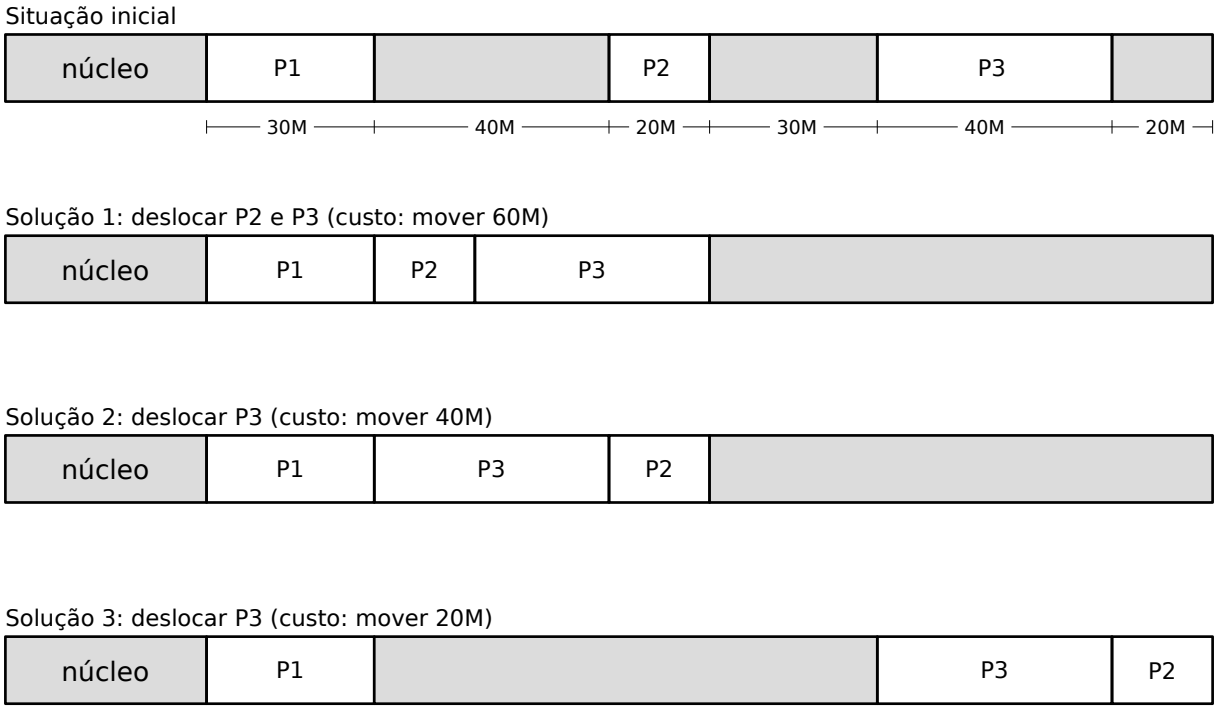

Figura 22: Possibilidades de desfragmentação.

Além da fragmentação externa, que afeta as áreas livres entre os processos, as estratégias de alocação de memória também podem apresentar a *fragmentação interna*, que pode ocorrer dentro das áreas alocadas aos processos. A Figura 23 apresenta uma situação onde ocorre esse problema: um novo processo requisita uma área de memória com 4.900 KBytes. Todavia, a área livre disponível tem 5.000 KBytes. Se for alocada

exatamente a área solicitada pelo processo (situação A), sobrará um fragmento residual com 100 KBytes, que é praticamente inútil para o sistema, pois é muito pequeno para acomodar novos processos. Além disso, essa área residual de 100 KBytes deve ser incluída na lista de áreas livres, o que representa um custo de gerência desnecessário. Outra possibilidade consiste em "arredondar" o tamanho da área solicitada pelo processo para 5.000 KBytes, ocupando totalmente aquela área livre (situação B). Assim, haverá uma pequena área de 100 KBytes no final da memória do processo, que provavelmente não será usada por ele.

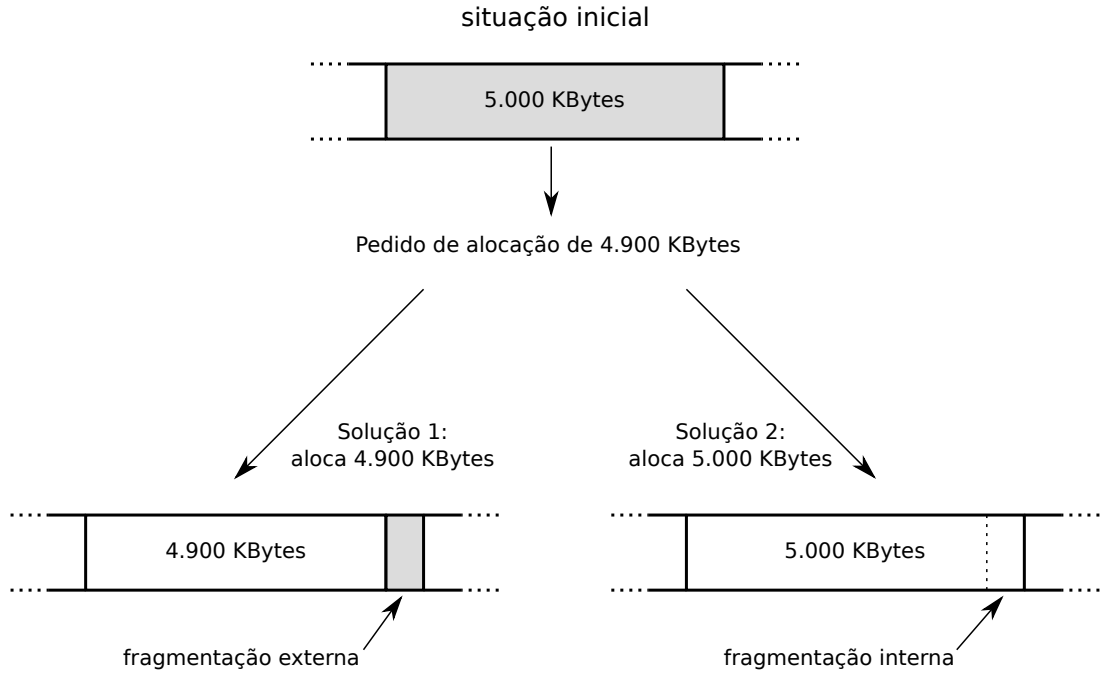

Figura 23: Fragmentação interna.

A fragmentação interna afeta todas as formas de alocação; as alocações contígua e segmentada sofrem menos com esse problema, pois o nível de arredondamento das alocações pode ser decidido caso a caso. No caso da alocação paginada, essa decisão não é possível, pois as alocações são feitas em páginas inteiras. Assim, em um sistema com páginas de 4 KBytes (4.096 bytes), um processo que solicite a alocação de 550.000 bytes (134,284 páginas) receberá 552.960 bytes (135 páginas), ou seja, 2.960 bytes a mais que o solicitado.

Em média, para cada processo haverá uma perda de 1/2 página de memória por fragmentação interna. Assim, uma forma de minimizar a perda por fragmentação interna seria usar páginas de menor tamanho (2K, 1K, 512 bytes ou ainda menos). Todavia, essa abordagem implica em ter mais páginas por processo, o que geraria tabelas de páginas maiores e com maior custo de gerência.

6 Compartilhamento de memória

A memória RAM é um recurso escasso, que deve ser usado de forma eficiente. Nos sistemas atuais, é comum ter várias instâncias do mesmo programa em execução, como várias instâncias de editores de texto, de navegadores, etc. Em servidores, essa situação pode ser ainda mais frequente, com centenas ou milhares de instâncias do mesmo programa carregadas na memória. Por exemplo, em um servidor de e-mail UNIX, cada cliente que se conecta através dos protocolos POP3 ou IMAP terá um processo correspondente no servidor, para atender suas consultas de e-mail (Figura 24). Todos esses processos operam com dados distintos (pois atendem a usuários distintos), mas executam o mesmo código. Assim, centenas ou milhares de cópias do mesmo código executável poderão coexistir na memória do sistema.

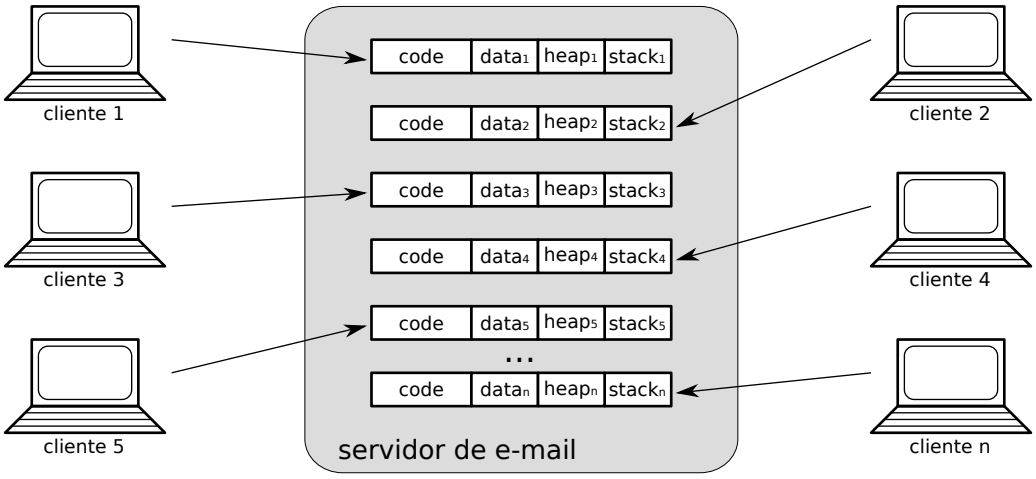

Figura 24: Várias instâncias do mesmo processo.

Conforme visto na Seção 2.2, a estrutura típica da memória de um processo contém áreas separadas para código, dados, pilha e *heap*. Normalmente, a área de código não precisa ter seu conteúdo modificado durante a execução, portanto geralmente essa área é protegida contra escritas (*read-only*). Assim, seria possível *compartilhar* essa área entre todos os processos que executam o mesmo código, economizando memória física.

O compartilhamento de código entre processos pode ser implementado de forma muito simples e transparente para os processos envolvidos, através dos mecanismos de tradução de endereços oferecidos pela MMU, como segmentação e paginação. No caso da segmentação, bastaria fazer com que todos os segmentos de código dos processos apontem para o mesmo segmento da memória física, como indica a Figura 25. É importante observar que o compartilhamento é transparente para os processos: cada processo continua a acessar endereços lógicos em seu próprio segmento de código, buscando suas instruções a executar.

No caso da paginação, a unidade básica de compartilhamento é a página. Assim, as entradas das tabelas de páginas dos processos envolvidos são ajustadas para referenciar os mesmos quadros de memória física. É importante observar que, embora referenciem os mesmos endereços físicos, as páginas compartilhadas podem ter endereços lógicos distintos. A Figura 26 ilustra o compartilhamento de páginas entre processos.

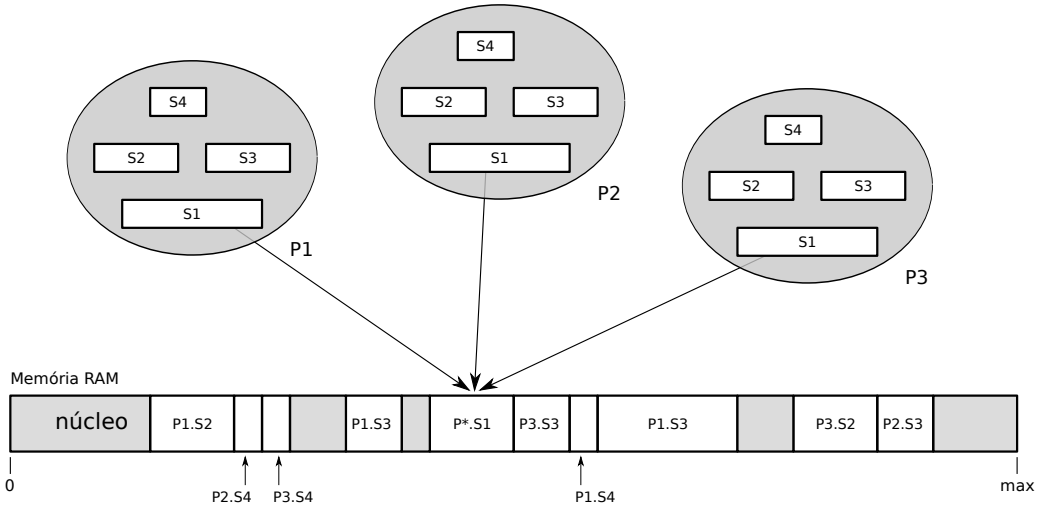

Figura 25: Compartilhamento de segmentos.

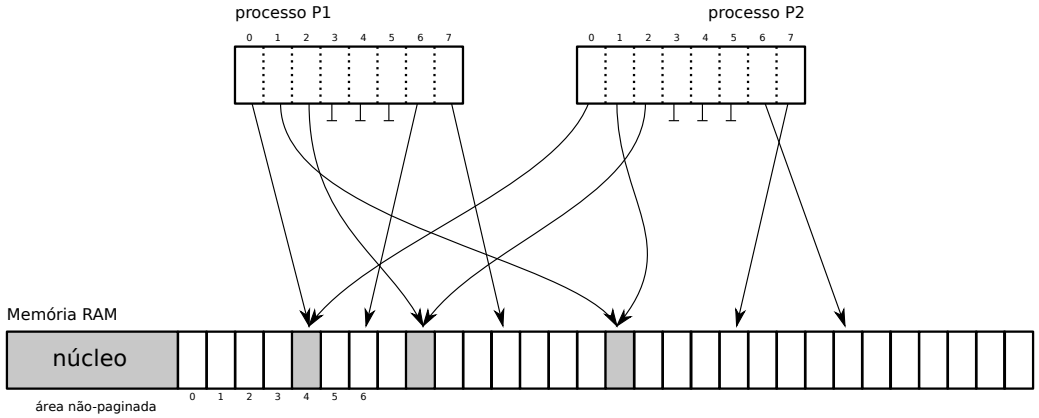

Figura 26: Compartilhamento de páginas.

O compartilhamento das áreas de código permite proporcionar uma grande economia no uso da memória física, sobretudo em servidores e sistemas multiusuários. Por exemplo: consideremos um processador de textos que necessite de 100 MB de memória para executar, dos quais 60 MB são ocupados por código executável. Sem o compartilhamento de áreas de código, 10 instâncias do editor consumiriam 1.000 MB de memória; com o compartilhamento, esse consumo cairia para 460 MB ($60MB + 10 \times 40MB$).

O mecanismo de compartilhamento de memória não é usado apenas com áreas de código; em princípio, toda área de memória protegida contra escrita pode ser compartilhada, o que poderia incluir áreas de dados constantes, como tabelas de constantes, textos de ajuda, etc., proporcionando ainda mais economia de memória.

Uma forma mais agressiva de compartilhamento de memória é proporcionada pelo mecanismo denominado *copiar-ao-escrever* (*COW - Copy-On-Write*). Nele, todas as áreas de memória de um processo (segmentos ou páginas) são passíveis de compartilhamento por outros processos, à condição que ele ainda não tenha modificado seu conteúdo. A ideia central do mecanismo é simples:

1. ao carregar um novo processo em memória, o núcleo protege todas as áreas de memória do processo contra escrita (inclusive dados, pilha e *heap*), usando os flags da tabela de páginas (ou de segmentos);

2. quando o processo tentar escrever na memória, a MMU gera uma interrupção (negação de escrita) para o núcleo do sistema operacional;

3. o sistema operacional ajusta então os flags daquela área para permitir a escrita e devolve a execução ao processo, para ele poder continuar;

4. processos subsequentes idênticos ao primeiro, ao serem carregados em memória, serão mapeados sobre as mesmas áreas de memória física do primeiro processo que ainda estiverem protegidas contra escrita, ou seja, que ainda não foram modificadas por ele;

5. se um dos processos envolvidos tentar escrever em uma dessas áreas compartilhadas, a MMU gera uma interrupção para o núcleo;

6. o núcleo então faz uma cópia separada daquela área física para o processo que deseja escrever nela e desfaz seu compartilhamento, ajustando as tabelas do processo que provocou a interrupção. Os demais processos continuam compartilhando a área inicial.

Todo esse procedimento é feito de forma transparente para os processos envolvidos, visando compartilhar ao máximo as áreas de memória dos processos e assim otimizar o uso da RAM. Esse mecanismo é mais efetivo em sistemas baseados em páginas, porque normalmente as páginas são menores que os segmentos. A maioria dos sistemas operacionais atuais (Linux, Windows, Solaris, FreeBSD, etc.) usa esse mecanismo.

Áreas de memória compartilhada também podem ser usadas para permitir a comunicação entre processos. Para tal, dois ou mais processos solicitam ao núcleo o mapeamento de uma área de memória comum, sobre a qual podem ler e escrever. Como os endereços lógicos acessados nessa área serão mapeados sobre a mesma área de memória física, o que cada processo escrever nessa área poderá ser lido pelos demais, imediatamente. É importante observar que os endereços lógicos em cada processo poderão ser distintos, pois isso depende do mapeamento feito pela tabela de páginas (ou de segmentos) de cada processo; apenas os endereços físicos serão iguais. Portanto, ponteiros (variáveis que contêm endereços lógicos) armazenados na área compartilhada terão significado para o processo que os escreveu, mas não necessariamente para os demais processos que acessam aquela área. A Seção **??** traz informações mais detalhadas sobre a comunicação entre processos através de memória compartilhada.

7 Memória virtual

Um problema constante nos computadores é a disponibilidade de memória física: os programas se tornam cada vez maiores e cada vez mais processos executam simultaneamente, ocupando a memória disponível. Além disso, a crescente manipulação

de informações multimídia (imagens, áudio, vídeo) contribui para esse problema, uma vez que essas informações são geralmente volumosas e seu tratamento exige grandes quantidades de memória livre. Como a memória RAM é um recurso caro (cerca de U$50/GByte no mercado americano, em 2007) e que consome uma quantidade significativa de energia, aumentar sua capacidade nem sempre é uma opção factível.

Observando o comportamento de um sistema computacional, constata-se que nem todos os processos estão constantemente ativos, e que nem todas as áreas de memória estão constantemente sendo usadas. Por isso, as áreas de memória pouco acessadas poderiam ser transferidas para um meio de armazenamento mais barato e abundante, como um disco rígido (U$0,50/GByte) ou um banco de memória *flash* (U$10/GByte)[5], liberando a memória RAM para outros usos. Quando um processo proprietário de uma dessas áreas precisar acessá-la, ela deve ser transferida de volta para a memória RAM. O uso de um armazenamento externo como extensão da memória RAM se chama *memória virtual*; essa estratégia pode ser implementada de forma eficiente e transparente para processos, usuários e programadores.

7.1 Mecanismo básico

Nos primeiros sistemas a implementar estratégias de memória virtual, processos inteiros eram transferidos da memória para o disco rígido e vice-versa. Esse procedimento, denominado *troca* (*swapping*) permite liberar grandes áreas de memória a cada transferência, e se justifica no caso de um armazenamento com tempo de acesso muito elevado, como os antigos discos rígidos. Os sistemas atuais raramente transferem processos inteiros para o disco; geralmente as transferências são feitas por páginas ou grupos de páginas, em um procedimento denominado *paginação* (*paging*), detalhado a seguir.

Normalmente, o mecanismo de memória virtual se baseia em páginas ao invés de segmentos. As páginas têm um tamanho único e fixo, o que permite simplificar os algoritmos de escolha de páginas a remover, os mecanismos de transferência para o disco e também a formatação da área de troca no disco. A otimização desses fatores seria bem mais complexa e menos efetiva caso as operações de troca fossem baseadas em segmentos, que têm tamanho variável.

A ideia central do mecanismo de memória virtual em sistemas com memória paginada consiste em retirar da memória principal as páginas menos usadas, salvando-as em uma área do disco rígido reservada para esse fim. Essa operação é feita periodicamente, de modo reativo (quando a quantidade de memória física disponível cai abaixo de um certo limite) ou proativo (aproveitando os períodos de baixo uso do sistema para retirar da memória as páginas pouco usadas). As páginas a retirar são escolhidas de acordo com algoritmos de substituição de páginas, discutidos na Seção 7.3. As entradas das tabelas de páginas relativas às páginas transferidas para o disco devem então ser ajustadas de forma a referenciar os conteúdos correspondentes no disco rígido. Essa situação está ilustrada de forma simplificada na Figura 27.

O armazenamento externo das páginas pode ser feito em um disco exclusivo para esse fim (usual em servidores de maior porte), em uma partição do disco principal

[5]Estes valores são apenas indicativos, variando de acordo com o fabricante e a tecnologia envolvida.

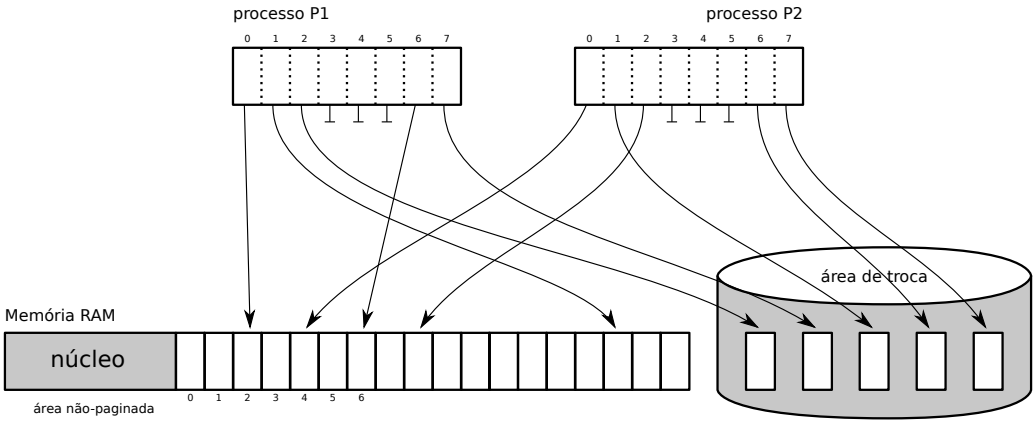

Figura 27: Memória virtual paginada.

(usual no Linux e outros UNIX) ou em um arquivo reservado dentro do sistema de arquivos do disco principal da máquina, geralmente oculto (como no Windows NT e sucessores). Em alguns sistemas, é possível usar uma área de troca remota, em um servidor de arquivos de rede; todavia, essa solução apresenta baixo desempenho. Por razões históricas, essa área de disco é geralmente denominada *área de troca* (*swap area*), embora armazene páginas. No caso de um disco exclusivo ou partição de disco, essa área geralmente é formatada usando uma estrutura de sistema de arquivos otimizada para o armazenamento e recuperação rápida das páginas.

As páginas que foram transferidas da memória para o disco provavelmente serão necessárias no futuro, pois seus processos proprietários provavelmente continuam vivos. Quando um processo tentar acessar uma página ausente, esta deve ser transferida de volta para a memória para permitir seu acesso, de forma transparente ao processo. Conforme exposto na Seção 3.4, quando um processo acessa uma página, a MMU verifica se a mesma está mapeada na memória RAM e, em caso positivo, faz o acesso ao endereço físico correspondente. Caso contrário, a MMU gera uma interrupção de falta de página (*page fault*) que força o desvio da execução para o sistema operacional. Nesse instante, o sistema deve verificar se a página solicitada não existe ou se foi transferida para o disco, usando os flags de controle da respectiva entrada da tabela de páginas. Caso a página não exista, o processo tentou acessar um endereço inválido e deve ser abortado. Por outro lado, caso a página solicitada tenha sido transferida para o disco, o processo deve ser suspenso enquanto o sistema transfere a página de volta para a memória RAM e faz os ajustes necessários na tabela de páginas. Uma vez a página carregada em memória, o processo pode continuar sua execução. O fluxograma da Figura 28 apresenta as principais ações desenvolvidas pelo mecanismo de memória virtual.

Nesse procedimento aparentemente simples há duas questões importantes. Primeiro, caso a memória principal já esteja cheia, uma ou mais páginas deverão ser removidas para o disco antes de trazer de volta a página faltante. Isso implica em mais operações de leitura e escrita no disco e portanto em mais demora para atender o pedido do processo. Muitos sistemas, como o Linux e o Solaris, mantém um processo *daemon*

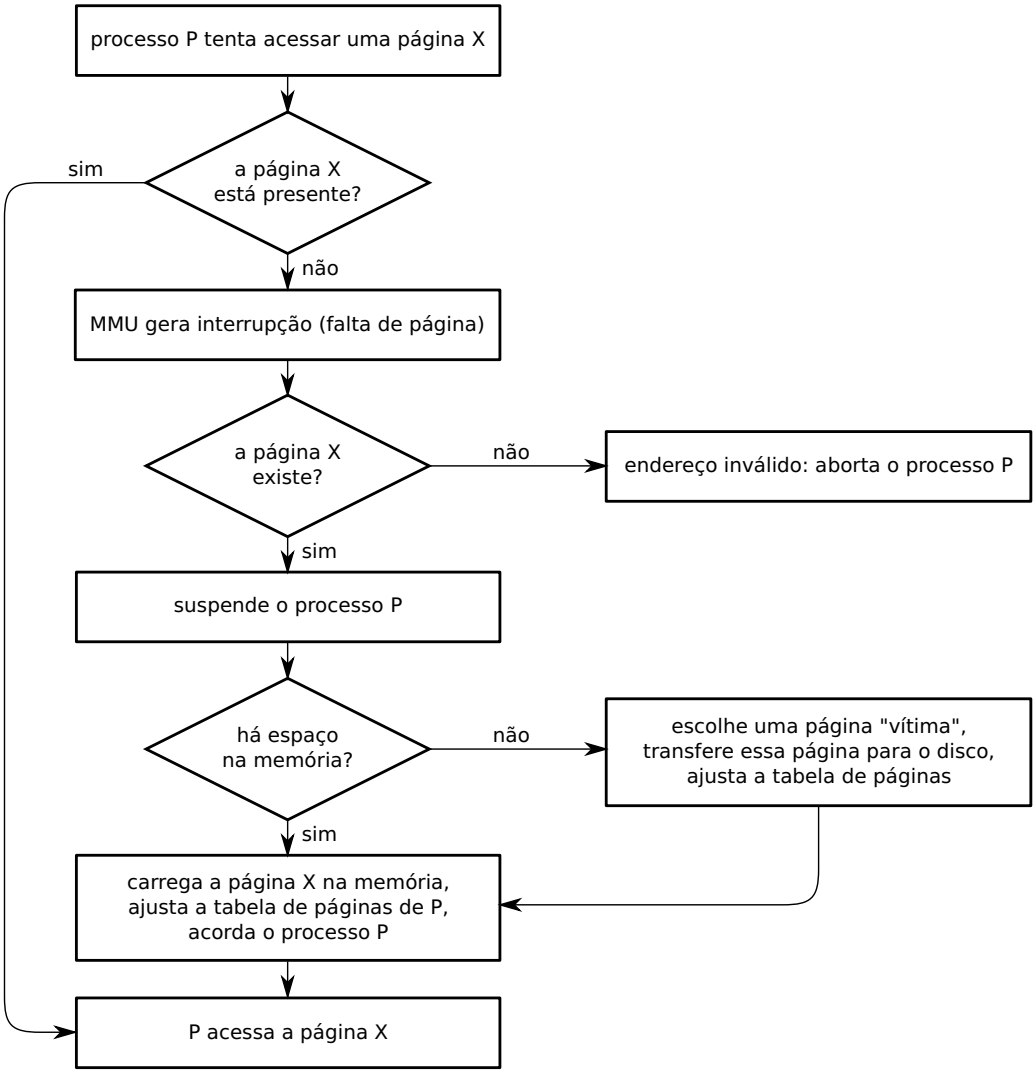

Figura 28: Ações do mecanismo de memória virtual.

com a finalidade de escolher e transferir páginas para o disco, ativado sempre que a quantidade de memória livre estiver abaixo de um limite mínimo.

Segundo, retomar a execução do processo que gerou a falta de página pode ser uma tarefa complexa. Como a instrução que gerou a falta de página não foi completada, ela deve ser reexecutada. No caso de instruções simples, envolvendo apenas um endereço de memória sua reexecução é trivial. Todavia, no caso de instruções que envolvam várias ações e vários endereços de memória, deve-se descobrir qual dos endereços gerou a falta de página, que ações da instrução foram executadas e então executar somente o que estiver faltando. A maioria dos processadores atuais provê registradores especiais que auxiliam nessa tarefa.

7.2 Eficiência de uso

O mecanismo de memória virtual permite usar o disco como uma extensão de memória RAM, de forma transparente para os processos. Seria a solução ideal para as limitações da memória principal, se não houvesse um problema importante: o tempo de acesso dos discos utilizados. Conforme os valores indicados na Tabela 1, um disco rígido típico tem um tempo de acesso cerca de 100.000 vezes maior que a memória RAM. Cada falta de página provocada por um processo implica em um acesso ao disco, para buscar a página faltante (ou dois acessos, caso a memória RAM esteja cheia e outra página tenha de ser removida antes). Assim, faltas de página muito frequentes irão gerar muitos acessos ao disco, aumentando o tempo médio de acesso à memória e, em consequência, diminuindo o desempenho geral do sistema.

Para demonstrar o impacto das faltas de página no desempenho, consideremos um sistema cuja memória RAM tem um tempo de acesso de 60 ns ($60 \times 10^{-9} s$) e cujo disco de troca tem um tempo de acesso de 6 ms ($6 \times 10^{-3} s$), no qual ocorre uma falta de página a cada milhão de acessos (10^6 acessos). Caso a memória não esteja saturada, o tempo médio de acesso será:

$$\begin{aligned} t_{médio} &= \frac{(999.999 \times 60ns) + 6ms + 60ns}{1.000.000} \\ &= \frac{10^6 \times 60 \times 10^{-9} + 6 \times 10^{-3}}{10^6} \\ t_{médio} &= 66ns \end{aligned}$$

Caso a memória esteja saturada, o tempo médio será maior:

$$\begin{aligned} t_{médio} &= \frac{(999.999 \times 60ns) + 2 \times 6ms + 60ns}{1.000.000} \\ &= \frac{10^6 \times 60 \times 10^{-9} + 2 \times 6 \times 10^{-3}}{10^6} \\ t_{médio} &= 72ns \end{aligned}$$

Caso a frequência de falta de páginas aumente para uma falta a cada 100.000 acessos (10^5 acessos), o tempo médio de acesso à memória subirá para 120 ns no primeiro caso (memória não saturada) e 180 ns no segundo caso (memória saturada).

A frequência de faltas de página depende de vários fatores, como:

- O tamanho da memória RAM, em relação à demanda dos processos em execução: sistemas com memória insuficiente, ou muito carregados, podem gerar muitas faltas de página, prejudicando o seu desempenho e podendo ocasionar o fenômeno conhecido como *thrashing* (Seção 7.6).

- o comportamento dos processos em relação ao uso da memória: processos que agrupem seus acessos a poucas páginas em cada momento, respeitando a localidade de referências (Seção 4), necessitam usar menos páginas simultaneamente e geram menos faltas de página.

- A escolha das páginas a remover da memória: caso sejam removidas páginas usadas com muita frequência, estas serão provavelmente acessadas pouco tempo após sua remoção, gerando mais faltas de página. A escolha das páginas a remover é tarefa dos algoritmos apresentados na Seção 7.3.

7.3 Algoritmos de substituição de páginas

A escolha correta das páginas a remover da memória física é um fator essencial para a eficiência do mecanismo de memória virtual. Más escolhas poderão remover da memória páginas muito usadas, aumentando a taxa de faltas de página e e diminuindo o desempenho do sistema. Vários critérios podem ser usados para escolher "vítimas", ou seja, páginas a transferir da memória para a área de troca no disco:

Idade da página : há quanto tempo a página está na memória; páginas muito antigas talvez sejam pouco usadas.

Frequência de acessos à página : páginas muito acessadas em um passado recente possivelmente ainda o serão em um futuro próximo.

Data do último acesso : páginas há mais tempo sem acessar possivelmente serão pouco acessadas em um futuro próximo (sobretudo se os processos respeitarem o princípio da localidade de referências).

Prioridade do processo proprietário : processos de alta prioridade, ou de tempo-real, podem precisar de suas páginas de memória rapidamente; se elas estiverem no disco, seu desempenho ou tempo de resposta poderão ser prejudicados.

Conteúdo da página : páginas cujo conteúdo seja código executável exigem menos esforço do mecanismo de memória virtual, porque seu conteúdo já está mapeado no disco (dentro do arquivo executável correspondente ao processo). Por outro lado, páginas de dados que tenham sido alteradas precisam ser salvas na área de troca.

Páginas especiais : páginas contendo *buffers* de operações de entrada/saída podem ocasionar dificuldades ao núcleo caso não estejam na memória no momento em que ocorrer a transferência de dados entre o processo e o dispositivo físico. O processo também pode solicitar que certas páginas contendo informações sensíveis (como senhas ou chaves criptográficas) não sejam copiadas na área de troca, por segurança.

Existem vários algoritmos para a escolha de páginas a substituir na memória, visando reduzir a frequência de falta de páginas, que levam em conta alguns dos fatores acima enumerados. Os principais serão apresentados na sequência.

Uma ferramenta importante para o estudo desses algoritmos é a *cadeia de referências* (*reference string*), que indica a sequência de páginas acessadas por um processo durante sua execução. Ao submeter a cadeia de referências de uma execução aos vários algoritmos, podemos calcular quantas faltas de página cada um geraria naquela execução em particular e assim comparar sua eficiência.

Cadeias de referências de execuções reais podem ser muito longas: considerando um tempo de acesso à memória de 50 ns, em apenas um segundo de execução ocorrem por volta de 20 milhões de acessos à memória. Além disso, a obtenção de cadeias de referências confiáveis é uma área de pesquisa importante, por envolver técnicas complexas de coleta, filtragem e compressão de dados de execução de sistemas [Uhlig and Mudge, 1997]. Para possibilitar a comparação dos algoritmos de substituição de páginas apresentados na sequência, será usada a seguinte cadeia de referências fictícia, extraída de [Tanenbaum, 2003]:

$$0, 2, 1, 3, 5, 4, 6, 3, 7, 4, 7, 3, 3, 5, 5, 3, 1, 1, 1, 7, 1, 3, 4, 1$$

Deve-se observar que acessos consecutivos a uma mesma página não são relevantes para a análise dos algoritmos, porque somente o primeiro acesso em cada grupo de acessos consecutivos provoca uma falta de página.

7.3.1 Algoritmo FIFO

Um critério básico a considerar para a escolha das páginas a substituir poderia ser sua "idade", ou seja, o tempo em que estão na memória. Assim, páginas mais antigas podem ser removidas para dar lugar a novas páginas. Esse algoritmo é muito simples de implementar: basta organizar as páginas em uma fila de números de páginas com política FIFO (*First In, First Out*). Os números das páginas recém carregadas na memória são registrados no final da lista, enquanto os números das próximas páginas a substituir na memória são obtidos no início da lista.

A aplicação do algoritmo FIFO à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 2. Nesse caso, o algoritmo gera no total 15 faltas de página.

Apesar de ter uma implementação simples, na prática este algoritmo não oferece bons resultados. Seu principal defeito é considerar somente a idade da página, sem levar em conta sua importância. Páginas carregadas na memória há muito tempo podem estar sendo frequentemente acessadas, como é o caso de páginas contendo bibliotecas dinâmicas compartilhadas por muitos processos, ou páginas de processos servidores lançados durante a inicialização (*boot*) da máquina.

7.3.2 Algoritmo Ótimo

Idealmente, a melhor página a remover da memória em um dado instante é aquela que ficará mais tempo sem ser usada pelos processos. Esta ideia simples define o *algoritmo ótimo* (OPT). Entretanto, como o comportamento futuro dos processos não pode ser previsto com precisão, este algoritmo não é implementável. Mesmo assim ele é importante, porque define um limite mínimo conceitual: se para uma dada cadeia

	antes				depois				
t	q_o	q_1	q_2	**página**	q_o	q_1	q_2	**faltas**	**ação realizada**
1	-	-	-	0	[0]	-	-	★	p_0 é carregada no quadro vazio q_0
2	0	-	-	2	0	[2]	-	★	p_2 é carregada em q_1
3	0	2	-	1	0	2	[1]	★	p_1 é carregada em q_2
4	0	2	1	3	[3]	2	1	★	p_3 substitui p_0 (a mais antiga na memória)
5	3	2	1	5	3	[5]	1	★	p_5 substitui p_2 (idem)
6	3	5	1	4	3	5	[4]	★	p_4 substitui p_1
7	3	5	4	6	[6]	5	4	★	p_6 substitui p_3
8	6	5	4	3	6	[3]	4	★	p_3 substitui p_5
9	6	3	4	7	6	3	[7]	★	p_7 substitui p_4
10	6	3	7	4	[4]	3	7	★	p_4 substitui p_6
11	4	3	7	7	4	3	7		p_7 está na memória
12	4	3	7	3	4	3	7		p_3 está na memória
13	4	3	7	3	4	3	7		p_3 está na memória
14	4	3	7	5	4	[5]	7	★	p_5 substitui p_3
15	4	5	7	5	4	5	7		p_5 está na memória
16	4	5	7	3	4	5	[3]	★	p_3 substitui p_7
17	4	5	3	1	[1]	5	3	★	p_1 substitui p_4
18	1	5	3	1	1	5	3		p_1 está na memória
19	1	5	3	1	1	5	3		p_1 está na memória
20	1	5	3	7	1	[7]	3	★	p_7 substitui p_5
21	1	7	3	1	1	7	3		p_1 está na memória
22	1	7	3	3	1	7	3		p_3 está na memória
23	1	7	3	4	1	7	[4]	★	p_4 substitui p_3
24	1	7	4	1	1	7	4		p_1 está na memória

Tabela 2: Aplicação do algoritmo de substituição FIFO.

de referências, o algoritmo ótimo gera 17 faltas de página, nenhum outro algoritmo irá gerar menos de 17 faltas de página ao tratar a mesma cadeia. Assim, seu resultado serve como parâmetro para a avaliação dos demais algoritmos.

A aplicação do algoritmo ótimo à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 3. Nesse caso, o algoritmo gera 11 faltas de página, ou seja, 4 a menos que o algoritmo FIFO.

7.3.3 Algoritmo LRU

Uma aproximação implementável do algoritmo ótimo é proporcionada pelo algoritmo LRU (*Least Recently Used*, menos recentemente usado). Neste algoritmo, a escolha recai sobre as páginas que estão na memória há mais tempo sem ser acessadas. Assim, páginas antigas e menos usadas são as escolhas preferenciais. Páginas antigas mas de uso frequente não são penalizadas por este algoritmo, ao contrário do que ocorre no algoritmo FIFO.

Pode-se observar facilmente que este algoritmo é simétrico do algoritmo OPT em relação ao tempo: enquanto o OPT busca as páginas que serão acessadas "mais longe"

	antes				depois				
t	q_o	q_1	q_2	**página**	q_o	q_1	q_2	**faltas**	**ação realizada**
1	-	-	-	0	[0]	-	-	★	p_0 é carregada no quadro vazio q_0
2	0	-	-	2	0	[2]	-	★	p_2 é carregada em q_1
3	0	2	-	1	0	2	[1]	★	p_1 é carregada em q_2
4	0	2	1	3	[3]	2	1	★	p_3 substitui p_0 (não será mais acessada)
5	3	2	1	5	3	[5]	1	★	p_5 substitui p_2 (não será mais acessada)
6	3	5	1	4	3	5	[4]	★	p_4 substitui p_1 (só será acessada em $t = 17$)
7	3	5	4	6	3	[6]	4	★	p_6 substitui p_5 (só será acessada em $t = 14$)
8	3	6	4	3	3	6	4		p_3 está na memória
9	3	6	4	7	3	[7]	4	★	p_7 substitui p_6 (não será mais acessada)
10	3	7	4	4	3	7	4		p_4 está na memória
11	3	7	4	7	3	7	4		p_7 está na memória
12	3	7	4	3	3	7	4		p_3 está na memória
13	3	7	4	3	3	7	4		p_3 está na memória
14	3	7	4	5	3	7	[5]	★	p_5 substitui p_4 (só será acessada em $t = 23$)
15	3	7	5	5	3	7	5		p_5 está na memória
16	3	7	5	3	3	7	5		p_3 está na memória
17	3	7	5	1	3	7	[1]	★	p_1 substitui p_5 (não será mais acessada)
18	3	7	1	1	3	7	1		p_1 está na memória
19	3	7	1	1	3	7	1		p_1 está na memória
20	3	7	1	7	3	7	1		p_7 está na memória
21	3	7	1	1	3	7	1		p_1 está na memória
22	3	7	1	3	3	7	1		p_3 está na memória
23	3	7	1	4	[4]	7	1	★	p_4 substitui p_3 (não será mais acessada)
24	4	7	1	1	4	7	1		p_1 está na memória

Tabela 3: Aplicação do algoritmo de substituição ótimo.

no futuro do processo, o algoritmo LRU busca as páginas que foram acessadas "mais longe" no seu passado.

A aplicação do algoritmo LRU à cadeia de referências apresentada na Seção anterior, considerando uma memória física com 3 quadros, é apresentada na Tabela 4. Nesse caso, o algoritmo gera 14 faltas de página (três faltas a mais que o algoritmo ótimo).

O gráfico da Figura 29 permite a comparação dos algoritmos OPT, FIFO e LRU sobre a cadeia de referências em estudo, em função do número de quadros existentes na memória física. Pode-se observar que o melhor desempenho é do algoritmo OPT, enquanto o pior desempenho é proporcionado pelo algoritmo FIFO.

O algoritmo LRU parte do pressuposto que páginas recentemente acessadas no passado provavelmente serão acessadas em um futuro próximo, e então evita removê-las da memória. Esta hipótese se verifica na prática, sobretudo se os processos respeitam o princípio da localidade de referência (Seção 4).

Embora possa ser implementado, o algoritmo LRU básico é pouco usado na prática, porque sua implementação exigiria registrar as datas de acesso às páginas a cada leitura ou escrita na memória, o que é difícil de implementar de forma eficiente em software e com custo proibitivo para implementar em hardware. Além disso, sua implementação exigiria varrer as datas de acesso de todas as páginas para buscar a página com acesso

	antes				depois				
t	q_o	q_1	q_2	**página**	q_o	q_1	q_2	**faltas**	**ação realizada**
1	-	-	-	0	[0]	-	-	★	p_0 é carregada no quadro vazio q_0
2	0	-	-	2	0	[2]	-	★	p_2 é carregada em q_1
3	0	2	-	1	0	2	[1]	★	p_1 é carregada em q_2
4	0	2	1	3	[3]	2	1	★	p_3 substitui p_0 (há mais tempo sem acessos)
5	3	2	1	5	3	[5]	1	★	p_5 substitui p_2 (idem)
6	3	5	1	4	3	5	[4]	★	p_4 substitui p_1 (...)
7	3	5	4	6	[6]	5	4	★	p_6 substitui p_3
8	6	5	4	3	6	[3]	4	★	p_3 substitui p_5
9	6	3	4	7	6	3	[7]	★	p_7 substitui p_4
10	6	3	7	4	[4]	3	7	★	p_4 substitui p_6
11	4	3	7	7	4	3	7		p_7 está na memória
12	4	3	7	3	4	3	7		p_3 está na memória
13	4	3	7	3	4	3	7		p_3 está na memória
14	4	3	7	5	[5]	3	7	★	p_5 substitui p_4
15	5	3	7	5	5	3	7		p_5 está na memória
16	5	3	7	3	5	3	7		p_3 está na memória
17	5	3	7	1	5	3	[1]	★	p_1 substitui p_7
18	5	3	1	1	5	3	1		p_1 está na memória
19	5	3	1	1	5	3	1		p_1 está na memória
20	5	3	1	7	[7]	3	1	★	p_7 substitui p_5
21	7	3	1	1	7	3	1		p_1 está na memória
22	7	3	1	3	7	3	1		p_3 está na memória
23	7	3	1	4	[4]	3	1	★	p_4 substitui p_7
24	4	3	1	1	4	3	1		p_1 está na memória

Tabela 4: Aplicação do algoritmo de substituição LRU.

mais antigo (ou manter uma lista de páginas ordenadas por data de acesso), o que exigiria muito processamento. Portanto, a maioria dos sistemas operacionais reais implementa algoritmos baseados em aproximações do LRU.

As próximas seções apresentam alguns algoritmos simples que permitem se aproximar do comportamento LRU. Por sua simplicidade, esses algoritmos têm desempenho limitado e por isso somente são usados em sistemas operacionais mais simples. Como exemplos de algoritmos de substituição de páginas mais sofisticados e com maior desempenho podem ser citados o LIRS [Jiang and Zhang, 2002] e o ARC [Bansal and Modha, 2004].

7.3.4 Algoritmo da segunda chance

O algoritmo FIFO move para a área de troca as páginas há mais tempo na memória, sem levar em conta seu histórico de acessos. Uma melhoria simples desse algoritmo consiste em analisar o bit de referência (Seção 3.4.1) de cada página candidata, para saber se ela foi acessada recentemente. Caso tenha sido, essa página recebe uma "segunda chance", voltando para o fim da fila com seu bit de referência ajustado para zero. Dessa forma, evita-se substituir páginas antigas mas muito acessadas. Todavia, caso todas as

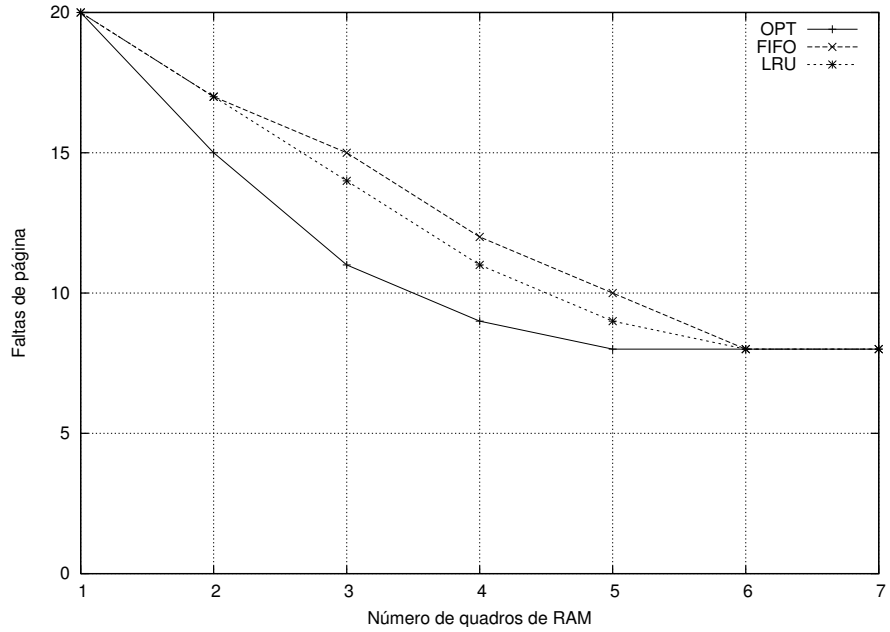

Figura 29: Comparação dos algoritmos de substituição de páginas.

páginas sejam muito acessadas, o algoritmo vai varrer todas as páginas, ajustar todos os bits de referência para zero e acabará por escolher a primeira página da fila, como faria o algoritmo FIFO.

Uma forma eficiente de implementar este algoritmo é através de uma fila circular de números de página, ordenados de acordo com seu ingresso na memória. Um ponteiro percorre a fila sequencialmente, analisando os bits de referência das páginas e ajustando-os para zero à medida em que avança. Quando uma página vítima é encontrada, ela é movida para o disco e a página desejada é carregada na memória no lugar da vítima, com seu bit de referência ajustado para zero. Essa implementação é conhecida como *algoritmo do relógio* e pode ser vista na Figura 30.

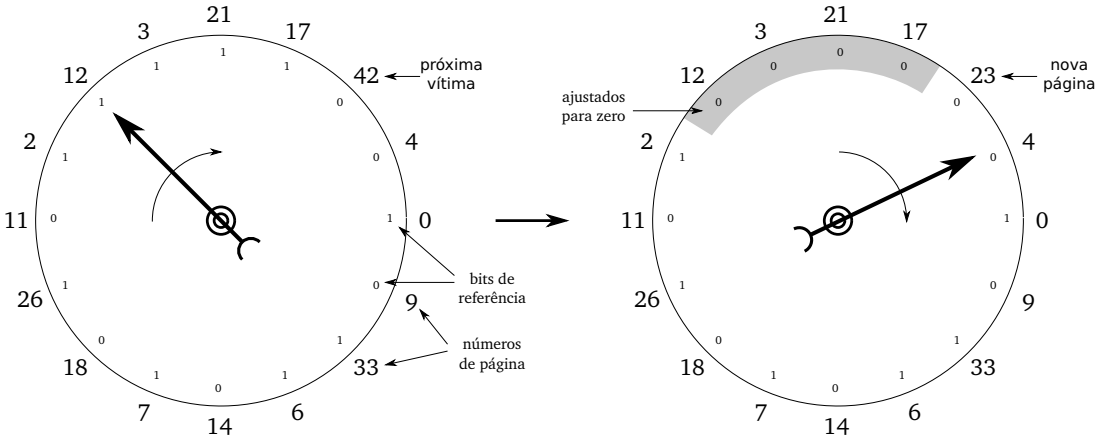

Figura 30: Algoritmo da segunda chance (ou do relógio).

7.3.5 Algoritmo NRU

O algoritmo da segunda chance leva em conta somente o bit de referência de cada página ao escolher as vítimas para substituição. O algoritmo NRU (*Not Recently Used*, ou *não usada recentemente*) melhora essa escolha, ao considerar também o bit de modificação (*dirty bit*, vide Seção 3.4.1), que indica se o conteúdo de uma página foi modificado após ela ter sido carregada na memória.

Usando os bits R (referência) e M (modificação), é possível classificar as páginas em memória em quatro níveis de importância:

- 00 ($R = 0, M = 0$): páginas que não foram referenciadas recentemente e cujo conteúdo não foi modificado. São as melhores candidatas à substituição, pois podem ser simplesmente retiradas da memória.
- 01 ($R = 0, M = 1$): páginas que não foram referenciadas recentemente, mas cujo conteúdo já foi modificado. Não são escolhas tão boas, porque terão de ser gravadas na área de troca antes de serem substituídas.
- 10 ($R = 1, M = 0$): páginas referenciadas recentemente, cujo conteúdo permanece inalterado. São provavelmente páginas de código que estão sendo usadas ativamente e serão referenciadas novamente em breve.
- 11 ($R = 1, M = 1$): páginas referenciadas recentemente e cujo conteúdo foi modificado. São a pior escolha, porque terão de ser gravadas na área de troca e provavelmente serão necessárias em breve.

O algoritmo NRU consiste simplesmente em tentar substituir primeiro páginas do nível 0; caso não encontre, procura candidatas no nível 1 e assim sucessivamente. Pode ser necessário percorrer várias vezes a lista circular até encontrar uma página adequada para substituição.

7.3.6 Algoritmo do envelhecimento

Outra possibilidade de melhoria do algoritmo da segunda chance consiste em usar os bits de referência das páginas para construir *contadores de acesso* às mesmas. A cada página é associado um contador inteiro com N bits (geralmente 8 bits são suficientes). Periodicamente, o algoritmo varre as tabelas de páginas, lê os bits de referência e agrega seus valores aos contadores de acessos das respectivas páginas. Uma vez lidos, os bits de referência são ajustados para zero, para registrar as referências de páginas que ocorrerão durante próximo período.

O valor lido de cada bit de referência não deve ser simplesmente somado ao contador, por duas razões: o contador chegaria rapidamente ao seu valor máximo (*overflow*) e a simples soma não permitiria diferenciar acessos recentes dos mais antigos. Por isso, outra solução foi encontrada: cada contador é deslocado para a direita 1 bit, descartando o bit menos significativo (LSB - *Least Significant Bit*). Em seguida, o valor do bit de referência é colocado na primeira posição à esquerda do contador, ou seja, em seu bit mais significativo (MSB - *Most Significant Bit*). Dessa forma, acessos mais recentes

têm um peso maior que acessos mais antigos, e o contador nunca ultrapassa seu valor máximo.

O exemplo a seguir mostra a evolução dos contadores para quatro páginas distintas, usando os valores dos respectivos bits de referência R. Os valores decimais dos contadores estão indicados entre parênteses, para facilitar a comparação. Observe que as páginas acessadas no último período (p_2 e p_4) têm seus contadores aumentados, enquanto aquelas não acessadas (p_1 e p_3) têm seus contadores diminuídos.

$$\begin{array}{c} \\ p_1 \\ p_2 \\ p_3 \\ p_4 \end{array}\!\!\begin{bmatrix} R \\ \boxed{0} \\ \boxed{1} \\ \boxed{0} \\ \boxed{1} \end{bmatrix} \text{ com } \begin{bmatrix} \textit{contadores} & \\ \boxed{0000\ 0011} & (3) \\ \boxed{0011\ 1101} & (61) \\ \boxed{1010\ 1000} & (168) \\ \boxed{1110\ 0011} & (227) \end{bmatrix} \Longrightarrow \begin{bmatrix} R \\ \boxed{0} \\ \boxed{0} \\ \boxed{0} \\ \boxed{0} \end{bmatrix} \text{ e } \begin{bmatrix} \textit{contadores} & \\ \boxed{\mathbf{0}000\ 0001} & (1) \\ \boxed{\mathbf{1}001\ 1110} & (158) \\ \boxed{\mathbf{0}101\ 0100} & (84) \\ \boxed{\mathbf{1}111\ 0001} & (241) \end{bmatrix}$$

O contador construído por este algoritmo constitui uma aproximação razoável do algoritmo LRU: páginas menos acessadas "envelhecerão", ficando com contadores menores, enquanto páginas mais acessadas permanecerão "jovens", com contadores maiores. Por essa razão, esta estratégia é conhecida como *algoritmo do envelhecimento* [Tanenbaum, 2003], ou *algoritmo dos bits de referência adicionais* [Silberschatz et al., 2001].

7.4 Conjunto de trabalho

A localidade de referências (estudada na Seção 4) mostra que os processos normalmente acessam apenas uma pequena fração de suas páginas a cada instante. O conjunto de páginas acessadas na história recente de um processo é chamado *Conjunto de Trabalho* (*Working Set*, ou *ws*) [Denning, 1980, Denning, 2006]. A composição do conjunto de trabalho é dinâmica, variando à medida em que o processo executa e evolui seu comportamento, acessando novas páginas e deixando de acessar outras. Para ilustrar esse conceito, consideremos a cadeia de referências apresentada na Seção 7.3. Considerando como história recente as últimas n páginas acessadas pelo processo, a evolução do conjunto de trabalho *ws* do processo que gerou aquela cadeia é apresentada na Tabela 5.

O tamanho e a composição do conjunto de trabalho dependem do número de páginas consideradas em sua história recente (o valor n na Tabela 5). Em sistemas reais, essa dependência não é linear, mas segue uma proporção exponencial inversa, devido à localidade de referências. Por essa razão, a escolha precisa do tamanho da história recente a considerar não é crítica. Esse fenômeno pode ser observado na Tabela 5: assim que a localidade de referências se torna mais forte (a partir de $t = 12$), os três conjuntos de trabalho ficam muito similares. Outro exemplo é apresentado na Figura 31, que mostra o tamanho médio dos conjuntos de trabalhos observados na execução do programa *gThumb* (analisado na Seção 4), em função do tamanho da história recente considerada (em número de páginas referenciadas).

Se um processo tiver todas as páginas de seu conjunto de trabalho carregadas na memória, ele sofrerá poucas faltas de página, pois somente acessos a novas páginas poderão gerar faltas. Essa constatação permite delinear um algoritmo simples para

t	página	$ws(n=3)$	$ws(n=4)$	$ws(n=5)$
1	0	{0}	{0}	{0}
2	2	{0, 2}	{0, 2}	{0, 2}
3	1	{0, 2, 1}	{0, 2, 1}	{0, 2, 1}
4	3	{2, 1, 3}	{0, 2, 1, 3}	{0, 2, 1, 3}
5	5	{1, 3, 5}	{2, 1, 3, 5}	{0, 2, 1, 3, 5}
6	4	{3, 5, 4}	{1, 3, 5, 4}	{2, 1, 3, 5, 4}
7	6	{5, 4, 6}	{3, 5, 4, 6}	{1, 3, 5, 4, 6}
8	3	{4, 6, 3}	{5, 4, 6, 3}	{5, 4, 6, 3}
9	7	{6, 3, 7}	{4, 6, 3, 7}	{5, 4, 6, 3, 7}
10	4	{3, 7, 4}	{6, 3, 7, 4}	{6, 3, 7, 4}
11	7	{4, 7}	{3, 4, 7}	{6, 3, 4, 7}
12	3	{4, 7, 3}	{4, 7, 3}	{4, 7, 3}
13	3	{7, 3}	{4, 7, 3}	{4, 7, 3}
14	5	{3, 5}	{7, 3, 5}	{4, 7, 3, 5}
15	5	{3, 5}	{3, 5}	{7, 3, 5}
16	3	{5, 3}	{5, 3}	{5, 3}
17	1	{5, 3, 1}	{5, 3, 1}	{5, 3, 1}
18	1	{3, 1}	{5, 3, 1}	{5, 3, 1}
19	1	{1}	{3, 1}	{5, 3, 1}
20	7	{1, 7}	{1, 7}	{3, 1, 7}
21	1	{7, 1}	{7, 1}	{3, 7, 1}
22	3	{7, 1, 3}	{7, 1, 3}	{7, 1, 3}
23	4	{1, 3, 4}	{7, 1, 3, 4}	{7, 1, 3, 4}
24	1	{3, 4, 1}	{3, 4, 1}	{7, 3, 4, 1}

Tabela 5: Conjuntos de trabalho *ws* para $n = 3$, $n = 4$ e $n = 5$.

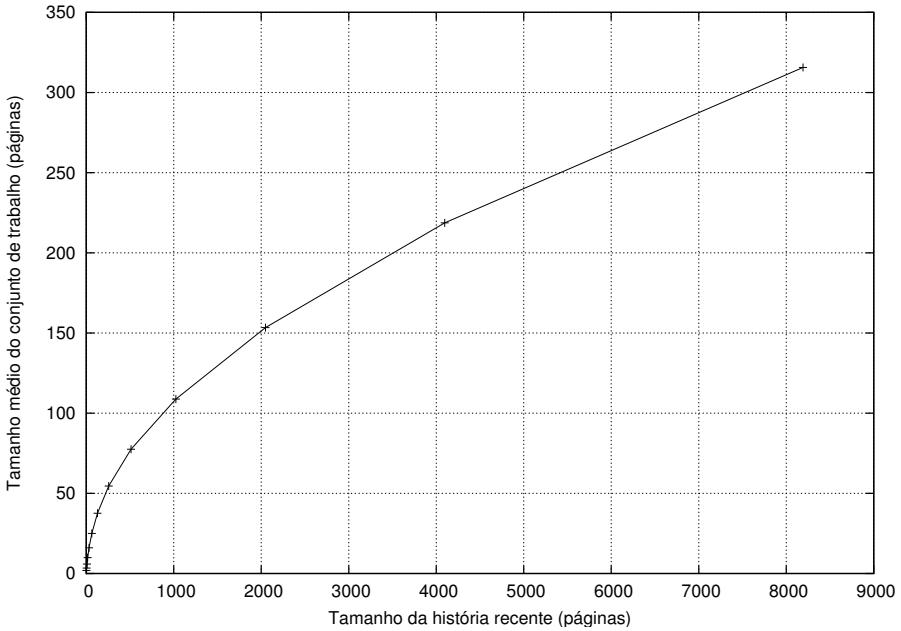

Figura 31: Tamanho do conjunto de trabalho do programa *gThumb*.

substituição de páginas: só substituir páginas que não pertençam ao conjunto de

trabalho de nenhum processo ativo. Contudo, esse algoritmo é difícil de implementar, pois exigiria manter atualizado o conjunto de trabalho de cada processo a cada acesso à memória, o que teria um custo computacional proibitivo.

Uma alternativa mais simples e eficiente de implementar seria verificar que páginas cada processo acessou recentemente, usando a informação dos respectivos bits de referência. Essa é a base do algoritmo *WSClock* (*Working Set Clock*) [Carr and Hennessy, 1981], que modifica o algoritmo do relógio (Seção 7.3.4) da seguinte forma:

1. Uma data de último acesso $t_a(p)$ é associada a cada página p da fila circular; essa data pode ser atualizada quando o ponteiro do relógio visita a página.

2. Define-se um prazo de validade τ para as páginas, geralmente entre dezenas e centenas de milissegundos; a "idade" $i(p)$ de uma página p é definida como a diferença entre sua data de último acesso $t_a(p)$ e o instante corrente t_c ($i(p) = t_c - t_a(p)$).

3. Quando há necessidade de substituir páginas, o ponteiro percorre a fila buscando páginas candidatas:

   (a) Ao encontrar uma página referenciada (com $R = 1$), sua data de último acesso é atualizada com o valor corrente do tempo ($t_a(p) = t_c$), seu bit de referência é limpo ($R = 0$) e o ponteiro do relógio avança, ignorando aquela página.

   (b) Ao encontrar uma página não referenciada (com $R = 0$), se sua idade for menor que τ, a página está no conjunto de trabalho; caso contrário, ela é considerada fora do conjunto de trabalho e pode ser substituída.

4. Caso o ponteiro dê uma volta completa na fila e não encontre páginas com idade maior que τ, a página mais antiga (que tiver o menor $t_a(p)$) encontrada na volta anterior é substituída.

5. Em todas as escolhas, dá-se preferência a páginas não modificadas ($M = 0$), pois seu conteúdo já está salvo no disco.

O algoritmo *WSClock* pode ser implementado de forma eficiente, porque a data último acesso de cada página não precisa ser atualizada a cada acesso à memória, mas somente quando a referência da página na fila circular é visitada pelo ponteiro do relógio (caso $R = 1$). Todavia, esse algoritmo não é uma implementação "pura" do conceito de conjunto de trabalho, mas uma composição de conceitos de vários algoritmos: FIFO e segunda chance (estrutura e percurso do relógio), Conjuntos de trabalho (divisão das páginas em dois grupos conforme sua idade), LRU (escolha das páginas com datas de acesso mais antigas) e NRU (preferência às páginas não modificadas).

7.5 A anomalia de Belady

Espera-se que, quanto mais memória física um sistema possua, menos faltas de página ocorram. Todavia, esse comportamento intuitivo não se verifica em todos os algoritmos de substituição de páginas. Alguns algoritmos, como o FIFO, podem apresentar um comportamento estranho: ao aumentar o número de quadros de memória,

o número de faltas de página geradas pelo algoritmo aumenta, ao invés de diminuir. Esse comportamento atípico de alguns algoritmos foi estudado pelo matemático húngaro Laslo Belady nos anos 60, sendo por isso denominado *anomalia de Belady*.

A seguinte cadeia de referências permite observar esse fenômeno; o comportamento dos algoritmos OPT, FIFO e LRU ao processar essa cadeia pode ser visto na Figura 32, que exibe o número de faltas de página em função do número de quadros de memória disponíveis no sistema. A anomalia pode ser observada no algoritmo FIFO: ao aumentar a memória de 4 para 5 quadros, esse algoritmo passa de 22 para 24 faltas de página.

$$0, 1, 2, 3, 4, 0, 1, 2, 5, 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 0, 1, 2, 5, 0, 1, 2, 3, 4, 5$$

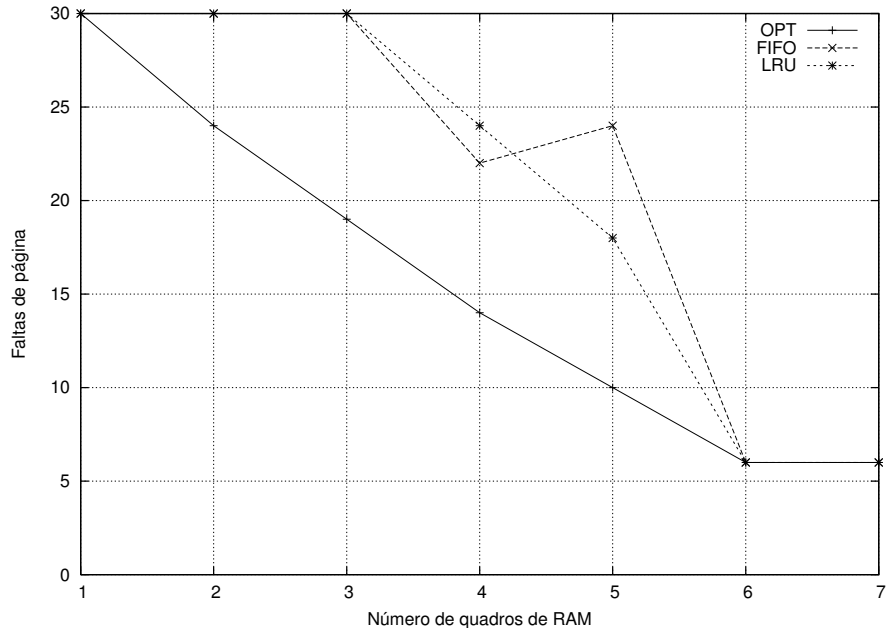

Figura 32: A anomalia de Belady.

Estudos demonstraram que uma família de algoritmos denominada *algoritmos de pilha* (à qual pertencem os algoritmos OPT e LRU, entre outros) não apresenta a anomalia de Belady [Tanenbaum, 2003].

7.6 Thrashing

Na Seção 7.2, foi demonstrado que o tempo médio de acesso à memória RAM aumenta significativamente à medida em que aumenta a frequência de faltas de página. Caso a frequência de faltas de páginas seja muito elevada, o desempenho do sistema como um todo pode ser severamente prejudicado.

Conforme discutido na Seção 7.4, cada processo tem um conjunto de trabalho, ou seja, um conjunto de páginas que devem estar na memória para sua execução naquele momento. Se o processo tiver uma boa localidade de referência, esse conjunto é pequeno e varia lentamente. Caso a localidade de referência seja ruim, o conjunto de trabalho

geralmente é grande e muda rapidamente. Enquanto houver espaço na memória RAM para os conjuntos de trabalho dos processos ativos, não haverão problemas. Contudo, caso a soma de todos os conjuntos de trabalho dos processos prontos para execução seja maior que a memória RAM disponível no sistema, poderá ocorrer um fenômeno conhecido como *thrashing* [Denning, 1980, Denning, 2006].

No *thrashing*, a memória RAM não é suficiente para todos os processos ativos, portanto muitos processos não conseguem ter seus conjuntos de trabalho totalmente carregados na memória. Cada vez que um processo recebe o processador, executa algumas instruções, gera uma falta de página e volta ao estado suspenso, até que a página faltante seja trazida de volta à RAM. Todavia, para trazer essa página à RAM será necessário abrir espaço na memória, transferindo algumas páginas (de outros processos) para o disco. Quanto mais processos estiverem nessa situação, maior será a atividade de paginação e maior o número de processos no estado suspenso, aguardando páginas.

A Figura 33 ilustra o conceito de *thrashing*: ela mostra a taxa de uso do processador (quantidade de processos na fila de prontos) em função do número de processos ativos no sistema. Na zona à esquerda não há *thrashing*, portanto a taxa de uso do processador aumenta com o aumento do número de processos. Caso esse número aumente muito, a memória RAM não será suficiente para todos os conjuntos de trabalho e o sistema entra em uma região de *thrashing*: muitos processos passarão a ficar suspensos aguardando a paginação, diminuindo a taxa de uso do processador. Quanto mais processos ativos, menos o processador será usado e mais lento ficará o sistema. Pode-se observar que um sistema ideal com memória infinita não teria esse problema, pois sempre haveria memória suficiente para todos os processos ativos.

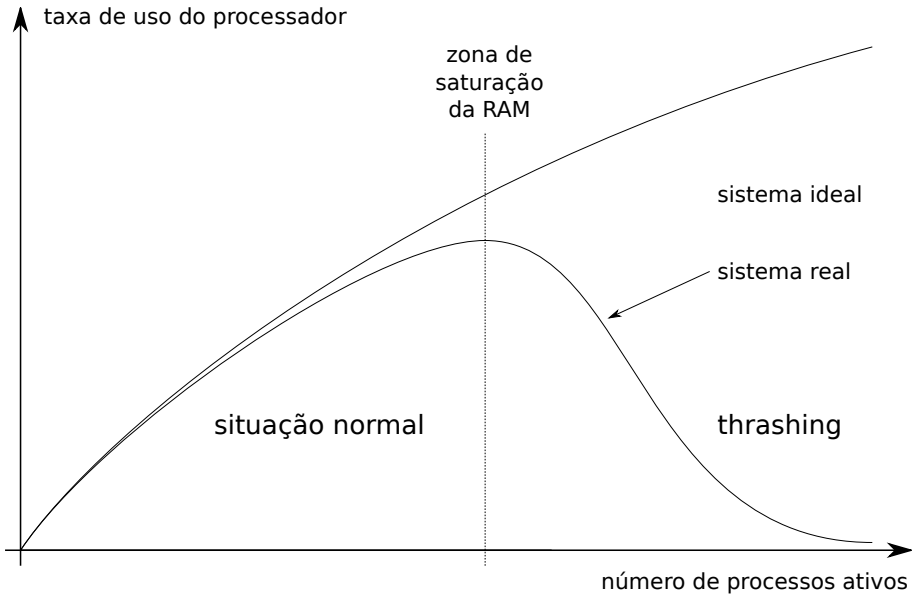

Figura 33: *Thrashing* no uso da memória RAM.

Um sistema operacional sob *thrashing* tem seu desempenho muito prejudicado, a ponto de parar de responder ao usuário e se tornar inutilizável. Por isso, esse fenômeno deve ser evitado. Para tal, pode-se aumentar a quantidade de memória RAM do sistema, limitar a quantidade máxima de processos ativos, ou mudar a política de escalonamento

dos processos durante o *thrashing*, para evitar a competição pela memória disponível. Vários sistemas operacionais adotam medidas especiais para situações de *thrashing*, como suspender em massa os processos ativos, adotar uma política de escalonamento de processador que considere o uso da memória, aumentar o *quantum* de processador para cada processo ativo, ou simplesmente abortar os processos com maior alocação de memória ou com maior atividade de paginação.

Referências

[Bansal and Modha, 2004] Bansal, S. and Modha, D. (2004). CAR: Clock with adaptive replacement. In *USENIX Conference on File and Storage Technologies*.

[Carr and Hennessy, 1981] Carr, R. and Hennessy, J. (1981). WSclock - a simple and effective algorithm for virtual memory management. In *ACM symposium on Operating systems principles*.

[Denning, 1980] Denning, P. (1980). Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84.

[Denning, 2006] Denning, P. J. (2006). The locality principle. In Barria, J., editor, *Communication Networks and Computer Systems*, chapter 4, pages 43–67. Imperial College Press.

[Jiang and Zhang, 2002] Jiang, S. and Zhang, X. (2002). LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Intl Conference on Measurement and Modeling of Computer Systems*, pages 31–42.

[Johnstone and Wilson, 1999] Johnstone, M. S. and Wilson, P. R. (1999). The memory fragmentation problem: solved? *ACM SIGPLAN Notices*, 34(3):26–36.

[Levine, 2000] Levine, J. (2000). *Linkers and Loaders*. Morgan Kaufmann.

[Navarro et al., 2002] Navarro, J., Iyer, S., Druschel, P., and Cox, A. (2002). Practical, transparent operating system support for superpages. In *5th USENIX Symposium on Operating Systems Design and Implementation*, pages 89–104.

[Patterson and Henessy, 2005] Patterson, D. and Henessy, J. (2005). *Organização e Projeto de Computadores*. Campus.

[Silberschatz et al., 2001] Silberschatz, A., Galvin, P., and Gagne, G. (2001). *Sistemas Operacionais – Conceitos e Aplicações*. Campus.

[Tanenbaum, 2003] Tanenbaum, A. (2003). *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall.

[Uhlig and Mudge, 1997] Uhlig, R. and Mudge, T. (1997). Trace-driven memory simulation: a survey. *ACM Computing Surveys*, 29(2):128–170.