Capítulo 14

Hardware de memória

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 e bibliotecas compartilhadas, 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 memórias RAM, caches, unidade de gerência, barramentos, etc, o que exige um esforço de gerência significativo por parte do sistema operacional.

Neste capítulo serão apresentados os principais elementos de hardware que compõe o sistema de memória de um computador e os mecanismos básicos implementados pelo hardware e controlados pelo sistema operacional para a sua gerência.

14.1 Tipos 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 informação. 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 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 informação.

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, o custo por byte armazenado e a volatilidade. Essas características permitem definir uma *hierarquia de memória*, geralmente representada na forma de uma pirâmide (Figura 14.1).

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. Além disso, as memórias mais rápidas são *voláteis*, ou seja, perdem seu conteúdo ao ficarem sem energia, quando o computador é desligado. Memórias que preservam seu conteúdo mesmo quando não tiverem energia, como as unidades *Flash* e os discos rígidos, são denominadas *memórias não-voláteis*.

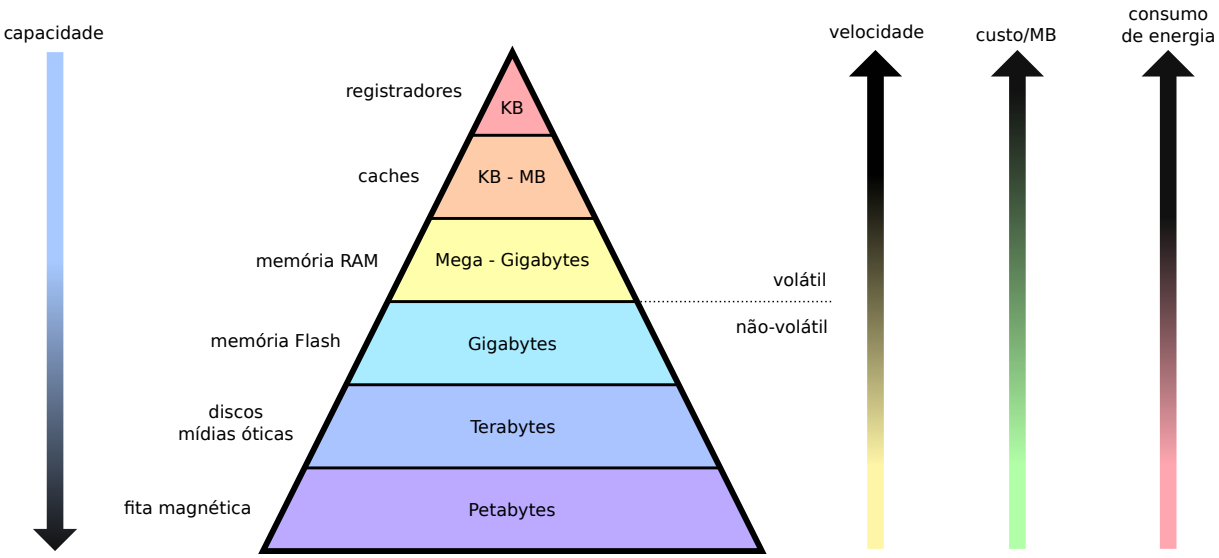

Figura 14.1: Hierarquia de memória.

Outra característica importante das memórias é a rapidez de seu funcionamento, que pode ser traduzida em duas grandezas: o *tempo de acesso* (ou *latência*) e a *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 14.1 traz valores de tempo de acesso e taxa de transferência típicos de alguns meios de armazenamento usuais.

Meio	**Tempo de acesso**	**Taxa de transferência**
Cache L2	1 ns	1 GB/s (1 ns por byte)
Memória RAM	60 ns	1 GB/s (1 ns por byte)
Memória *flash* (NAND)	2 ms	10 MB/s (100 ns por byte)
Disco rígido SATA	5 ms (tempo para o ajuste da cabeça de leitura e a rotação do disco até o setor desejado)	100 MB/s (10 ns por 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 por byte)

Tabela 14.1: Tempos de acesso e taxas de transferência típicas [Patterson and Henessy, 2005].

Este e os próximos capítulos são dedicados aos mecanismos envolvidos na gerência da memória principal do computador, que geralmente é constituída por um grande espaço de memória do tipo RAM (*Random Access Memory*). 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].

14.2 A memória física

A memória principal do computador é uma área de RAM composta por uma grande sequência de bytes, que é a menor unidade de memória usada pelo processador. Cada byte da memória RAM possui um endereço, que é usado para acessá-lo. Um computador convencional atual possui alguns GBytes de memória RAM, usados para conter o sistema operacional e os processos em execução, além de algumas áreas para finalidades específicas, como buffers de dispositivos de entrada/saída. A quantidade de memória RAM disponível em um computador constitui seu **espaço de memória física**.

A Figura 14.2 ilustra a organização (simplificada) da memória RAM de um computador PC atual com 16 GBytes de memória RAM instalados. Nessa figura, as áreas livres (*free RAM*) podem ser usadas pelo sistema operacional e as aplicações; as demais áreas têm finalidades específicas e geralmente só são acessadas pelo hardware e pelo sistema operacional, para gerenciar o computador e realizar operações de entrada/saída.

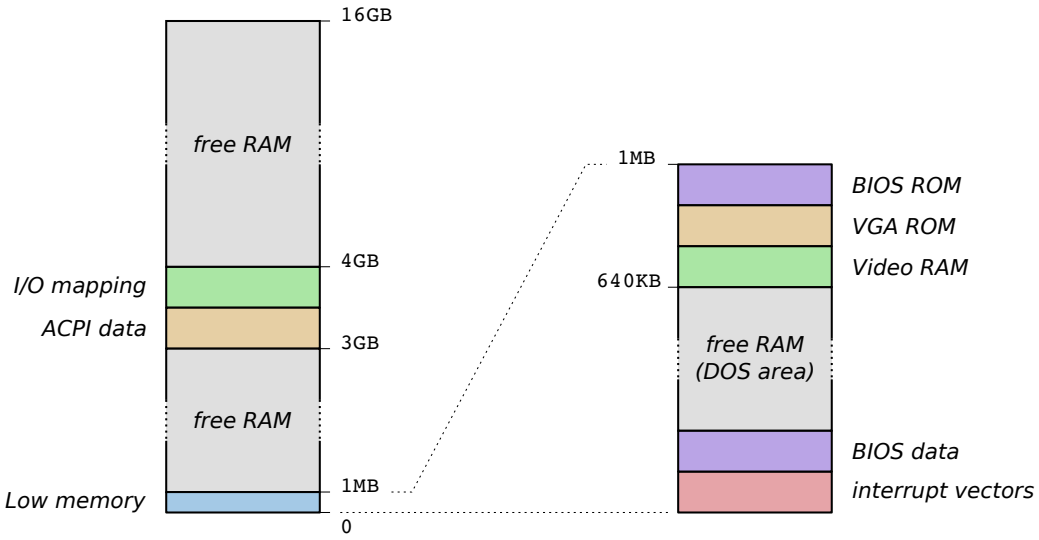

Figura 14.2: Layout da memória física de um computador.

Nos sistemas atuais, o layout da memória física apresentado na figura 14.2 não é visível ao usuário. Como regra geral, a execução de programas diretamente sobre a memória física é pouco usada, com exceção de sistemas muito simples, como em sistemas embarcados baseados em microcontroladores, ou muito antigos (como o MS-DOS – *Disk Operating System*). Os sistemas atuais mais sofisticados usam os conceitos de espaços de endereçamento e de memória virtual, vistos nas próximas seções, para desacoplar a visão da memória pelos processos da estrutura física da memória no hardware. Esse desacoplamento visa tornar mais simples e flexível o uso da memória pelos processos e pelo sistema operacional.

14.3 Espaço de endereçamento

O processador acessa a memória RAM através de barramentos de dados, de endereços e de controle. O barramento de endereços (como os demais) possui um número fixo de vias, que define a quantidade total de endereços de memória que podem ser gerados pelo processador: um barramento de dados com n vias consegue gerar 2^n

endereços distintos (pois cada via define um bit do endereço), no intervalo $[0 \ldots 2^n - 1]$. Por exemplo, um processador Intel 80386 possui 32 vias de endereços, o que o permite acessar até 2^{32} bytes (4 GBytes) de memória, no intervalo $[0 \ldots 2^{32} - 1]$. Já um processador Intel Core i7 usa 48 vias para endereços e portanto pode endereçar até 2^{48} bytes, ou seja, 256 Terabytes de memória física. O conjunto de endereços de memória que um processador pode produzir é chamado de **espaço de endereçamento**.

É fundamental ter em mente que o espaço de endereçamento do processador é independente da quantidade de memória RAM disponível no sistema, podendo ser muito maior que esta. Assim, um endereço gerado pelo processador pode ser válido, quando existe um byte de memória RAM acessível naquele endereço, ou inválido, quando não há memória instalada naquele endereço. Dependendo da configuração da memória RAM, o espaço de endereçamento pode conter diversas áreas válidas e outras inválidas.

14.4 A memória virtual

Para ocultar a organização complexa da memória física e simplificar os procedimentos de alocação da memória aos processos, os sistemas de computação modernos implementam a noção de **memória virtual**, na qual existem dois tipos de endereços de memória **distintos**:

Endereços físicos (ou reais) são os endereços dos bytes de memória física do computador. Estes endereços são definidos pela quantidade de memória disponível na máquina, de acordo com o diagrama da figura 14.2.

Endereços lógicos (ou virtuais) são os endereços de memória usados pelos processos e pelo sistema operacional e, portanto, usados pelo processador durante a execução. Estes endereços são definidos de acordo com o espaço de endereçamento do processador.

Ao executar, os processos "enxergam" somente a memória virtual. Assim, durante a execução de um programa, o processador gera endereços lógicos para acessar a memória. Esses endereços devem então ser traduzidos para os endereços físicos correspondentes na memória RAM, onde as informações desejadas se encontram. Por questões de desempenho, a tradução de endereços lógicos em físicos é feita por um componente específico do hardware do computador, denominado **Unidade de Gerência de Memória** (MMU – *Memory Management Unit*). Na maioria dos processadores atuais, a MMU se encontra integrada ao chip da própria CPU.

A MMU intercepta os endereços lógicos emitidos pelo processador e os traduz para os endereços físicos correspondentes na memória da máquina, permitindo então seu acesso pelo processador. Caso o acesso a um determinado endereço lógico não seja possível (por não estar associado a um endereço físico, por exemplo), a MMU gera uma interrupção de hardware para notificar o processador sobre a tentativa de acesso indevido. O comportamento da MMU e as regras de tradução de endereços são configurados pelo núcleo do sistema operacional.

O funcionamento básico da MMU está ilustrado na Figura 14.3. Observa-se que a MMU intercepta o acesso do processador ao barramento de endereços, recebendo os endereços lógicos gerados pelo mesmo e enviando os endereços físicos correspondentes

ao barramento de endereços. Além disso, a MMU também tem acesso ao barramento de controle, para identificar operações de leitura e de escrita na memória.

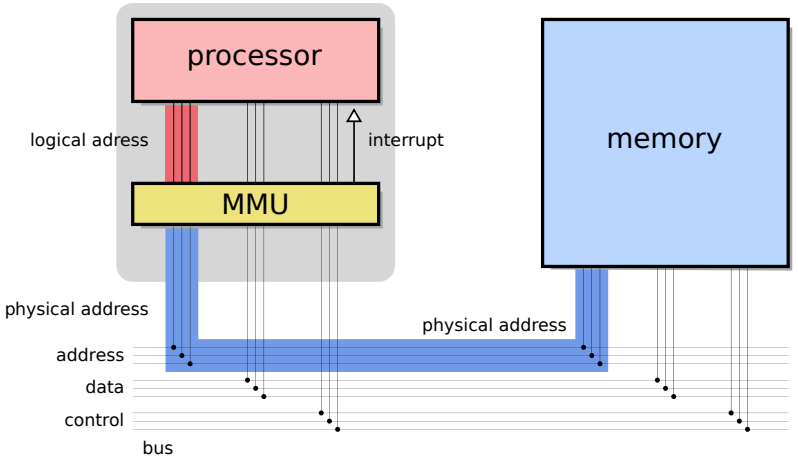

Figura 14.3: Funcionamento básico de uma MMU.

Além de desacoplar os endereços lógicos dos endereços físicos e realizar a tradução entre ambos, a noção de memória virtual também permite implementar a proteção de memória do núcleo e dos processos entre si, fundamentais para a segurança e estabilidade do sistema. Para implementar a proteção de memória entre processos, o núcleo mantém regras distintas de tradução de endereços lógicos para cada processo e reconfigura a MMU a cada troca de contexto. Assim, o processo em execução em cada instante tem sua própria área de memória e é impedido pela MMU de acessar áreas de memória dos demais processos.

Além disso, a configuração das MMUs mais sofisticadas inclui a definição de permissões de acesso às áreas de memória. Essa funcionalidade permite implementar as permissões de acesso às diversas áreas de cada processo (conforme visto na Seção 15.2), bem como impedir os processos de acessar áreas exclusivas do núcleo do sistema operacional.

Nas próximas seções serão estudadas as principais estratégias de tradução de endereços usadas pelas MMUs: por partições (usada nos primeiros sistemas de memória virtual), por segmentos e por páginas, usada nos sistemas atuais.

14.5 Memória virtual por partições

Uma das formas mais simples de organização da memória e tradução de endereços lógicos em físicos consiste em dividir a memória física em N partições, que podem ter tamanhos iguais ou distintos, fixos ou variáveis. Em cada partição da memória física é carregado um processo. O processo ocupando uma partição de tamanho T bytes terá um espaço de endereçamento com até T bytes, com endereços lógicos no intervalo $[0 \ldots T-1]$. A Figura 14.4 ilustra essa estratégia. Nela, quatro processos ocupam partições distintas na memória RAM.

Na tradução de endereços lógicos em um esquema por partições, a MMU possui dois registradores: um *registrador base* (B), que define o endereço físico inicial da partição

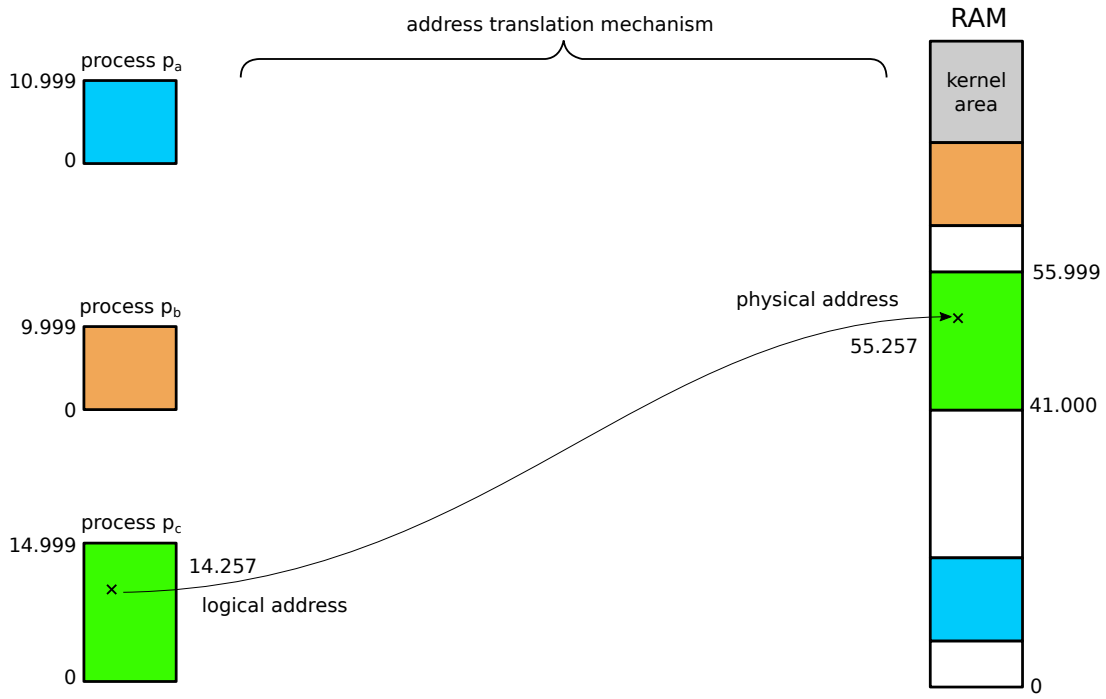

Figura 14.4: Memória virtual por partições.

ativa[1], e um *registrador limite* (L), que define o tamanho em bytes dessa partição. O algoritmo implementado pela MMU é simples: cada endereço lógico e_l gerado pelo processador é comparado ao valor do registrador limite; caso seja menor que este ($e_l < L$), o endereço lógico é somado ao valor do registrador base, para a obtenção do endereço físico correspondente ($e_f = e_l + B$). Caso contrário ($e_l \geq L$), uma interrupção é gerada pela MMU, indicando um endereço lógico inválido.

A Figura 14.5 apresenta o funcionamento da MMU usando essa estratégia. Na Figura, o processo ativo, ocupando a partição 2, tenta acessar o endereço lógico 14.257. A MMU verifica que esse endereço é válido, pois é inferior ao limite da partição (15.000). Em seguida o endereço lógico é somado à base da partição (41.000) para obter o endereço físico correspondente (55.257). A tabela de partições reside na memória RAM, sendo usada para atualizar os registradores de base e limite da MMU quando houver uma troca de contexto.

Os valores dos registradores base e limite da MMU devem ser ajustados pelo núcleo sempre que for necessário trocar de espaço de endereçamento, ou seja, a cada troca de contexto. Os valores de base e limite para cada processo do sistema podem estar armazenados em uma tabela de partições ou no TCB do processo (*Task Control Block*, vide Seção 5.1). Quando o núcleo estiver executando, os valores de base e limite podem ser ajustados respectivamente para 0 e ∞, permitindo 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 $[0 \ldots L(p_i) - 1]$, que correspondem a endereços físicos no intervalo $[B(p_i) \ldots B(p_i) + L(p_i) - 1]$. Ao detectar uma tentativa de acesso a um endereço lógico fora desse intervalo, a MMU irá gerar uma solicitação de interrupção (IRq - *Interrupt Request*, vide Seção 2.2.2) para o processador,

[1] A partição ativa é aquela onde se encontra o código em execução naquele instante.

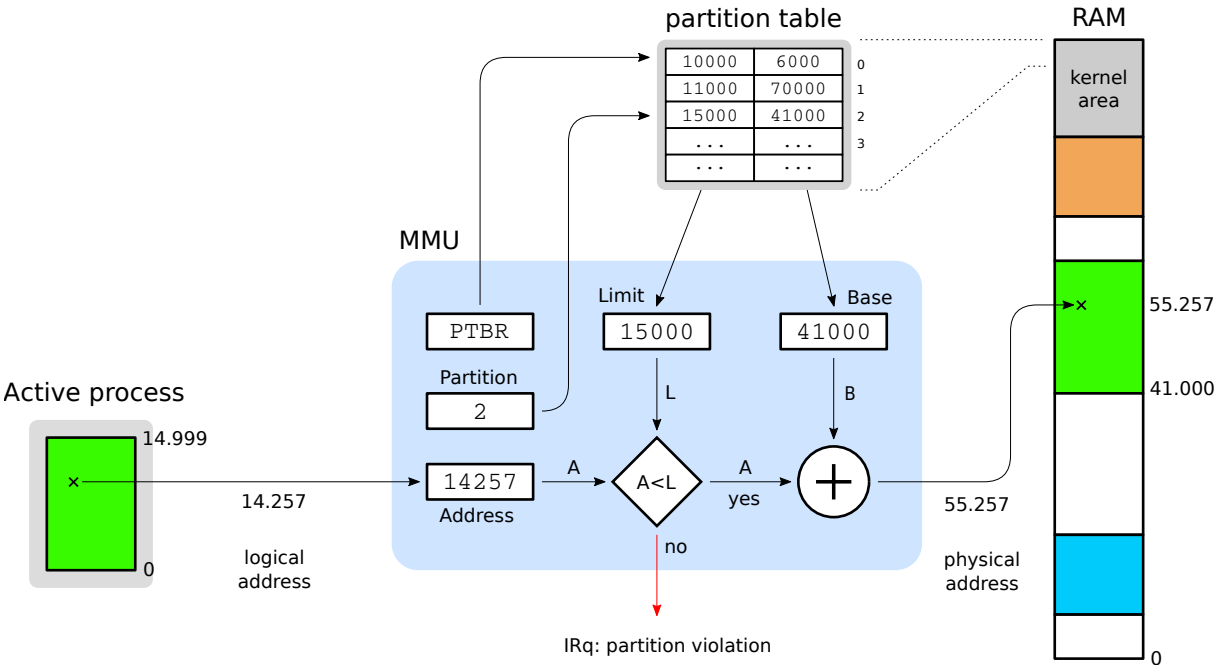

Figura 14.5: MMU com partições.

sinalizado um acesso a endereço inválido. Ao receber a interrupção, o processador interrompe a 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 tradução por partições é sua simplicidade: por depender apenas de dois registradores e de uma lógica simples para a tradução de endereços, ela pode ser implementada em hardware de baixo custo, ou mesmo incorporada a processadores mais simples. Todavia, é uma estratégia pouco flexível e está sujeita a um fenômeno denominado *fragmentação externa*, que será discutido na Seção 16.3. Ela foi usada no OS/360, um sistema operacional da IBM usado nas décadas de 1960-70 [Tanenbaum, 2003].

14.6 Memória virtual por segmentos

A tradução por segmentos é uma extensão da tradução por partições, na qual as seções de memória do processo (`TEXT`, `DATA`, etc.) são mapeadas em áreas separadas na memória física. Além das seções funcionais básicas da memória do processo discutidas na Seção 15.2, também podem ser definidas áreas para itens específicos, como bibliotecas compartilhadas, vetores, matrizes, pilhas de *threads*, buffers de entrada/saída, etc.

Nesta abordagem, o espaço de endereçamento de cada processo não é mais visto como uma sequência linear de endereços lógicos, mas como uma coleção de áreas de tamanhos diversos e políticas de acesso distintas, denominadas *segmentos*. Cada segmento se comporta como uma partição de memória independente, com seus próprios endereços lógicos. A Figura 14.6 apresenta a visão lógica da memória de um processo e a sua forma de mapeamento para a memória física.

No modelo de memória virtual por segmentos, os endereços lógicos gerados pelos processos são compostos por pares [*segmento* : *offset*], onde *segmento* indica o número do segmento e *offset* indica a posição dentro daquele segmento. Os valores de

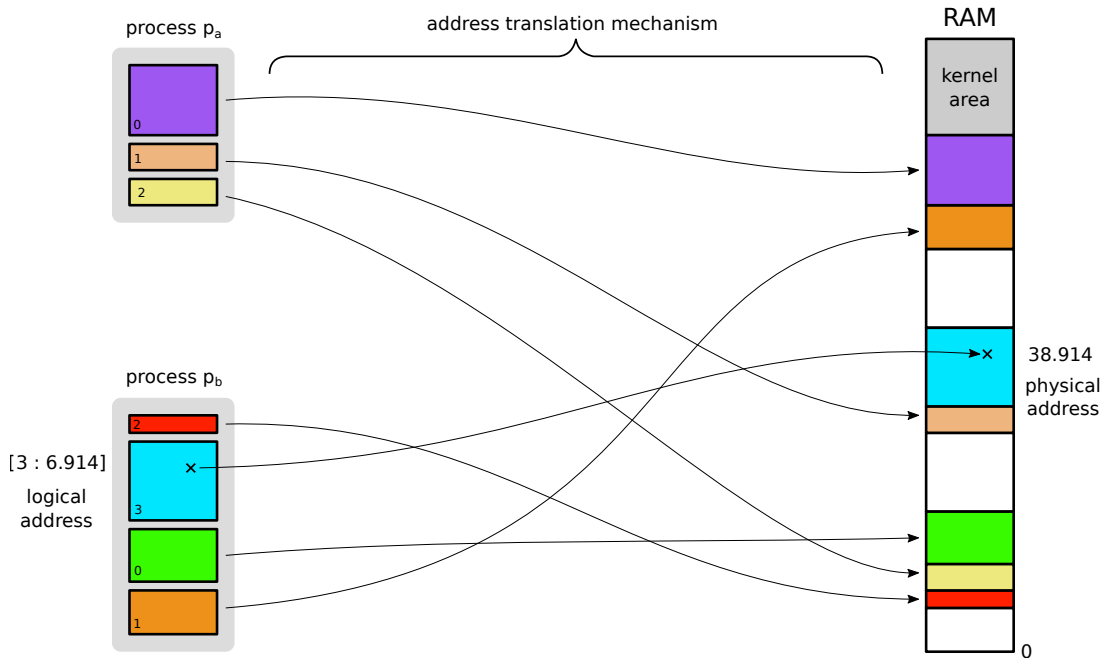

Figura 14.6: Memória virtual por segmentos.

offset em um segmento S variam no intervalo $[0 \ldots T(S) - 1]$, onde $T(S)$ é o tamanho do segmento. A Figura 14.6 mostra o endereço lógico [3 : 6.914], que corresponde ao *offset* 6.914 no segmento 3 do processo p_b. Nada impede de existir outros endereços 6.914 em outros segmentos do mesmo processo.

A tradução de endereços lógicos por segmentos é similar à tradução por partições. Contudo, como os segmentos são partições, 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 o segmento, como permissões de acesso, etc. (vide Seção 14.7.2). A MMU possui dois registradores para indicar a localização da tabela de segmentos ativa na memória RAM e seu tamanho: *STBR* (*Segment Table Base Register*) e *STLR* (*Segment Table Length Register*). A Figura 14.7 apresenta os principais elementos envolvidos na tradução de endereços lógicos em físicos usando segmentos.

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 manter as tabelas na memória RAM. O processador Intel 80386 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 18.1). Cada uma dessas duas tabelas comporta até 8.192 segmentos. A cada troca de contexto, os registradores que indicam a tabela de segmentos ativa são 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 ao endereço lógico acessado. Todavia, como as tabelas de segmentos normalmente se

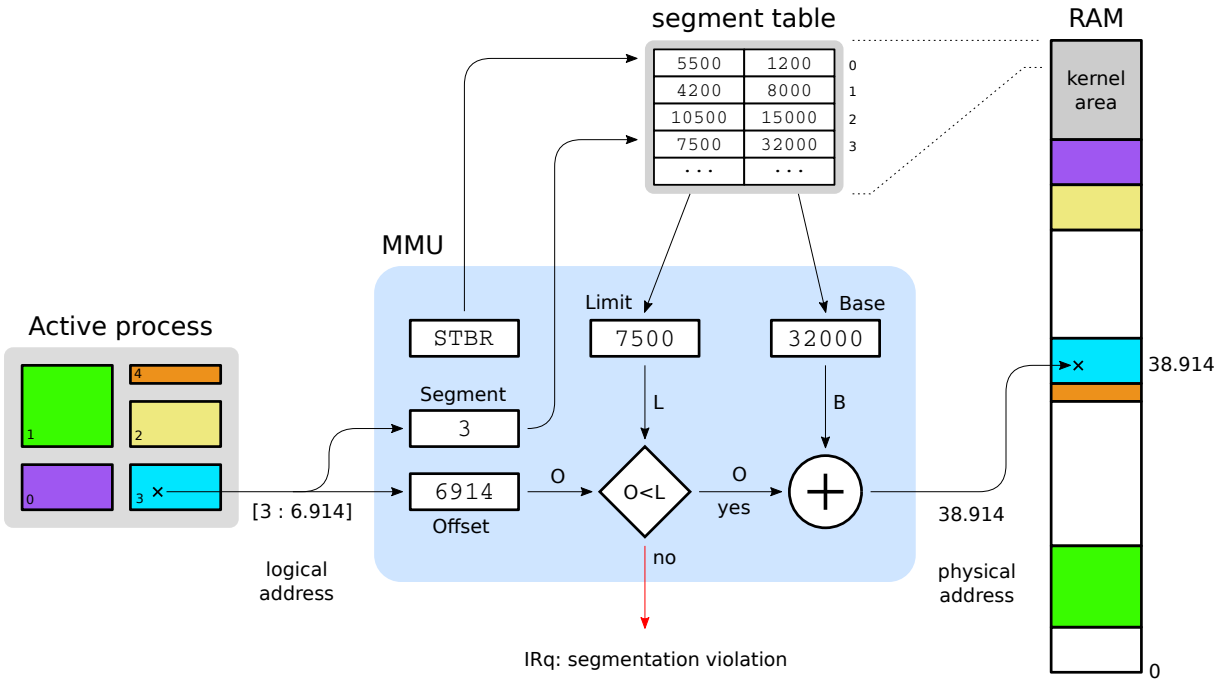

Figura 14.7: MMU com segmentação.

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 o tempo todo, o que mantém o desempenho de acesso à memória em um nível satisfatório. O processador Intel 80386 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 de memória 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.

O modelo de memória virtual por segmentos foi muito utilizado nos anos 1970-90, sobretudo nas arquiteturas Intel e AMD de 32 bits. Hoje em dia esse modelo é raramente utilizado em processadores de uso geral, sendo dada preferência ao modelo baseado em páginas (apresentado na próxima seção).

14.7 Memória virtual por páginas

Conforme visto na seção anterior, a organização da 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 organização bastante suscetível à fragmentação externa, conforme será discutido na Seção 16.3. Essas deficiências levaram os projetistas de hardware a desenvolver outras técnicas para a organização da memória principal.

Na organização da memória por páginas, ou *memória paginada*, o espaço de endereçamento lógico dos processos é mantido linear e unidimensional. Internamente, de forma transparente para o processador, o espaço de endereçamento lógico 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]. A memória física também é dividida em blocos de mesmo tamanho que as páginas, denominados *quadros* (do inglês *frames*).

O mapeamento do espaço de endereçamento lógico na memória física é então feito simplesmente indicando em que quadro da memória física se encontra cada página, conforme ilustra a Figura 14.8. É importante observar que uma página pode estar em qualquer posição da memória física disponível, ou seja, pode estar associada a qualquer quadro, o que permite uma grande flexibilidade no uso da memória física.

14.7.1 A tabela de páginas

O mapeamento entre as páginas e os quadros correspondentes na memória física é feita através de *tabelas de páginas* (*page tables*), nas quais cada entrada corresponde a uma página do processo e contém o número do quadro onde ela se encontra. Cada processo possui sua própria tabela de páginas; a tabela de páginas ativa, que corresponde ao processo em execução no momento, é referenciada por um registrador da MMU 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.

[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 mais complexos. O artigo [Navarro et al., 2002] traz uma discussão mais detalhada sobre esse tema.

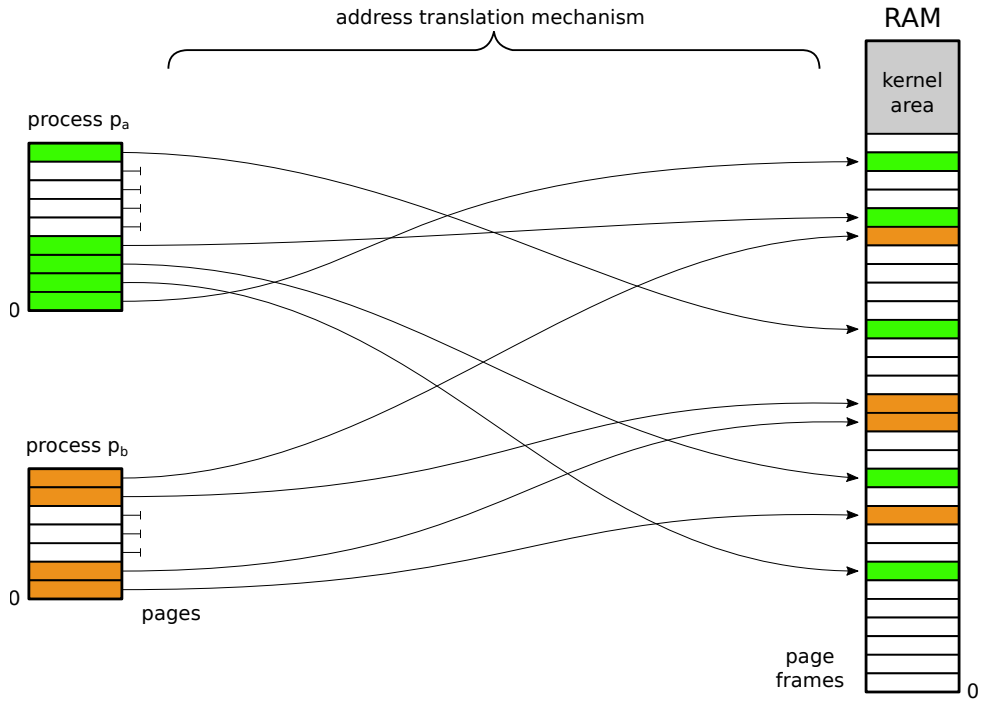

Figura 14.8: Organização da memória em páginas.

Por exemplo, o processador Intel 80386 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 do endereço lógico $01803E9A_h$ nesse sistema[3]:

$$01803E9A_h \rightarrow \overbrace{0000\ 0001\ 1000\ 0000\ 0011}^{page:\ 20\ bits}\overbrace{1110\ 1001\ 1010_2}^{offset:\ 12\ bits}$$

$$\rightarrow \overbrace{0000\ 0001\ 1000\ 0000\ 0011}^{page=01803_h}\overbrace{1110\ 1001\ 1010_2}^{offset=E9A_h}$$

$$\rightarrow page = 01803_h \quad offset = E9A_h$$

Para traduzir um endereço lógico no endereço físico correspondente, a MMU efetua os seguintes passos, que são ilustrados na Figura 14.9:

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.

[3] A notação NNN_h indica um número em hexadecimal.

Pode-se observar que as páginas de memória não utilizadas pelo processo são representadas por entradas vazias na tabela de páginas e portanto não são mapeadas em quadros de memória física. Se o processo tentar acessar essas páginas, a MMU irá gerar uma interrupção de falta de página (*page fault*). Essa interrupção provoca o desvio da execução para o núcleo do sistema operacional, que deve então tratar a falta de página, abortando o processo ou tomando outra medida.

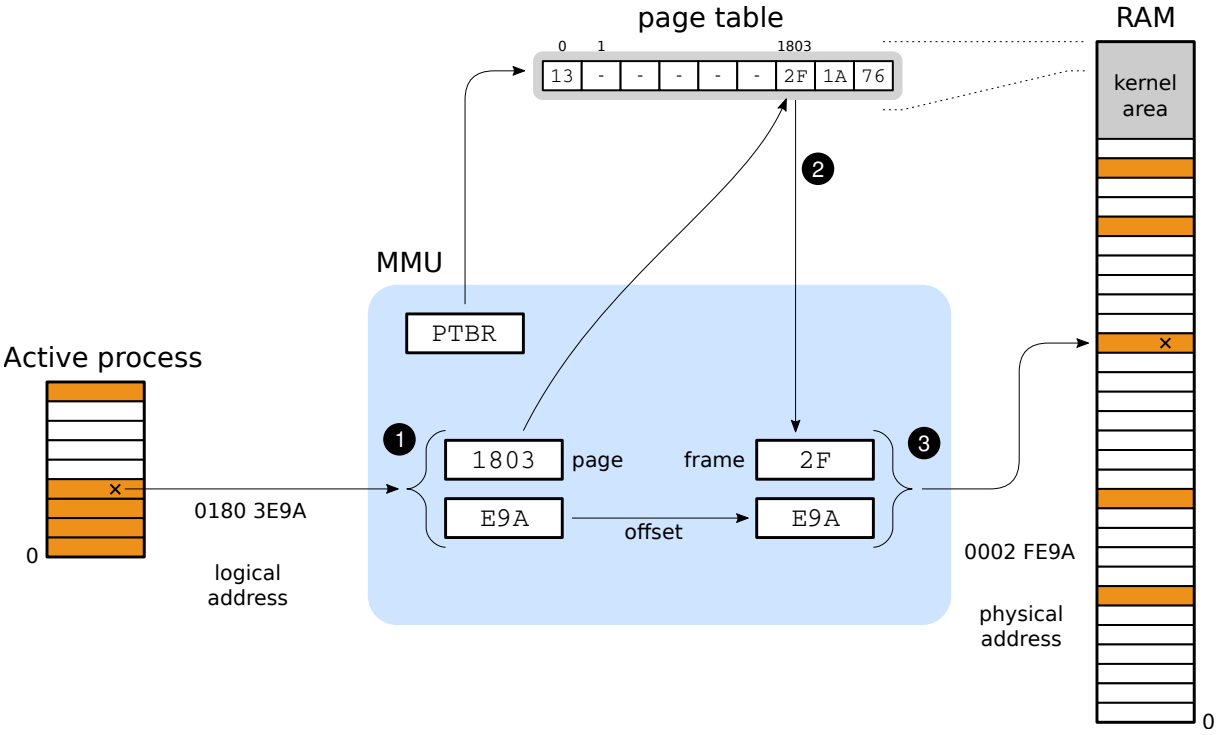

Figura 14.9: MMU com paginação.

14.7.2 Flags de status e controle

Além do número do quadro correspondente na memória física, cada entrada de uma tabela de páginas contém um conjunto de *flags* (bits) de status ou de controle relativos à página, com diversas finalidades. Os mais usuais são:

- *Valid*: indica se a página é válida, ou seja, se existe no espaço de endereçamento daquele processo; se este bit estiver em 0, tentativas de acesso à pagina irão gerar uma interrupção de falta de página (*page fault*);
- *Writable*: controla se a página pode ser acessada em leitura e escrita (1) ou somente em leitura (0);
- *User*: se estiver ativo (1), código executando em modo usuário pode acessar a página; caso contrário, a página só é acessível ao núcleo do sistema;
- *Present*: indica se a página está presente na memória RAM ou se foi transferida para um armazenamento secundário, como ocorre nos sistemas com paginação em disco (Seção 17.2);

- *Accessed*: indica se a página foi acessada recentemente; este bit é ativado pela MMU a cada acesso à página e pode ser desativado pelo núcleo quando desejado; essa informação é usada pelos algoritmos de paginação em disco;
- *Dirty*: este bit é ativado pela MMU após uma escrita na página, para informar que ela foi modificada (que foi "suja"); também é usado pelos algoritmos de paginação em disco.

Além destes, podem existir outros bits, indicando a política de *caching* aplicável à página, se a página pode ser movida para disco, o tamanho da página (no caso de sistemas que permitam mais de um tamanho de página), além de bits genéricos que podem ser usados pelos algoritmos do núcleo. O conteúdo exato de cada entrada da tabela de páginas depende da arquitetura do hardware considerado.

14.7.3 Tabelas multiníveis

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 informações e *flags* de controle). Considerando que cada tabela de páginas tem 2^{20} páginas, uma 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 poderá ocupar mais espaço na memória que o próprio processo.

A Figura 14.10 mostra a tabela de páginas de um processo pequeno, com 100 páginas mapeadas no início de seu espaço de endereçamento (para as seções `TEXT`, `DATA` e `HEAP`) e 20 páginas mapeadas no final (para a seção `STACK`)[4]. Esse processo ocupa 120 páginas em RAM, ou 480 KBytes, enquanto sua tabela de páginas é quase 10 vezes maior, ocupando 4 MBytes. Além disso, a maior parte das entradas da tabela é vazia, ou seja, não aponta para quadros válidos.

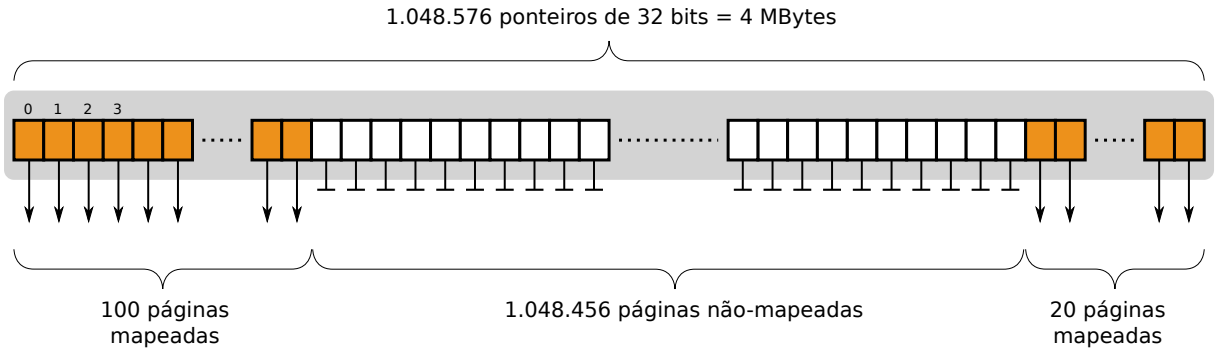

Figura 14.10: Tabela de páginas linear.

Para resolver esse problema, são usadas *tabelas de páginas multiníveis*, estruturadas na forma de árvores: uma primeira tabela de páginas (ou *diretório de páginas*) contém ponteiros para tabelas de páginas secundárias e assim por diante, até chegar à tabela que contém o número do quadro desejado. Quando uma tabela secundária não contiver entradas válidas, ela não precisa ser alocada; isso é representado por uma entrada nula na tabela principal.

[4] A organização interna da memória de um processo é discutida na Seção 15.2.

A Figura 14.11 apresenta uma tabela de páginas com dois níveis que armazena as mesmas informações que a tabela linear da Figura 14.10, mas de forma muito mais compacta (12 KBytes ao invés de 4 MBytes). Cada sub-tabela contém 1.024 entradas.

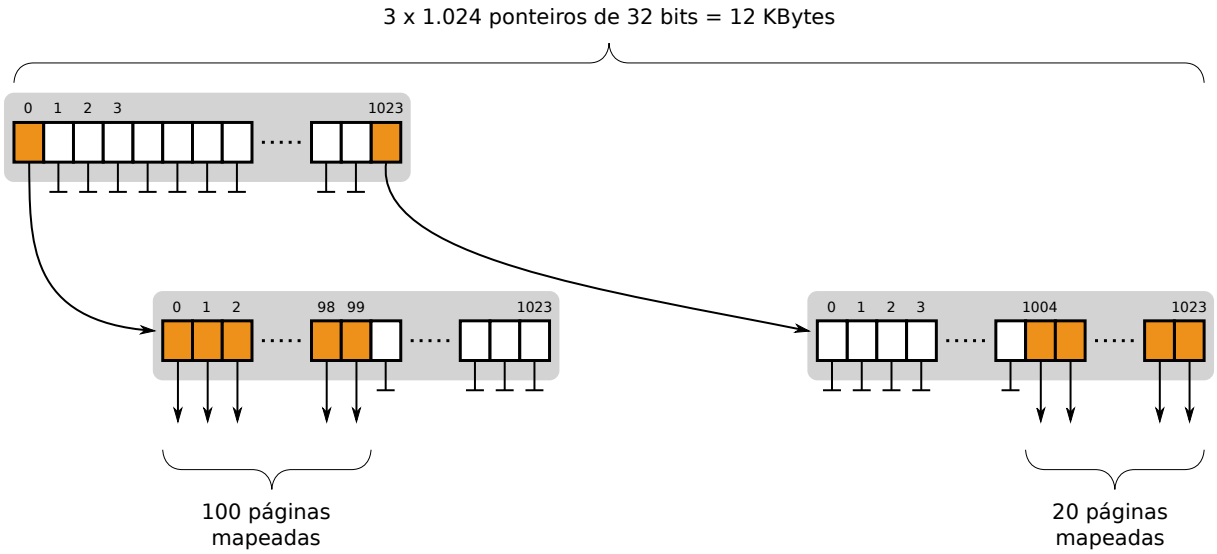

Figura 14.11: Tabela de páginas com dois níveis.

Para percorrer essa árvore, o número de página precisa ser dividido em duas ou mais partes, que são usadas como índices em cada nível de tabela, até encontrar o número de quadro desejado. Um exemplo permite explicar melhor esse mecanismo: considerando uma arquitetura de 32 bits com páginas de 4 KBytes, 20 bits são usados para acessar a tabela de páginas. Esses 20 bits são divididos em dois grupos de 10 bits (p_1 e p_2) que são usados como índices em uma tabela de páginas com dois níveis:

$$
\begin{aligned}
01803E9A_h \;\rightarrow\; & \overbrace{0000\ 0001\ 10}^{p_2\text{:10 }bits}\overbrace{00\ 0000\ 0011}^{p_1\text{:10 }bits}\overbrace{1110\ 1001\ 1010}^{offset\text{:12}bits} \\
\rightarrow\; & \overbrace{0000\ 0001\ 10}^{p_2=0006_h}\overbrace{00\ 0000\ 0011}^{p_1=0003_h}\overbrace{1110\ 1001\ 1010}^{offset=E9A_h} \\
\rightarrow\; & p_2 = 0006_h \quad p_1 = 0003_h \quad offset = E9A_h
\end{aligned}
$$

A tradução de endereços lógicos em físicos usando uma tabela de páginas com dois níveis é efetuada através dos seguintes passos, que são ilustrados na Figura 14.12:

1. o endereço lógico 0180 $3E9A_h$ é decomposto em um *offset* de 12 bits ($E9A_h$) e dois números de página de 10 bits cada, que serão usados como índices: índice do nível externo p_2 (006_h) e o índice do nível interno p_1 (003_h);

2. o índice p_2 é usado como índice na tabela de páginas externa, para encontrar o endereço de uma tabela de páginas interna;

3. em seguida, o índice p_1 é usado na tabela de páginas interna indicada por p_2, para encontrar a entrada contendo o número de quadro ($2F_h$) que corresponde a $[p_2 p_1]$;

4. o número de quadro é combinado ao *offset* para obter o endereço físico (0002 $FE9A_h$) correspondente ao endereço lógico solicitado.

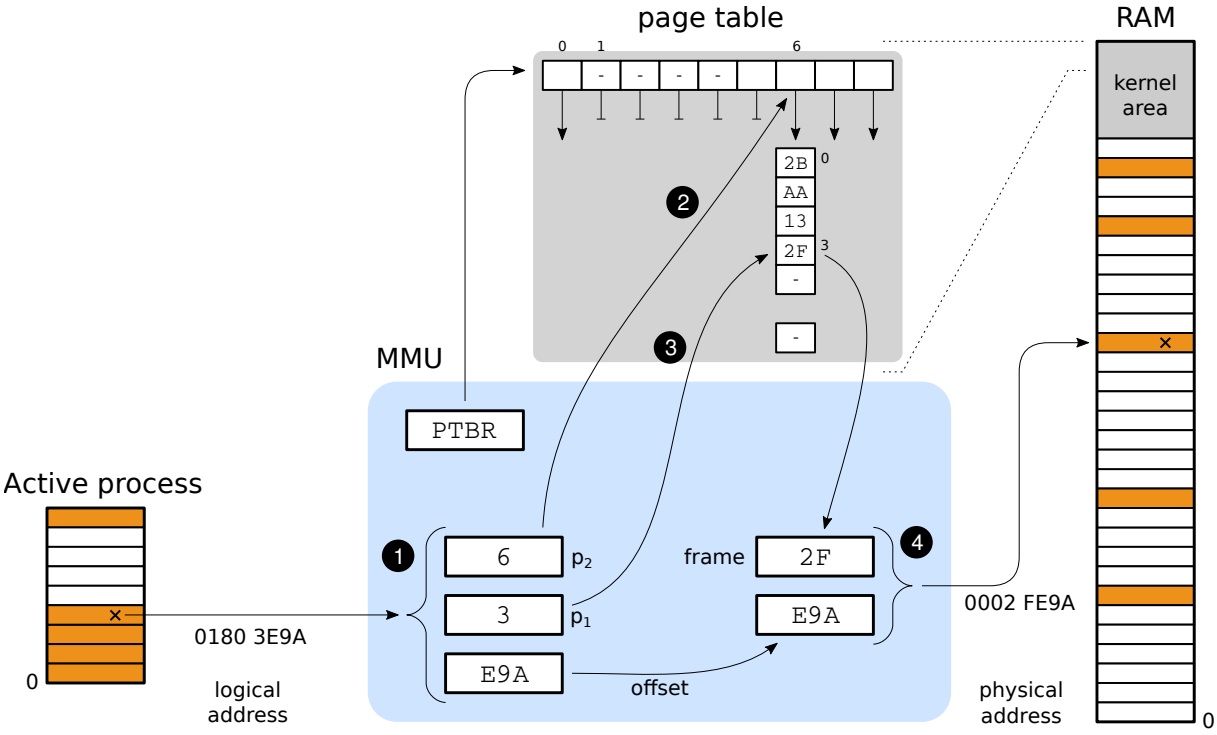

Figura 14.12: MMU com paginação multinível.

A estruturação da tabela de páginas em níveis reduz significativamente a quantidade de memória necessária para armazená-la, sobretudo no caso de processos pequenos. As Figuras 14.10 e 14.11 evidenciam essa redução, de 4 MBytes para 12 KBytes. Por outro lado, se um processo ocupar todo o seu espaço de endereçamento, seriam necessárias uma tabela de primeiro nível e 1.024 tabelas de segundo nível, que ocupariam $(1 + 1.024) \times 4KB$, ou seja, 0,098% a mais que se a tabela linear ($1.024 \times 4KB$).

O número de níveis da tabela de páginas depende da arquitetura considerada: processadores *Intel 80386* usam tabelas com dois níveis, cada tabela com 1.024 entradas de 4 bytes. Processadores de 64 bits mais recentes, como o *Intel Core i7*, usam tabelas com 4 níveis, cada tabela contendo 512 entradas de 8 bytes. Em ambos os casos, cada subtabela ocupa exatamente uma página de 4 KBytes.

14.7.4 Cache da tabela de páginas

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

Para atenuar esse problema, consultas recentes à tabela de páginas podem ser armazenadas em um *cache* dentro da própria MMU, evitando ter de repeti-las e

assim diminuindo o tempo de acesso à memória RAM. 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 erro, caso não contenha informação sobre p.

A tradução de endereços lógicos em físicos usando TLB se torna mais rápida, mas também mais complexa. Os seguintes passos são necessários, ilustrados na Figura 14.13, são necessários:

1. A MMU decompõe o endereço lógico em números de página e *offset*;
2. a MMU consulta os números de página em seu cache TLB;
3. caso o número do quadro correspondente seja encontrado (*TLB hit*), ele é usado para compor o endereço físico;
4. caso contrário (*TLB miss*), uma busca completa na tabela de páginas[5] deve ser realizada para obter o número do quadro (passos 4-6);
7. o número de quadro obtido é usado para compor o endereço físico;
8. o número de quadro é adicionado ao TLB para agilizar as próximas consultas.

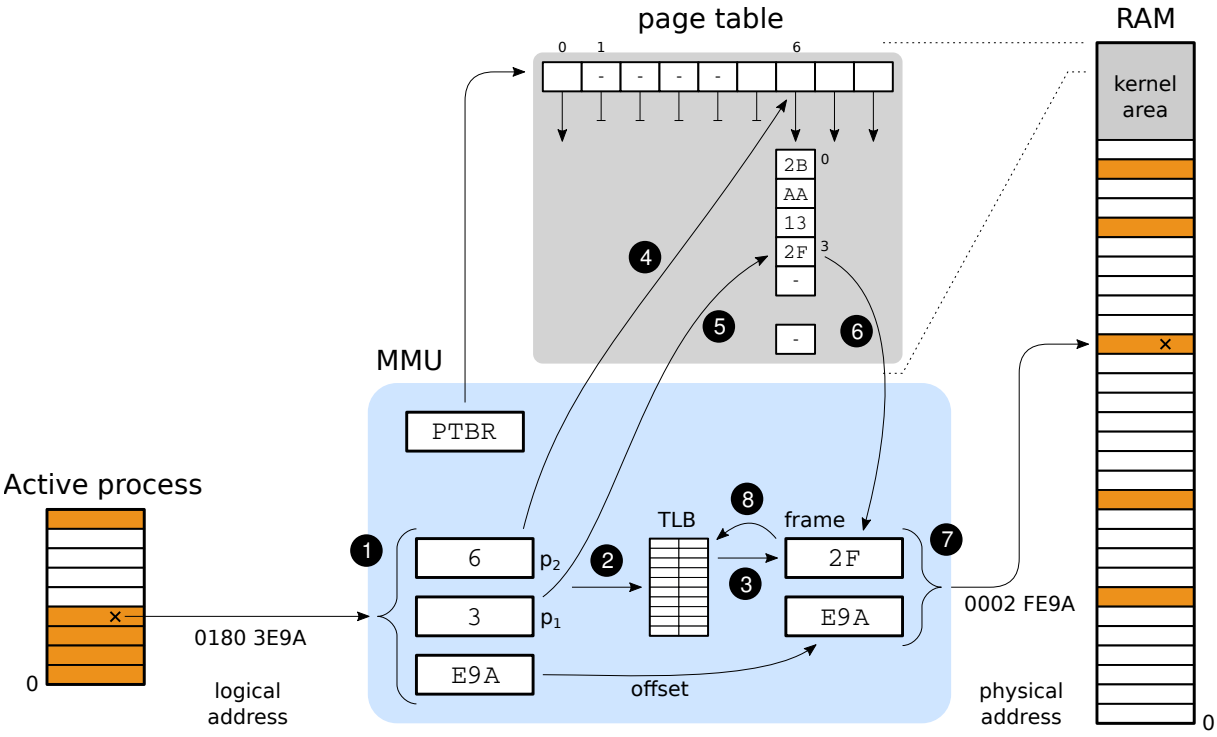

Figura 14.13: MMU com paginação e TLB.

O benefício do TLB pode ser estimado através do cálculo do tempo médio de acesso à memória, que é a média ponderada entre o tempo de acesso com acerto de TLB

[5] A busca de um número de quadro nas tabelas de páginas é comumente denominada *page table walk*; vários processadores têm hardware dedicado para realizar essa busca de forma otimizada.

(*hit*) e o tempo de acesso com erro (*miss*). Deve-se observar, entretanto, que o uso do TLB também adiciona custos em tempo: em processadores típicos, um *TLB hit* custa cerca de 1 ciclo de relógio, enquanto um *TLB miss* pode custar entre 10 e 100 ciclos (considerando o custo para acessar os vários níveis da tabela de páginas e atualizar o TLB).

Considerando como exemplo um sistema operando a 2 GHz (relógio de 0,5 ns) com tempo de acesso à memória RAM de 50 *ns* e um TLB com custo de acerto de 0,5 *ns* (um ciclo de relógio), custo de erro de 30 *ns* (60 ciclos de relógio) e uma taxa de acerto de TLB de 95%, o tempo médio de acesso à memória pode ser estimado como segue:

$$
\begin{aligned}
t_{médio} &= 95\% \times 0,5ns && \text{\# custo do acerto no TLB} \\
&+ (1 - 95\%) \times 30ns && \text{\# custo do erro no TLB} \\
&+ 50ns && \text{\# custo do acesso ao quadro em RAM} \\
t_{médio} &= 51,975ns
\end{aligned}
$$

Este resultado indica que o sistema de paginação multinível aumenta em 1,975 *ns* (4,0%) 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 caísse a 80%, o custo adicional seria de 12,8%; caso a taxa subisse a 99%, o custo adicional cairia para 1,6%.

Percebe-se então que, quanto maior a taxa de acertos do TLB (*TLB hit ratio*), melhor é o desempenho dos acessos à memória física. A taxa de acertos de um TLB é influenciada por diversos fatores:

Tamanho do TLB: 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, a arquitetura *Intel 80386* tinha um TLB com 64 entradas para páginas de dados e 32 entradas para páginas de código; por sua vez, o *Intel Core i7* possui 96 entradas para páginas de dados e 142 entradas para páginas de código.

Padrão de acessos à memória: processos que concentram seus acessos em poucas páginas de cada vez farão um uso eficiente do TLB, enquanto processos que acessam muitas páginas distintas em um curto período irão gerar frequentes erros de TLB, prejudicando seu desempenho no acesso à memória. Essa propriedade é conhecida como *localidade de referência* e será abordada na Seção 14.9.

Trocas de contexto: o conteúdo do TLB reflete a tabela de páginas do processo ativo em um dado momento. A cada troca de contexto, a tabela de páginas é substituída e portanto o TLB deve ser esvaziado, pois seu conteúdo não é mais válido. Em consequência, trocas de contexto muito frequentes prejudicam a eficiência de acesso à memória, tornando o sistema mais lento.

Política de substituição de entradas: o que ocorre quando há um erro de TLB e não há mais entradas livres no TLB? Em alguns processadores, a associação [*página*, *quadro*] que gerou o erro é adicionada ao TLB, substituindo a entrada mais antiga; todavia, na maioria dos processadores mais recentes, cada erro de TLB provoca uma interrupção, que transfere ao sistema operacional a tarefa de gerenciar o conteúdo do TLB [Patterson and Henessy, 2005].

14.8 Segmentos e páginas

Cada uma das principais formas de organização de memória vistas até agora tem suas vantagens: a organização por partições prima pela simplicidade e rapidez; a organização por segmentos oferece múltiplos espaços de endereçamento para cada processo, oferecendo flexibilidade ao programador; a organização por páginas oferece um grande espaço de endereçamento linear, enquanto elimina a fragmentação externa.

Vários processadores permitem combinar mais de uma forma de organização. Por exemplo, os processadores *Intel x86* permitem combinar a organização por segmentos com a organização por páginas, visando oferecer a flexibilidade dos segmentos com a baixa fragmentação das páginas. Nessa abordagem, os processos veem a memória estruturada em segmentos, conforme indicado na Figura 14.6. A MMU inicialmente converte os endereços lógicos na forma [*segmento*:*offset*] em endereços lógicos lineares (unidimensionais), usando as tabelas de descritores de segmentos (Seção 14.6). Em seguida, converte esse endereços lógicos lineares nos endereços físicos correspondentes, usando as tabelas de páginas.

Apesar do processador *Intel x86* oferecer as duas formas de organização 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 organização por páginas. O antigo DOS e o Windows 3.* usavam somente a organização por segmentos. O OS/2 da IBM foi um dos poucos sistemas operacionais comerciais a fazer uso pleno das possibilidades de organização de memória nessa arquitetura, combinando segmentos e páginas.

14.9 Localidade de referências

A forma como os processos acessam a memória tem um impacto direto na eficiência dos mecanismos de gerência de memória, sobretudo os caches de memória física, o cache da tabela de páginas (TLB, Seção 14.7.4) e o mecanismo de paginação em disco (Capítulo 17). 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, de 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;

Localidade espacial: um recurso será mais provavelmente acessado se outro recurso próximo a ele já foi acessado;

Localidade sequencial: é um caso particular da localidade espacial, no qual há uma predominância de acesso sequencial aos recursos: após o acesso a um recurso na posição p, há maior probabilidade de acessar um recurso na posição $p + 1$. É útil na otimização de sistemas de arquivos, por exemplo.

A Figura 14.14 ilustra o conceito de localidade de referências. Ela mostra as páginas acessadas durante uma execução do visualizador gráfico *gThumb*, ao abrir um arquivo de imagem JPEG. 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 (discutidas na Seção 15.2). Nessa execução, pode-se observar que os acessos à memória em cada momento da execução são concentrados em certas áreas do espaço de endereçamento. Quanto maior a concentração de acessos em poucas áreas, melhor a localidade de referências de um programa.

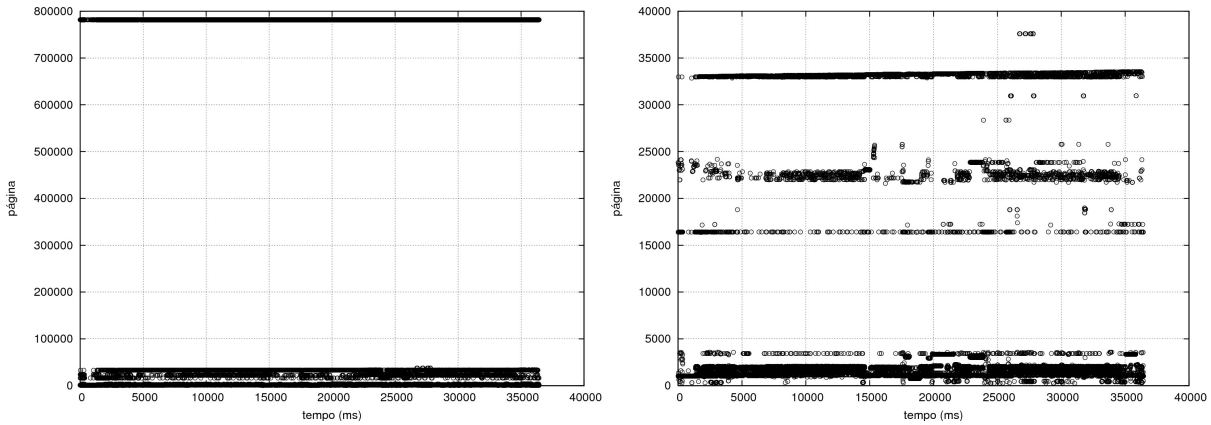

Figura 14.14: Distribuição dos acessos à memória do programa *gThumb*: visão geral (à esquerda) e detalhe da parte inferior (à direita).

Como exemplo da importância da localidade de referências, consideremos 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 ;   // preenche com algum valor
10 }
```

Também é possível preencher a matriz percorrendo-a coluna por coluna:

```
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 ;   // preenche com algum valor
10 }
```

Embora percorram a matriz de forma distinta, os dois programas são equivalentes e geram o mesmo resultado. 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 Figura 14.15 mostra o padrão de acesso à memória dos dois programas.

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 5 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.

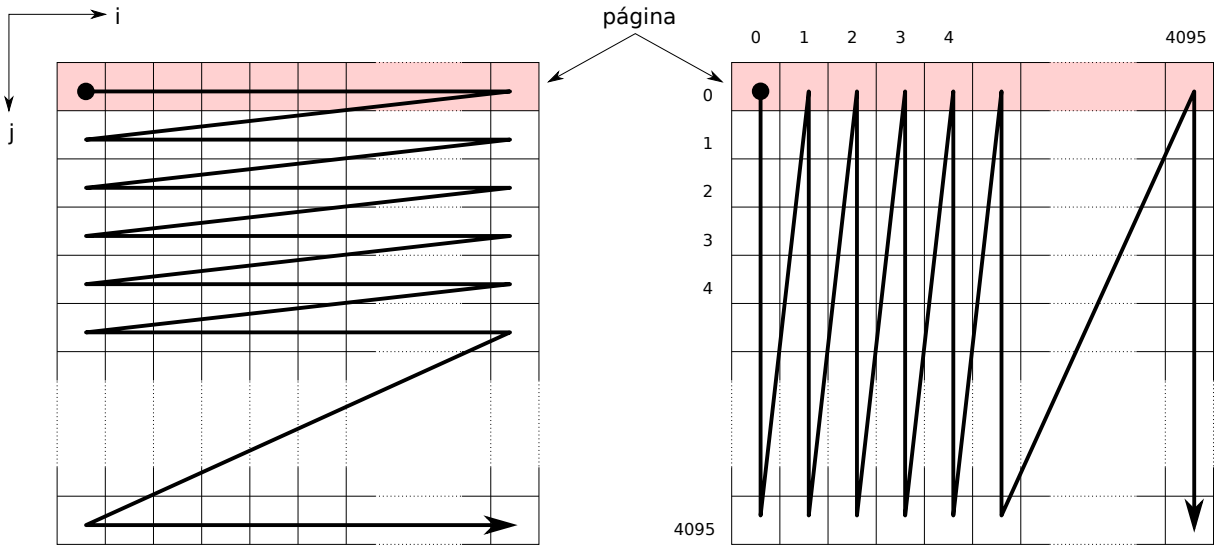

Figura 14.15: Comportamento dos programas no acesso à memória.

A diferença de comportamento das duas execuções pode ser observada na Figura 14.16, que mostra a distribuição dos endereços de memória acessados pelos dois códigos[6]. Nos gráficos, percebe-se claramente que a primeira implementação tem uma localidade de referências muito melhor 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.

[6]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.

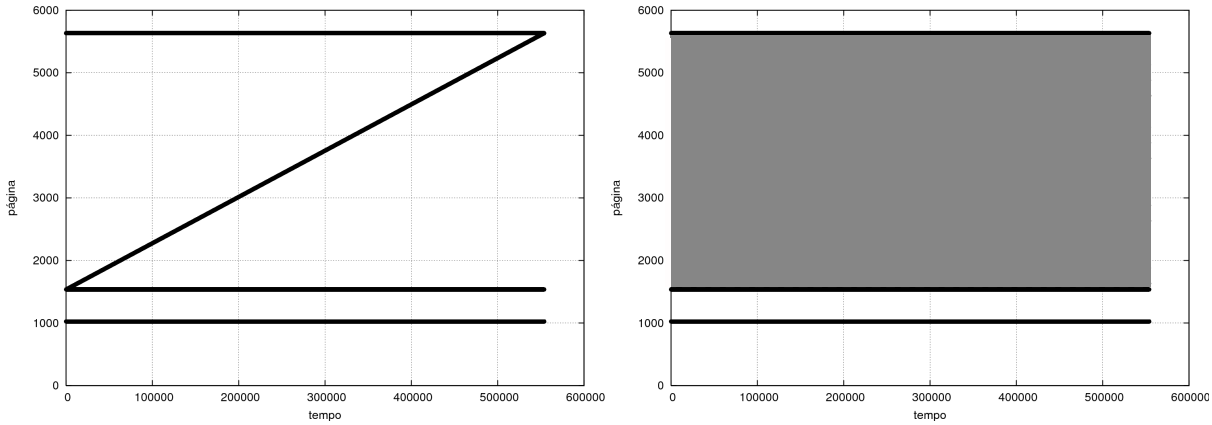

Figura 14.16: Localidade de referências nas duas execuções.

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. O compilador também pode alinhar as estruturas de dados mais usadas em relação às páginas.

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 aplicações interativas, etc.

Exercícios

1. Explique a diferença entre endereços *lógicos* e endereços *físicos* e as razões que justificam o uso de endereços lógicos.
2. O que é uma MMU – *Memory Management Unit*?
3. Seria possível e/ou viável implementar as conversões de endereços realizadas pela MMU em software, ao invés de usar um hardware dedicado? Por que?
4. Por que os tamanhos de páginas e quadros são sempre potências de 2?

5. Considerando a tabela de segmentos a seguir (com valores em decimal), calcule os endereços físicos correspondentes aos endereços lógicos 0:45, 1:100, 2:90, 3:1.900 e 4:200.

Segmento	0	1	2	3	4
Base	44	200	0	2.000	1.200
Limite	810	200	1.000	1.000	410

6. Considerando a tabela de páginas a seguir, com páginas de 500 bytes[7], informe os endereços físicos correspondentes aos endereços lógicos 414, 741, 1.995, 4.000 e 6.633, indicados em decimal.

página	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
quadro	3	12	6	–	9	–	2	–	0	5	–	–	–	7	–	1

7. Considere um sistema com endereços físicos e lógicos de 32 bits, que usa tabelas de páginas com três níveis. Cada nível de tabela de páginas usa 7 bits do endereço lógico, sendo os restantes usados para o *offset*. Cada entrada das tabelas de páginas ocupa 32 bits. Calcule, indicando seu raciocínio:

   (a) O tamanho das páginas e quadros, em bytes.

   (b) O tamanho máximo de memória que um processo pode ter, em bytes e páginas.

   (c) O espaço ocupado pela tabela de páginas para um processo com apenas uma página de código, uma página de dados e uma página de pilha. As páginas de código e de dados se encontram no inicio do espaço de endereçamento lógico, enquanto a pilha se encontra no final do mesmo.

   (d) Idem, caso todas as páginas do processo estejam mapeadas na memória.

8. Explique o que é TLB, qual a sua finalidade e como é seu funcionamento.

9. Sobre as afirmações a seguir, relativas à alocação por páginas, indique quais são incorretas, justificando sua resposta:

   (a) Um endereço lógico com N bits é dividido em P bits para o número de página e $N - P$ bits para o deslocamento em cada página.

   (b) As tabelas de páginas multiníveis permitem mais rapidez na conversão de endereços lógicos em físicos.

   (c) O bit de referência R associado a cada página é "ligado" pela MMU sempre que a página é acessada.

   (d) O cache TLB é usado para manter páginas frequentemente usadas na memória.

[7] Um tamanho de página de 500 bytes permite fazer os cálculos mentalmente, sem a necessidade de converter os endereços para binário e vice-versa, bastando usar divisões inteiras (com resto) entre os endereços e o tamanho de página.

(e) O bit de modificação *M* associado a cada página é "ligado" pelo núcleo sempre que um processo modificar o conteúdo da mesma.

(f) O cache TLB deve ser esvaziado a cada troca de contexto entre processos.

10. Por que é necessário limpar o cache TLB após cada troca de contexto entre processos? Por que isso não é necessário nas trocas de contexto entre threads?

11. Um sistema de memória virtual paginada possui tabelas de página com três níveis e tempo de acesso à memória RAM de 100 ns. O sistema usa um cache TLB de 64 entradas com taxa estimada de acerto de 98% e tempo de consulta de 10 ns. Considerando que na ocorrência um *cache miss* a tabela de páginas terá de ser consultada, calcule o tempo médio estimado de acesso à memória pelo processador e explique seu raciocínio.

12. Considerando um sistema de 32 bits com páginas de 4 KBytes e um TLB com 64 entradas, calcule quantos erros de cache TLB são gerados pela execução de cada um dos laços a seguir. Considere somente os acessos à matriz `buffer` (linhas 5 e 9), ignorando páginas de código, *heap* e *stack*. Indique seu raciocínio.

```
1 unsigned char buffer[4096][4096] ;
2
3 for (int i=0; i<4096; i++)  // laço 1
4    for (int j=0; j<4096; j++)
5       buffer[i][j] = 0 ;
6
7 for (int j=0; j<4096; j++)  // laço 2
8    for (int i=0; i<4096; i++)
9       buffer[i][j] = 0 ;
```

Referências

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

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

D. Patterson and J. Henessy. *Organização e Projeto de Computadores*. Campus, 2005.

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