Capítulo 3

Gerenciamento de memória

A memória principal (RAM) é um recurso importante que deve ser gerenciado com muito cuidado. Apesar de atualmente os computadores pessoais possuírem memórias dez mil vezes maiores que o IBM 7094 (o maior computador do mundo no início dos anos 1960), os programas tornam-se maiores muito mais rapidamente do que as memórias. Parafraseando a Lei de Parkinson, pode-se afirmar que "programas tendem a se expandir a fim de ocupar toda a memória disponível". Neste capítulo, estudaremos como os sistemas operacionais criam abstrações a partir da memória e como as gerenciam.

O que todo programador deseja é dispor de uma memória infinitamente grande, rápida e não volátil, ou seja, uma memória que não perdesse seu conteúdo quando faltasse energia. E por que não também a um baixo custo? Infelizmente, a tecnologia atual não comporta essas memórias. Talvez você seja capaz de desenvolvê-las.

Qual é a segunda opção? Ao longo dos anos, as pessoas descobriram o conceito de **hierarquia de memórias**, em que os computadores têm alguns megabytes de memória cache muito rápida, de custo alto e volátil, alguns gigabytes de memória principal volátil de velocidade e custo médios e alguns terabytes de armazenagem em disco não volátil de velocidade e custo baixos, para não falar do armazenamento removível, como DVDs e dispositivos USB. A função do sistema operacional é abstrair essa hierarquia em um modelo útil e, então, gerenciar a abstração.

A parte do sistema operacional que gerencia (parcialmente) a hierarquia de memórias é denominada **gerenciador de memória**. Sua função é gerenciar a memória de modo eficiente: manter o controle de quais partes da memória estão em uso e quais não estão, alocando memória aos processos quando eles precisam e liberando-a quando esses processos terminam.

Neste capítulo conheceremos vários esquemas diferentes de gerenciamento de memória, desde os mais simples aos mais sofisticados. Visto que o gerenciamento do nível inferior da memória cache normalmente é feito pelo hardware, o foco deste capítulo será o modelo da memória principal do programador e como pode ser bem gerenciado. As abstrações para armazenamento permanente — o disco — e o gerenciamento dessas abstrações são o tema do próximo capítulo. Começaremos pelo sistema mais simples possível e, então, avançaremos gradualmente para os mais elaborados.

3.1 Sem abstração de memória

A abstração de memória mais simples é a ausência de abstração. Os primeiros computadores de grande porte (antes de 1960), microcomputadores (antes de 1970) e computadores pessoais (antes de 1980) não possuíam abstração de memória. Cada programa simplesmente considerava a memória física. Quando um programa executava uma instrução como

```
MOV REGISTER1,1000
```

o computador apenas movia o conteúdo da memória física da posição 1000 para *REGISTER1*. Assim, o modelo de memória apresentado ao programador era simplesmente a memória física, um conjunto de endereços de 0 a algum máximo, cada endereço correspondendo a uma célula que continha certo número de bits, normalmente oito.

Nessas condições, não era possível executar dois programas na memória simultaneamente. Se o primeiro programa escrevesse um novo valor para a posição 2000, por exemplo, apagaria qualquer valor que o segundo programa estivesse armazenando ali. Nenhum deles funcionaria e os dois programas quebrariam quase imediatamente.

Ainda que o modelo da memória fosse apenas a memória física, havia várias opções possíveis. São mostradas três variações na Figura 3.1. O sistema operacional pode estar na parte inferior da memória em RAM (*random acess memory* — memória de acesso aleatório), como mostrado na Figura 3.1(a) ou pode estar em ROM (*read-only memory* — memória apenas para leitura) na parte superior da memória, como mostrado na Figura 3.1(b) ou os drivers de dispositivo podem estar na parte superior da memória em ROM e o resto do sistema em RAM embaixo, como mostrado na Figura 3.1(c). O primeiro modelo era usado antigamente em computadores de grande porte e minicomputadores e raramente foi utilizado depois disso. O segundo modelo é usado em alguns computadores portáteis e sistemas embarcados. O terceiro modelo foi empregado nos primeiros computadores pessoais (por exemplo, executando o MS-DOS), em que a porção do sistema na ROM é chamada de **BIOS** (*basic input output system* — sistema básico de E/S). Os modelos (a) e (c) apresentam a desvantagem da possibilidade de que um erro no programa do usuário apague o sistema operacional, possivelmente com resultados desastrosos (como a adulteração do disco).

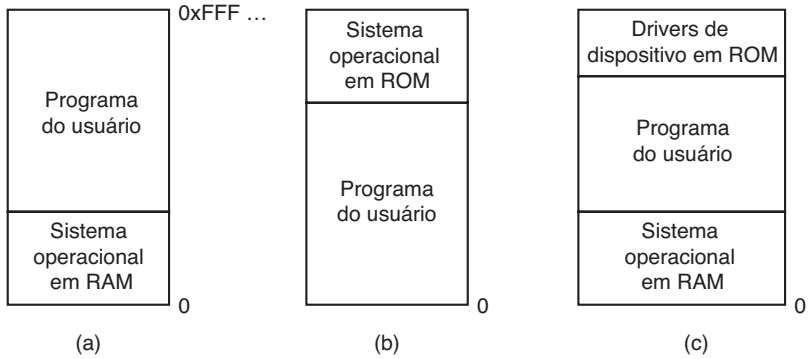

Figura 3.1 Três modos simples de organizar a memória com um sistema operacional e um processo de usuário. Também existem outras possibilidades.

Quando o sistema é organizado dessa forma, geralmente apenas um processo pode ser executado por vez. Assim que o usuário digita um comando, o sistema operacional copia o programa solicitado do disco para a memória e o executa. Quando o processo termina, o sistema operacional exibe um prompt e espera por um novo comando. Quando recebe o novo comando, carrega um novo programa na memória, sobrescrevendo o primeiro.

Um modo de obter algum grau de paralelismo em um sistema sem abstração de memória é programar com múltiplos threads. Uma vez que se suponha que todos os threads em um processo consideram a mesma imagem da memória, o fato destes serem forçados não é um problema. Embora essa ideia funcione, sua utilidade é limitada porque as pessoas normalmente desejam que programas *não relacionados* sejam executados ao mesmo tempo, algo que a abstração de threads não proporciona. Além do mais, é improvável que sistemas tão primitivos, que não comportam abstração de memória, sejam capazes de fornecer abstração de threads.

Executando múltiplos programas sem abstração de memória

Mesmo sem abstração de memória, entretanto, é possível executar múltiplos programas simultaneamente. O que o sistema operacional deve fazer é salvar o conteúdo completo da memória em um arquivo de disco e, em seguida, introduzir e executar o próximo programa. Contanto que haja apenas um programa por vez na memória, não há conflitos. Esse conceito (troca de processos, *swapping*) será discutido adiante.

O acréscimo de hardware especial possibilita executar múltiplos programas simultaneamente, mesmo sem troca de processos. Os primeiros modelos do IBM 360 resolveram o problema do seguinte modo. A memória foi dividida em blocos de 2 KB e a cada um foi atribuída uma chave de proteção de 4 bits mantida em registradores especiais dentro da CPU. Uma máquina com uma memória de 1 MB precisava de apenas 512 desses registradores de 4 bits para um total de 256 bytes de armazenamento de chaves. A PSW (*program status word* — palavra de status do programa) também continha uma chave de 4 bits. O hardware 360 interrompia qualquer tentativa de um processo em execução de acessar a memória com um código de proteção diferente da chave PSW. Visto que apenas o sistema operacional poderia alterar as chaves de proteção, impedia-se que os processos do usuário interferissem no funcionamento mútuo e do próprio sistema operacional.

Entretanto, essa solução tem uma desvantagem importante, ilustrada na Figura 3.2. Temos aqui dois programas, cada um com 16 KB, como mostrado na Figura 3.2(a) e (b). A primeira é sombreada para indicar que tem uma chave de memória diferente da segunda. O primeiro programa inicializa saltando para o endereço 24, que contém uma instrução MOV. O segundo programa inicializa saltando para o endereço 28, que contém uma instrução CMP. As instruções irrelevantes para essa discussão não são mostradas. Quando os dois programas são carregados sucessivamente na memória, começando no endereço 0, temos a situação da Figura 3.2(c). Neste exemplo, supomos que o sistema operacional está na região alta memória e, dessa forma, não é mostrado.

Depois de serem carregados, os programas podem ser executados. Visto que têm chaves de memória diferentes, um não pode danificar o outro. Mas o problema é de natureza diferente. Quando o primeiro programa é inicializado, executa a instrução JMP 24, que salta para a instrução como esperado. Esse programa funciona normalmente.

No entanto, após ter executado o primeiro programa por tempo suficiente, o sistema operacional pode decidir executar o segundo programa, que foi carregado acima do primeiro no endereço 16.384. A primeira instrução executada é JMP 28, que salta para a instrução ADD do primeiro programa, em vez de saltar para a instrução esperada, a CMP. O programa provavelmente quebrará muito antes de 1 segundo.

O problema central nesse caso é que ambos os programas referenciam a memória física absoluta e não é isso o que queremos. Desejamos que cada programa referencie um conjunto privado de endereços que seja local para

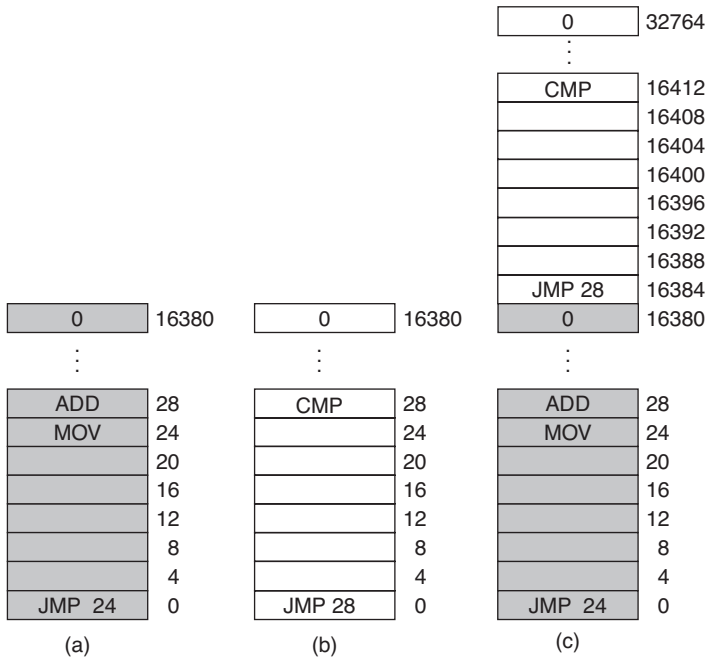

Figura 3.2 Ilustração do problema de realocação. (a) Um programa de 16 KB. (b) Outro programa de 16 KB. (c) Os dois programas carregados consecutivamente na memória.

o programa. Explicaremos em poucas palavras como isso pode ser realizado. A solução temporária encontrada pelo IBM 360 foi modificar o segundo programa dinamicamente à medida que o carregava na memória, usando uma técnica conhecida como **realocação estática**, que funciona da seguinte forma: quando um programa era carregado no endereço 16.384, a constante 16.384 era adicionada a todos os endereços de programa durante o processo de carregamento. Embora esse mecanismo funcione se executado corretamente, não é uma solução muito comum e deixa mais lento o carregamento. Além do mais, requer informações adicionais em todos os programas executáveis para indicar quais palavras contêm ou não endereços (relocalizáveis). Afinal, o '28' na Figura 3.2(b) deve ser relocalizado, mas uma instrução como

```
MOV REGISTER1,28
```

que move o número 28 para *REGISTER1* não deve ser relocalizada. O carregador precisa de algum modo para dizer o que é um endereço e o que é uma constante.

Por fim, como indicamos no Capítulo 1, a história tende a se repetir no ramo da computação. Embora o endereçamento direto de memória física seja uma memória distante (perdoem-me) em computadores de grande porte, minicomputadores, computadores de mesa e notebooks, a ausência de abstração de memória ainda é comum em sistemas embarcados e de cartões inteligentes (*smart cards*). Dispositivos como rádios, máquinas de lavar e fornos de micro-ondas atualmente têm muitos softwares (na ROM) e, na maioria dos casos, o software se endereça à memória absoluta. Isso funciona porque todos os programas são conhecidos antecipadamente e os usuários não são livres para executar seu próprio software na torradeira.

Sistemas embarcados sofisticados (como telefones celulares) têm sistemas operacionais elaborados, ao passo que sistemas mais simples não os têm. Em alguns casos, há um sistema operacional, mas é apenas uma biblioteca vinculada com o programa aplicativo e que fornece chamadas de sistema para executar E/S e outras tarefas comuns. O popular sistema operacional **e-cos** é um exemplo comum de um sistema operacional como biblioteca.

3.2 Abstração de memória: espaços de endereçamento

De modo geral, a exposição da memória física a processos apresenta várias desvantagens importantes. Primeiro, se os programas do usuário podem endereçar cada byte de memória, podem, intencional ou acidentalmente, danificar o sistema e paralisá-lo facilmente (a menos que haja hardwares especiais como o bloqueio e esquema de chave do IBM 360). Esse problema existe mesmo se apenas um programa de usuário (aplicação) estiver sendo executado. Segundo, com esse modelo, é difícil executar múltiplos programas simultaneamente (alternando-se, se houver apenas uma CPU). Em computadores pessoais, é comum ter vários programas abertos ao mesmo tempo (um processador de texto, um programa de e-mail e um navegador da Web, por exemplo), com um deles como foco no momento

e os demais reativados pelo click do mouse. Visto que essa situação dificilmente é alcançada quando não há abstração da memória física, algo deve ser feito.

3.2.1 | A noção de espaço de endereçamento

Para que múltiplas aplicações estejam na memória simultaneamente sem interferência mútua, dois problemas devem ser resolvidos: proteção e realocação. Examinamos uma solução primitiva para o primeiro, usada no IBM 360: rotular blocos da memória com uma chave de proteção e comparar a chave do processo em execução com a de cada palavra da memória recuperada. Entretanto, essa abordagem por si só não resolve o último problema, embora ele possa ser resolvido realocando os programas quando são carregados, mas essa é uma solução lenta e complicada.

Uma solução melhor é inventar uma abstração para a memória: o espaço de endereçamento. Assim como o conceito de processo cria um tipo de CPU abstrata para executar programas, o espaço de endereçamento cria um tipo de memória abstrata para abrigá-los. Um **espaço de endereçamento** é o conjunto de endereços que um processo pode usar para endereçar a memória. Cada processo tem seu próprio espaço de endereçamento, independente dos que pertencem a outros processos (exceto em algumas circunstâncias especiais em que os processos desejam compartilhar seus espaços de endereçamento).

O conceito de espaço de endereçamento é muito geral e ocorre em muitos contextos. Considere os números de telefone. Nos Estados Unidos e em muitos outros países, um número de telefone local é normalmente um número de 7 dígitos. O espaço de endereçamento para números de telefone varia, desse modo, de 0.000.000 a 9.999.999, embora alguns números, como os começados por 000, não sejam usados. Com o aumento de telefones celulares, modems e máquinas de fax, esse espaço está se tornando muito pequeno, caso em que mais dígitos têm de ser usados. O espaço de endereçamento para portas de E/S no Pentium varia de 0 a 16383. Endereços IPv4 são números de 32 bits; portanto, seu espaço de endereçamento varia de 0 a $2^{32} - 1$ (novamente com alguns números reservados).

Os espaços de endereçamento não precisam ser numéricos. O conjunto de domínios da Internet *.com* também é um espaço de endereçamento. Esse espaço de endereçamento consiste de todas as cadeias de comprimento de 2 a 63 caracteres que possam ser feitas usando letras, números e hífens, seguidas por *.com*. Agora você já deve ter uma ideia do que são os espaços de endereçamento. É algo bastante simples.

Dar a cada programa seu próprio espaço de endereçamento, de modo que o endereço 28 em um programa signifique uma localização física diferente do endereço 28 em outro programa, é um pouco mais difícil. Adiante, discutiremos um modo simples, antes bastante comum, mas que caiu em desuso em decorrência da capacidade de colocação de esquemas muito mais complicados (e melhores) em chips de CPUs modernas.

Registradores-base e registradores-limite

Essa solução simples usa uma versão particularmente simples da **realocação dinâmica**, que mapeia cada espaço de endereçamento do processo em uma parte diferente da memória física de modo simples. A solução clássica, que foi usada em máquinas desde o CDC 6600 (o primeiro supercomputador do mundo) ao Intel 8088 (o coração do PC IBM original), é equipar cada CPU com dois registradores de hardware especiais, normalmente chamados de **registradores-base** e **registradores-limite**. Quando registradores-base e registradores-limite são usados, os programas são localizados em posições consecutivas na memória, onde haja espaço e sem realocação durante o carregamento, como mostrado na Figura 3.2(c). Quando um processo é executado, o registrador-base é carregado com o endereço físico onde seu programa começa na memória e o registrador-limite é carregado com o comprimento do programa. Na Figura 3.2(c), o valor-base e o valor-limite que seriam carregados nesses registradores do hardware quando o primeiro programa é executado são 0 e 16.384, respectivamente. Os valores usados quando o segundo programa é executado são 16.384 e 32.768, respectivamente. Se um terceiro programa de 16 KB fosse carregado diretamente acima do segundo e executado, o registrador-base e o registrador-limite seriam 32.768 e 16.384.

Cada vez que um processo referencia a memória, seja para recuperar uma instrução, seja para ler ou escrever uma palavra de dados, o hardware da CPU automaticamente acrescenta o valor-base ao endereço gerado pelo processo antes de enviar o endereço ao barramento da memória. Simultaneamente, ele verifica se o endereço oferecido é igual ou maior que o valor do registrador-limite, caso no qual um erro é gerado e o acesso é abortado. Desse modo, no caso da primeira instrução do segundo programa na Figura 3.2(c), o processo executa uma instrução

```
JMP 28
```

mas o hardware a trata como se fosse

```
JMP 16412
```

de modo que ele pula para a instrução CMP como era esperado. As configurações do registrador-base e do registrador-limite durante a execução do segundo programa da Figura 3.2(c) são mostradas na Figura 3.3.

Usar registradores-limite e registradores-base é um modo fácil de dar a cada processo seu próprio espaço de endereçamento privado porque cada endereço de memória gerado automaticamente tem o conteúdo do registrador-base acrescentado a ele antes de ser enviado à memória. Em muitas implementações, o registrador-base e o registrador-limite são protegidos de modo que apenas o sistema

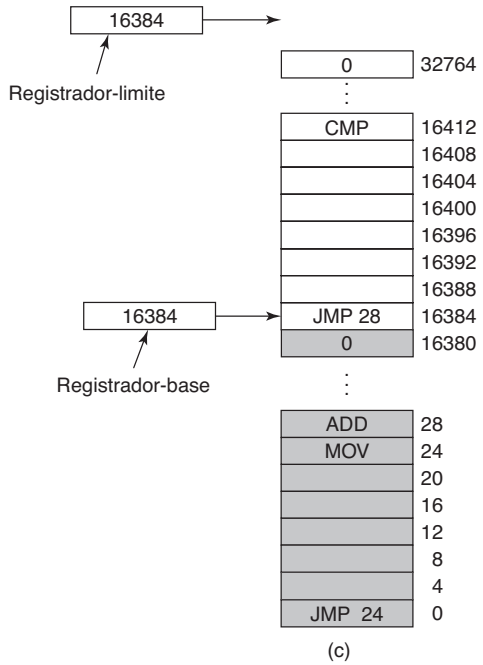

Figura 3.3 O registrador-limite e o registrador-base podem ser usados para dar a cada processo um espaço de endereçamento independente.

operacional possa modificá-los. Esse foi o caso do CDC 6600, mas não do Intel 8088, que nem mesmo tinha o registrador-limite. Ele tinha, contudo, múltiplos registradores-base, que permitiam que dados e textos de programas, por exemplo, fossem realocados independentemente, mas que não ofereciam proteção contra referências à memória além da capacidade.

Uma desvantagem da realocação usando registradores-limite e registradores-base é a necessidade de executar uma adição e uma comparação em cada referência à memória. As comparações podem ser feitas rapidamente, mas, a menos que circuitos de adição especiais sejam utilizados, as adições são demoradas em virtude do tempo de propagação do 'vai um'.

3.2.2 Troca de memória

Se a memória física do computador for grande o suficiente para armazenar todos os processos, os esquemas descritos até agora bastarão. Mas, na prática, a quantidade total de RAM necessária para todos os processos é frequentemente muito maior do que a memória pode comportar. Em um sistema Windows ou Linux típico, algo como 40–60 processos ou mais podem ser inicializados quando o computador é inicializado. Por exemplo, quando uma aplicação do Windows é instalada, frequentemente emite comandos de modo que, em subsequentes inicializações do sistema, seja lançado um processo que não faça nada além de verificar atualizações para a aplicação. Tal processo pode facilmente ocupar 5–10 MB de memória. Outros processos secundários verificam a chegada de correspondência eletrônica, de conexões do sistema de rede e muitas outras coisas. E tudo isso antes que o primeiro programa do usuário seja inicializado. Atualmente, programas sérios de aplicação de usuários podem utilizar facilmente de 50 a 200 MB ou mais. Consequentemente, a manutenção de todos esses processos na memória o tempo todo requer uma quantidade de memória enorme e não pode ser realizada se a memória for insuficiente.

Dois métodos gerais para lidar com a sobrecarga de memória têm sido desenvolvidos com o passar dos anos. A estratégia mais simples, denominada troca de processos (**swapping**), consiste em trazer, em sua totalidade, cada processo para a memória, executá-lo durante um certo tempo e, então, devolvê-lo ao disco. Processos ociosos muitas vezes são armazenados no disco, de forma que não ocupem qualquer espaço na memória quando não estão executando (embora alguns deles 'acordem' periodicamente para fazer seu trabalho, e, em seguida, vão 'dormir' novamente). A outra estratégia, denominada **memória virtual**, permite que programas possam ser executados mesmo que estejam apenas parcialmente carregados na memória principal. Estudaremos a seguir troca de processos; na Seção 3.3, trataremos de memória virtual.

O funcionamento de um sistema de troca de processos é ilustrado na Figura 3.4. Inicialmente, somente o processo *A* está na memória. Em seguida, os processos *B* e *C* são criados ou trazidos do disco. Na Figura 3.4(d), o processo *A* é devolvido ao disco. Então, o processo *D* entra na memória e, em seguida, o processo *B* é retirado. Por fim, o processo *A* é novamente trazido do disco para a memória. Como o processo *A* está agora em uma localização diferente na memória, os endereços nele contidos devem ser relocados via software durante a carga na memória ou, mais provavelmente, via hardware durante a execução do programa. Por exemplo, registradores-base e registradores-limite funcionariam bem aqui.

Quando as trocas de processos deixam muitos espaços vazios na memória, é possível combiná-los todos em um único espaço contíguo de memória, movendo-os, o máximo possível, para os endereços mais baixos. Essa técnica é denominada **compactação de memória**. Ela geralmente não é usada em virtude do tempo de processamento necessário. Por exemplo, uma máquina com 1 GB de memória e que possa copiar 4 bytes em 20 ns gastaria cerca de 5 segundos para compactar toda a memória.

Um ponto importante a ser considerado relaciona-se com a quantidade de memória que deve ser alocada a um processo quando este for criado ou trazido do disco para a memória. Se processos são criados com um tamanho fixo, inalterável, então a alocação é simples: o sistema operacional alocará exatamente aquilo que é necessário, nem mais nem menos.

Contudo, se a área de dados do processo puder crescer — por exemplo, alocando dinamicamente memória de

uma área temporária (*heap*), como em muitas linguagens de programação —, problemas poderão ocorrer sempre que um processo tentar crescer. Se houver um espaço livre disponível adjacente ao processo, ele poderá ser alocado e o processo poderá crescer nesse espaço. Por outro lado, se estiver adjacente a outro processo, o processo que necessita crescer poderá ser movido para uma área de memória grande o suficiente para contê-lo ou um ou mais processos terão de ser transferidos para o disco a fim de criar essa área disponível. Se o processo não puder crescer na memória e a área de troca de disco (*swap*) estiver cheia, o processo deverá ser supenso até que algum espaço seja liberado (ou pode ser terminado).

Se o esperado é que a maioria dos processos cresça durante a execução, provavelmente será uma boa ideia alocar uma pequena memória extra sempre que se fizer a transferência de um processo para a memória ou a movimentação dele na memória, a fim de reduzir o custo (extra) associado à movimentação ou à transferência de processos que não mais cabem na memória alocada a eles. Contudo, quando os processos forem transferidos de volta para o disco, somente a memória realmente em uso deverá ser transferida, pois é desperdício efetuar também a transferência da memória extra. Na Figura 3.5(a) vemos uma configuração de memória na qual o espaço para crescimento foi alocado a dois processos.

Se os processos puderem ter duas áreas em expansão — por exemplo, a área de dados sendo usada como área temporária (*heap*) para variáveis dinamicamente alocadas e liberadas e uma área de pilha para variáveis locais comuns e para endereços de retorno —, então uma solução poderá ser a mostrada na Figura 3.5(b). Nessa figura, vemos que cada processo tem uma pilha no topo de sua memória alo-

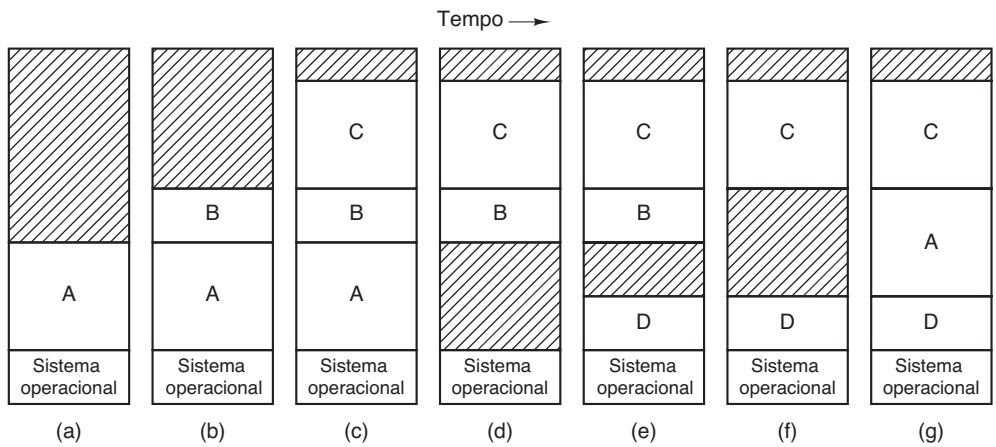

Figura 3.4 Alterações na alocação de memória à medida que processos entram e saem dela. As regiões sombreadas correspondem a regiões da memória não utilizadas naquele instante.

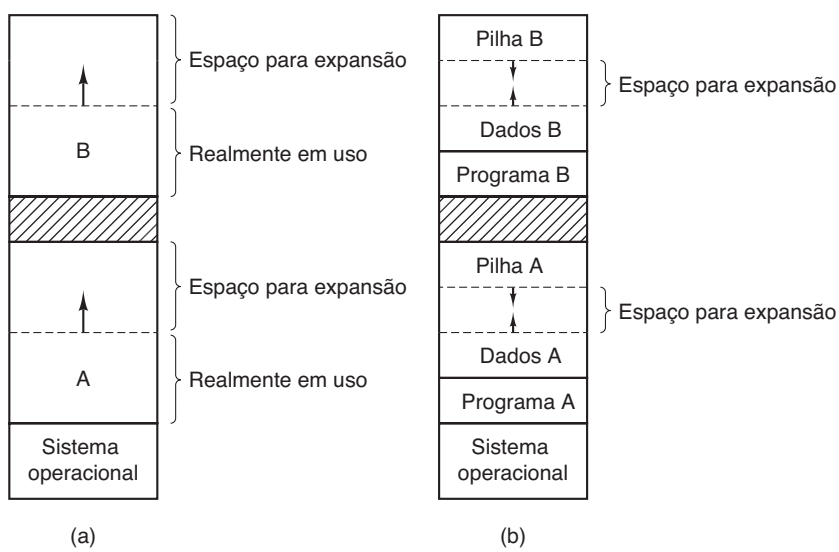

Figura 3.5 (a) Alocação de espaço para um segmento de dados em expansão. (b) Alocação de espaço para uma pilha e um segmento de dados em crescimento.

cada, crescendo para baixo, e uma área de dados adjacente ao código do programa, crescendo para cima. A porção de memória situada entre essas duas áreas pode ser usada por ambas. Se essa porção de memória assim situada for toda ocupada, então o processo terá de ser movido para outra área com espaço suficiente, ser transferido para disco e esperar até que uma área de memória grande o bastante possa ser criada ou ser terminado.

3.2.3 Gerenciando a memória livre

Quando a memória é atribuída dinamicamente, o sistema operacional deve gerenciá-la. De modo geral, há dois modos de verificar a utilização da memória: mapas de bits e listas livres. Nesta seção e na próxima examinaremos esses dois métodos.

Gerenciamento de memória com mapa de bits

Com um mapa de bits, a memória é dividida entre unidades de alocação tão pequenas quanto palavras ou tão grandes como vários kilobytes. A cada unidade de alocação corresponde um bit no mapa de bits, o qual é 0 se a unidade estiver livre e 1 se estiver ocupada (ou vice-versa). A Figura 3.6 mostra parte da memória e o mapa de bits correspondente.

O tamanho da unidade de alocação é um item importante no projeto. Quanto menor a unidade de alocação, maior será o mapa de bits. Contudo, mesmo com uma unidade de alocação tão pequena quanto 4 bytes, 32 bits de memória necessitarão de apenas 1 bit no mapa de bits. Assim, uma memória com 32n bits usará um mapa de n bits, de modo que o mapa de bits ocupará somente 1/33 da memória. Se a unidade de alocação for definida como grande, o mapa de bits será menor, mas uma quantidade apreciável de memória poderá ser desperdiçada na última unidade do processo se o tamanho do processo não for exatamente múltiplo da unidade de alocação.

Um mapa de bits é uma maneira simples de gerenciar palavras na memória em uma quantidade fixa de memória, pois o tamanho desse mapa de bits só depende do tamanho da memória e da unidade de alocação. O principal problema com essa técnica é que, quando se decide carregar na memória um processo com tamanho de k unidades, o gerenciador de memória precisa encontrar espaço disponível na memória procurando no mapa de bits uma sequência de k bits consecutivos em 0. Essa operação de busca por uma sequência de ∅s é muito lenta (porque esta sequência pode ultrapassar limites de palavras no mapa), o que constitui um argumento contra os mapas de bits.

Gerenciamento de memória com listas encadeadas

Outra maneira de gerenciar o uso de memória é manter uma lista encadeada de segmentos de memória alocados e disponíveis. Um segmento é tanto uma área de memória alocada a um processo como uma área de memória livre situada entre as áreas de dois processos. A memória da Figura 3.6(a) é representada na Figura 3.6(c) como uma lista encadeada de segmentos. Cada elemento dessa lista encadeada especifica um segmento de memória livre (L) ou um segmento de memória alocado a um processo (P), o endereço onde se inicia esse segmento, seu comprimento e um ponteiro para o próximo elemento da lista.

Nesse exemplo, a lista de segmentos é mantida ordenada por endereço. Essa ordenação apresenta a vantagem de permitir uma atualização rápida e simples da lista sempre que um processo terminar sua execução ou for removido da memória. Um processo que termina sua execução geralmente tem dois vizinhos na lista encadeada de segmentos de memória (exceto quando estiver no início ou no fim dessa lista). Esses vizinhos podem ser ou segmentos de memória alocados a processos ou segmentos de memória livres e, assim, as quatro combinações da Figura 3.7 são possíveis. Na Figura 3.7(a), um segmento de memória é liberado e a atualização da lista é feita pela substituição de um P por um L. Nas figuras 3.7(b) e 3.7(c), dois segmentos de memória são concatenados em um único; a lista fica,

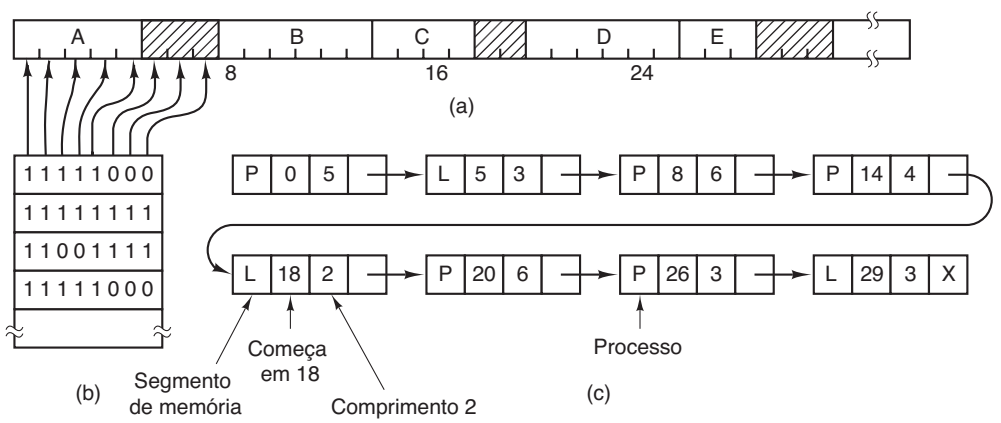

Figura 3.6 (a) Parte da memória com cinco processos e três segmentos de memória. As marcas mostram as unidades de alocação de memória. As regiões sombreadas (0 no mapa de bits) estão livres. (b) O mapa de bits correspondente. (c) A mesma informação como lista.

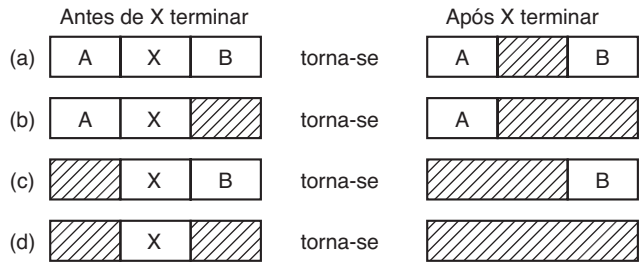

Figura 3.7 Quatro combinações de vizinhos para o processo que termina, *X*.

desse modo, com um item a menos. Na Figura 3.7(d), três segmentos de memória são transformados em um único e dois itens são eliminados da lista.

Como a entrada da tabela de processos referente a um processo em estado de término de execução geralmente aponta para a entrada da lista encadeada associada a esse referido processo, uma lista duplamente encadeada pode ser mais adequada, em vez da lista com encadeamento simples da Figura 3.6(c). Uma lista duplamente encadeada torna mais fácil encontrar o item anterior na lista, a fim de verificar a possibilidade de uma mistura.

Quando os segmentos de memória alocados a processos e segmentos de memória livres são mantidos em uma lista ordenada por endereço, é possível utilizar diversos algoritmos para alocar memória a um processo recém-criado (ou a um processo já existente em disco que esteja sendo transferido para a memória). Vamos supor que o gerenciador de memória saiba quanta memória deve ser alocada ao processo. O algoritmo mais simples é o **first fit** (primeiro encaixe). O gerenciador de memória procura ao longo da lista de segmentos de memória por um segmento livre que seja suficientemente grande para esse processo. Esse segmento é, então, quebrado em duas partes, uma das quais é alocada ao processo, e a parte restante transforma-se em um segmento de memória livre de tamanho menor, exceto no caso improvável de o tamanho do segmento de memória alocado ao processo se ajustar exatamente ao tamanho do segmento de memória livre original. O algoritmo *first fit* é rápido porque procura o menos possível.

Uma alteração menor no algoritmo *first fit* resulta no algoritmo **next fit** (próximo encaixe). O algoritmo *next fit* funciona da mesma maneira que o algoritmo *first fit*, exceto pelo fato de sempre memorizar a posição em que encontra um segmento de memória disponível de tamanho suficiente. Quando o algoritmo *next fit* tornar a ser chamado para encontrar um novo segmento de memória livre, ele inicializará sua busca a partir desse ponto, em vez de procurar sempre a partir do início da lista, tal como faz o algoritmo *first fit*. Simulações feitas por Bays (1977) mostraram que o algoritmo *next fit* fornece um desempenho ligeiramente inferior ao do algoritmo *first fit*.

Outro algoritmo bastante conhecido é o **best fit** (melhor encaixe). Esse algoritmo pesquisa a lista inteira e escolhe o menor segmento de memória livre que seja adequado ao processo. Em vez de escolher um segmento de memória disponível grande demais que poderia ser necessário posteriormente, o algoritmo *best fit* tenta encontrar um segmento de memória disponível cujo tamanho seja próximo do necessário ao processo, para que haja maior correspondência entre a solicitação e os segmentos disponíveis.

Como exemplo de algoritmos *first fit* e *best fit*, observe a Figura 3.6 novamente. Se um segmento de memória de tamanho 2 for necessário, o algoritmo *first fit* alocará o segmento de memória disponível que se inicia na unidade 5, mas o algoritmo *best fit* alocará aquele que se inicia em 18.

O algoritmo *best fit* é mais lento que o algoritmo *first fit*, pois precisa pesquisar a lista inteira cada vez que for chamado. O algoritmo *best fit*, surpreendentemente, também resulta em maior desperdício de memória do que os algoritmos *first fit* e *next fit*, pois tende a deixar disponíveis inúmeros segmentos minúsculos de memória e consequentemente inúteis. Em média, o algoritmo *first fit* gera segmentos de memória disponíveis maiores.

Para evitar o problema de alocar um segmento de memória disponível de tamanho quase exato ao requisitado pelo processo e, assim, gerar outro minúsculo segmento de memória disponível, seria possível pensar em um algoritmo **worst fit** (pior encaixe), isto é, em que sempre se escolhesse o maior segmento de memória disponível, de modo que, quando dividido, o segmento de memória disponível restante, após a alocação ao processo, fosse suficientemente grande para ser útil depois. Entretanto, simulações têm mostrado que o algoritmo *worst fit* não é uma ideia muito boa.

Todos os quatro algoritmos poderiam se tornar mais rápidos se segmentos de memória alocados a processos e segmentos de memória disponíveis fossem mantidos em listas separadas. Nesse caso, todos esses algoritmos se voltariam para inspecionar segmentos de memória ainda disponíveis, e não segmentos de memória já alocados a processos. O preço inevitavelmente pago por esse aumento de desempenho na alocação de memória é a complexidade adicional e a redução no desempenho da liberação de memória, visto que um segmento de memória liberado tem de ser removido da lista

de segmentos de memória alocados a processos e inserido na lista de segmentos de memória disponíveis.

Se listas distintas forem usadas para segmentos de memória alocados a processos e para segmentos de memória disponíveis, a lista de segmentos de memória disponíveis poderá ser mantida ordenada por tamanho, tornando o algoritmo *best fit* mais rápido. Quando o algoritmo *best fit* pesquisar a lista de segmentos de memória disponíveis do menor para o maior, ao encontrar um segmento de tamanho adequado, terá também a certeza de que se trata do menor segmento de memória disponível e, portanto, o *best fit*. Nenhuma procura adicional será necessária, como o seria no esquema de lista única. Com a lista de segmentos de memória disponíveis ordenada por tamanho, os algoritmos *first fit* e *best fit* são igualmente rápidos, e fica sem sentido utilizar o algoritmo *next fit*.

Quando se mantém a lista de segmentos de memória ainda disponíveis separada da lista de segmentos de memória já alocados a processos, uma pequena otimização é possível. Em vez de se ter um conjunto separado de estruturas de dados para a manutenção da lista de segmentos de memória disponíveis, como é feito na Figura 3.6(c), esses próprios segmentos podem ser usados para essa finalidade. A primeira palavra de cada segmento de memória disponível poderia conter o tamanho desse segmento, e a segunda palavra, um ponteiro para o segmento disponível seguinte. Os nós da lista da Figura 3.6(c) — os quais necessitam de três palavras e um bit (P/L) — não seriam mais necessários.

Ainda existe outro algoritmo de alocação, denominado **quick fit** (encaixe mais rápido), que mantém listas separadas para alguns dos tamanhos de segmentos de memória disponíveis em geral mais solicitados. Por exemplo, pense em uma tabela com *n* entradas, na qual a primeira entrada seria um ponteiro para o início de uma lista de segmentos de memória disponíveis de 4 KB; a segunda, um ponteiro para uma lista de segmentos de 8 KB; a terceira, um ponteiro para uma lista de segmentos de 12 KB, e assim por diante. Os segmentos de memória disponíveis de 21 KB poderiam ser colocados na lista de segmentos de memória disponíveis de 20 KB ou em uma lista de segmentos de tamanhos especiais.

Com o algoritmo *quick fit*, buscar um segmento de memória disponível de um determinado tamanho é extremamente rápido, mas apresenta a mesma desvantagem de todos os esquemas que ordenam por tamanho de segmento, ou seja, quando um processo termina sua execução ou é removido da memória (devolvido ao disco ou deletado), é dispendioso descobrir quais são os segmentos de memória vizinhos aos segmentos desse processo, a fim de verificar a possibilidade de uma mistura de segmentos de memória disponíveis. Se a mistura não for feita, a memória rapidamente será fragmentada em uma grande quantidade de minúsculos segmentos de memória disponíveis, inúteis a qualquer processo.

3.3 Memória virtual

Embora registradores-base e registradores-limite possam ser usados para criar a abstração de espaços de endereçamento, há outro problema a ser resolvido: gerenciar o bloatware.[1] O tamanho das memórias está aumentando rapidamente, mas o tamanho dos softwares está aumentando muito mais rápido. Na década de 1980, muitas universidades executavam um sistema de tempo compartilhado com dezenas de usuários (mais ou menos satisfeitos) simultâneos em um VAX 4 MB. Agora a Microsoft recomenda pelo menos 512 MB para que um único usuário do sistema Vista execute aplicações simples e 1 GB se estiver fazendo algo sério. A tendência à multimídia gera ainda mais demandas sobre a memória.

Como consequência desses desenvolvimentos, há a necessidade de executar programas grandes demais para se encaixarem na memória, e certamente há a necessidade de haver sistemas que possam dar suporte a múltiplos programas em execução simultânea, cada um dos quais é comportado pela memória individualmente, mas que, coletivamente, excedam a memória. A troca de processos não é uma opção atrativa, visto que um disco SATA típico tem uma taxa de transferência de pico de, no máximo, 100 MB/s, o que significa que leva pelo menos 10 segundos para sair de um programa de 1 GB e outros 10 segundos para inicializar um programa de 1 GB.

O problema de programas maiores que a memória está presente desde o início da computação, ainda que em áreas limitadas, como ciência e engenharia (a simulação da criação do universo ou mesmo a simulação de uma nova aeronave ocupam muita memória). Uma solução adotada na década de 1960 foi a divisão do programa em módulos, denominados **sobreposições** (*overlays*) (módulos de sobreposição). Quando um programa era inicializado, tudo o que era carregado na memória era o gerenciador de sobreposições, que imediatamente carregava e executava a sobreposição 0. Feito isso, ele diria ao gerenciador de sobreposições para carregar a sobreposição 1, seja acima da sobreposição 0 na memória (se houvesse espaço para ele), seja na parte superior da sobreposição 0 (se não houvesse espaço). Alguns sistemas de sobreposições eram altamente complexos, permitindo muitos sobreposições na memória ao mesmo tempo. As sobreposições eram mantidos em disco e carregadas (ou removidas) dinamicamente na memória pelo gerenciador de sobreposições.

Embora o trabalho real de troca de sobreposição do disco para a memória e vice-versa fosse feito pelo sistema operacional, a divisão do programa em módulos tinha de ser feita manualmente pelo programador. A divisão de programas grandes em módulos menores era um trabalho lento, enfadonho e propenso a erros. Poucos programadores

1 Bloatware é o termo utilizado para definir softwares que usam quantidades excessivas de memória (N.R.T.).

eram bons nisso. Não demorou muito para se pensar em também atribuir essa tarefa ao computador.

Para isso, foi concebido um método (Fotheringham, 1961), que ficou conhecido como **memória virtual**. A ideia básica por trás da memória virtual é que cada programa tem seu próprio espaço de endereçamento, que é dividido em blocos chamados **páginas**. Cada página é uma série contígua de endereços. Essas páginas são mapeadas na memória física, mas nem todas precisam estar na memória física para executar o programa. Quando o programa referencia uma parte de seu espaço de endereçamento que está na memória física, o hardware executa o mapeamento necessário dinamicamente. Quando o programa referencia uma parte de seu espaço de endereçamento que *não* está em sua memória física, o sistema operacional é alertado para obter a parte que falta e reexecutar a instrução que falhou.

De certo modo, a memória virtual é uma generalização da ideia do registrador-limite e do registrador-base. O 8088 tinha registradores-base separados (mas nenhum registrador-limite) para texto e dados. Com a memória virtual, em vez de realocação separada apenas para os segmentos de texto e dados, o espaço de endereçamento completo pode ser mapeado na memória física em unidades razoavelmente pequenas. Explicaremos como a memória virtual é implementada a seguir.

A memória virtual também funciona bem em um sistema com multiprogramação, com pedaços e partes de diferentes programas simultaneamente na memória. Se um programa estiver esperando por outra parte de si próprio ser carregada na memória, a CPU poderá ser dada a outro processo.

3.3.1 | Paginação

A maioria dos sistemas com memória virtual utiliza uma técnica denominada **paginação**. Em qualquer computador existe um conjunto de endereços de memória que os programas podem gerar ao serem executados. Quando um programa usa uma instrução do tipo

```
MOV REG,1000
```

ele deseja copiar o conteúdo do endereço de memória 1000 para o registrador REG (ou o contrário, dependendo do computador). Endereços podem ser gerados com o uso da indexação, de registradores-base, registradores de segmento ou outras técnicas.

Esses endereços gerados pelo programa são denominados **endereços virtuais** e constituem o **espaço de endereçamento virtual**. Em computadores sem memória virtual, o endereço virtual é idêntico ao endereço físico e, assim, para ler ou escrever uma posição de memória, ele é colocado diretamente no barramento da memória. Quando a memória virtual é usada, o endereço virtual não é colocado diretamente no barramento da memória. Em vez disso, ele vai a uma **MMU** (*memory management unit* — unidade de gerenciamento de memória), que mapeia endereços virtuais em endereços físicos, como ilustrado na Figura 3.8.

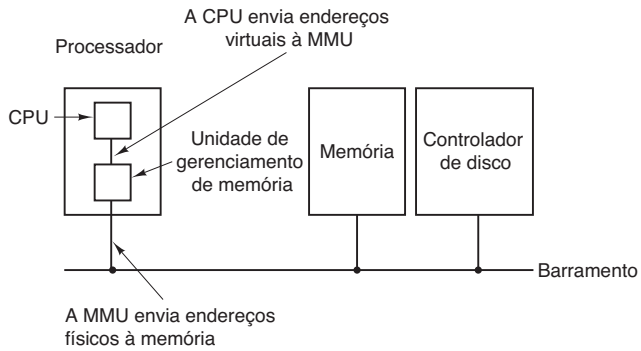

Figura 3.8 A posição e a função da MMU. Aqui a MMU é mostrada como parte do chip da CPU (processador) porque isso é comum atualmente. Contudo, em termos lógicos, poderia ser um chip separado, como ocorria no passado.

Um exemplo muito simples de como esse mapeamento funciona é mostrado na Figura 3.9. Nela, temos um computador que pode gerar endereços virtuais de 16 bits, de 0 a 64 K – 1. Contudo, esse computador tem somente 32 KB de memória física. Assim, embora seja possível escrever programas de 64 KB, eles não podem ser totalmente carregados na memória para serem executados. Uma cópia completa do código do programa, de até 64 KB, deve, entretanto, estar presente em disco, de modo que partes possam ser carregadas dinamicamente na memória quando necessário.

O espaço de endereçamento virtual é dividido em unidades denominadas **páginas** (*pages*). As unidades correspondentes na memória física são denominadas **molduras de página** (*page frames*). As páginas e as molduras de página são sempre do mesmo tamanho. No exemplo dado, as páginas têm 4 KB, mas páginas de 512 bytes a 64 KB têm sido utilizadas em sistemas reais. Com 64 KB de espaço de endereçamento virtual e 32 KB de memória física, podemos ter 16 páginas virtuais e oito molduras de página. As transferências entre memória e disco são sempre em páginas completas.

A notação na Figura 3.9 é a seguinte: a série 0K–4K significa que os endereços físicos ou virtuais nessa página são 0 a 4095. A série 4K–8K refere-se aos endereços 4096 a 8191 e assim por diante. Cada página contém exatamente 4096 endereços começando com um múltiplo de 4096 e terminando antes de um múltiplo de 4096.

Quando um programa tenta acessar o endereço 0, por exemplo, usando a instrução

```
MOV REG,0
```

o endereço virtual 0 é enviado à MMU, que detecta que esse endereço virtual situa-se na página virtual 0 (de 0 a 4095), que, de acordo com seu mapeamento, corresponde à moldura de página 2 (de 8192 a 12287). A MMU transforma, assim, o endereço virtual 0, que lhe foi entregue pela CPU, no endereço físico 8192 e o envia à memória por

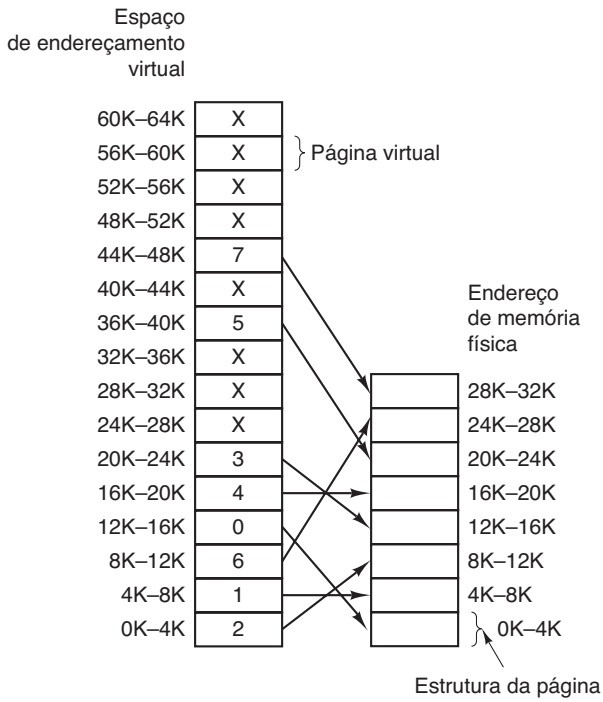

Figura 3.9 A relação entre endereços virtuais e endereços de memória física é dada pela tabela de páginas. Cada página começa com um múltiplo de 4096 e termina 4095 endereços acima; assim, 4K–8K na verdade significa 4096–8191 e 8K–12K significa 8192–12287.

meio do barramento. A memória desconhece a existência da MMU e somente enxerga uma solicitação de leitura ou escrita no endereço 8192, a qual ela executa. Desse modo, a MMU mapeia todos os endereços virtuais de 0 a 4095 em endereços físicos localizados de 8192 a 12287.

De maneira similar, a instrução

```
MOV REG,8192
```

é efetivamente transformada em

```
MOV REG,24576
```

pois o endereço virtual 8192 (na página virtual 2) está mapeado em 24576 (na moldura de página física 6). Como outro exemplo, o endereço virtual 20500 está localizado 20 bytes depois do início da página virtual 5 (endereços virtuais de 20480 a 24575) e é mapeado no endereço físico 12288 + 20 = 12308.

Por si só, essa habilidade de mapear as 16 páginas virtuais em qualquer uma das oito molduras de página, por meio da configuração apropriada do mapa da MMU, não resolve o problema de o espaço de endereçamento virtual ser maior do que a memória física disponível. Como temos apenas oito molduras de página física, somente oito páginas virtuais da Figura 3.9 podem ser simultaneamente mapeadas na memória física. As outras páginas, rotuladas com um X na figura, não são mapeadas. No hardware real, um **bit Presente/ausente** em cada entrada da tabela de páginas informa se a página está fisicamente presente ou não na memória.

O que acontece se um programa tenta usar uma página virtual não mapeada, por exemplo, empregando a instrução

```
MOV REG,32780
```

na qual 32780 corresponde ao décimo segundo byte dentro da página virtual 8 (que se inicia em 32768)? A MMU constata que essa página virtual não está mapeada (o que é indicado por um X na figura) e força o desvio da CPU para o sistema operacional. Essa interrupção é denominada **falta de página** (*page fault*). O sistema operacional, então, escolhe uma moldura de página (*page frame*) pouco usada e a salva em disco, ou seja, escreve seu conteúdo de volta no disco (se já não estiver lá). Em seguida, ele carrega a página virtual referenciada pela instrução na moldura de página recém-liberada, atualiza o mapeamento da tabela de páginas e reinicializa a instrução causadora da interrupção.

Por exemplo, se o sistema operacional decidir escolher a moldura de página 1 para ser substituída, deverá então carregar a página virtual 8 a partir do endereço físico 8192 e fazer duas alterações no mapeamento da MMU. Primeiro, o sistema operacional marcará, na tabela de páginas virtuais, a entrada da página virtual 1 como 'não mapeada' e, consequentemente, qualquer acesso futuro a endereços virtuais entre 4096 e 8191 provocará uma interrupção do tipo falta de página. Em seguida, ele marcará, na tabela de páginas virtuais, a entrada da página virtual 8 como 'mapeada' (substitui o X por 1), de modo que, quando a instrução causadora da interrupção for reexecutada, a MMU transformará o endereço virtual 32780 no endereço físico 4108 (4096 + 12).

Vamos dar uma olhada no funcionamento da MMU para também entender por que escolhemos um tamanho de página que é uma potência de 2. Na Figura 3.10 vemos um exemplo de endereço virtual, 8196 (0010000000000100 em binário), sendo mapeado por meio do emprego do mapeamento da MMU da Figura 3.9. O endereço virtual de 16 bits que chega à MMU é nela dividido em um número de página de 4 bits e um deslocamento de 12 bits. Com 4 bits para número de página, podemos ter 16 páginas virtuais e, com 12 bits para o deslocamento, é possível endereçar todos os 4096 bytes dessa página.

O número da página é usado como um índice para a **tabela de páginas**, a fim de obter a moldura de página física correspondente àquela página virtual. Se o bit *Presente/ausente* da página virtual estiver em 0, ocorrerá uma interrupção do tipo falta de página, desviando-se, assim, para o sistema operacional. Se o bit *Presente/ausente* estiver em 1, o número da moldura de página encontrado na tabela de páginas será copiado para os três bits mais significativos do registrador de saída, concatenando-os ao deslocamento de 12 bits, que é copiado sem alteração do endereço virtual de entrada. Juntos constituem o endereço físico de 15 bits da

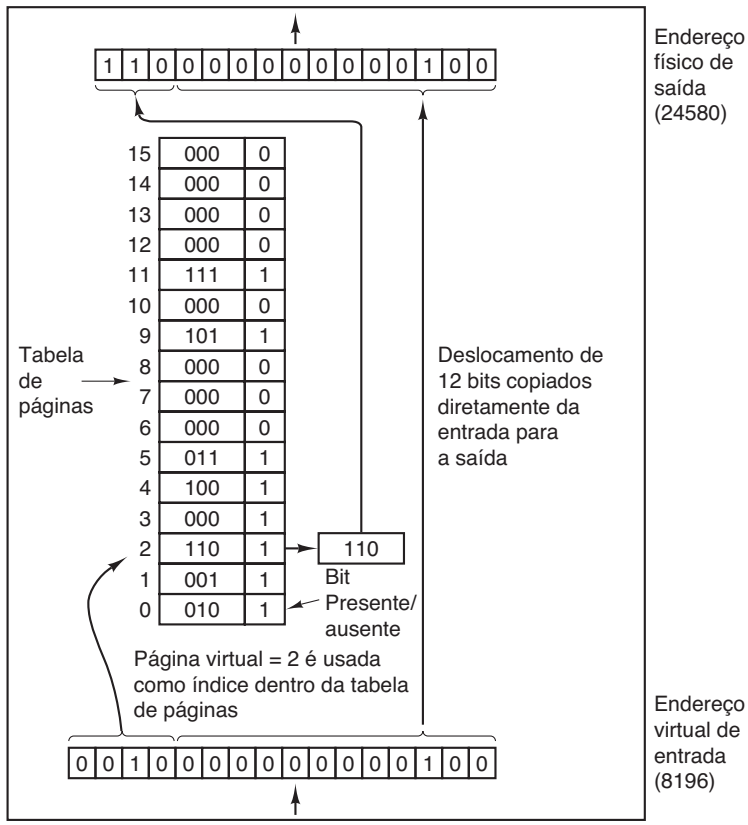

Figura 3.10 Operação interna da MMU com 16 páginas de 4 KB.

memória. O registrador de saída envia esse endereço físico de 15 bits à memória por meio de um barramento.

3.3.2 | Tabelas de páginas

No caso mais simples, o mapeamento dos endereços virtuais em endereços físicos pode ser resumido da seguinte forma: o endereço virtual é dividido em um número de página virtual (bits mais significativos) e um deslocamento (bits menos significativos). Por exemplo, com um endereço de 16 bits e um tamanho de página de 4 KB, os 4 bits superiores poderiam especificar uma dentre as 16 páginas virtuais e os 12 bits inferiores especificariam, então, o deslocamento do byte (de 0 a 4095) dentro da página selecionada. Contudo, também é possível uma divisão com 3 ou 5 bits ou algum outro número de bits para endereçamento de página. Diferentes divisões implicam diferentes tamanhos de páginas.

O número da página virtual é usado como índice dentro da tabela de páginas para encontrar a entrada dessa tabela associada à página virtual em questão. A partir dessa entrada, chega-se ao número da moldura de página física correspondente (caso ela já exista). O número da moldura de página física é concatenado aos bits do deslocamento, substituindo, assim, o número da página virtual pelo da moldura de página física, para formar o endereço físico que será enviado à memória.

Desse modo, o objetivo da tabela de páginas é mapear páginas virtuais em molduras de página física. Matematicamente falando, a tabela de páginas é uma função que usa o número da página virtual como argumento e tem o número da moldura de página física correspondente como resultado. Usando o resultado dessa função, o campo que endereça a página virtual em um endereço virtual pode ser substituído pelo campo que endereça a moldura de uma página física, formando, assim, um endereço da memória física.

Estrutura de uma entrada da tabela de páginas

Passemos, então, da análise da estrutura das tabelas de páginas como um todo para o detalhamento de uma única entrada da tabela de páginas. O esquema exato de uma entrada é altamente dependente da máquina, mas o tipo de informação presente é aproximadamente o mesmo de máquina para máquina. Na Figura 3.11 mostramos um exemplo de uma entrada da tabela de páginas. O tamanho varia de um computador para o outro, mas 32 bits são um tamanho comum. O campo mais importante é o *Número da moldura de página*. Afinal de contas, o objetivo do mapeamento de páginas é localizar esse valor. Próximo a ele, temos o bit *Presente/ausente*. Se esse bit for 1, a entrada será

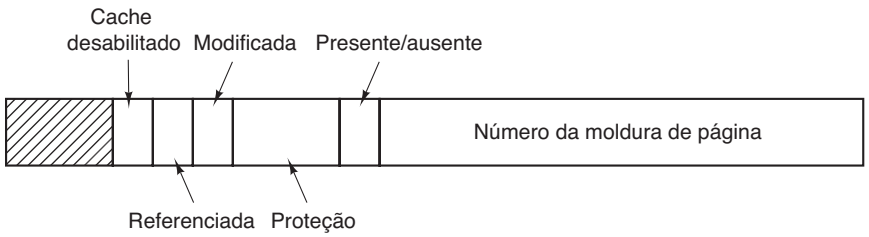

Figura 3.11 Entrada típica de uma tabela de páginas.

válida e poderá ser usada. Se ele for 0, a página virtual à qual a entrada pertence não estará presente na memória no referido instante. O acesso à entrada da tabela de páginas com esse bit em 0 causa uma falta de página.

Os bits *de proteção* dizem quais tipos de acesso são permitidos à página. Em sua configuração mais simples, esse campo contém 1 bit, com 0 para leitura/escrita e 1 somente para leitura. Uma forma mais sofisticada tem 3 bits, 1 bit para habilitar/desabilitar cada uma das operações básicas na página: leitura, escrita e execução.

Os bits *Modificada* e *Referenciada* controlam o uso da página. Ao escrever na página, o hardware automaticamente faz o bit *Modificada* igual a 1 na entrada correspondente da tabela de páginas. Esse bit é importante quando o sistema operacional decide reivindicar uma moldura de página. Se a página física dentro dessa moldura foi modificada (isto é, ficou 'suja'), ela também deve ser atualizada no disco. Do contrário (isto é, se continua 'limpa'), ela pode ser simplesmente abandonada, visto que a imagem em disco continua válida. Esse bit é chamado algumas vezes de **bit sujo**, já que reflete o estado da página.

Ao bit *Referenciada* é atribuído o valor 1 quando a página física é referenciada, para leitura ou escrita. Esse bit ajuda o sistema operacional a escolher uma página a ser substituída quando ocorre uma falta de página. As páginas físicas que não estão sendo utilizadas são melhores candidatas do que as que estão, e esse bit desempenha um papel importante em vários dos algoritmos de substituição de páginas que estudaremos posteriormente, ainda neste capítulo.

Por fim, o último bit permite que o mecanismo de cache seja desabilitado para a página em questão. Essa propriedade é mais importante para as páginas que mapeiam em registradores de dispositivos do que na memória. Se o sistema operacional estiver preso em um laço (*loop*) à espera da resposta de algum dispositivo de E/S, é essencial que o hardware mantenha a busca da palavra a partir do dispositivo, e não a partir de uma cópia antiga da cache. Com esse bit, o uso da cache pode ser desabilitado. Máquinas com espaços de endereçamento separados para E/S e que não usem E/S mapeada em memória não precisam desse bit.

Note que o endereço em disco empregado para armazenar a página quando esta não está presente na memória não faz parte da tabela de páginas. A justificativa para isso é simples: a tabela de páginas contém somente as informações necessárias ao hardware para traduzir um endereço virtual em um endereço físico. As informações de que o sistema operacional precisa para tratar as faltas de página são mantidas em tabelas em software dentro do sistema operacional. O hardware não necessita dessas informações.

Antes de passar a mais problemas de implementação, convém enfatizar novamente que o que a memória virtual faz, fundamentalmente, é criar uma nova abstração — o espaço de endereçamento — o qual é uma abstração da memória física, assim como um processo é uma abstração do processador físico (CPU). A memória virtual pode ser implementada dividindo o espaço de endereçamento virtual em páginas e mapeando cada uma delas em alguma moldura de página da memória física ou não mapeando-as (temporariamente). Dessa forma, este capítulo é basicamente sobre uma abstração criada pelo sistema operacional e sobre como essa abstração é gerenciada.

3.3.3 Acelerando a paginação

Acabamos de ver os princípios básicos da memória virtual e da paginação. Agora entraremos em detalhes sobre implementações possíveis. Em qualquer sistema de paginação, dois problemas importantes devem ser enfrentados:

1. O mapeamento do endereço virtual para endereço físico deve ser rápido.
2. Se o espaço de endereço virtual for grande, a tabela de páginas será grande.

O primeiro ponto é consequência do fato de que o mapeamento virtual-físico deve ser feito em cada referência à memória. Em última instância, todas as instruções devem vir da memória e muitas delas referenciam operandos na memória também. Consequentemente, é necessário fazer uma, duas ou algumas vezes mais referências à tabela de páginas por instrução. Se a execução de uma instrução leva 1 ns, por exemplo, a busca na tabela de páginas deve ser feita em menos de 0,2 ns para evitar que o mapeamento se torne um gargalo significativo.

O segundo ponto decorre do fato de que todos os computadores modernos usam endereços virtuais de pelo menos 32 bits, com 64 bits se tornando cada vez mais comuns. Com um tamanho de página de 4 KB, por exemplo, um

espaço de endereços de 32 bits tem 1 milhão de páginas, e um espaço de endereços de 64 bits tem mais do que você gostaria de considerar. Com 1 milhão de páginas no espaço de endereçamento virtual, a tabela de páginas deve ter 1 milhão de entradas. E lembre-se de que cada processo precisa de sua própria tabela de páginas (porque tem seu próprio espaço de endereço virtual).

A necessidade de extensos e rápidos mapeamentos de páginas é uma restrição significativa ao modo como os computadores são construídos. O projeto mais simples (pelo menos em termos conceituais) é ter uma tabela de páginas consistindo de um arranjo de registradores de hardware rápidos, com uma entrada para cada página virtual, indexada pelo número dessa página, como mostrado na Figura 3.10. Quando um processo é inicializado, o sistema operacional carrega os registradores com a tabela de páginas do processo, retirada de uma cópia mantida na memória principal. Durante a execução do processo, não são mais necessárias referências à memória para a tabela de páginas. As vantagens desse processo são que ele é direto e não requer referências à memória durante o mapeamento. Uma desvantagem é que é excessivamente caro se a tabela de páginas for grande. Outra desvantagem é que a necessidade de carregar a tabela de páginas completa a cada alternância de contexto prejudica o desempenho.

No outro extremo, a tabela de páginas pode estar inteiramente na memória principal. Tudo de que o hardware precisa, nesse caso, é de um registrador único que aponte para o início da tabela de páginas. O projeto permite que o mapa virtual-físico seja mudado em uma alternância de contexto por meio do carregamento de um registro. Naturalmente há a desvantagem de requerer uma ou mais referências à memória para ler as entradas na tabela de páginas durante a execução de cada instrução, tornando-a muito lenta.

TLB ou memória associativa

Examinemos agora esquemas amplamente implementados para acelerar a paginação e para lidar com grandes espaços de endereçamento virtual, começando com o primeiro tipo. O ponto de partida da maior parte das técnicas de otimização é o posicionamento da tabela de páginas na memória. Potencialmente, essa decisão tem um enorme impacto no desempenho. Considere, por exemplo, uma instrução de 1 byte que copie o conteúdo de um registrador para outro. Na ausência de paginação, essa instrução faz um único acesso à memória para buscar a própria instrução. Com a paginação, será necessária pelo menos uma referência adicional à memória para acessar a tabela de páginas. Como a velocidade de execução geralmente é limitada pela frequência com que a CPU pode acessar instruções e dados na memória, o fato de haver a necessidade de dois acessos à memória por referência à memória reduz o desempenho pela metade. Nessas condições, ninguém usaria paginação.

Os projetistas de computadores estudam esse problema há anos e encontraram uma solução, com base na observação de que a maioria dos programas tende a fazer um grande número de referências a um mesmo pequeno conjunto de páginas virtuais. Assim, somente uma reduzida parte das entradas da tabela de páginas é intensamente lida; as entradas restantes raramente são referenciadas.

A solução concebida foi equipar os computadores com um pequeno dispositivo em hardware para mapear os endereços virtuais para endereços físicos sem passar pela tabela de páginas. Esse dispositivo, denominado **TLB** (*translation lookaside buffer* — buffer para tradução de endereços) ou às vezes **memória associativa**, é ilustrado na Tabela 3.1. Esse dispositivo geralmente se localiza dentro da MMU e consiste em um pequeno número de entradas — oito no exemplo dado —, mas raramente mais do que 64. Cada entrada contém informações sobre uma página — incluindo o número da página virtual —, um bit que é colocado em 1 quando a página é modificada, o código de proteção (permissão de leitura/escrita/execução) e a moldura de página em que está localizada. Esses campos têm uma correspondência de um para um com os campos na tabela de páginas, exceto para o número de página virtual, que não é necessário na tabela de páginas. Outro bit indica se a entrada é válida (em uso) ou não.

Um exemplo que pode gerar a TLB da Tabela 3.1 é um processo em um laço (*loop*) que referencia constantemente as páginas virtuais 19, 20 e 21, de modo que essas entradas na TLB apresentam código de proteção para leitura e execução. Os dados principais referenciados em um dado instante (por exemplo, um arranjo) estão localizados nas páginas 129 e 130. A página 140 contém os índices usados no processamento desse arranjo. Por fim, a pilha localiza-se nas páginas 860 e 861.

Vejamos agora como a TLB funciona. Quando um endereço virtual é apresentado à MMU para tradução, o hardware primeiro verifica se o número de sua página virtual está presente na TLB comparando-o com todas as entradas da TLB simultaneamente (isto é, em paralelo). Se uma correspon-

Válida	Página virtual	Modificada	Proteção	Moldura da página
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Tabela 3.1 Uma TLB para acelerar a paginação.

dência válida é encontrada e o acesso não viola os bits de proteção, o número da moldura de página é, então, obtido diretamente da TLB, sem necessidade de buscá-lo na tabela de páginas. Se o número da página virtual estiver presente na TLB, mas a instrução estiver tentando escrever em uma página que permita somente leitura, uma falta por violação de proteção (*protection fault*) é gerada.

É interessante notar o que acontece quando o número da página virtual não está presente na TLB. A MMU detecta a ausência de página (*page miss*) e, então, faz uma busca comum na tabela de páginas. A MMU então destitui uma das entradas da TLB e a substitui por essa entrada da tabela de páginas que acabou de ser buscada. Assim, se a mesma página virtual for referenciada novamente, haverá uma presença de página (*page hit*) em vez de uma ausência de página. Quando uma entrada é retirada da TLB, apenas o bit modificado deve ser copiado de volta na entrada correspondente da tabela de páginas na memória. Os demais valores já estão lá, exceto o bit referenciado. Quando uma entrada da TLB é carregada a partir da tabela de páginas, todos os campos dessa entrada devem ser trazidos da memória.

Gerenciamento da TLB por software

Até agora temos partido do pressuposto de que toda máquina com memória virtual paginada tem reconhecimento por hardware das tabelas de páginas e também uma TLB. Nesse projeto, o gerenciamento e o tratamento das faltas na TLB são totalmente feitos pelo hardware da MMU. Interrupções para o sistema operacional só ocorrem quando uma página não está na memória.

Antigamente essa suposição era verdadeira. Hoje, porém, muitas das máquinas RISC, dentre elas SPARC, MIPS, Alpha e HP PA, fazem quase todo esse gerenciamento por software. Nessas máquinas, as entradas da TLB são explicitamente carregadas pelo sistema operacional. Quando ocorre uma ausência de página na TLB, em vez de a própria MMU buscar na tabela de páginas a página virtual requisitada, ela apenas gera uma interrupção e repassa o problema ao sistema operacional. Este deve, então, encontrar a página virtual na tabela de páginas, destituir uma das entradas da TLB, inserir aí a nova página virtual e reinicializar a instrução interrompida. Obviamente, tudo isso deve ser feito para muitas instruções, pois as ausências de página na TLB ocorrem muito mais frequentemente do que as faltas de páginas na tabela de páginas.

Mas, se a TLB for suficientemente grande (digamos, com 64 entradas) para que se reduza a taxa de ausências de página, o gerenciamento da TLB por software acaba tendo uma eficiência aceitável. O ganho principal aqui é ter uma MMU muito mais simples, o que libera bastante área no chip da CPU para caches e outros recursos que possam melhorar o desempenho. O gerenciamento da TLB por software é discutido por Uhlig et al. (1994).

Diversas estratégias têm sido desenvolvidas para melhorar o desempenho de máquinas que fazem o gerenciamento da TLB por software. Uma delas tenta tanto reduzir o número de ausências de página na TLB quanto o custo dessas ausências quando elas ocorrem (Bala et al., 1994). Para reduzir o número de ausências na TLB, muitas vezes o sistema operacional pode usar sua 'intuição' para descobrir quais páginas virtuais têm mais probabilidade de serem usadas e, então, antecipar seu carregamento na TLB. Por exemplo, quando um processo cliente envia uma mensagem a um processo de servidor na mesma máquina, é bem provável que o processo de servidor tenha de ser executado logo. Sabendo desse fato, enquanto o processamento é desviado para o send, o sistema operacional também pode verificar onde se encontram as páginas do código, os dados e a pilha do processo de servidor e, então, mapeá-los antes que eles possam provocar ausências de página na TLB.

O modo normal de tratar uma ausência de página na TLB, seja por hardware, seja por software, é acessar a tabela de páginas e executar as operações de indexação para localizar a página referenciada não encontrada na TLB. O problema em se fazer essa pesquisa por software é que as páginas que contêm a tabela de páginas podem não estar mapeadas na TLB, ocasionando ausências adicionais na TLB durante o processamento. Essas ausências podem ser reduzidas se uma grande cache (de 4 KB, por exemplo), gerenciada por software e contendo entradas do tipo TLB, for mantida em uma localização fixa com sua página sempre mapeada na TLB. Verificando primeiro essa cache, o sistema operacional pode reduzir substancialmente as ausências de página na TLB.

Quando o gerenciamento da TLB por software é usado, é essencial compreender a diferença entre dois tipos de ausência. Uma **ausência leve** (*soft miss*) ocorre quando a página referenciada não está na TLB, mas na memória. Aqui é necessário apenas atualizar a TLB, e os discos de E/S não são utilizados. Normalmente, uma ausência temporária leva de 10 a 20 instruções da máquina para ser concluída em alguns nanossegundos. Por outro lado, uma **ausência completa** (*hard miss*) ocorre quando a própria página não está na memória (e, é claro, também não está na TLB). Requer-se acesso ao disco para trazer a página, o que leva vários milissegundos. Uma ausência definitiva é milhões de vezes mais lenta que uma ausência temporária.

3.3.4 Tabelas de páginas para memórias grandes

As TLBs podem ser usadas para acelerar a tradução de endereços virtuais para endereços físicos em relação ao esquema original de tabela de páginas na memória. Mas esse não é o único problema que temos de atacar. Há ainda o problema de como lidar com espaços de endereço virtual muito grandes. Discutiremos adiante dois modos de lidar com tais espaços.

Tabelas de páginas multinível

Como método inicial, considere o uso de uma tabela de páginas multinível. Um exemplo simples desse método é mostrado na Figura 3.12. Na Figura 3.12(a), vê-se um endereço virtual de 32 bits dividido em um campo *PT1* de 10 bits, um campo *PT2* de 10 bits e um campo *Deslocamento* de 12 bits. Como o campo *Deslocamento* tem 12 bits, as páginas são de tamanho 4 KB. Os outros dois campos têm conjuntamente 20 bits, o que possibilita um total de 2^{20} páginas virtuais.

O segredo para o método de tabela de páginas multinível é evitar que todas elas sejam mantidas na memória o tempo todo, especialmente as que não são necessárias. Suponha, por exemplo, que um processo necessite de 12 megabytes: 4 megabytes da base da memória para o código do programa, outros 4 megabytes para os dados do programa e 4 megabytes do topo da memória para a pilha. Sobra, entre o topo dos dados e a base da pilha, um gigantesco espaço não usado.

A Figura 3.12(b) mostra como funciona a tabela de páginas com dois níveis nesse exemplo. No lado esquerdo, vemos a tabela de páginas de nível 1, com 1024 entradas, correspondente ao campo *PT1* de 10 bits. Quando um endereço virtual chega à MMU, ela primeiro extrai o campo *PT1* e o utiliza como índice da tabela de páginas de nível 1. Cada uma dessas 1024 entradas representa 4 M, pois o espaço total de endereçamento virtual de 4 gigabytes (isto é, 32 bits) foi dividido em segmentos de 4096 bytes.

A entrada da tabela de páginas de nível 1, que é localizada por meio do campo *PT1* do endereço virtual, aponta para o endereço ou a moldura de página de uma tabela de páginas de nível 2. A entrada 0 da tabela de páginas de nível 1 aponta para a tabela de páginas de nível 2 relativa ao código do programa; a entrada 1 aponta para a tabela de páginas de nível 2 relativa aos dados e a entrada 1023 aponta para a tabela de páginas de nível 2 relativa à pilha. As outras entradas (sombreadas) da tabela de páginas virtuais de nível 1 não são usadas. O campo *PT2* é então empregado como índice da tabela de páginas de nível 2 selecionada para localizar o número da moldura de página física correspondente.

Por exemplo, considere o endereço virtual de 32 bits 0x00403004 (4.206.596 em decimal), o qual corresponde à localização 12.292 contando-se a partir do início dos dados, ou seja, a partir de 4 M. Esse endereço virtual corresponde a

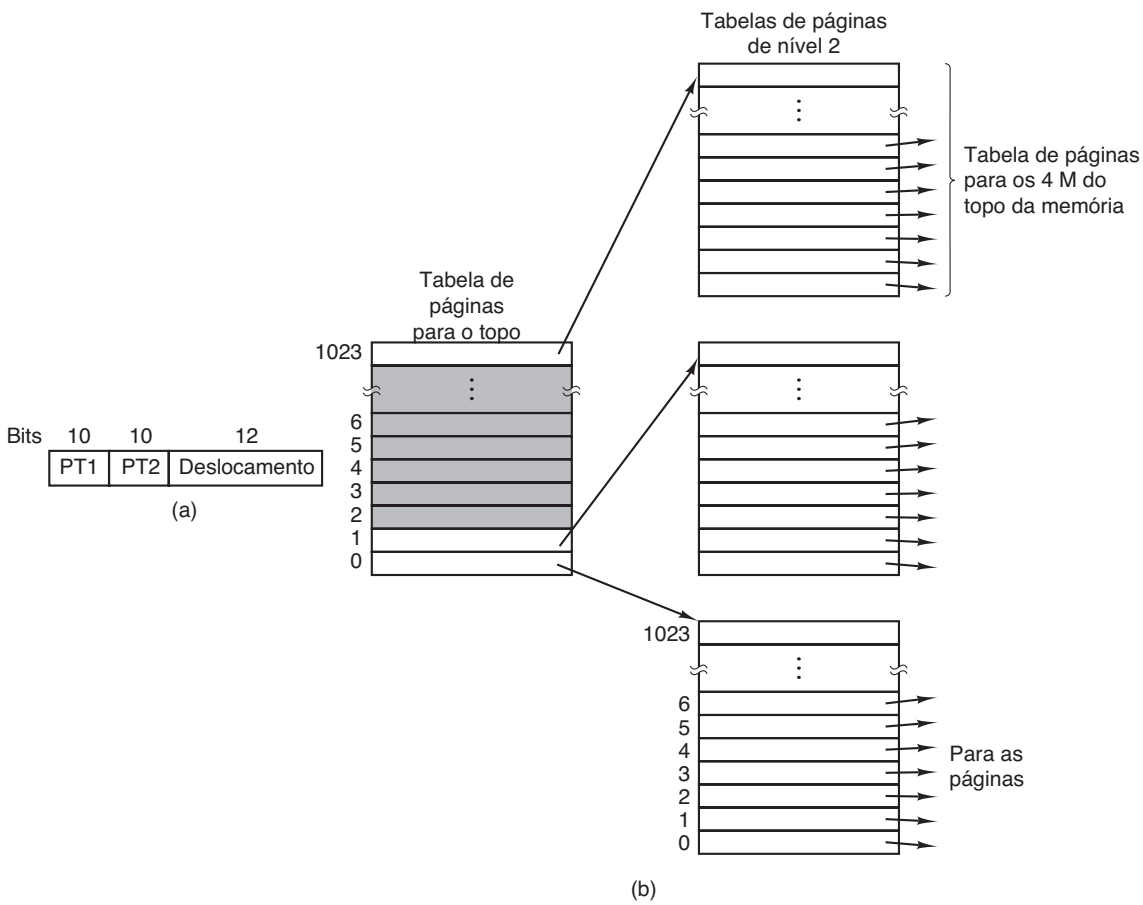

Figura 3.12 (a) Um endereço de 32 bits com dois campos de tabela de páginas. (b) Tabelas de páginas de dois níveis.

PT1 = 1, *PT2* = 2 e *Deslocamento* = 4. A MMU primeiro utiliza *PT1* como índice da tabela de páginas de nível 1 e obtém a entrada 1, que corresponde ao endereço de uma tabela de nível 2 que contém os endereços de 4 M – 1 a 8 M – 1. A MMU então utiliza *PT2* como índice dessa tabela de nível 2 recém-localizada e obtém a entrada 3, que corresponde aos endereços de 12288 a 16383 dentro de seu pedaço de 4 M (isto é, endereços absolutos de 4.206.592 a 4.210.687). Essa entrada contém o número de moldura de página com a página que contém o endereço virtual 0x00403004. Se essa página não estiver presente na memória, então o bit *Presente/ausente* na entrada dessa tabela de páginas será zero, o que causará uma falta de página. Se a página estiver na memória, o número da moldura de página tirado da tabela de páginas de nível 2 será combinado com o deslocamento (4) para construir o endereço físico. Esse endereço é colocado no barramento e enviado à memória.

É interessante notar na Figura 3.12 que, embora o espaço de endereçamento contenha mais de um milhão de páginas virtuais, somente quatro tabelas de páginas são realmente necessárias: a tabela de nível 1 e as três tabelas de nível 2 referentes aos endereços de 0 a 4 M (para o código do programa), de 4 M a 8 M (para os dados) e aos 4 M do topo (para a pilha). Os bits *Presente/ausente* nas 1021 entradas da tabela de páginas de nível 1 são marcados com 0, forçando uma falta de página se forem acessados. Se isso ocorrer, o sistema operacional saberá que o processo está tentando referenciar uma parte não permitida da memória e tomará uma decisão apropriada — como enviar um sinal ao processo ou eliminá-lo. Nesse exemplo, escolhemos números redondos para os diversos tamanhos e *PT1* do mesmo tamanho que *PT2*, mas, na prática, outros valores são obviamente possíveis.

Esse sistema da tabela de páginas em dois níveis da Figura 3.12 pode ser expandido para três, quatro ou mais níveis. Níveis adicionais permitem maior flexibilidade, embora seja duvidoso que essa complexidade adicional do hardware continue vantajosa além dos três níveis.

Tabelas de páginas invertidas

Para espaços de endereçamento virtuais de 32 bits, a tabela de páginas multinível funciona razoavelmente bem. Contudo, à medida que aparecem computadores de 64 bits, a situação muda drasticamente. Como o espaço de endereçamento virtual agora é de 2^{64} bytes, se adotarmos páginas de 4 KB, precisaremos de uma tabela de páginas com 2^{52} entradas. Se cada entrada contiver 8 bytes, essa tabela de páginas terá mais de 30 milhões de gigabytes (30 PB). Reservar 30 milhões de gigabytes somente para a tabela de páginas não é factível, nem agora nem nos próximos anos, e talvez nunca. Consequentemente, é necessária uma solução diferente para tratar espaços de endereçamento virtuais paginados de 64 bits.

Uma possível solução é a **tabela de páginas invertidas**: nela existe apenas uma entrada por moldura de página na memória real, em vez de uma entrada por página do espaço de endereçamento virtual. Por exemplo, com endereços virtuais de 64 bits, uma página de 4 KB e 1 GB de RAM, uma tabela de páginas invertidas requer apenas 262.144 entradas. Cada entrada informa que o par (processo, página virtual) está localizado na moldura de página.

Embora as tabelas de páginas invertidas possam economizar muito espaço — principalmente quando o espaço de endereçamento virtual é muito maior que a memória física —, elas apresentam um problema sério: a tradução de virtual para físico torna-se muito mais difícil. Quando o processo *n* referencia a página virtual *p*, o hardware não pode mais encontrar a página física usando *p* como índice da tabela de páginas. Em vez disso, ele deve pesquisar toda a tabela de páginas invertidas em busca de uma entrada (*n*, *p*). Além desse procedimento, essa pesquisa deve ser feita a cada referência à memória, e não somente nas faltas de página. Pesquisar uma tabela de 256 K a cada referência à memória não é a melhor maneira de tornar sua máquina rápida.

Uma solução possível para esse dilema é a utilização da TLB. Se esta puder conter todas as páginas mais intensamente usadas, a tradução pode ocorrer tão rapidamente quanto nas tabelas de páginas convencionais. Ocorrendo uma ausência na TLB, contudo, a tabela de páginas invertidas deve ser pesquisada no software. Um modo de realizar essa pesquisa é ter uma tabela de espalhamento (*hash*) nos endereços virtuais. Todas as páginas virtuais atualmente presentes na memória e que tiverem o mesmo valor de espalhamento serão encadeadas juntas, como mostra a Figura 3.13. Se a tabela de espalhamento apresentar tantas entradas quantas páginas físicas a máquina tiver, o comprimento médio de encadeamento será de somente uma entrada, agilizando muito o mapeamento. Ao encontrar o número da moldura de página, a nova dupla (virtual, física) é inserida na TLB.

Tabelas de páginas invertidas são comuns em máquinas de 64 bits porque, mesmo com um tamanho de página muito grande, o número de entradas de tabelas de páginas é enorme. Por exemplo, com páginas de 4 MB e endereços virtuais de 64 bits, são necessárias 2^{42} entradas de tabelas de páginas. Outros métodos para lidar com grandes memórias virtuais podem ser encontrados em Talluri et al. (1995).

3.4 Algoritmos de substituição de páginas

Quando ocorre uma falta de página, o sistema operacional precisa escolher uma página a ser removida da memória a fim de liberar espaço para uma nova página a ser trazida para a memória. Se a página a ser removida tiver sido modificada enquanto estava na memória, ela deverá ser reescrita no disco com o propósito de atualizar a cópia virtual lá existente. Se, contudo, a página não tiver sido

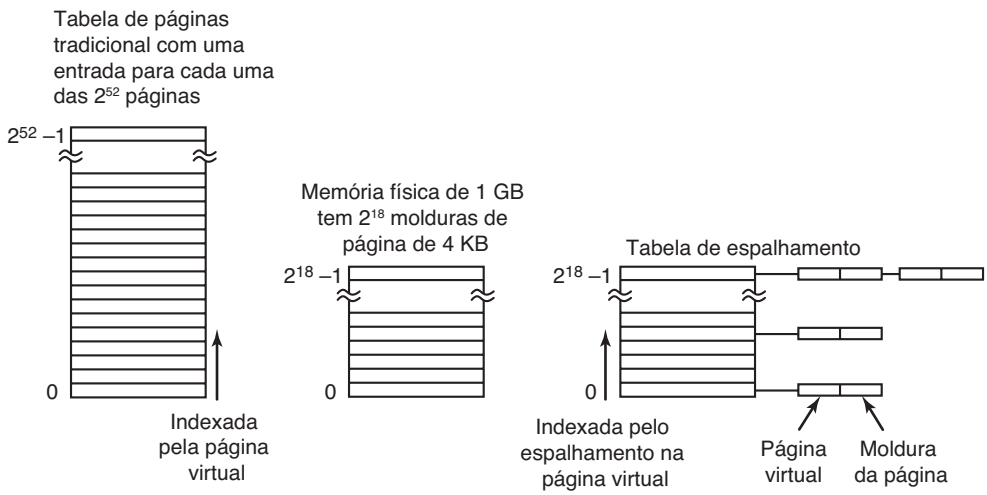

Figura 3.13 Comparação entre uma tabela de páginas tradicional e uma tabela de páginas invertidas.

modificada (por exemplo, uma página de código), a cópia em disco já estará atualizada e, assim, não será necessário reescrevê-la. A página a ser trazida para a memória simplesmente sobrescreve a página que está sendo destituída.

Embora seja possível escolher aleatoriamente uma página a ser descartada a cada falta de página, o desempenho do sistema será muito melhor se a página escolhida for uma que não estiver sendo muito usada. Se uma página intensamente usada for removida, é provável que logo ela precise ser trazida de volta, ocasionando custos extras. Muitos trabalhos, teóricos e experimentais, têm se voltado para os algoritmos de substituição de páginas. A seguir, descreveremos alguns dos algoritmos mais importantes.

É importante notar que o problema da substituição de páginas também ocorre em outras áreas da concepção de computadores. Por exemplo, a maioria dos computadores tem uma ou mais caches com os blocos de memória mais recentemente usados — blocos que contêm 32 ou 64 bytes cada um. Se a cache estiver cheia, um dos blocos será escolhido para ser removido. Esse problema é precisamente análogo ao da substituição de páginas, diferindo apenas na duração de tempo envolvida (na cache, a substituição do conteúdo de um bloco de memória é realizada em poucos nanossegundos, e não em milissegundos, como na substituição de página). A razão para essa redução de tempo é que uma falta de bloco (*block miss*) na cache é satisfeita a partir da memória principal, que não tem retardos resultantes do tempo de rotação e de latência rotacional do disco.

Um segundo exemplo é um servidor da Web. O servidor pode manter um certo número de páginas da Web intensamente usadas em sua cache na memória. Contudo, quando a cache estiver cheia e uma nova página for referenciada, será preciso decidir qual página da Web descartar. São considerações similares àquelas concernentes às páginas de memória virtual, exceto pelo fato de que páginas da Web nunca são modificadas na cache e, assim, suas cópias em disco estão sempre atualizadas, ao passo que, em um sistema com memória virtual, as páginas na memória podem estar limpas ou sujas.

Em todos os algoritmos de substituição de páginas estudados a seguir, surge a seguinte questão: quando uma página vai ser removida da memória, devemos remover uma das páginas do próprio processo que causou a falta ou podemos remover uma página pertencente a outro processo? No primeiro caso, estamos efetivamente limitando cada processo a um número fixo de páginas; no segundo, não o fazemos. Há as duas possibilidades. Retornaremos a esse assunto na Seção 3.5.1.

3.4.1 O algoritmo ótimo de substituição de página

O melhor dos algoritmos de substituição de página é fácil de descrever, mas impossível de implementar. Ele funciona da seguinte maneira: no momento em que ocorre uma falta de página, existe um determinado conjunto de páginas na memória. Uma delas será referenciada na próxima instrução, ou seja, trata-se da mesma página que contém a instrução que gerou a falta de página. Outras páginas podem ser referenciadas até dez, cem ou talvez mil instruções mais tarde. Cada página pode ser rotulada com o número de instruções que serão executadas antes de aquela página ser referenciada pela primeira vez.

O algoritmo ótimo diz apenas que se deve remover a página com o maior rótulo. Se determinada página só for usada após oito milhões de instruções e outra página só for usada após seis milhões de instruções, a primeira deverá ser removida antes da segunda. Dessa maneira, o algoritmo ótimo de substituição de página adia a ocorrência da próxima falta de página o máximo possível. Computadores,

assim como pessoas, tentam adiar o máximo possível a ocorrência de eventos desagradáveis.

O único problema com esse algoritmo é que ele é irrealizável. Na ocorrência de uma falta de página, o sistema operacional não tem como saber quando cada uma das páginas será referenciada novamente. (Já vimos situação similar no caso do algoritmo de escalonamento tabela curta primeiro — como o sistema operacional pode saber qual das tarefas é a mais curta?) Entretanto, executar primeiro o programa em um simulador e guardar todas as referências às páginas possibilita implementar o algoritmo ótimo na *segunda* execução desse programa, usando as informações coletadas durante a *primeira* execução.

Desse modo, torna-se possível comparar o desempenho dos algoritmos realizáveis com o do melhor possível. Se um sistema operacional tiver um algoritmo de substituição de página com desempenho de, digamos, somente 1 por cento pior que o do algoritmo ótimo, todos os esforços despendidos para melhorar esse algoritmo proporcionarão uma melhora de, no máximo, 1 por cento.

Para evitar qualquer possível confusão, é preciso deixar claro que esse registro (*log*) de referências às páginas é concernente apenas à execução de um programa específico com dados de entrada específicos. O algoritmo de substituição de página derivado daí é específico daquele programa e daqueles dados. Embora esse método auxilie na avaliação de algoritmos de substituição de página, ele é inútil para uso em sistemas práticos. A seguir, estudaremos algoritmos úteis em sistemas reais.

3.4.2 | O algoritmo de substituição de página não usada recentemente (NRU)

A maioria dos computadores com memória virtual tem 2 bits de status — o bit referenciado (*R*) e o bit modificado (*M*) —, associados a cada página virtual, que permitem que o sistema operacional saiba quais páginas físicas estão sendo usadas e quais não estão. O bit *R* é colocado em 1 sempre que a página é referenciada (lida ou escrita). O bit *M* é colocado em 1 sempre que se escreve na página (isto é, a página é modificada). Os bits estão contidos em cada entrada da tabela de páginas, como mostra a Figura 3.11. É importante perceber que esses bits devem ser atualizados em todas as referências à memória, de modo que é essencial que essa atualização se dê por hardware. Uma vez que um bit é colocado em 1 por hardware, ele permanece em 1 até o sistema operacional reinicializá-lo.

Se o hardware não possui esses bits, estes podem ser simulados da seguinte maneira: um processo, ao ser inicializado, tem todas as suas entradas da tabela de páginas marcadas como não presentes na memória. Tão logo uma de suas páginas virtuais seja referenciada, ocorre uma falta de página. Então, o sistema operacional coloca o bit *R* em 1 (em suas tabelas internas), altera a entrada da tabela de páginas a fim de apontar para a página física correta, com modo SOMENTE LEITURA, e reinicializa a instrução. Se a página for subsequentemente modificada, outra falta de página ocorrerá, permitindo que o sistema operacional coloque o bit *M* em 1 e altere o modo da página para LEITURA/ESCRITA.

Os bits *R* e *M* podem ser usados para construir um algoritmo de paginação simples, tal como segue. Quando um processo é inicializado, os dois bits citados, para todas as suas páginas, são colocados em 0 pelo sistema operacional. Periodicamente (por exemplo, a cada tique do relógio), o bit *R* é limpo, de modo que diferencie as páginas que não foram referenciadas recentemente daquelas que foram.

Quando acontece uma falta de página, o sistema operacional inspeciona todas as páginas e as separa em quatro categorias, com base nos valores atuais dos bits *R* e *M*:

Classe 0: não referenciada, não modificada.

Classe 1: não referenciada, modificada.

Classe 2: referenciada, não modificada.

Classe 3: referenciada, modificada.

Embora as páginas da classe 1 pareçam, à primeira vista, impossíveis de ocorrer, elas surgem quando uma página da classe 3 tem seu bit *R* limpo por uma interrupção do relógio. As interrupções do relógio não limpam o bit *M*, pois essa informação é necessária para saber se a página deve ou não ser reescrita em disco. A limpeza do bit *R*, mas não do bit *M*, conduz à classe 1.

O algoritmo **NRU** (*not recently used* — não usada recentemente) remove aleatoriamente uma página da classe de ordem mais baixa que não esteja vazia. Está implícito nesse algoritmo que é melhor remover uma página modificada, mas não referenciada, a pelo menos um tique do relógio (em geral, 20 ms) do que uma página não modificada que está sendo intensamente referenciada. A principal vantagem do algoritmo NRU é ser fácil de entender e de implementar e, além disso, fornece um desempenho que, apesar de não ser ótimo, pode ser adequado.

3.4.3 | O algoritmo de substituição de página primeiro a entrar, primeiro a sair

O algoritmo de substituição de página **primeiro a entrar, primeiro a sair** (*first-in, first-out* — FIFO) é um algoritmo de baixo custo. Para ilustrar seu funcionamento, imagine um supermercado que tenha diversas prateleiras para acomodar exatamente *k* produtos diferentes. Um dia, uma empresa lança um produto — iogurte orgânico, seco e congelado, com reconstituição instantânea no forno de micro-ondas. É um sucesso imediato, de modo que nosso supermercado, que tem espaço limitado, se vê obrigado a se livrar de um produto velho para conseguir espaço para o novo produto.

Uma solução seria descobrir qual produto o supermercado vem estocando há mais tempo (por exemplo, algo que ele começou a vender 120 anos atrás) e se livrar dele supondo que ninguém mais se interessa por ele. Por sorte, o supermercado mantém uma lista encadeada de todos os produtos que atualmente vende na ordem em que eles foram introdu-

zidos. O novo produto entrará no final dessa lista encadeada; o primeiro produto introduzido na lista será eliminado.

Pode-se aplicar a mesma ideia a um algoritmo de substituição de página. O sistema operacional mantém uma lista de todas as páginas atualmente na memória, com a página mais antiga na cabeça da lista e a página que chegou mais recentemente situada no final dessa lista. Na ocorrência de uma falta de página, a primeira página da lista é removida e a nova página é adicionada no final da lista. Quando se aplica o algoritmo FIFO a armazéns, pode-se tanto remover itens pouco vendidos, como cera para bigodes, quanto itens muito vendidos, como farinha, sal ou manteiga. Quando se aplica o algoritmo FIFO a computadores, surge o mesmo problema. Por essa razão, o algoritmo de substituição de página FIFO, em sua configuração pura, raramente é usado.

3.4.4 O algoritmo de substituição de página segunda chance

Uma modificação simples no algoritmo de substituição de página FIFO evita o problema de se jogar fora uma página intensamente usada, e isso é feito simplesmente inspecionando o bit *R* da página mais antiga, ou seja, a primeira página da fila. Se o bit *R* for 0, essa página, além de ser a mais antiga, não estará sendo usada, de modo que será substituída imediatamente. Se o bit *R* for 1, ele será colocado em 0, a página será posta no final da lista de páginas e seu tempo de carregamento (chegada) será atualizado como se ela tivesse acabado de ser carregada na memória. A pesquisa então continua.

O funcionamento desse algoritmo, chamado de **segunda chance**, é mostrado na Figura 3.14. Na Figura 3.14(a), vemos as páginas de *A* a *H* mantidas em uma lista encadeada e ordenada por tempo de chegada na memória.

Suponha que uma falta de página ocorra no instante 20. A página mais antiga é a página *A*, que chegou no instante 0 quando o processo foi inicializado. Se o bit *R* da página *A* for 0, ela será retirada da memória, tendo sua cópia em disco atualizada se houver sido modificada (suja), ou será simplesmente abandonada se não tiver sido modificada (se estiver limpa). Por outro lado, se o bit *R* for 1, a página *A* será colocada no final da lista e seu 'instante de carregamento' será atualizado com o valor atual (20). O bit *R* é colocado em 0, e a busca por uma página a ser substituída continua então a partir da página *B*.

O que o algoritmo segunda chance faz é procurar uma página antiga que não tenha sido referenciada no intervalo de relógio anterior. Se todas as páginas foram referenciadas, o segunda chance degenera-se para o FIFO puro. Especificamente, imagine que todas as páginas na Figura 3.14(a) tenham seus bits *R* em 1. Uma a uma, as páginas são reinseridas no final da lista pelo sistema operacional, e o bit *R* de cada página é zerado. Quando a página *A* for novamente a página mais antiga — ou seja, quando estiver de novo na cabeça da lista —, ela terá seu bit *R* em 0 e poderá, então, ser substituída. Assim o algoritmo sempre termina.

3.4.5 O algoritmo de substituição de página do relógio

Embora o segunda chance seja um algoritmo razoável, ele é desnecessariamente ineficaz, pois permanece constantemente reinserindo páginas no final da lista. Uma estratégia melhor é manter todas as páginas em uma lista circular em forma de relógio, como mostra a Figura 3.15. Um ponteiro aponta para a página mais antiga, ou seja, para a 'cabeça' da lista.

Quando ocorre uma falta de página, a página indicada pelo ponteiro é examinada. Se o bit *R* for 0, a página é removida, a nova página é inserida em seu lugar no relógio e o ponteiro avança uma posição. Se *R* for 1, ele é zerado e o ponteiro avança para a próxima página. Esse processo é repetido até que uma página seja encontrada com $R = 0$. Não é de surpreender que esse algoritmo seja chamado de **relógio**.

3.4.6 Algoritmo de substituição de página usada menos recentemente (LRU)

Uma boa aproximação do algoritmo ótimo de substituição de página é baseada na observação de que as páginas muito utilizadas nas últimas instruções provavelmente serão muito utilizadas novamente nas próximas instruções.

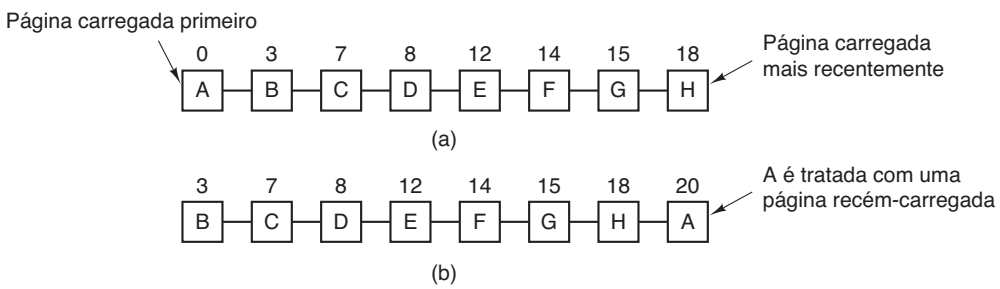

Figura 3.14 Operação de segunda chance. (a) Páginas na ordem FIFO. (b) Lista de páginas se uma falta de página ocorre no tempo 20 e o bit *R* de *A* possui o valor 1. Os números acima das páginas são seus tempos de carregamento.

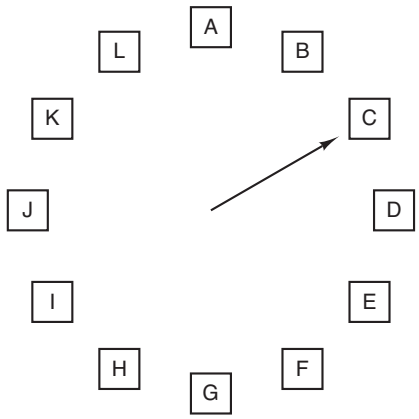

Figura 3.15 Algoritmo de substituição de página relógio.

Ao contrário, páginas que não estão sendo utilizadas por um longo período de tempo provavelmente permanecerão inutilizadas por muito tempo. Essa ideia sugere um algoritmo realizável: quando ocorrer uma falta de página, elimine a página não utilizada pelo período de tempo mais longo. Essa estratégia é chamada de paginação **LRU** (*least recently used* — usada menos recentemente).

Embora o LRU seja teoricamente realizável, não é barato. Para implementar completamente o LRU, é necessário manter uma lista vinculada de todas as páginas na memória, com a página usada mais recentemente na dianteira e a página usada menos recentemente na parte de trás. A dificuldade é que a lista deve ser atualizada em cada referência à memória. Encontrar uma página na lista, deletá--la e posicioná-la na dianteira é uma operação demorada, mesmo no hardware (supondo que um hardware assim possa ser construído).

Entretanto, há outros modos de implementar o LRU com hardwares especiais. Consideremos o modo mais simples primeiro. Esse método requer equipar o hardware com um contador de 64 bits, *C*, que é automaticamente incrementado após cada instrução. Além do mais, cada entrada na tabela de páginas também deve ter um campo grande o suficiente para acomodar o contador. Após cada referência à memória, o valor atual de *C* é armazenado na entrada da tabela de páginas para a página que acaba de ser referenciada. Quando ocorre uma falta de página, o sistema operacional examina todos os contadores na tabela de páginas para encontrar o menor deles. A página correspondente a esse menor valor será a "usada menos recentemente".

Examinemos agora uma segunda maneira de implementar o algoritmo LRU com o auxílio de um hardware especial. Para uma máquina com n molduras de página, esse hardware auxiliar pode conter uma matriz de $n \times n$ bits, inicialmente todos com o valor 0. Sempre que a moldura de página k for referenciada, esse hardware auxiliar primeiro marcará todos os bits da linha k com o valor 1 e, em seguida, todos os bits da coluna k com o valor 0. Em um instante qualquer, a linha que possuir o menor valor binário será a página LRU — ou seja, a página usada menos recentemente —, e a linha cujo valor binário seja o mais próximo do menor será a segunda usada menos recentemente e assim por diante. O funcionamento desse algoritmo é mostrado na Figura 3.16 para quatro molduras de página e referências a páginas na ordem

0 1 2 3 2 1 0 3 2 3

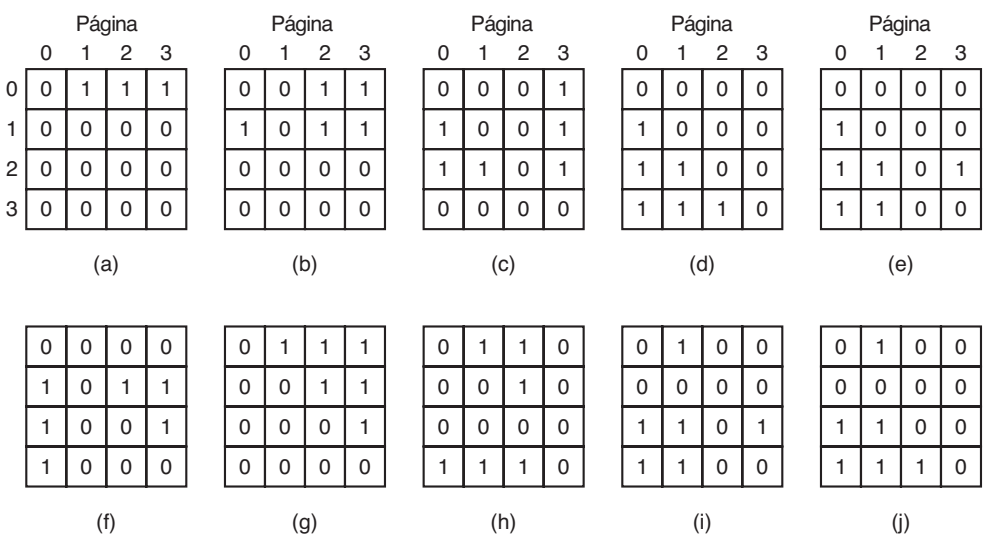

Figura 3.16 LRU usando uma matriz em que as páginas são referenciadas na ordem 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

Após a página 0 ser referenciada, temos a situação da Figura 3.16(a). Após a página 1 ser referenciada, temos a situação da Figura 3.16(b) e assim sucessivamente.

3.4.7 | Simulação do LRU em software

Embora ambas as implementações anteriores ao LRU sejam (em princípio) perfeitamente realizáveis, poucas máquinas — talvez nenhuma — têm esse hardware. Assim, é necessário encontrar uma solução implementável em software. Uma possibilidade é empregar o algoritmo de substituição de página não usada frequentemente (**NFU** — *not frequently used*). A implementação desse algoritmo requer contadores em software, cada um deles associado a uma página, inicialmente zerados. A cada interrupção de relógio, o sistema operacional percorre todas as páginas na memória. Para cada página, o bit *R*, que pode estar em 0 ou 1, é adicionado ao contador correspondente. Assim, esses contadores constituem uma tentativa de saber quantas vezes cada página já foi referenciada. Quando ocorrer uma falta de página, a página que tiver a menor contagem será selecionada para a substituição.

O problema principal com o algoritmo NFU é que ele nunca se esquece de nada. Por exemplo, em um compilador de múltiplos passos, páginas que foram intensamente referenciadas durante o passo 1 podem ainda ter um contador alto em passos bem posteriores. De fato, se acontecer de o passo 1 possuir o tempo de execução mais longo de todos os passos, as páginas que contiverem o código para os passos seguintes poderão ter sempre contadores menores do que as páginas do passo 1. Consequentemente, o sistema operacional removerá páginas que ainda estiverem sendo referenciadas, em vez daquelas que não o estão mais.

Felizmente, uma pequena modificação no algoritmo NFU possibilita a simulação do algoritmo LRU. Essa modificação tem dois passos. Primeiro, os contadores são deslocados um bit à direita. Em seguida, o bit *R* de cada página é adicionado ao bit mais à esquerda do contador correspondente, em vez de ao bit mais à direita.

A Figura 3.17 ilustra como funciona esse algoritmo modificado, também conhecido como **algoritmo de envelhecimento** (*aging*). Suponha que, após a primeira interrupção de relógio, os bits *R* das páginas 0 a 5 tenham, respectivamente, os valores 1, 0, 1, 0, 1 e 1 (página 0 é 1, página 1 é 0, página 2 é 1 etc.). Em outras palavras, entre as interrupções de relógio 0 e 1, as páginas 0, 2, 4 e 5 foram referenciadas e, assim, seus bits *R* foram colocados em 1, enquanto os bits *R* das outras páginas permaneceram em 0. Após os seis contadores correspondentes terem sido deslocados um bit à direita e cada bit *R* ter sido inserido à esquerda, esses contadores terão os valores mostrados na Figura 3.17(a). As quatro colunas restantes mostram os seis contadores após as quatro interrupções de relógio seguintes.

Quando ocorre uma falta de página, a página que tem a menor contagem é removida. É claro que a página que não tiver sido referenciada por, digamos, quatro interrupções de relógio terá quatro zeros nas posições mais significativas de seu contador e, assim, possuirá um valor menor do que um contador que não tiver sido referenciado por três interrupções de relógio.

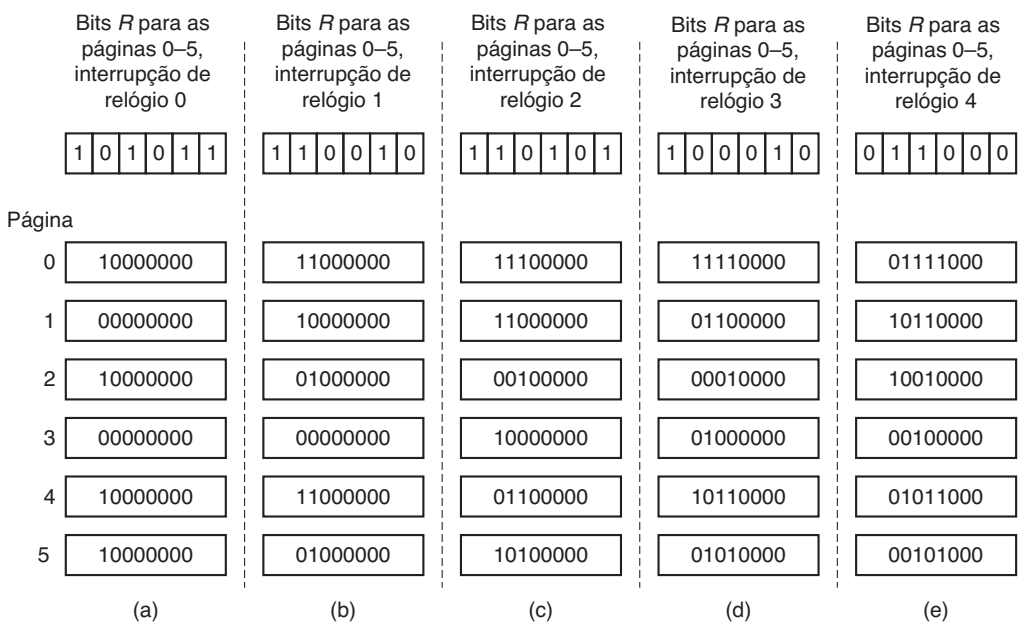

Figura 3.17 O algoritmo de envelhecimento simula o LRU em software. São mostradas seis páginas para cinco interrupções de relógio. As cinco interrupções de relógio são representadas de (a) até (e).

Esse algoritmo difere do LRU de duas maneiras. Observe as páginas 3 e 5 na Figura 3.17(e). Nenhuma delas foi referenciada por duas interrupções de relógio; mas ambas foram referenciadas na interrupção anterior àquelas. De acordo com o LRU, se uma página tiver de ser substituída, deveremos escolher uma das duas. O problema é que não sabemos qual dessas páginas foi referenciada por último no intervalo entre as interrupções de relógio 1 e 2. Registrando somente um bit por intervalo de tempo, perdemos a capacidade de distinguir a ordem das referências dentro de um mesmo intervalo. Tudo o que podemos fazer é remover a página 3, pois a página 5 foi referenciada duas interrupções de relógio antes e a página 3, não.

A segunda diferença entre o algoritmo LRU e o de envelhecimento é que, neste último, os contadores têm um número finito de bits (8 bits no exemplo dado). Imagine duas páginas, ambas com seus contadores zerados. Só nos resta substituir uma delas aleatoriamente. Na realidade, é bastante possível que uma das páginas tenha sido referenciada pela última vez nove intervalos atrás e a outra o tenha sido há mil intervalos. Não temos como verificar isso. Na prática, porém, 8 bits geralmente são suficientes se a interrupção de relógio ocorrer a cada 20 ms. Se uma página ficar sem ser referenciada durante 160 ms, provavelmente ela não é tão importante.

3.4.8 O algoritmo de substituição de página do conjunto de trabalho

No modo mais puro de paginação, os processos são inicializados sem qualquer de suas páginas presentes na memória. Assim que a CPU tenta buscar a primeira instrução, ela detecta uma falta de página, fazendo o sistema operacional carregar na memória a referida página que contém essa primeira instrução. Outras faltas de página, para as variáveis globais e a pilha, geralmente ocorrem logo em seguida. O processo, depois de um certo tempo, tem a maioria das páginas de que necessita para ser executado com relativamente poucas faltas de página. Essa estratégia é denominada **paginação por demanda**, pois as páginas só são carregadas à medida que são solicitadas, e não antecipadamente.

Obviamente, é fácil escrever um programa de teste que leia sistematicamente todas as páginas em um grande espaço de endereçamento e gere muitas faltas de página, de maneira que não exista memória suficiente para conter todas elas. Felizmente, a maioria dos processos não funciona assim. Eles apresentam uma propriedade denominada **localidade de referência**, a qual diz que, durante qualquer uma das fases de sua execução, o processo só vai referenciar uma fração relativamente pequena de suas páginas. Por exemplo, em cada passo de um compilador de múltiplos passos, somente uma fração de todas as suas páginas é referenciada, e essa fração é diferente a cada passo.

O conjunto de páginas que um processo está usando atualmente é denominado **conjunto de trabalho** (*working set*) (Denning, 1968a; Denning, 1980). Se todo esse conjunto estiver presente na memória, o processo será executado com poucas faltas de página até mudar para outra fase de execução (por exemplo, o próximo passo em um compilador). Se a memória disponível for muito pequena para conter todo esse conjunto de trabalho, o processo sofrerá muitas faltas de página e será executado lentamente, pois a execução de uma instrução leva apenas uns poucos nanossegundos, mas trazer uma página do disco para a memória consome, em geral, cerca de 10 milissegundos. Executar apenas uma ou duas instruções a cada 10 milissegundos faria demorar uma eternidade para finalizar o processamento. Diz-se que um programa que gere faltas de página frequente e continuamente provoca **ultrapaginação** (*thrashing*) (Denning, 1968b).

Em um sistema multiprogramado, os processos muitas vezes são transferidos para disco, ou seja, todas as suas páginas são retiradas da memória, para que outros processos possam utilizar a CPU. Uma questão que surge é: o que fazer quando as páginas relativas a um processo são trazidas de volta à memória? Tecnicamente, nada precisa ser feito. O processo simplesmente causará faltas de página até que seu conjunto de páginas tenha sido novamente carregado na memória. O problema é que, havendo 20, cem ou mesmo mil faltas de página toda vez que um processo é carregado, a execução se torna bastante lenta e é desperdiçado um considerável tempo de CPU, pois o sistema operacional gasta alguns milissegundos de tempo de CPU para processar uma falta de página.

Portanto, muitos sistemas de paginação tentam gerenciar o conjunto de trabalho de cada processo e assegurar que ele esteja presente na memória antes de o processo ser executado. Essa prática, denominada **modelo do conjunto de trabalho** (*working set model*) (Denning, 1970), foi concebida para reduzir substancialmente a frequência de faltas de página. Carregar páginas de um processo na memória *antes* de ele ser posto em execução também se denomina **pré-paginação**. Note que o conjunto de páginas se altera no tempo.

Não é de hoje que se sabe que a maioria dos programas não referencia seus espaços de endereçamento uniformemente, mas que essas referências tendem a se agrupar em um pequeno número de páginas. Uma referência à memória pode ocorrer para buscar uma instrução, buscar dados ou armazenar dados. Em qualquer instante de tempo, t, existe um conjunto que é constituído de todas as páginas usadas pelas k referências mais recentes à memória. Esse conjunto, $w(k, t)$, como vimos anteriormente, é o conjunto de trabalho. Como as $k > 1$ referências mais recentes devem ter empregado todas as páginas usadas pelas $k = 1$ referências mais recentes — e possivelmente outras —, $w(k, t)$ é uma função monotonicamente não decrescente de k. O

limite de $w(k, t)$, quando k cresce, é finito, pois um programa não pode referenciar mais páginas do que aquelas que seu espaço de endereçamento contém e poucos programas usam todas as páginas. A Figura 3.18 mostra o tamanho do conjunto de trabalho como função de k.

O fato de a maioria dos programas acessar aleatoriamente um pequeno número de páginas e esse conjunto se alterar lentamente no tempo explica a rápida subida inicial da curva e, em seguida, o crescimento lento para k maiores. Por exemplo, um programa que estiver executando um laço que ocupe duas páginas de código e acessando dados contidos em quatro páginas poderá referenciar todas essas seis páginas a cada mil instruções, mas sua referência mais recente a alguma outra página poderá ter acontecido um milhão de instruções atrás, durante a fase de inicialização. Em virtude desse comportamento assintótico, o conteúdo do conjunto de trabalho não é sensível ao valor escolhido de k, ou seja, existe uma ampla faixa de valores de k para os quais o conjunto de trabalho não se altera. Como o conjunto de trabalho varia em um ritmo lento, é possível saber com razoável segurança quais páginas serão necessárias quando o programa puder continuar sua execução, desde que se conheça o conjunto de trabalho do processo no instante em que a execução anterior foi interrompida. A pré-paginação consiste no carregamento dessas páginas na memória antes de reinicializar o processo.

Para implementar o modelo do conjunto de trabalho, é necessário que o sistema operacional saiba quais são as páginas pertencentes ao conjunto de trabalho. A posse dessa informação também leva imediatamente a esse possível algoritmo de substituição de página: ao ocorrer uma falta de página, encontre uma página não pertencente ao conjunto de trabalho e a remova da memória. Para implementar esse algoritmo, necessitamos de uma maneira precisa de determinar, a qualquer instante, quais páginas pertencem ao conjunto de trabalho e quais não pertencem. Por definição, o conjunto de trabalho é o conjunto das páginas usadas nas k mais recentes referências à memória (alguns autores preferem as k mais recentes referências à página, mas a escolha é arbitrária). Para implementar um algoritmo com base no conjunto de trabalho, é preciso escolher antecipadamente um valor para k. Uma vez escolhido o valor de k, o conjunto de trabalho — ou seja, o conjunto de páginas referenciadas nas últimas k referências à memória — é, após cada referência à memória, determinado de modo singular.

Obviamente, o fato de haver uma definição operacional do conjunto de trabalho não significa que exista uma maneira eficiente de gerenciá-lo em tempo real, ou seja, durante a execução do programa. Seria possível pensar um registrador de deslocamento de comprimento k, em que cada referência à memória deslocasse esse registrador de uma posição à esquerda e inserisse à direita o número da página referenciada mais recentemente. O conjunto de todos os k números de páginas presentes nesse registrador de deslocamento constituiria o conjunto de trabalho. Na teoria, em uma falta de página, o conteúdo desse registrador de deslocamento estaria apto a ser lido e ordenado. Páginas duplicadas poderiam, então, ser removidas. O que sobrasse constituiria o conjunto de trabalho. Contudo, manter um registrador de deslocamento e processá-lo a cada falta de página tem um custo proibitivo, o que faz com que essa técnica nunca seja usada.

Em vez disso, empregam-se várias aproximações. Uma delas, bastante comum, é a seguinte: abandona-se a ideia da contagem de k referências à memória e usa-se, em vez disso, o tempo de execução. Por exemplo, no lugar de definir que o conjunto de trabalho é constituído por aquelas páginas usadas nas últimas dez milhões de referências à memória, podemos considerar que ele seja constituído daquelas páginas referenciadas nos últimos 100 ms do tempo de execução. Na prática, essa definição é igualmente boa e, além disso, mais simples de usar. Note que, para cada processo, somente seu próprio tempo de execução é considerado. Assim, se um processo inicializar sua execução no instante T e até o instante $T + 100$ ms tiver utilizado 40 ms de tempo de CPU, para os propósitos do conjunto de trabalho, o tempo considerado para esse processo será de 40 ms. Em geral, dá-se o nome de **tempo virtual atual** a essa quantidade de tempo de CPU que um processo realmente empregou desde que foi inicializado. Usando essa

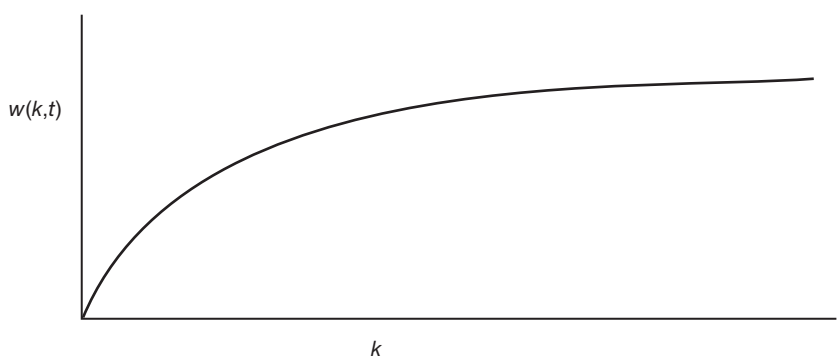

Figura 3.18 O conjunto de trabalho é o conjunto das páginas usadas pelas k referências mais recentes à memória. A função $w(k, t)$ é o tamanho do conjunto de trabalho no instante t.

aproximação, o conjunto de trabalho de um processo pode ser visto como o conjunto de páginas que ele referenciou durante os últimos τ segundos de tempo virtual.

Examinemos agora o algoritmo de substituição de página com base no conjunto de trabalho. A ideia principal é encontrar uma página que não esteja presente no conjunto de trabalho e removê-la da memória. Na Figura 3.19 vemos parte de uma tabela de páginas de uma máquina. Como somente páginas presentes na memória são consideradas candidatas à remoção, páginas ausentes da memória são ignoradas por esse algoritmo. Cada entrada contém (no mínimo) dois itens de informação: o instante aproximado em que a página foi referenciada pela última vez e o bit *R* (Referenciada). O retângulo branco vazio representa campos não necessários a esse algoritmo, como número da moldura de página, os bits de proteção e o bit *M* (Modificada).

O algoritmo funciona da seguinte maneira: suponha que o hardware inicialize os bits *R* e *M* de acordo com nossa discussão anterior. Do mesmo modo, suponha que uma interrupção de relógio periódica ative a execução do software que limpe o bit *Referenciada* em cada tique de relógio. A cada falta de página, a tabela de páginas é varrida à procura de uma página adequada para ser removida da memória.

O bit *R* de cada entrada da tabela de páginas é examinado. Se o bit *R* for 1, o tempo virtual atual é copiado no campo *Instante de último uso* na tabela de páginas, indicando que a página estava em uso no instante em que ocorreu a falta de página. Como a página foi referenciada durante a interrupção de relógio atual, ela está certamente presente no conjunto de trabalho e não é uma candidata a ser removida da memória (supõe-se que o intervalo τ corresponda a várias interrupções de relógio).

Se *R* é 0, a página não foi referenciada durante a interrupção de relógio atual e pode ser candidata à remoção da memória. Para saber se ela deve ou não ser removida da memória, sua idade — o tempo virtual atual menos seu *Instante de último uso* dessa página — é computada e comparada com o intervalo τ. Se a idade for maior do que o intervalo τ, faz muito tempo que essa página está ausente do conjunto de trabalho. Ela é, então, removida da memória e a nova página é carregada aí. Dá-se continuidade à atualização das entradas restantes.

Contudo, se *R* é 0, mas a idade é menor ou igual ao intervalo τ, a página ainda está no conjunto de trabalho. A página é temporariamente poupada; no entanto, a página com a maior idade (menor *Instante do último uso*) é marcada. Se a tabela toda é varrida e não se encontra nenhuma candidata à remoção, isso significa que todas as páginas estão no conjunto de trabalho. Nesse caso, se uma ou mais páginas com $R = 0$ são encontradas, aquela com maior idade será removida. Na pior das hipóteses, todas as páginas foram referenciadas durante a interrupção de relógio atual (e, portanto, todas têm $R = 1$); assim, uma delas será escolhida aleatoriamente para remoção, preferivelmente uma página não referenciada (limpa), caso exista.

3.4.9 O algoritmo de substituição de página WSClock

O algoritmo básico do conjunto de trabalho é enfadonho, pois é preciso pesquisar em cada falta de página toda a tabela de páginas para que seja localizada uma página adequada para ser substituída. Há um algoritmo melhorado, com base no algoritmo do relógio, que também usa informações do conjunto de trabalho: é o chamado **WSClock**

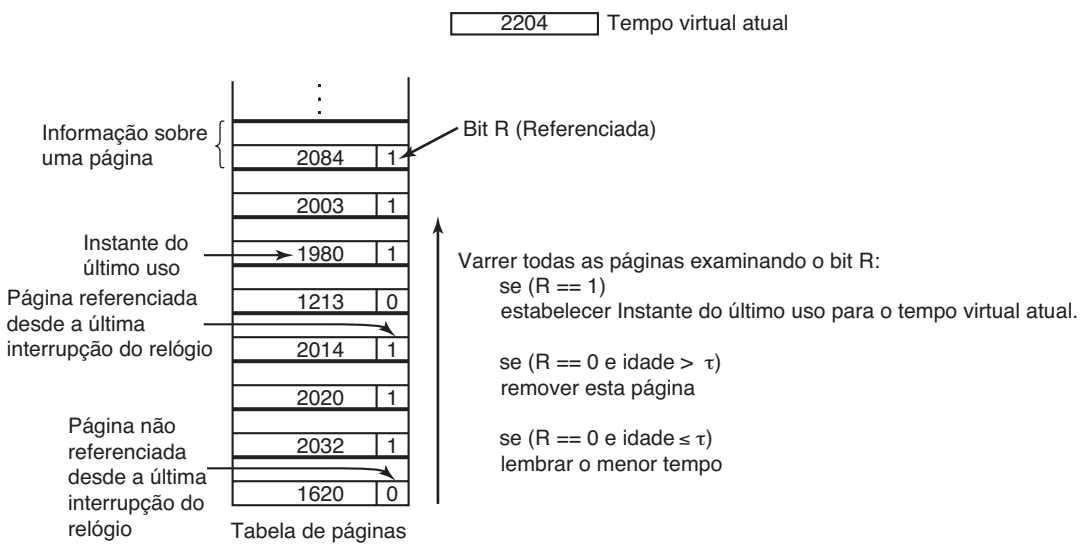

Figura 3.19 Algoritmo do conjunto de trabalho.

(Carr e Hennessey, 1981). Em virtude da simplicidade de implementação e do bom desempenho, esse algoritmo é amplamente utilizado.

A estrutura de dados necessária é uma lista circular de molduras de página (assim como no algoritmo do relógio), como mostra a Figura 3.20(a). Inicialmente, essa lista circular encontra-se vazia. Quando a primeira página é carregada, ela é inserida nessa lista. À medida que mais páginas são carregadas na memória, elas também são inseridas na lista para formar um anel. Cada entrada dessa lista contém o campo *Instante do último uso*, do algoritmo do conjunto de trabalho básico, bem como o bit *R* (mostrado) e o bit *M* (não mostrado).

Assim como ocorre com o algoritmo do relógio, a cada falta de página, a página que estiver sendo apontada será examinada primeiro. Se seu bit *R* for 1, a página foi referenciada durante a interrupção de relógio atual e, assim, não será candidata à remoção da memória. O bit *R* é, então, colocado em zero, o ponteiro avança para a página seguinte e o algoritmo é repetido para essa nova página. O estado posterior a essa sequência de eventos é mostrado na Figura 3.20(b).

Agora observe o que acontece se a página que estiver sendo apontada tiver seu bit $R = 0$, como ilustra a Figura 3.20(c). Se sua idade for maior do que o intervalo τ e se essa página estiver limpa, ela não se encontrará no conjunto de trabalho e haverá uma cópia válida em disco. A moldura de página é simplesmente reivindicada e a nova página é colocada lá, conforme se verifica na Figura 3.20(d). Por outro lado, se a página estiver suja, ela não poderá ser reivindicada imediatamente, já que não há uma cópia válida em disco. Para evitar um chaveamento de processo, a escrita em disco é escalonada, mas o ponteiro é avançado e o algoritmo continua com a próxima página. Afinal de contas, pode haver uma página velha e limpa mais adiante, apta a ser usada imediatamente.

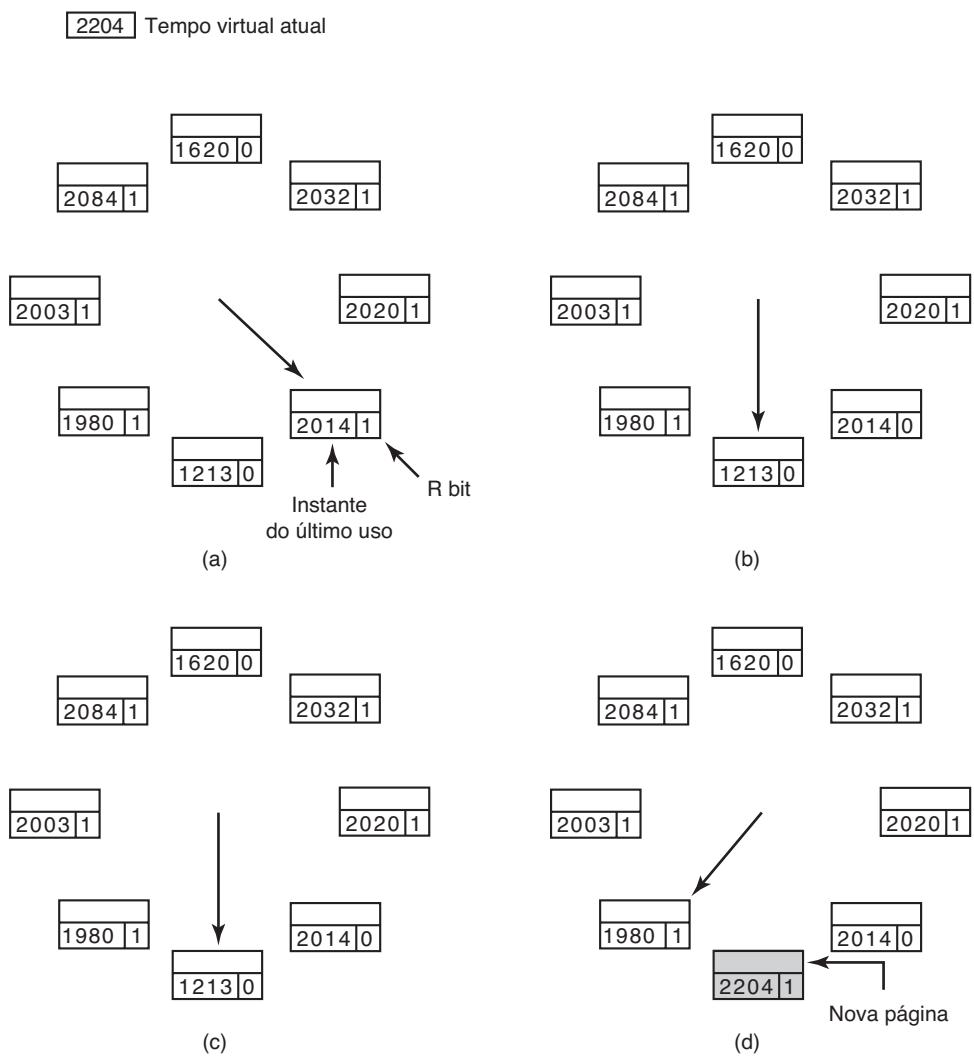

Figura 3.20 Funcionamento do algoritmo WSClock. (a) e (b) exemplificam o que acontece quando $R = 1$. (c) e (d) exemplificam a situação $R = 0$.

Em princípio, todas as páginas podem ser escalonadas para E/S em disco a cada volta do relógio. Para reduzir o tráfego de disco, pode-se estabelecer um limite, permitindo que um número máximo de *n* páginas sejam reescritas em disco. Uma vez alcançado esse limite, não mais serão escalonadas novas escritas em disco.

O que acontece se o ponteiro deu uma volta completa retornando a seu ponto de partida? Existem duas possibilidades a serem consideradas:

1. Pelo menos uma escrita foi escalonada.
2. Nenhuma escrita foi escalonada.

No primeiro caso, o ponteiro simplesmente continua a se mover, procurando por uma página limpa. Visto que uma ou mais escritas em disco foram escalonadas, uma dessas escritas acabará por se completar, e sua página será, então, marcada como limpa. A primeira página limpa encontrada é removida. Essa página não é necessariamente a primeira escrita escalonada, pois o driver de disco pode reordenar as escritas em disco para otimizar o desempenho.

No segundo caso, todas as páginas pertencem ao conjunto de trabalho; caso contrário, pelo menos uma escrita em disco teria sido escalonada. Em razão da falta de informações adicionais, a coisa mais simples a fazer é reivindicar qualquer página limpa e usá-la. A localização de uma página limpa pode ser registrada durante a varredura. Se nenhuma página limpa existir, então a página atual será escolhida e reescrita em disco.

3.4.10 Resumo dos algoritmos de substituição de página

Acabamos de analisar vários algoritmos de substituição de página. Nesta seção, faremos uma breve revisão. A lista de algoritmos discutidos é mostrada na Tabela 3.2.

O algoritmo ótimo substitui, entre as páginas atuais, a página que será referenciada por último. Infelizmente, não há como determinar qual página será a última, de modo que, na prática, esse algoritmo não pode ser usado. Entretanto, ele é útil como uma medida-padrão de desempenho, à qual outros algoritmos podem ser comparados.

O algoritmo NRU divide as páginas em quatro classes, dependendo do estado dos bits *R* e *M*. Uma página aleatória da classe de ordem mais baixa é escolhida aleatoriamente. Esse algoritmo é fácil de implementar, mas ainda bastante rudimentar. Existem outros melhores.

O algoritmo FIFO gerencia a ordem em que as páginas foram carregadas na memória, mantendo-as em uma lista encadeada. A remoção da página mais velha torna-se, assim, trivial, mas essa página ainda pode estar em uso, de modo que o FIFO não é uma escolha adequada.

Uma modificação do FIFO é o algoritmo segunda chance, que verifica se a página está em uso antes de removê-la. Se estiver, a página será poupada. Essa modificação melhora bastante o desempenho. O algoritmo do relógio é simplesmente uma implementação diferente do segunda chance: ele tem as mesmas propriedades de desempenho, mas gasta um pouco menos de tempo para executar o algoritmo.

O LRU é um excelente algoritmo, mas não pode ser implementado sem hardware especial. Se o hardware não está disponível, ele não pode ser usado. O NFU é uma tentativa rudimentar, não muito boa, de aproximação do LRU. Por outro lado, o algoritmo do envelhecimento é uma aproximação muito melhor do LRU e pode ser implementado eficientemente, constituindo uma boa escolha.

Os dois últimos algoritmos utilizam o conjunto de trabalho, que tem desempenho razoável, mas é de implementação um tanto cara. O WSClock é uma variante que não somente provê um bom desempenho, mas também é eficiente em sua implementação.

Algoritmo	Comentário
Ótimo	Não implementável, mas útil como um padrão de desempenho
NRU (não usada recentemente)	Aproximação muito rudimentar do LRU
FIFO (primeiro a entrar, primeiro a sair)	Pode descartar páginas importantes
Segunda chance	Algoritmo FIFO bastante melhorado
Relógio	Realista
LRU (usada menos recentemente)	Excelente algoritmo, porém difícil de ser implementado de maneira exata
NFU (não frequentemente usada)	Aproximação bastante rudimentar do LRU
Envelhecimento (*aging*)	Algoritmo eficiente que aproxima bem o LRU
Conjunto de trabalho	Implementação um tanto cara
WSClock	Algoritmo bom e eficiente

Tabela 3.2 Algoritmos de substituição de página discutidos no texto.

Entre todos eles, os dois melhores algoritmos são o envelhecimento e o WSClock. Eles baseiam-se, respectivamente, no LRU e no conjunto de trabalho. Ambos oferecem um bom desempenho de paginação e podem ser implementados de forma eficiente. Há poucos algoritmos além dos citados aqui, mas esses dois são provavelmente os mais importantes na prática.

3.5 Questões de projeto para sistemas de paginação

Nas seções anteriores, explicamos como funciona a paginação, introduzimos os algoritmos básicos de substituição de página e mostramos como modelá-los. No entanto, apenas conhecer a mecânica básica não basta. Para projetar um sistema de paginação e fazê-lo funcionar bem, é preciso saber muito mais. É como a diferença entre saber como mover a torre, o cavalo, o bispo e outras peças do xadrez e ser um bom jogador. Nas seções a seguir, analisaremos outras questões que os projetistas de sistemas operacionais devem considerar cuidadosamente para obter um bom desempenho de um sistema de paginação.

3.5.1 Política de alocação local *versus* global

Nas seções anteriores, discutimos vários algoritmos de escolha da página a ser substituída quando da ocorrência de uma falta de página. A maior questão associada a essa escolha (cuja discussão adiamos até agora) é a seguinte: como a memória deve ser alocada entre processos concorrentes em execução?

Dê uma olhada na Figura 3.21(a). Nela, três processos — *A*, *B* e *C* — constituem o conjunto de processos executáveis. Suponha que *A* tenha uma falta de página. O algoritmo de substituição de página deve tentar encontrar a página usada menos recentemente, levando em conta somente as seis páginas atualmente alocadas para *A*, ou deve considerar todas as páginas na memória? Se esse algoritmo considerar somente as páginas alocadas para *A*, a página com o menor valor de idade será *A5*, de modo que obteremos a situação da Figura 3.21(b).

Por outro lado, se a página com o menor valor de idade for removida sem considerar a quem pertence, a página *B3* será escolhida e teremos então a situação da Figura 3.21(c). O algoritmo da Figura 3.21(b) é um algoritmo de substituição local, ao passo que o algoritmo da Figura 3.21(c) é um algoritmo de substituição **global**. Algoritmos de substituição local alocam uma fração fixa de memória para cada processo. Algoritmos de substituição global alocam molduras de página entre os processos em execução. Assim, o número de molduras de página alocadas a cada processo varia no tempo.

Em geral, os algoritmos globais funcionam melhor, especialmente quando o tamanho do conjunto de trabalho varia durante o tempo de vida de um processo. Se um algoritmo local é usado e o conjunto de trabalho cresce durante a execução do processo, uma ultrapaginação (*thrashing*) pode ocorrer mesmo que existam muitas molduras de página disponíveis na memória. Se o conjunto de trabalho diminuir durante a execução do processo, os algoritmos locais desperdiçam memória. Se um algoritmo global é usado, o sistema deve decidir continuamente quantas molduras de página alocar para cada processo. Uma das maneiras de fazer isso é o monitoramento do tamanho do conjunto de trabalho desse processo, conforme indicado pelos bits de envelhecimento (*aging*), mas essa estratégia não necessariamente evita a ultrapaginação. O conjunto de trabalho pode variar de tamanho em questão de microssegundos, enquanto os bits de envelhecimento representam uma medida rudimentar estendida a um número de interrupções de relógio.

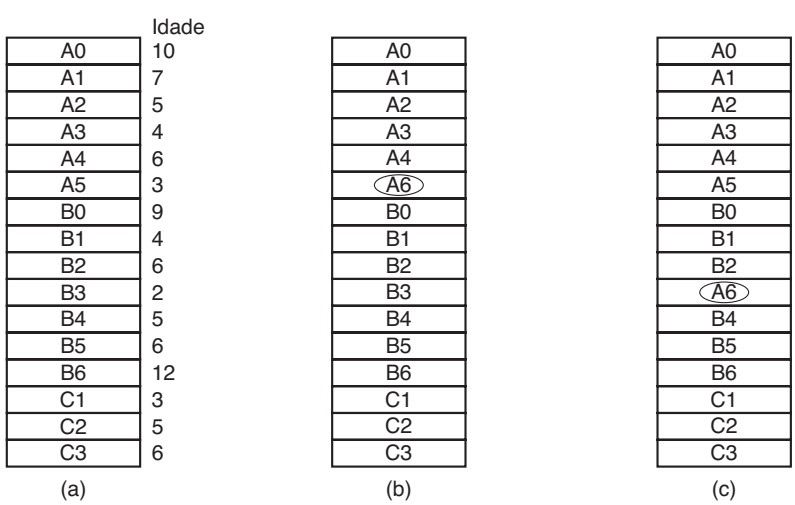

Figura 3.21 Substituição de página local *versus* global. (a) Configuração original. (b) Substituição de página local. (c) Substituição de página global.

Outra prática se baseia em um algoritmo de alocação de molduras de página para processos. Uma possibilidade é determinar periodicamente o número de processos em execução e alocar para cada processo o mesmo número de molduras de página. Assim, com 12.416 molduras de página disponíveis (isto é, excluídas as do sistema operacional) e dez processos, cada processo obtém 1.241 molduras de página. As seis molduras restantes vão para uma área comum (*pool*) para serem usadas na ocorrência de falta de página.

Embora esse método pareça justo, não tem muito sentido alocar a mesma quantidade de memória para um processo de 10 KB e para um de 300 KB. Em vez disso, é possível alocar molduras de página proporcionalmente ao tamanho total de cada processo — ou seja, um processo de 300 KB obteria 30 vezes a quantidade alocada a um processo de 10 KB. Parece razoável alocar para cada processo um número mínimo de molduras de página, de modo que ele possa ser executado, não importando o quão pequeno seja. Por exemplo, em algumas máquinas, uma única instrução de dois operandos pode precisar de seis molduras de página, pois a instrução em si, o operando-fonte e o operando-destino podem todos extrapolar os limites de página. Com a alocação de apenas cinco molduras de página, programas que contenham essas instruções não poderão ser executados.

Se um algoritmo global for usado, pode ser possível inicializar cada processo com um número de molduras de página proporcional a seu tamanho, mas a alocação tem de ser atualizada dinamicamente durante a execução do processo. Uma maneira de gerenciar a alocação é usar o algoritmo **PFF** (*page fault frequency* — frequência das faltas de página). Esse algoritmo informa quando aumentar ou diminuir a alocação de páginas de um processo, mas nada diz acerca de quais páginas substituir quando ocorrerem faltas de página. Ele somente controla o tamanho do conjunto de alocação.

Para uma classe extensa de algoritmos de substituição de página, incluindo o LRU, sabe-se que a frequência de faltas de página diminui à medida que mais molduras de página são alocadas ao processo, conforme discutimos anteriormente. Essa é a suposição por trás do PFF. Essa propriedade é ilustrada na Figura 3.22.

Medir a frequência de faltas de página é direto: basta contar o número de faltas por segundo e depois calcular a frequência média de faltas por segundo. Em seguida, para cada segundo, somar à média existente — ou seja, à frequência de faltas de página atual — o número de faltas ocorridas nesse novo segundo e, em seguida, dividir por dois, a fim de obter a nova média de faltas — ou seja, a nova frequência de faltas de página atual. A linha tracejada *A* corresponde a uma frequência de faltas de página inaceitavelmente alta, de modo que o processo que gerou a faltas de página deve receber mais molduras de página para reduzir essa taxa. A linha tracejada *B* corresponde a uma frequência de faltas de página tão baixa que permite concluir que o processo dispõe de muita memória. Nesse caso, algumas molduras de página podem ser retiradas desse processo. Assim, o algoritmo PFF tentará manter a frequência de paginação para cada processo em limites aceitáveis.

É importante notar que alguns algoritmos de substituição de página podem funcionar tanto com uma política de substituição local como uma global. Por exemplo, o FIFO pode substituir a página mais antiga em toda a memória (algoritmo global) ou a página mais antiga possuída pelo processo atual (algoritmo local). Da mesma maneira, o LRU — ou algum algoritmo aproximado — pode substituir a página menos usada recentemente em toda a memória (algoritmo global) ou a página menos usada recentemente possuída pelo processo atual (algoritmo local). A escolha de local *versus* global é independente, em alguns casos, do algoritmo escolhido.

Por outro lado, para os demais algoritmos de substituição de página, somente uma estratégia local tem sentido. Por exemplo, os algoritmos conjunto de trabalho e WSClock referem-se a algum processo específico e, portanto, devem ser aplicados naquele contexto. Para esses algoritmos, não existe um conjunto de trabalho para a máquina como um todo; a tentativa de unir todos os conjuntos de trabalho dos processos causaria a perda da propriedade da localidade e, consequentemente, esses algoritmos não funcionariam bem.

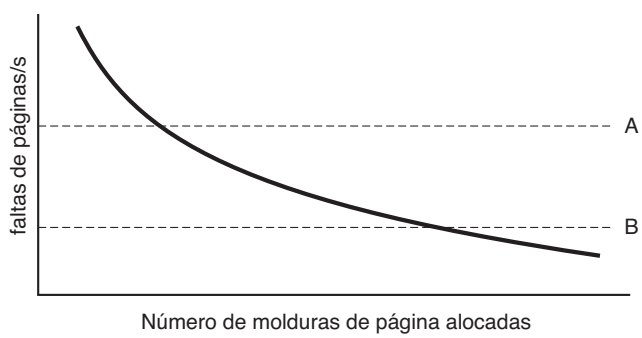

Figura 3.22 Frequência de faltas de página como função do número de molduras de página alocadas.

3.5.2 | Controle de carga

Mesmo com o melhor algoritmo de substituição de página e uma ótima alocação global de molduras de página a processos, pode ocorrer ultrapaginação (*thrashing*). Na verdade, sempre que os conjuntos de trabalho de todos os processos combinados excedem a capacidade da memória, pode-se esperar a ocorrência de ultrapaginação. Um sintoma dessa situação é que o algoritmo PFF indica que alguns processos precisam de mais memória, mas que nenhum outro processo necessita de menos memória. Nesse caso, não existe como alocar mais memória a processos que dela precisam sem que, com isso, se prejudiquem alguns outros processos. A única solução possível é livrar-se temporariamente de alguns dos processos.

Um bom modo de reduzir o número de processos que competem por memória é levar alguns deles para disco e liberar a memória a eles alocada. Por exemplo, um processo pode ser levado para disco e suas molduras de página serem distribuídas entre outros processos que estão sofrendo ultrapaginação. Se esta parar, o sistema pode continuar a execução durante um certo tempo. Se a ultrapaginação não acabar, outro processo terá de ser levado para disco e assim por diante, até que a ultrapaginação cesse. Assim, mesmo com paginação, a troca de processos entre a memória e o disco ainda é necessária. A diferença é que agora a troca de processos é usada para reduzir a demanda potencial por memória, em vez de reivindicar páginas.

A ideia de remover processos para disco a fim de aliviar a carga sobre a memória origina-se do escalonamento de dois níveis, no qual alguns processos são colocados em disco e um escalonador de curto prazo é empregado para escalonar os processos restantes na memória. É claro que as duas ideias podem ser combinadas, de modo que se remova somente um número suficiente de processos para disco, a fim de fazer com que a frequência de faltas de página fique aceitável. Periodicamente, alguns processos são trazidos do disco para a memória, e outros, presentes na memória, são levados para disco.

Contudo, um outro fator a ser considerado é o grau de multiprogramação. Quando o número de processos presentes na memória principal é muito pequeno, a CPU pode ficar ociosa durante consideráveis períodos de tempo. Ao escolher quais processos devem ser levados para disco, é preciso considerar não só o tamanho dos processos e a taxa de paginação, mas também outros aspectos, como se o processo é do tipo limitado pela CPU ou por E/S e quais as características que os processos remanescentes na memória possuem.

3.5.3 | Tamanho de página

O tamanho de página é um parâmetro que frequentemente pode ser escolhido pelo sistema operacional. Mesmo se o hardware tiver sido projetado com, por exemplo, páginas de 512 bytes, o sistema operacional poderá facilmente considerar os pares de página 0 e 1, 2 e 3, 4 e 5 etc., como páginas de 1 KB, simplesmente alocando sempre duas páginas consecutivas de 512 bytes.

A determinação do melhor tamanho de página requer o balanceamento de vários fatores conflitantes, o que leva a não se conseguir um tamanho ótimo geral. Há dois argumentos a favor de um tamanho pequeno de página. O primeiro é o seguinte: é provável que um segmento de código, dados ou pilha escolhido aleatoriamente não ocupe um número inteiro de páginas. Em média, metade da última página permanecerá vazia e, portanto, esse espaço será desperdiçado. Esse desperdício é denominado **fragmentação interna**. Com n segmentos na memória e um tamanho de página de p bytes, $np/2$ bytes serão perdidos com a fragmentação interna. Essa é uma razão para ter um tamanho de página pequeno.

O segundo argumento a favor de um tamanho reduzido de página fica óbvio se pensarmos, por exemplo, em um programa que seja constituído de oito fases sequenciais de 4 KB cada. Se o tamanho de página for 32 KB, esse programa demandará 32 KB durante todo o tempo de execução. Se o tamanho de página for 16 KB, ele requererá somente 16 KB. Se o tamanho de página se mostrar igual ou inferior a 4 KB, esse programa requisitará somente 4 KB em um instante qualquer. Em geral, tamanhos grandes de página farão com que partes do programa não usadas ocupem a memória desnecessariamente.

Por outro lado, páginas pequenas implicam muitas páginas e, consequentemente, uma grande tabela de páginas. Um programa de 32 KB precisa de apenas quatro páginas de 8 KB, mas 64 páginas de 512 bytes. As transferências entre memória e disco geralmente são de uma página por vez, e a maior parte do tempo é gasta no posicionamento da cabeça de leitura/gravação na trilha correta e no tempo de rotação necessário para que a cabeça de leitura/gravação atinja o setor correto, de modo que se gasta muito mais tempo na transferência de páginas pequenas do que na de páginas grandes. A transferência de 64 páginas de 512 bytes, do disco para a memória, pode levar 64 × 10 ms, mas a transferência de quatro páginas de 8 KB talvez leve somente 4 × 12 ms.

Em algumas máquinas, a tabela de páginas deve ser carregada em registradores de hardware sempre que a CPU chavear de um processo para outro. Assim, nessas máquinas, o tempo necessário para carregar os registradores com a tabela de páginas aumenta à medida que se diminui o tamanho da página. Além disso, o espaço ocupado pela tabela de páginas também aumenta quando o tamanho da página diminui.

Este último ponto pode ser analisado matematicamente. Seja de s bytes o tamanho médio de processo e de p bytes o tamanho de página. Além disso, suponha que cada entrada da tabela de páginas requeira e bytes. O número aproximado de páginas necessárias por processo será então

de *s*/*p* páginas, ocupando, assim, *se*/*p* bytes do espaço da tabela de páginas. A memória desperdiçada na última página em virtude da fragmentação interna é de *p*/2 bytes. Desse modo, o custo adicional total em decorrência da tabela de páginas e da perda pela fragmentação interna é dado pela soma destes dois termos:

$$\text{custo adicional} = se/p + p/2$$

O primeiro termo (tamanho da tabela de páginas) é grande quando o tamanho de página *p* é pequeno. O segundo termo (fragmentação interna), ao contrário, é grande quando o tamanho de página *p* também é grande. O valor ótimo para o tamanho de página deve ser um valor intermediário. Calculando a derivada primeira com relação a *p* e fazendo-a igual a zero, obtemos a seguinte equação:

$$-se/p^2 + 1/2 = 0$$

Dessa equação, poderemos obter a expressão que dá o tamanho ótimo de página (considerando somente a memória desperdiçada em virtude da fragmentação e do tamanho da tabela de páginas). O resultado é:

$$p = \sqrt{2se}$$

Para $s = 1$ MB e $e = 8$ bytes por entrada da tabela de páginas, o tamanho ótimo de página é 4 KB. Computadores comercialmente disponíveis têm usado tamanhos de página que variam de 512 bytes a 64 KB. Um valor típico de tamanho de página era 1 KB, mas atualmente 4 KB ou 8 KB são mais comuns. À medida que as memórias se tornam maiores, o tamanho de página também tende a ficar maior (mas não linearmente). Quadruplicando-se o tamanho da memória, raramente duplica-se o tamanho de página.

3.5.4 Espaços separados de instruções e dados

A maioria dos computadores tem um espaço único de endereçamento para programas e dados, como mostra a Figura 3.23(a). Se esse espaço de endereçamento for suficientemente grande, tudo funcionará bem. No entanto, muitas vezes esse espaço é muito pequeno, o que obriga os programadores a suar a camisa para fazer tudo caber no espaço de endereçamento.

Uma solução, que apareceu pioneiramente no PDP-11 (16 bits), consiste em espaços de endereçamento separados para instruções (código do programa) e dados. Esses espaços são denominados, respectivamente, **espaço I** e **espaço D**. Cada espaço de endereçamento se situa entre 0 e um valor máximo — em geral $2^{16} - 1$ ou $2^{32} - 1$. Na Figura 3.23(b), vemos esses dois espaços de endereçamento. O ligador (*linker*) deve saber quando espaços separados estão sendo usados, pois, nessas situações, os dados são realocados para o endereço virtual 0, independentemente de inicializarem após o programa.

Em um computador com esse projeto, ambos os espaços de endereçamento podem ser paginados, independentemente um do outro. Cada um deles possui sua própria tabela de páginas, que contém o mapeamento individual de páginas virtuais para molduras de página física. Quando o hardware busca uma instrução, ele sabe que deve usar o espaço I e a tabela de páginas do espaço I. Da mesma maneira, referências aos dados devem acontecer por intermédio da tabela de páginas do espaço D. A não ser por essa distinção, usar espaços separados de instruções e dados não causa nenhuma complicação especial e duplica o espaço de endereçamento disponível.

3.5.5 Páginas compartilhadas

Outro aspecto importante do projeto é o compartilhamento de páginas. Em grandes sistemas com multiprogramação, é comum haver vários usuários executando simultaneamente o mesmo programa. É nitidamente mais eficiente compartilhar páginas para evitar a situação de existirem duas cópias ou mais da mesma página presentes na memória. Um problema é que nem todas as páginas são compartilháveis. Em particular, as páginas somente de leitura — como as que contêm código de programa — são compartilháveis, mas páginas com dados alteráveis durante a execução não o são.

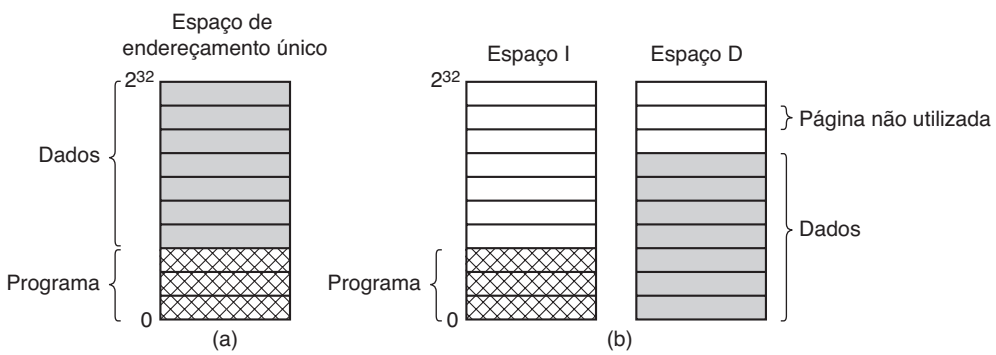

Figura 3.23 (a) Um espaço de endereçamento. (b) Espaços I e D independentes.

Se o sistema der suporte aos espaços I e D, o compartilhamento de programas será obtido de modo relativamente direto, fazendo com que dois ou mais processos usem, no espaço I, a mesma tabela de páginas em vez de tabelas de páginas individuais, mas diferentes tabelas de páginas no espaço D. Em geral, em uma implementação que suporte compartilhamento desse tipo, tabelas de páginas são estruturas de dados independentes da tabela de processos. Cada processo tem, assim, dois ponteiros em sua tabela de processos: um aponta para a tabela de páginas do espaço I e outro, para a tabela de páginas do espaço D, como mostra a Figura 3.24. Quando o escalonador escolher um processo para ser executado, ele usará esses ponteiros para localizar as tabelas de páginas apropriadas e ativar a MMU. Mesmo sem espaços I e D separados, os processos ainda podem compartilhar programas (ou, às vezes, bibliotecas), mas o mecanismo é mais complicado.

Quando dois ou mais processos compartilham o mesmo código, um problema ocorre com as páginas compartilhadas. Suponha que os processos *A* e *B* estejam ambos executando o editor e compartilhando suas páginas. Se o escalonador decide remover *A* da memória, descartando todas as suas páginas e carregando as molduras de página vazias com outro programa, ele leva o processo *B* a causar muitas faltas de página, até que suas páginas estejam novamente presentes na memória.

De maneira semelhante, quando o processo *A* termina sua execução, é essencial que o sistema operacional saiba que as páginas utilizadas pelo processo *A* ainda estão sendo utilizadas por outro processo, a fim de que o espaço em disco ocupado por essas páginas não seja liberado acidentalmente. Em geral, é muito trabalhoso pesquisar todas as tabelas de páginas para descobrir se uma determinada página é compartilhada, de modo que são necessárias estruturas de dados especiais para manter o controle das páginas compartilhadas — especialmente se a unidade de compartilhamento for de uma página individual (ou de um conjunto delas), em vez de toda uma tabela de páginas.

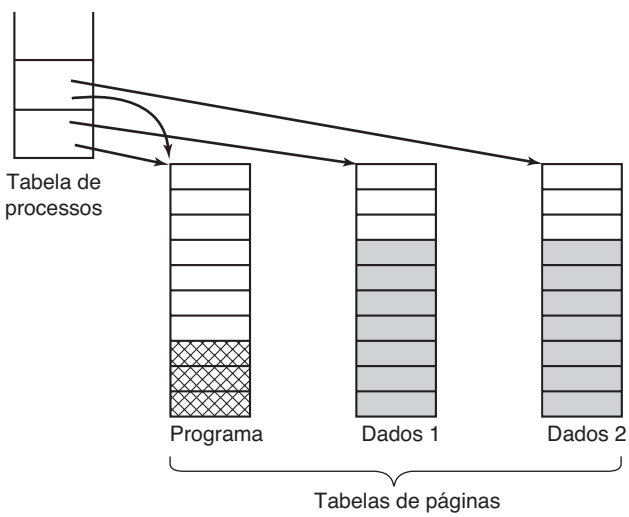

Figura 3.24 Dois processos que compartilham o mesmo programa compartilhando sua tabela de páginas.

Compartilhar dados é mais complicado do que compartilhar código, mas não chega a ser impossível. Por exemplo, no UNIX, após uma chamada ao sistema fork, o processo pai e o processo filho compartilham código e dados. Em um sistema com paginação, o que muitas vezes se faz é fornecer a cada um desses dois processos sua própria tabela de páginas, de modo que ambos possam apontar para o mesmo conjunto de páginas. Assim, nenhuma cópia de páginas é feita durante a execução do comando fork. Contudo, todas as páginas de dados são mapeadas em ambos os processos, pai e filho, como SOMENTE PARA LEITURA (*read-only*).

Enquanto os processos estiverem apenas lendo seus dados, sem modificá-los, essa situação pode perdurar. Assim que um dos dois processos atualizar uma palavra da memória, a violação da proteção contra gravação (*read-only*) causará uma interrupção, desviando-se, assim, para o sistema operacional. Então, é feita uma cópia da página, de modo que cada processo agora tem sua cópia particular. As duas cópias são então marcadas para LEITURA-ESCRITA (*read-write*) — portanto, operações de escrita subsequentes em uma das duas cópias prosseguirão sem interrupções. Essa estratégia significa que páginas não modificadas (incluindo todas as páginas de código) não precisam ser copiadas. Somente páginas de dados que foram realmente modificadas devem ser copiadas separadamente. Esse método, denominado **copiar-se-escrita** (*copy on write*), melhora o desempenho por meio da redução do número de cópias.

3.5.6 Bibliotecas compartilhadas

O compartilhamento pode ser feito em outras granularidades além de páginas individuais. Se um programa for inicializado duas vezes, a maioria dos sistemas operacionais automaticamente compartilhará todas as páginas de texto, de modo que apenas uma cópia esteja na memória. As páginas de texto sempre são apenas para leitura, por isso não há nenhum problema nesse caso. Dependendo do sistema operacional, cada processo pode obter sua própria cópia privada das páginas de dados ou eles podem ser compartilhados e marcados como somente leitura. Se qualquer processo modifica uma página de dados, uma cópia privada será feita para ele, ou seja, o método copiar-se-escrita será aplicado.

Em sistemas modernos, há muitas bibliotecas grandes usadas por muitos processos; por exemplo, a biblioteca que trata a caixa de diálogo de busca de arquivos para abrir e de bibliotecas gráficas múltiplas. Ligar estaticamente todas essas bibliotecas a cada programa executável no disco as tornaria ainda mais infladas do que já são.

Em vez disso, uma técnica comum é usar **bibliotecas compartilhadas** (que são chamadas de **DLL** ou **bibliotecas de ligação dinâmica** no Windows). Para esclarecer a

ideia de uma biblioteca compartilhada, primeiro considere a ligação tradicional. Quando um programa é ligado, um ou mais arquivos do objeto e possivelmente algumas bibliotecas são nomeados no comando ao vinculador, como o comando do UNIX

```
ld *.o –lc –lm
```

que liga todos os arquivos (do objeto) *.o* no diretório atual e em seguida varre duas bibliotecas, */usr/lib/libc.a* e */usr/lib/libm.a*. Quaisquer funções chamadas nos arquivos de objeto mas ausentes ali (por exemplo, *printf*) recebem o nome de **externas indefinidas** e são buscadas nas bibliotecas. Se encontradas, são incluídas no arquivo binário executável. Por exemplo, se *printf* precisa de *write,* e *write* não está incluída, o vinculador a buscará e incluirá quando encontrar. Quando o vinculador acaba, um arquivo binário executável é escrito no disco contendo todas as funções necessárias. As funções presentes na biblioteca mas não chamadas não são incluídas. Quando o programa é carregado na memória e executado, todas as funções de que necessita estão ali.

Agora suponha que programas comuns usem 20–50 MB em gráficos e funções de interface com o usuário. Ligar estaticamente centenas de programas com todas essas bibliotecas gastaria uma quantidade enorme de espaço no disco, bem como desperdiçaria espaço em RAM quando eles fossem carregados, uma vez que o sistema não teria como saber que poderia compartilhá-los. É aqui que entram as bibliotecas compartilhadas. Quando um programa é ligado com bibliotecas compartilhadas (que são ligeiramente diferentes das estáticas), em vez de incluir a função atual chamada, o vinculador inclui uma pequena rotina de *stub* que liga à função chamada no instante de execução. Dependendo do sistema e dos detalhes de configuração, as bibliotecas compartilhadas são carregadas quando o programa é carregado ou quando funções nelas são chamadas pela primeira vez. É claro que, se outro programa já tiver carregado a biblioteca compartilhada, não há necessidade de carregá-la novamente — isso é o mais importante. Note que, quando uma biblioteca é carregada ou usada, não é a biblioteca inteira que é lida de uma só vez. As páginas entram uma a uma, de acordo com a necessidade; assim, as funções que não são chamadas não serão trazidas à RAM.

Além de reduzir os arquivos executáveis e de economizar espaço na memória, as bibliotecas compartilhadas têm outra vantagem: se uma função em uma biblioteca compartilhada for atualizada para remover um erro, não é necessário recompilar os programas que a chamam. Os antigos arquivos binários continuam a funcionar. Essa característica é especialmente importante para softwares comerciais, em que o código-fonte não é distribuído ao cliente. Por exemplo, se a Microsoft encontra e repara um erro de segurança em algum DLL padrão, o Windows Update fará o download do novo DLL e substituirá o antigo, e todos os programas que usam o DLL automaticamente usarão a nova versão da próxima vez que forem inicializados.

Contudo, as bibliotecas compartilhadas apresentam um pequeno problema que deve ser resolvido. O problema é ilustrado na Figura 3.25. Aqui vemos dois processos compartilhando uma biblioteca de tamanho de 20 KB (supondo que cada caixa tenha 4 KB). Entretanto, a biblioteca está localizada em um endereço diferente em cada processo, presumivelmente porque os próprios programas não são do mesmo tamanho. No processo 1, a biblioteca começa no endereço 36 KB; no processo 2, começa em 12 K. Suponha que a primeira coisa a ser feita pela primeira função seja saltar para o endereço 16 na biblioteca. Se a biblioteca não fosse compartilhada, poderia ser realocada dinamicamente quando carregada, de modo que o salto (no processo 1) poderia ser para o endereço virtual 36 K + 16. Note que o endereço físico na RAM onde a biblioteca é localizada não importa, uma vez que todas as páginas são mapeadas de endereços físicos para virtuais pela MMU no hardware.

No entanto, visto que a biblioteca é compartilhada, a realocação dinâmica não funcionará. Afinal, quando a primeira função é chamada pelo processo 2 (no endereço 12 K), a instrução *jump* deve ir para 12 K + 16, e não para 36 K + 16. Esse é um pequeno problema. Um modo de resolvê-lo é usar o método copiar-se-escrita e criar novas páginas para cada processo compartilhando a biblioteca, realocando-as dina-

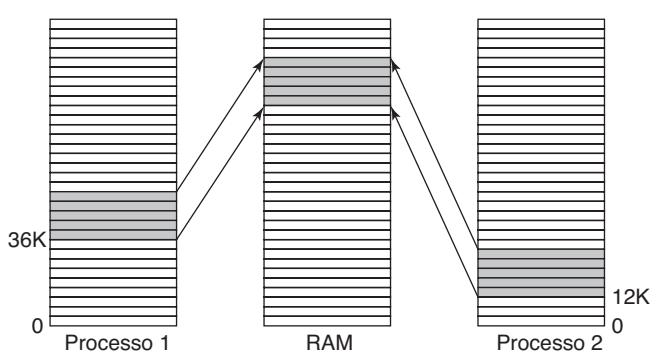

Figura 3.25 Uma biblioteca compartilhada sendo usada por dois processos.

micamente quando são criadas, mas é claro que esse esquema frustra o propósito de compartilhamento da biblioteca.

Uma solução melhor é compilar bibliotecas compartilhadas com uma *flag* de compilador especial, que diz ao compilador para não gerar instruções que usem endereços absolutos. Em vez disso, apenas instruções usando endereços relativos são empregadas. Por exemplo, quase sempre há uma instrução que diz: "salte adiante (ou para trás) *n* bytes (em oposição a uma instrução que fornece um endereço específico para o qual saltar)". Essa instrução funciona corretamente, não importa onde a biblioteca compartilhada esteja localizada no espaço de endereçamento virtual. Evitando endereços absolutos, o problema pode ser resolvido. O código que usa apenas deslocamentos relativos é chamado de **código independente da posição**.

3.5.7 | Arquivos mapeados

As bibliotecas compartilhadas são realmente um caso especial de um recurso mais geral, chamado **arquivos mapeados em memória**. A ideia aqui é que um processo pode emitir uma chamada ao sistema para mapear um arquivo em uma porção de seu espaço de endereçamento virtual. Na maior parte das implementações, nenhuma página é trazida durante o período do mapeamento, mas, à medida que as páginas são usadas, são paginadas, uma a uma, por demanda, usando o arquivo no disco como memória auxiliar. Quando o processo sai, ou explicitamente finaliza o mapeamento do arquivo, todas as páginas modificadas são escritas de volta no arquivo.

Os arquivos mapeados fornecem um modelo alternativo para E/S. Em vez de fazer leituras e gravações, o arquivo pode ser acessado como um grande arranjo de caracteres na memória. Em algumas situações, os programadores acham esse modelo mais conveniente.

Se dois ou mais processos mapeiam o mesmo arquivo ao mesmo tempo, eles podem se comunicar através da memória compartilhada. Gravações feitas por um processo na memória compartilhada são imediatamente visíveis quando o outro lê da parte de seu espaço de endereçamento virtual mapeado no arquivo. Dessa forma, esse arquivo fornece um canal de largura de banda elevada entre processos e muitas vezes é usado como tal (muitas vezes utiliza-se até um arquivo temporário). Agora deve estar claro que, se arquivos mapeados em memória estiverem disponíveis, as bibliotecas compartilhadas podem usar esse mecanismo.

3.5.8 | Política de limpeza

A paginação funciona melhor quando existe uma grande quantidade de molduras de página disponíveis prontas a serem requisitadas quando ocorrerem faltas de página. Se toda moldura de página estiver ocupada e, além disso, for modificada, antes de uma nova página ser carregada na memória, uma página antiga deverá primeiro ser escrita em disco. Para garantir um estoque abundante de molduras de página disponíveis, muitos sistemas de paginação executam um processo específico, denominado **daemon de paginação** (*paging daemon*), que dorme quase todo o tempo, mas que é acordado periodicamente para inspecionar o estado da memória. Se existirem apenas algumas molduras de página disponíveis, o daemon de paginação começa a selecionar as páginas a serem removidas da memória usando um algoritmo de substituição de página. Se essas páginas tiverem sido modificadas desde quando carregadas, elas serão escritas em disco.

Em qualquer eventualidade, o conteúdo anterior da página é lembrado. No caso de uma página descartada da memória ser novamente referenciada antes de sua moldura ter sido sobreposta por uma nova página, a moldura pode ser reclamada, ou seja, trazida de volta, retirando-a do conjunto de molduras de página disponível. Manter uma lista de molduras de página disponíveis fornece melhor desempenho do que pesquisar toda a memória em busca de uma moldura de página disponível todas as vezes em que isso se tornar necessário. O daemon de paginação garante, no mínimo, que todas as molduras de página disponíveis estejam limpas e que, assim, não precisem ser, quando requisitadas, escritas às pressas em disco.

Uma maneira de implementar essa política de limpeza é usar um relógio com dois ponteiros. O ponteiro da frente é controlado pelo daemon de paginação. Quando esse ponteiro aponta para uma página suja, essa página é escrita em disco e o ponteiro avança. Quando aponta para uma página limpa, ele apenas avança. O ponteiro de trás é usado para substituição de página, assim como no algoritmo-padrão do relógio, só que, agora, a probabilidade de o ponteiro de trás apontar para uma página limpa aumenta em virtude do trabalho do daemon de paginação.

3.5.9 | Interface da memória virtual

Até agora, supomos que a memória virtual seja transparente a processos e programadores — isto é, tudo o que se vê é um grande espaço de endereçamento virtual em um computador com memória física menor. Em muitos sistemas isso é verdade, mas, em alguns sistemas avançados, os programadores dispõem de algum controle sobre o mapa de memória e podem usá-lo de maneiras não tradicionais para aumentar o desempenho do programa. Nesta seção, discutiremos brevemente algumas dessas maneiras.

Uma razão para dar o controle do mapa de memória a programadores é permitir que dois ou mais processos compartilhem a mesma memória. Se os programadores puderem nomear regiões de memória, talvez se torne possível a um processo fornecer a outro o nome de uma região, de modo que esse segundo processo também possa ser mapeado na mesma região do primeiro. Com dois (ou mais) processos compartilhando as mesmas páginas, surge a possibilidade de um compartilhamento em largura de banda

elevada — ou seja, um processo escreve na memória compartilhada e o outro lê dessa mesma memória.

O compartilhamento de páginas também pode ser usado na implementação de um sistema de troca de mensagens de alto desempenho. Em geral, na troca de mensagens, dados são copiados de um espaço de endereçamento para outro, a um custo considerável. Se os processos puderem controlar seus mapeamentos, uma mensagem poderá ser trocada retirando-se a página relacionada do mapeamento do processo emissor e inserindo-a no mapeamento do processo receptor. Nesse caso, somente os nomes de páginas devem ser copiados, em vez de todos os dados.

Outra técnica avançada de gerenciamento de memória é a **memória compartilhada distribuída** (Feeley et al., 1995; Li, 1986; Li e Hudak, 1989; Zekauskas et al., 1994). A ideia é permitir que vários processos em uma rede compartilhem um conjunto de páginas — possível, mas não necessariamente — por intermédio de um único espaço de endereçamento linear compartilhado. Quando um processo referencia uma página não atualmente mapeada, isso causa uma falta de página. O manipulador de falta de página (que pode estar no núcleo ou no espaço de usuário) localiza a máquina que contém a referida página e manda um pedido para liberar essa página de seu mapeamento e enviá-la pela rede. Quando a página chega, ela é mapeada na memória e a instrução que causou a falta de página é reinicializada. No Capítulo 8, examinaremos mais detalhadamente a memória compartilhada distribuída.

3.6 Questões de implementação

Para a implementação de sistemas de memória virtual, os implementadores têm de escolher entre os principais algoritmos teóricos vistos, como: segunda chance *versus* envelhecimento (*aging*), alocação local *versus* global e paginação por demanda *versus* paginação antecipada. Mas eles também devem estar cientes de inúmeras outras questões práticas de implementação. Nesta seção, introduziremos os problemas mais comuns e algumas das soluções possíveis.

3.6.1 Envolvimento do sistema operacional com a paginação

Existem quatro circunstâncias em que o sistema operacional tem de se envolver com a paginação: na criação do processo, no tempo de execução do processo, na ocorrência de falta de página e na finalização do processo. Veremos rapidamente cada um desses momentos para saber como proceder.

Quando um novo processo é criado em um sistema com paginação, o sistema operacional deve determinar qual será o tamanho (inicial) do programa e de seus dados e criar uma tabela de páginas para eles. Um espaço precisa ser alocado na memória para a tabela de páginas, e esta deve ser inicializada. A tabela de páginas não precisa estar presente na memória quando o processo é levado para disco, mas ela tem de estar na memória quando o processo estiver em execução. Além disso, um espaço deve ser alocado na área de trocas do disco (*swap area*), de modo que, quando uma página é devolvida ao disco, ela tenha para onde ir. A área de trocas também deve ser inicializada com o código do programa e os dados, para que, quando o novo processo começar a causar faltas de página, as páginas possam ser trazidas do disco para a memória. Alguns sistemas paginam o programa diretamente do arquivo executável, economizando, assim, espaço em disco e tempo de inicialização. Por fim, informações acerca da tabela de páginas e da área de trocas de processos em disco devem ser registradas na tabela de processos.

Quando um processo é escalonado para execução, a MMU tem de ser reinicializada para o novo processo, e a TLB, esvaziada para livrar-se de resíduos do processo executado anteriormente. A tabela de páginas do novo processo deve tornar-se a tabela atual, o que em geral é feito copiando-se a tabela ou um ponteiro para ela em algum(ns) registrador(es) em hardware. Opcionalmente, algumas páginas do processo — ou todas elas — podem ser trazidas para a memória a fim de reduzir o número inicial de faltas de página.

Quando ocorre uma falta de página, o sistema operacional tem de ler o(s) registrador(es) em hardware para determinar o endereço virtual causador da falta de página. A partir dessa informação, ele precisa calcular qual página virtual é requisitada e, então, localizá-la em disco. Em seguida, ele procura uma moldura de página disponível para colocar a nova página, descartando, se necessário, alguma página antiga. No passo seguinte, ele deve carregar a página requisitada do disco para a moldura de página. Por fim, o sistema operacional tem de salvar o contador de programa para que ele aponte para a instrução que causou a falta de página, de modo que possa ser executada novamente.

Quando um processo termina, o sistema operacional deve liberar sua tabela de páginas, suas páginas e o espaço em disco que as páginas ocupam. Se algumas das páginas forem compartilhadas com outros processos, as páginas na memória e em disco só poderão ser liberadas quando o último processo que as usar for finalizado.

3.6.2 Tratamento de falta de página

Finalmente estamos em condições de descrever, com alguns detalhes, o que ocorre durante uma falta de página. A sequência de eventos acontece da seguinte maneira:

1. O hardware gera uma interrupção que desvia a execução para o núcleo, salvando o contador de programa na pilha. Na maioria das máquinas, algumas informações acerca do estado da instrução atualmente em execução também são salvas em registradores especiais na CPU.

2. Uma rotina em código de montagem é ativada para salvar o conteúdo dos registradores de uso geral e outras informações voláteis, a fim de impedir que o sistema operacional o destrua. Essa rotina chama o sistema operacional como um procedimento.
3. O sistema operacional descobre a ocorrência de uma falta de página e tenta identificar qual página virtual é necessária. Muitas vezes, um dos registradores em hardware contém essa informação. Do contrário, o sistema operacional deve resgatar o contador de programa, buscar a instrução e analisá-la por software para descobrir qual referência gerou essa falta de página.
4. Uma vez conhecido o endereço virtual causador da falta de página, o sistema operacional verifica se esse endereço é válido e se a proteção é consistente com o acesso. Se não, o processo recebe um sinal ou é eliminado. Se o endereço for válido e nenhuma violação de proteção tiver ocorrido, o sistema verificará se existe uma moldura de página disponível. Se nenhuma moldura de página estiver disponível, o algoritmo de substituição será executado para selecionar uma vítima.
5. Se o conteúdo da moldura de página tiver sido modificado, a página será escalonada para ser transferida para o disco; um chaveamento de contexto será realizado, ou seja, será suspenso o processo causador da falta de página e outro processo será executado até que a transferência da página para disco tenha sido completada. De qualquer modo, a moldura de página é marcada como não disponível para impedir que seja usada com outro propósito.
6. Tão logo a moldura de página seja limpa (imediatamente ou após ter sido escrita em disco), o sistema operacional buscará o endereço em disco onde está a página virtual solicitada e escalonará uma operação em disco para trazê-la para a memória. Enquanto a página estiver sendo carregada na memória, o processo causador da falta de página será mantido em suspenso e outro processo será executado, caso exista.
7. Quando a interrupção de disco indicar que a página chegou na memória, as tabelas de páginas serão atualizadas para refletir sua posição, e será indicado que a moldura de página está em estado normal.
8. A instrução causadora da falta de página é recuperada para o estado em que ela se encontrava quando começou sua execução, e o contador de programa é reinicializado, a fim de apontar para aquela instrução.
9. O processo causador da falta de página é escalonado para execução, e o sistema operacional retorna para a rotina, em linguagem de máquina, que a chamou.
10. Essa rotina recarrega os registradores e outras informações de estado e retorna ao espaço de usuário para continuar a execução, como se nada tivesse ocorrido.

3.6.3 Backup de instrução

Quando um programa referencia uma página não presente na memória, a instrução causadora da falta de página é bloqueada no meio de sua execução e ocorre uma interrupção, desviando-se assim para o sistema operacional. Após o sistema operacional buscar em disco a página necessária, ele deverá reinicializar a instrução causadora da interrupção. Isso é mais fácil de explicar do que de implementar.

Para entender melhor esse problema em seu pior grau, imagine uma CPU que tenha instruções de dois endereços, como o Motorola 680x0, amplamente utilizado em sistemas embarcados. Por exemplo, a instrução

```
MOV.L #6(A1),2(A0)
```

é de 6 bytes (veja a Figura 3.26). Para reinicializar essa instrução, o sistema operacional deve determinar onde se encontra o primeiro byte da instrução. O valor do contador de programa no instante da interrupção depende de qual operando causou a falta de página e de como o microcódigo da CPU foi implementado.

Na Figura 3.26, temos uma instrução que se inicializa no endereço 1000 e que faz três referências à memória: a própria palavra de instrução e os dois deslocamentos dos operandos. Dependendo de qual dessas três referências tiver causado a falta de página, o contador de programa pode ser 1000, 1002 ou 1004 no instante da falta. Muitas vezes é impossível ao sistema operacional determinar precisamente onde a instrução se inicializa. Se o contador de programa estiver na posição 1002 no instante da ocorrência da falta de página, o sistema operacional não disporá de meios para determinar se a palavra no endereço 1002 é um endereço de memória associado a uma instrução em 1000 (por exemplo, a posição de um operando) ou se é o próprio código de operação da instrução.

Apesar de esse já ser um grande problema, ele poderia ser ainda pior. Alguns modos de endereçamento do 680x0 usam autoincremento, o que significa que o efeito colateral da execução dessa instrução é incrementar um ou mais registradores. Instruções que empregam autoincremento também são capazes de causar falta de página. Dependendo dos detalhes do microcódigo, o incremento pode ser feito antes da referência à memória, obrigando o sistema operacional a realizar o decremento do registrador por software antes de reexecutar a instrução. Ou o autoincremento pode ser feito após a referência à memória, e assim o sistema

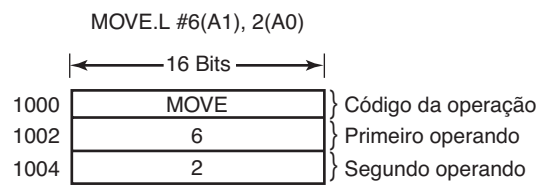

Figura 3.26 Uma instrução provocando uma falta de página.

operacional não tem de desfazê-lo nesse caso, pois o incremento ainda não terá ocorrido no momento da interrupção. Também pode existir o autodecremento, capaz de causar um problema similar. Detalhes precisos que informam se o autoincremento e o autodecremento são feitos antes ou depois da referência à memória podem diferir de instrução para instrução e de uma CPU para outra.

Felizmente, em algumas máquinas os projetistas da CPU fornecem uma solução, geralmente na forma de um registrador interno escondido em que o conteúdo do contador de programa é salvo antes de cada instrução ser executada. Essas máquinas também podem ter um segundo registrador, que informa quais registradores sofreram autoincremento ou autodecremento e de quanto é o valor. Com essas informações, o sistema operacional pode desfazer sem ambiguidade todos os efeitos da instrução causadora da falta de página, de modo que possa ser reexecutada. Se essas informações não estiverem disponíveis, o sistema operacional precisará se desdobrar para descobrir o que ocorreu antes de poder reinicializar a execução da instrução. É como se os projetistas de hardware fossem incapazes de resolver o problema e o tivessem deixado para os projetistas do sistema operacional resolverem. Caras legais.

3.6.4 Retenção de páginas na memória

Embora neste capítulo não tenhamos discutido muito acerca de E/S, o fato de um computador ter memória virtual não significa que não haja E/S. Memória virtual e E/S interagem de maneira sutil. Por exemplo, considere um processo que tenha emitido uma chamada ao sistema para ler de algum arquivo ou dispositivo para um buffer em seu espaço de endereçamento. Enquanto o processo está esperando que a E/S seja completada, ele é suspenso e outro processo entra em execução. Esse outro processo causa, então, uma falta de página.

Se o algoritmo de paginação é global, existe uma possibilidade pequena, mas não nula, de que a página que contém o buffer de E/S seja escolhida para ser removida da memória. Se um dispositivo de E/S estiver atualmente na fase de transferência via DMA para aquela página, sua remoção da memória fará com que uma parte dos dados seja escrita na página correta, e a outra parte, na nova página carregada na memória. Uma solução para esse problema é trancar as páginas envolvidas com E/S na memória, de modo que não possam ser removidas. Essa ação do sistema operacional é muitas vezes denominada **retenção de página** (*pinning*). Outra solução é fazer todas as operações de E/S para buffers no núcleo e, posteriormente, copiar os dados para as páginas do usuário.

3.6.5 Memória secundária

Em nossa discussão de algoritmos de substituição de página, vimos como uma página é selecionada para remoção. Mas não abordamos a localização em disco onde a página descartada da memória é colocada. Vamos então tratar de algumas das questões relacionadas ao gerenciamento de disco.

O algoritmo mais simples para a alocação de espaço em disco consiste na manutenção de uma área de troca (*swap area*) em disco ou, ainda melhor que isso, em um disco separado do sistema de arquivos (para balancear a carga de E/S). A maioria dos sistemas UNIX funciona dessa forma. Essa partição não tem um sistema de arquivos normal, o que elimina os custos indiretos de converter deslocamentos em arquivos para bloquear endereços. Em vez disso, são usados números de bloco relativos ao início da partição em todo o processo.

Quando o sistema operacional é inicializado, essa área de troca encontra-se vazia e é representada na memória como uma única entrada contendo sua localização e seu tamanho. No esquema mais simples, quando o primeiro processo é inicializado, reserva-se uma parte dessa área de troca, do tamanho desse processo, e a área de troca restante fica reduzida dessa quantidade. Quando novos processos são inicializados, a eles também são atribuídos pedaços da área de troca de tamanhos iguais aos ocupados por cada um deles. À medida que os processos vão terminando, seus espaços em disco são gradativamente liberados. A área de troca em disco é gerenciada como uma lista de pedaços disponíveis. Algoritmos melhores serão discutidos no Capítulo 10.

O endereço da área de troca de cada processo é mantido na tabela de processos. O cálculo do endereço para escrever uma página é simples: basta adicionar o deslocamento da página dentro do espaço de endereçamento virtual ao endereço inicial da área de troca. Contudo, antes que um processo possa começar sua execução, a área de troca deve ser inicializada. Para isso, copia-se o processo todo em sua área de troca em disco, carregando-o na memória de acordo com a necessidade. Ou, ainda, pode-se carregar o processo todo na memória e removê-lo para o disco quando necessário.

No entanto, esse modelo simples apresenta um problema: os processos podem aumentar de tamanho no decorrer de suas execuções. Embora o código do programa geralmente tenha um tamanho fixo, a área de dados algumas vezes pode crescer, mas a área de pilha certamente crescerá. Consequentemente, talvez seja melhor reservar áreas de troca separadas para código, dados e pilha e permitir que cada uma delas consista em mais de um pedaço em disco.

O outro extremo consiste em não alocar nada antecipadamente e apenas alocar o espaço em disco para cada página quando ela for enviada para lá e liberar o mesmo espaço quando a página for carregada na memória. Dessa maneira, os processos na memória não ficam amarrados a nenhuma área específica de troca. A desvantagem é a necessidade de manter na memória o endereço de cada página armazenada em disco. Em outras palavras, é preciso haver uma tabela para cada processo dizendo onde se encontra cada uma de suas páginas em disco. As duas alternativas são mostradas na Figura 3.27.

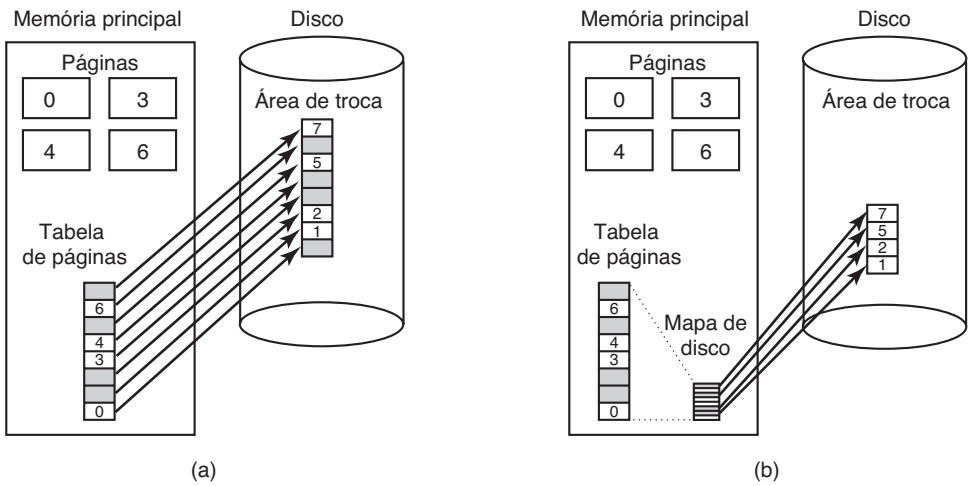

Figura 3.27 (a) Paginação para uma área de troca estática. (b) Recuperando páginas dinamicamente.

A Figura 3.27(a) ilustra uma tabela de páginas com oito páginas. As páginas 0, 3, 4 e 6 estão na memória principal. As páginas 1, 2, 5 e 7 estão em disco. A área de troca em disco é tão grande quanto o espaço de endereçamento virtual do processo (oito páginas); cada página tem uma localização fixa, para onde ela é reescrita quando é removida da memória principal. O cálculo desse endereço requer o conhecimento apenas do endereço de onde a área de paginação do processo começa, visto que as páginas são armazenadas de modo contíguo e na mesma ordem dos números das páginas virtuais. As páginas presentes na memória sempre têm uma cópia de sombra em disco, mas essa cópia pode estar desatualizada se a página tiver sido modificada desde seu carregamento na memória. As páginas sombreadas na memória indicam páginas ausentes da memória, e as páginas sombreadas no disco são (em princípio) substituídas pelas cópias na memória, embora a cópia em disco possa ser usada caso a página em memória tenha de ser enviada novamente ao disco e não tenha sido modificada desde que foi carregada.

Na Figura 3.27(b), as páginas não têm endereços fixos em disco. Quando uma página é levada para disco, uma página vazia em disco é escolhida livremente e o mapa do disco (que tem espaço para um endereço de disco por página virtual) é atualizado. As páginas que estão na memória não têm cópia em disco; suas entradas no mapa do disco contêm um endereço de disco inválido ou um bit que indica que elas não estão em uso.

Nem sempre é possível ter uma partição de área de troca fixa. Por exemplo, pode ser que não haja nenhuma partição de disco disponível. Nesse caso, um ou mais arquivos pré-alocados dentro do sistema de arquivos normal podem ser usados. O Windows usa esse método. Entretanto, uma otimização pode ser usada aqui para reduzir a quantidade de disco necessária. Uma vez que o texto do programa de cada processo vem de algum arquivo (executável) no sistema de arquivos, o arquivo executável pode ser usado como área de troca. Mais do que isso, visto que o texto do programa em geral é somente para leitura, quando a memória está cheia e páginas do programa têm de ser removidas da memória, elas são apenas descartadas e lidas novamente a partir do programa executável quando necessário. As bibliotecas compartilhadas também podem funcionar desse modo.

3.6.6 Separação da política e do mecanismo

Uma ferramenta importante para gerenciar a complexidade de qualquer sistema é separar a política do mecanismo. Esse princípio pode ser aplicado ao gerenciamento de memória fazendo com que muitos dos gerenciadores de memória sejam executados como processos no nível do usuário. Essa separação foi feita primeiro no Mach (Young et al., 1987). A discussão a seguir é ligeiramente baseada no Mach.

A Figura 3.28 traz um exemplo simples de como a política e o mecanismo podem ser separados. Aqui o sistema de gerenciamento de memória é dividido em três partes:

1. Um manipulador de MMU de baixo nível.
2. Um manipulador de falta de página que faz parte do núcleo.
3. Um paginador externo executado no espaço do usuário.

Todos os detalhes de como a MMU trabalha são escondidos em seu manipulador, que possui código dependente de máquina, devendo ser reescrito para cada nova plataforma que o sistema operacional executar. O manipulador de falta de página é um código independente de máquina e contém a maioria dos mecanismos para paginação. A política é em grande parte determinada pelo paginador externo, que executa como um processo do usuário.

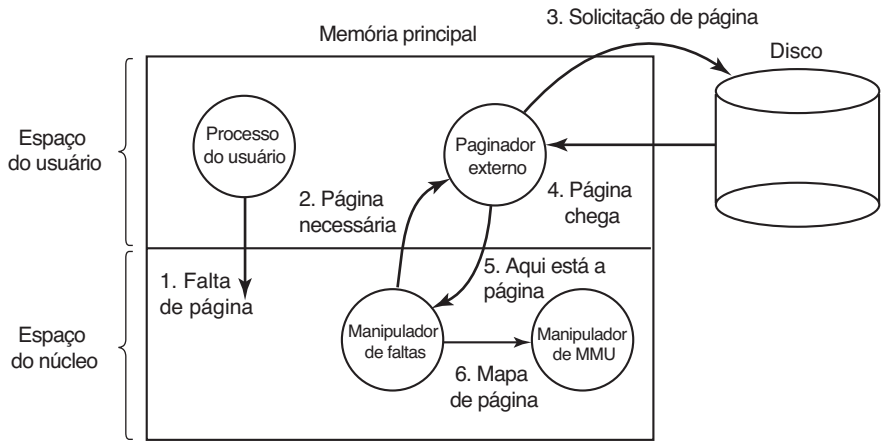

Figura 3.28 Tratamento de falta de página com um paginador externo.

Quando um processo é inicializado, o paginador externo é notificado para ajustar o mapa de páginas do processo e, se necessário, alocar uma área de troca no disco. À medida que o processo executa, ele pode mapear novos objetos em seu espaço de endereçamento, de modo que o paginador externo seja notificado novamente.

Uma vez que o processo inicie a execução, ele pode causar uma falta de página. O manipulador de faltas calcula qual página virtual é necessária e a envia para o paginador externo, informando qual é o problema. O paginador externo então lê a página necessária do disco e a copia para uma porção de seu próprio espaço de endereçamento. Com isso, ele diz ao tratador de faltas onde a página está. O tratador de faltas remove o mapeamento da página do espaço de endereçamento do paginador externo e solicita ao manipulador de MMU que coloque a página no local correto dentro do espaço de endereçamento do usuário. O processo do usuário pode, então, ser reinicializado.

Essa implementação deixa livre o local onde o algoritmo de substituição de página é colocado. Seria mais limpo tê-lo no paginador externo, mas existem alguns problemas com esse esquema. O principal deles é que o paginador externo não tem acesso aos bits *R* e *M* de todas as páginas. Esses bits ditam regras em muitos dos algoritmos de paginação. Assim, ou algum mecanismo é necessário para passar essa informação para o paginador externo ou o algoritmo de substituição deve estar no núcleo do sistema operacional. No segundo caso, o manipulador de faltas diz ao paginador externo qual página ele selecionou para a remoção e, então, passa os dados, mapeando-os no espaço de endereçamento do paginador externo ou incluindo-os em uma mensagem. Em qualquer método, o paginador externo escreve os dados no disco.

A vantagem principal dessa implementação é permitir um código mais modular e com maior flexibilidade. A principal desvantagem é a sobrecarga adicional causada pelos diversos chaveamentos entre o núcleo e o usuário e a sobrecarga nas trocas de mensagens entre as partes do sistema. Por enquanto, o assunto é altamente controverso, mas, como os computadores se tornam cada dia mais rápidos e os programas cada vez mais complexos, em uma execução demorada, o sacrifício de algum desempenho para aumentar a confiabilidade dos programas provavelmente será aceitável para a maioria dos programadores.

3.7 Segmentação

A memória virtual discutida até agora é unidimensional porque o endereçamento virtual vai de 0 a algum endereço máximo, um após o outro. Para muitos problemas, ter dois ou mais espaços de endereçamento separados pode ser muito melhor do que ter somente um. Por exemplo, um compilador tem muitas tabelas construídas em tempo de compilação, possivelmente incluindo:

1. O código-fonte sendo salvo para impressão (em sistemas em lote).
2. A tabela de símbolos que contém os nomes e os atributos das variáveis.
3. A tabela com todas as constantes usadas, inteiras e em ponto flutuante.
4. A árvore sintática, que contém a análise sintática do programa.
5. A pilha usada pelas chamadas de rotina dentro do compilador.

Cada uma das primeiras quatro tabelas cresce continuamente quando a compilação prossegue. A última cresce e diminui de modo imprevisível durante a compilação. Em uma memória unidimensional, essas cinco tabelas teriam de ser alocadas em regiões contíguas do espaço de endereçamento virtual, como se pode observar na Figura 3.29.

Imagine o que ocorre se um programa tem um número excepcionalmente grande de variáveis, mas uma quantidade normal de todo o restante. A região do espaço de

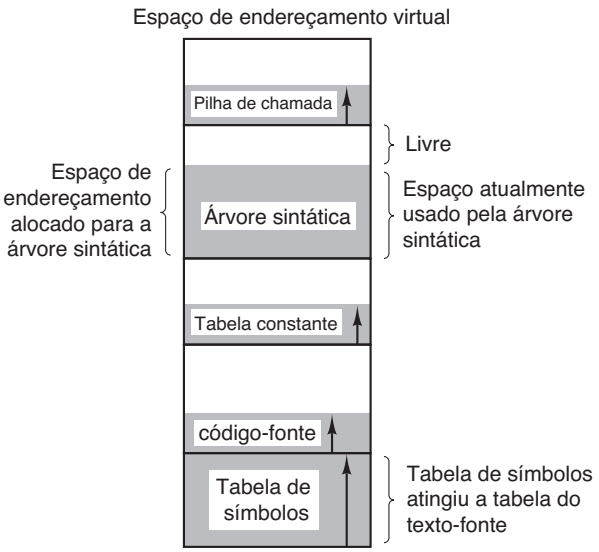

Figura 3.29 Em um espaço de endereçamento unidimensional com tabelas crescentes, uma tabela poderá atingir outra.

endereçamento alocada para a tabela de símbolos pode se esgotar, mas talvez existam várias entradas livres nas outras tabelas. Obviamente, o compilador poderia simplesmente emitir uma mensagem que informasse que a compilação não continuaria em virtude da grande quantidade de variáveis, mas essa atitude não parece muito interessante, pois se sabe que existem espaços não utilizados nas outras tabelas.

Outra possibilidade é imitar Robin Hood, tirando espaço das tabelas com excesso de entradas livres e dando para as tabelas com poucas entradas livres. Esse entrelaçamento pode ser feito, mas é como gerenciar os próprios sobreposições — um incômodo, na melhor das hipóteses; na pior delas, um trabalho ingrato e tedioso.

O que é realmente necessário é uma maneira de livrar o programador da obrigatoriedade de gerenciar a expansão e a contração de tabelas, do mesmo modo que a memória virtual elimina a preocupação de organizar o programa em sobreposições.

Uma solução extremamente abrangente e direta é prover a máquina com muitos espaços de endereçamento completamente independentes, chamados de **segmentos**. Cada segmento é constituído de uma sequência linear de endereços, de 0 a algum máximo. O tamanho de cada segmento pode ser qualquer um, de 0 ao máximo permitido. Segmentos diferentes podem ter diferentes tamanhos (e geralmente é o que ocorre). Além disso, os tamanhos dos segmentos podem variar durante a execução. O tamanho de cada segmento de pilha é passível de ser expandido sempre que algo é colocado sobre a pilha e diminuído toda vez que algo é retirado dela.

Pelo fato de cada segmento constituir um espaço de endereçamento separado, segmentos diferentes podem crescer ou reduzir independentemente, sem afetarem uns aos outros. Se uma pilha, em certo segmento, precisa de mais espaço de endereçamento para crescer, ela pode tê-lo, pois não existe nada em seu espaço de endereçamento capaz de colidir com ela. É óbvio que um segmento pode ser totalmente preenchido, mas geralmente os segmentos são grandes e esse tipo de ocorrência é raro. Para especificar um endereço nessa memória segmentada e bidimensional, o programa deve fornecer um endereço composto de duas partes: um número do segmento e um endereço dentro do segmento. A Figura 3.30 ilustra uma memória segmentada sendo usada pelas tabelas do compilador discutidas anteriormente. Cinco segmentos independentes são mostrados aqui.

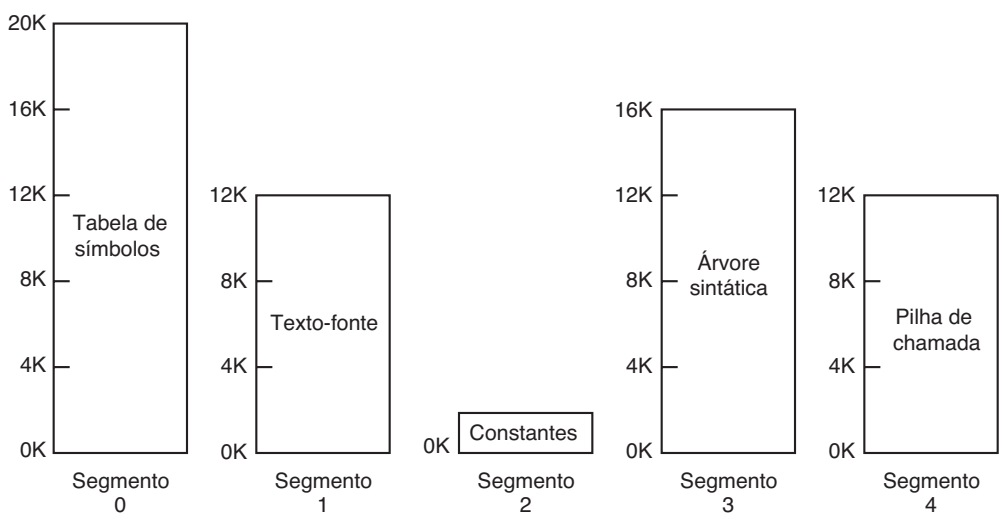

Figura 3.30 Uma memória segmentada permite que cada tabela cresça ou encolha de modo independente das outras tabelas.

É preciso enfatizar que um segmento é uma entidade lógica, que o programador conhece e usa como entidade lógica. Um segmento pode conter uma rotina, um arranjo, uma pilha ou um conjunto de variáveis escalares, mas em geral ele não apresenta uma mistura de diferentes tipos.

Uma memória segmentada tem outras vantagens além de simplificar o tratamento das estruturas de dados que estão crescendo ou se reduzindo. Se cada rotina ocupa um segmento separado, com o endereço 0 como endereço inicial, a ligação das rotinas compiladas separadamente é extremamente simplificada. Após todas as rotinas constituintes de um programa terem sido compiladas e ligadas, uma chamada para uma rotina no segmento *n* usará o endereço de duas partes (*n*, 0) para endereçar a palavra 0 (o ponto de entrada).

Se a rotina no segmento *n* é posteriormente modificada e recompilada, nenhuma outra rotina precisa ser alterada (pois nenhum endereço inicial foi modificado), mesmo se a nova versão for maior do que a primeira. Com uma memória unidimensional, as rotinas são fortemente empacotadas próximas umas às outras, sem qualquer espaço de endereçamento entre elas. Consequentemente, a variação do tamanho de uma rotina pode afetar os endereços iniciais das outras rotinas (não relacionadas). Isso, por sua vez, requer a modificação de todas as rotinas que fazem chamadas às rotinas que foram movidas, a fim de atualizar seus novos endereços iniciais. Se um programa contém centenas de rotinas, esse processo pode ser dispendioso.

A segmentação também facilita o compartilhamento de rotinas ou dados entre vários processos. Um exemplo comum é a biblioteca compartilhada. Muitas vezes, as estações de trabalho modernas, que executam sistemas avançados de janelas, têm bibliotecas gráficas extremamente grandes compiladas em quase todos os programas. Em um sistema segmentado, a biblioteca gráfica pode ser colocada em um segmento e compartilhada entre vários processos, eliminando a necessidade de sua replicação em cada espaço de endereçamento de cada processo. Apesar de ser possível haver bibliotecas compartilhadas em sistemas de paginação pura, essa situação é muito mais complicada. Portanto, esses sistemas preferem simular a segmentação.

Como cada segmento forma uma entidade lógica da qual o programador está ciente — como uma rotina, um arranjo ou uma pilha —, diferentes segmentos podem possuir diferentes tipos de proteção. Um segmento de rotina pode ser especificado como somente para execução, o que proíbe tentativas de leitura ou escrita nele. Um arranjo de ponto flutuante pode ser especificado como leitura/escrita, mas não de execução, e, assim, as tentativas de execução serão capturadas. Essa proteção é útil na captura de erros de programação.

É preciso compreender por que a proteção faz sentido em uma memória segmentada, mas não em uma memória unidimensional paginada: em uma memória segmentada, o usuário está ciente do que existe em cada segmento. Normalmente, um segmento não conteria uma rotina e uma pilha, por exemplo, mas ou um ou outro. Visto que cada segmento contém somente um tipo de objeto, o segmento pode ter a proteção apropriada para aquele tipo específico. A paginação e a segmentação são comparadas na Tabela 3.3.

De certo modo, os conteúdos de uma página são acidentais. O programador desconhece o fato de que a paginação está ocorrendo. Embora fosse possível colocar alguns bits em cada entrada da tabela de páginas, a fim de especificar o acesso permitido, para utilizar essa propriedade o programador deveria manter o controle de onde estão

Consideração	Paginação	Segmentação
O programador precisa saber que essa técnica está sendo usada?	Não	Sim
Há quantos espaços de endereçamento linear?	1	Muitos
O espaço de endereçamento total pode superar o tamanho da memória física?	Sim	Sim
Rotinas e dados podem ser distinguidos e protegidos separadamente?	Não	Sim
As tabelas cujo tamanho flutua podem ser facilmente acomodadas?	Não	Sim
O compartilhamento de rotinas entre os usuários é facilitado?	Não	Sim
Por que essa técnica foi inventada?	Para obter um grande espaço de endereçamento linear sem a necessidade de comprar mais memória física	Para permitir que programas e dados sejam divididos em espaços de endereçamento logicamente independentes e para auxiliar o compartilhamento e a proteção

Tabela 3.3 Comparação entre paginação e segmentação.

os limites da página em seu espaço de endereçamento. A paginação foi inventada, precisamente, para eliminar esse tipo de administração. Como o usuário de uma memória segmentada tem a ilusão de que todos os segmentos estão na memória principal durante todo o tempo — isto é, ele é capaz de endereçá-los como se eles aí estivessem —, ele pode proteger cada segmento separadamente, sem ter de se preocupar com a administração de sua sobreposição.

3.7.1 | Implementação de segmentação pura

A implementação da segmentação difere da paginação em um ponto essencial: as páginas têm tamanhos fixos e os segmentos, não. A Figura 3.31(a) mostra um exemplo de memória física que contém inicialmente cinco segmentos. Agora imagine o que ocorre se o segmento 1 é removido e o segmento 7, menor, é colocado em seu lugar. Chegaremos à configuração de memória da Figura 3.31(b). Entre o segmento 7 e o segmento 2 existe uma área não usada — isto é, uma lacuna. O segmento 4 é substituído pelo segmento 5, como na Figura 3.31(c), e o segmento 3 é substituído pelo segmento 6, como ilustra a Figura 3.31(d). Após o sistema ter executado por um tempo, a memória estará dividida em regiões, algumas com segmentos e outras com lacunas. Esse fenômeno, chamado de **fragmentação externa** (ou **checkerboarding**), desperdiça memória nas lacunas. Isso pode ser sanado com o uso de compactação, como mostra a Figura 3.31(e).

3.7.2 | Segmentação com paginação: MULTICS

Se os segmentos são grandes, talvez seja inconveniente — ou mesmo impossível — mantê-los na memória em sua totalidade. Isso gera a ideia de paginação dos segmentos, de modo que somente as páginas realmente necessárias tenham de estar na memória. Vários sistemas importantes têm suportado segmentos paginados. Nesta seção descreveremos o primeiro deles: o MULTICS. Na próxima seção, trataremos de um mais recente: o Pentium, da Intel.

O MULTICS foi executado em máquinas Honeywell 6000 e seus descendentes e provia de cada programa uma memória virtual de até 2^{18} segmentos (mais do que 250 mil), na qual cada segmento poderia ter até 65.536 (36 bits) palavras de comprimento. Para implementá-lo, os projetistas do MULTICS optaram por tratar cada segmento como uma memória virtual e, assim, paginá-lo, combinando as vantagens da paginação (tamanho uniforme de páginas sem necessidade de manter o segmento todo na memória no caso de somente parte dele estar sendo usado) com as vantagens da segmentação (facilidade de programação, modularidade, proteção e compartilhamento).

Cada programa no MULTICS tem uma tabela de segmentos, com um descritor para cada segmento. Visto que potencialmente existe mais do que um quarto de milhão de entradas na tabela, a tabela de segmentos forma, por si só, um segmento, que também é paginado. Um descritor de segmento informa se o segmento se encontra ou não na memória principal. Se qualquer parte do segmento está na memória, considera-se que o segmento está na memória e que sua tabela de páginas também estará. Se o segmento está na memória, seu descritor contém um ponteiro de 18 bits para sua tabela de páginas (veja a Figura 3.32(a)). Como o endereçamento físico é de 24 bits e as páginas são alinhadas em limites de 64 bytes (o que implica que os 6 bits de mais baixa ordem dos endereços das páginas sejam 000000), apenas 18 bits são necessários no descritor para armazenar um endereço da tabela de páginas. O descritor também pode conter o tamanho do segmento, os bits de proteção e alguns outros itens. A Figura 3.32(b) ilustra um descritor de segmento do MULTICS. O endereço do segmento na memória secundária não está no descritor do seg-

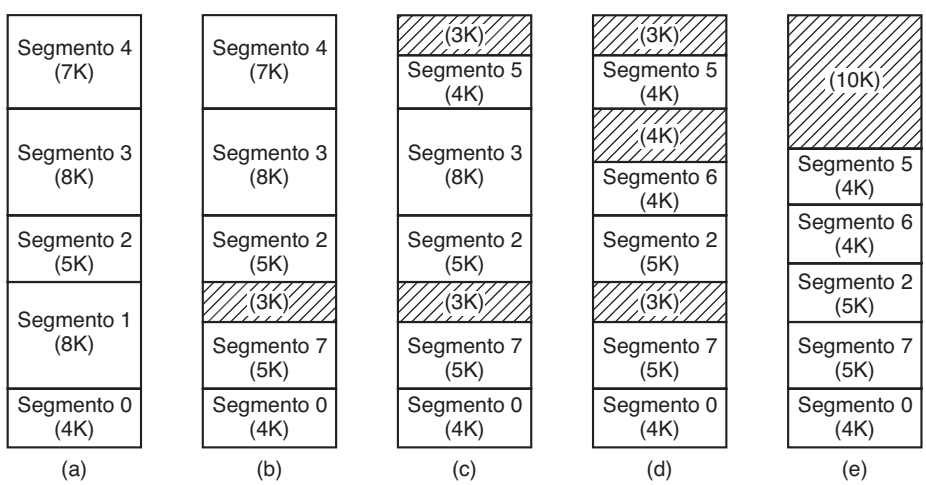

Figura 3.31 (a)–(d) Desenvolvimento de checkerboarding. (e) Remoção do checkerboarding por compactação.

mento, mas em outra tabela usada pelo manipulador de faltas de segmento.

Cada segmento é um espaço de endereçamento virtual comum, sendo também paginado do mesmo modo que a memória paginada não segmentada descrita anteriormente neste capítulo. O tamanho de uma página normal é 1.024 palavras (embora alguns poucos segmentos pequenos usados pelo próprio MULTICS não sejam paginados ou sejam paginados em unidades de 64 palavras para economizar memória física).

Um endereço no MULTICS consiste de duas partes: o segmento e o endereço dentro do segmento. O endereço dentro do segmento é, por sua vez, dividido em duas partes: um número de página e uma palavra dentro da página, como mostra a Figura 3.33. Quando ocorre uma referência à memória, o seguinte algoritmo é executado:

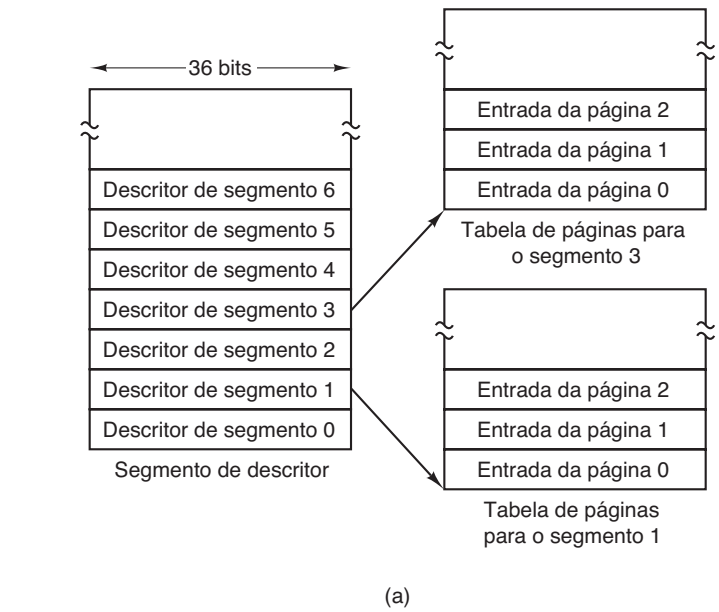

Figura 3.32 Memória virtual MULTICS. (a) O descritor de segmento aponta para tabelas de páginas. (b) Um descritor de segmento. Os números são o comprimento do campo.

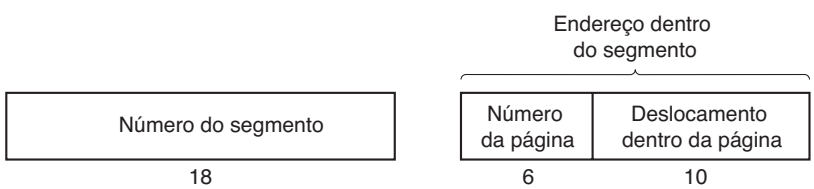

Figura 3.33 Um endereço virtual MULTICS de 34 bits.

1. O número do segmento é usado para encontrar o descritor do segmento.
2. Faz-se uma verificação para ver se a tabela de páginas do segmento está na memória. Se a tabela de páginas está na memória, ela é localizada. Do contrário, ocorre uma falta de segmento. Se existe uma violação de proteção, ocorre uma interrupção.
3. A entrada da tabela de páginas para a página virtual requisitada é examinada. Se a página não está na memória, ocorre uma falta de página. Se ela está na memória, o endereço da memória principal do início da página é extraído da entrada da tabela de páginas.
4. O deslocamento é adicionado ao início da página a fim de gerar o endereço da memória principal onde a palavra está localizada.
5. A leitura ou a escrita pode finalmente ser feita.

Esse processo é ilustrado na Figura 3.34. Para simplificar, foi omitido o fato de que o segmento de descritores de segmento é paginada. O que realmente ocorre é que um registrador (registrador-base do descritor) é usado para localizar a tabela de páginas do segmento de descritores, a qual aponta para as páginas do segmento de descritores. Uma vez encontrado o descritor para o segmento necessário, o endereçamento prossegue como mostrado na Figura 3.34.

Como você já deve ter percebido, se o algoritmo anterior fosse de fato utilizado pelo sistema operacional para cada instrução, os programas não executariam muito rapidamente. Na realidade, o hardware do MULTICS contém uma TLB de alta velocidade, com 16 palavras, que pode pesquisar todas as suas entradas em paralelo para uma dada chave. Essa TLB é ilustrada na Figura 3.35. Quando um endereço é apresentado para o computador, o hard-

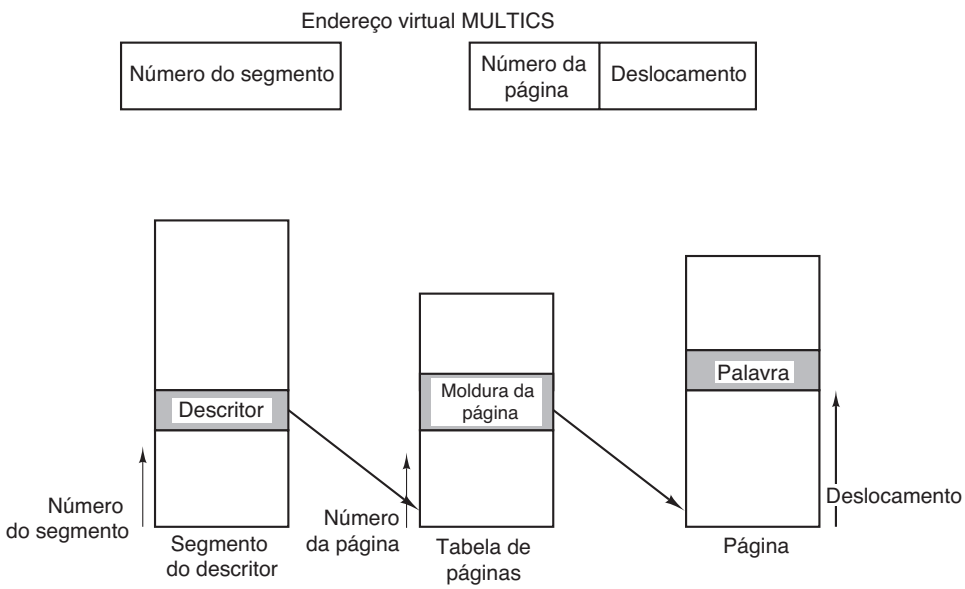

Figura 3.34 Conversão de um endereço de duas partes do MULTICS em um endereço de memória principal.

Campo de comparação					Esta entrada é usada?
Número do segmento	Página virtual	Moldura da página	Proteção	Idade	
4	1	7	Leitura/escrita	13	1
6	0	2	Somente leitura	10	1
12	3	1	Leitura/escrita	2	1
					0
2	1	0	Somente execução	7	1
2	2	12	Somente execução	9	1

Figura 3.35 Versão simplificada da TLB do MULTICS. A existência de dois tamanhos de páginas torna a TLB real mais complicada.

ware de endereçamento verifica inicialmente se o endereço virtual está na TLB. Em caso afirmativo, ele obtém o número da moldura de página diretamente da TLB e forma o endereço real da palavra referenciada sem precisar consultar o segmento de descritores ou a tabela de páginas.

Os endereços das 16 páginas mais recentemente referenciadas são mantidos na TLB. Os programas que tiverem um conjunto de trabalho menor do que o tamanho da TLB atingirão o equilíbrio com os endereços de todo o conjunto de trabalho na TLB e, portanto, executarão com eficiência. Se a página não está na TLB, as tabelas de descritores e de páginas são realmente acessadas para encontrar o endereço da moldura de página e, assim, a TLB é atualizada para incluir essa página, sendo descartada a página menos usada recentemente. O campo *Idade* mantém o controle de qual entrada é a menos usada recentemente. A TLB é empregada para comparar paralelamente os números de segmento e de página de todas as entradas.

3.7.3 Segmentação com paginação: o Pentium Intel

Em muitos aspectos, a memória virtual no Pentium se parece com a do MULTICS, incluindo a presença de segmentação e paginação. Enquanto o MULTICS tem 256 K segmentos independentes, cada um com até 64 K palavras de 36 bits, o Pentium possui 16 K segmentos independentes, cada um com até um bilhão de palavras de 32 bits. Embora o Pentium trabalhe com poucos segmentos, o tamanho do segmento maior é muito mais importante, pois poucos programas precisam de mais do que mil segmentos, mas muitos programas requerem segmentos grandes.

O coração da memória virtual do Pentium consiste em duas tabelas, a **LDT** (*local descriptor table* — tabela de descritores local) e a **GDT** (*global descriptor table* — tabela de descritores global). Cada programa tem sua própria LDT, mas existe uma única GDT, compartilhada por todos os programas do computador. A LDT descreve os segmentos locais a cada programa, incluindo código, dados, pilha e assim por diante, ao passo que a GDT descreve os segmentos do sistema, inclusive o próprio sistema operacional.

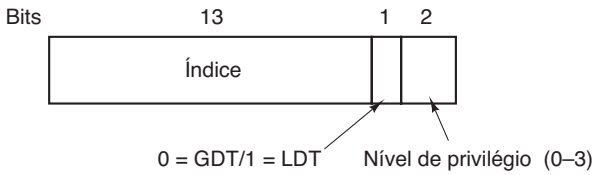

Figura 3.36 Seletor do Pentium.

Para acessar um segmento, um programa Pentium carrega primeiro um seletor para aquele segmento dentro de um dos seis registradores de segmento da máquina. Durante a execução, o registrador CS guarda o seletor para cada segmento de código (*code segment*) e o registrador DS guarda o seletor para o segmento de dados (*data segment*). Os outros registradores de segmentos são menos importantes. Cada seletor é um número de 16 bits, como mostra a Figura 3.36.

Um dos bits do seletor informa se determinado segmento é local ou global (isto é, se ele está na LDT ou na GDT). Treze outros bits especificam o número da entrada na LDT ou na GDT, de modo que cada tabela pode conter até 8 K descritores de segmentos. Os outros dois bits (0 e 1) relacionam-se à proteção e serão abordados posteriormente. O descritor nulo (0) é proibido. Ele pode ser seguramente carregado para um registrador de segmento para indicar que não está atualmente disponível. Ele causa uma interrupção quando usado.

No momento em que um seletor é carregado para um registrador de segmento, o descritor correspondente é buscado da LDT ou na GDT e armazenado em registradores internos do microprograma, de modo que possa ser acessado rapidamente. Um descritor é constituído de 8 bytes, incluindo o endereço-base do segmento, o tamanho e outras informações — como pode ser observado na Figura 3.37.

O formato do seletor foi cuidadosamente escolhido para facilitar a localização do descritor de segmento. Primeiro, seleciona-se a LDT ou a GDT, com base no bit 2 do seletor. Então, o seletor é copiado para um registrador provisório interno e os 3 bits de mais baixa ordem são marcados com 0. Por fim, o endereço inicial da LDT ou da GDT

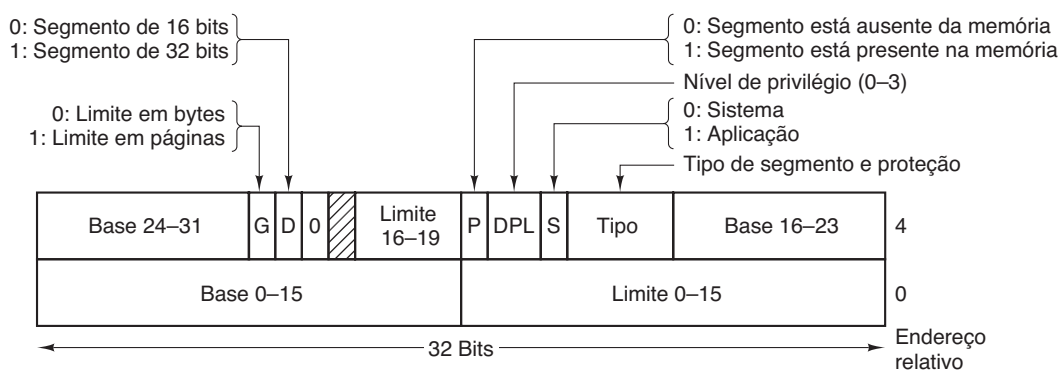

Figura 3.37 Descritor de segmento de código do Pentium. Os segmentos de dados são ligeiramente diferentes.

é adicionado a esse registrador, resultando em um ponteiro direto para o descritor do segmento pretendido. Por exemplo, o seletor 72 refere-se à entrada 9 na GDT, a qual está localizada no endereço GDT + 72.

Vamos traçar os passos pelos quais um par (seletor, deslocamento) é convertido para um endereço físico. Assim que o microprograma é informado sobre qual registrador de segmento está sendo usado, ele pode encontrar, em seus registradores internos, o descritor completo correspondente àquele seletor. Uma interrupção ocorre se o segmento não existe (seletor 0) ou se está atualmente em disco.

O hardware usa então o campo *Limite* para verificar se o deslocamento está além do final do segmento, caso em que também ocorre uma interrupção. Logicamente, deveria haver apenas um campo de 32 bits no descritor informando o tamanho do segmento, mas existem somente 20 bits disponíveis, de modo que então se emprega outro esquema. Se o bit G (granularidade) é 0, o campo *Limite* indica o tamanho do segmento exato, até 1 MB. Se ele é 1, o bit G indica o tamanho do segmento em páginas em vez de bytes. O tamanho da página do Pentium é fixado em 4 KB; portanto, 20 bits são suficientes para segmentos de até 2^{32} bytes (4 GB).

Presumindo que o segmento encontra-se na memória e o deslocamento está dentro do alcance, o Pentium então adiciona o campo *Base* de 32 bits do descritor ao deslocamento para formar o que é chamado de **endereço linear** — como mostra a Figura 3.38. O campo *Base* é dividido em três partes e espalhado pelo descritor para compatibilidade com o 286, no qual o campo *Base* é de somente 24 bits. Como consequência, ele permite que cada segmento inicie em um local arbitrário dentro do espaço de endereçamento linear de 32 bits.

Se a paginação é desabilitada (por um bit no registrador de controle global), o endereço linear é interpretado como o endereço físico e enviado para a memória para leitura ou escrita. Assim, com a paginação desabilitada, temos um esquema de segmentação pura, em que cada endereço de base do segmento é dado em seu descritor. É permitido aos segmentos se sobreporem de modo acidental — provavelmente porque implicaria preocupação e tempo demais para verificar se eles estão disjuntos.

Por outro lado, se a paginação está habilitada, o endereço linear é interpretado como um endereço virtual e mapeado sobre o endereço físico a partir do emprego de tabelas de páginas, da mesma maneira que nos exemplos anteriores. A única complicação real é que, com um endereço virtual de 32 bits e uma página de 4 KB, um segmento pode conter até um milhão de páginas, de modo que um mapeamento de dois níveis é usado para reduzir o tamanho da tabela de páginas para segmentos pequenos.

Cada programa em execução tem um **diretório de páginas**, que consiste em 1.024 entradas de 32 bits. Ele está localizado em um endereço apontado por um registrador global. Cada entrada nesse diretório aponta para uma tabela de páginas que também contém 1.024 entradas de 32 bits. O esquema é mostrado na Figura 3.39.

Na Figura 3.39(a), vemos um endereço linear dividido em três campos: *Dir*, *Página* e *Deslocamento*. O campo *Dir* é usado para indexar dentro do diretório de páginas a fim de localizar um ponteiro para a tabela de páginas correta. O campo *Página* é, então, usado como um índice dentro da tabela de páginas para encontrar o endereço físico da moldura de página. Por fim, o campo *Deslocamento* é adicionado ao endereço da moldura de página com o intuito de obter o endereço físico do byte ou da palavra necessária.

Cada uma das entradas das tabelas de páginas é de 32 bits, 20 dos quais contêm um número de moldura de página. Os bits restantes carregam informações de acesso e modificação, marcados pelo hardware para auxiliar o sistema operacional, bits de proteção e outros bits úteis.

Cada tabela de páginas tem entradas para 1.024 molduras de página de 4 KB, de modo que uma única tabela de páginas gerencia 4 megabytes de memória. Um segmento menor do que 4 M terá um diretório de páginas com uma única entrada: um ponteiro para sua única tabela de páginas. Dessa maneira, o custo para segmentos pequenos é de somente duas páginas — em vez dos milhões de páginas que seriam necessários em uma tabela de páginas de um nível.

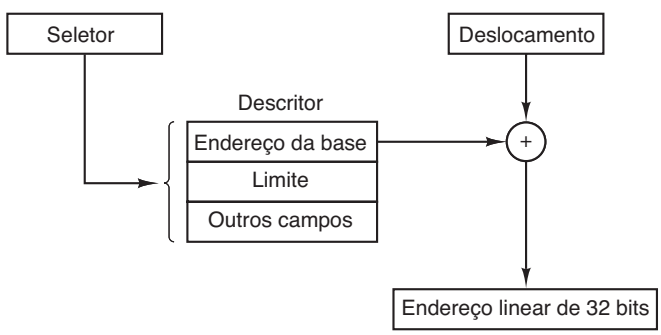

Figura 3.38 Conversão de um par (de seletores, deslocamentos) em um endereço linear.

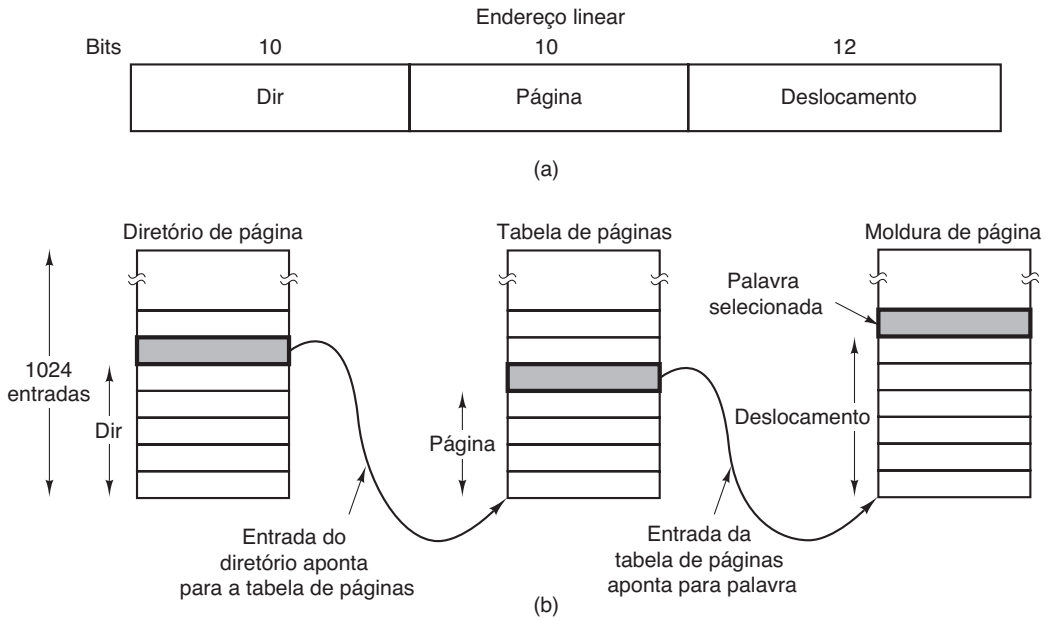

Figura 3.39 Mapeamento de um endereço linear em um endereço físico.

Para evitar referências repetidas à memória, o Pentium, como o MULTICS, tem uma pequena TLB, que mapeia diretamente a maioria das combinações *Dir-Página* mais usadas recentemente em endereços físicos da moldura de página. Somente quando a combinação atual não está presente na TLB, o mecanismo da Figura 3.39 é de fato executado e a TLB é atualizada. Enquanto as faltas na TLB são raras, o desempenho é bom.

Vale a pena notar que o modelo citado é válido para a aplicação que não precisa de segmentação e que é satisfeita com um espaço de endereçamento único paginado de 32 bits. Todos os registradores de segmento podem ser estabelecidos com o mesmo seletor, cujo descritor tem *Base* = 0 e *Limite* configurado no máximo. Então, o deslocamento da instrução será o endereço linear, com um único espaço de endereçamento usado — consequentemente, paginação normal. De fato, todos os sistemas operacionais atuais para o Pentium trabalham assim. O OS/2 foi o único a usar o poder total da arquitetura MMU da Intel.

De modo geral, é preciso fazer justiça aos projetistas do Pentium. Sabendo que as técnicas de paginação, segmentação e segmentação paginada possuem objetivos conflitantes e que a manutenção da compatibilidade com o 286 dificulta ainda mais a implementação, o projeto resultante é surpreendentemente simples e claro, ainda mais se considerarmos que tudo isso é feito eficientemente.

Embora tenhamos, ainda que de modo breve, visto a arquitetura completa da memória virtual do Pentium, é importante dizer algo sobre a proteção, uma vez que se trata de um assunto intimamente relacionado com a memória virtual. Assim como o esquema de memória, o sistema de proteção do Pentium também é modelado de maneira parecida com o do MULTICS. O Pentium suporta quatro níveis de proteção, dos quais o nível 0 é o mais privilegiado e o nível 3, o menos. Eles são mostrados na Figura 3.40. Em cada instante, um programa em execução está em certo nível, indicado por um campo de 2 bits em sua PSW. Cada segmento no sistema também tem um nível.

Enquanto um programa se restringe a utilizar segmentos em seu próprio nível, tudo trabalha bem. As tentativas de acesso aos dados em um nível mais alto são permitidas, mas as tentativas de acesso aos dados em um nível mais baixo são ilegais e causam interrupções. As tentativas de chamadas às rotinas em um nível diferente (maior ou menor) são permitidas, mas de uma maneira cuidadosamente controlada. Para fazer uma chamada internível, a instrução CALL deve conter um seletor em vez de um endereço. Esse seletor define um descritor chamado de **porta de chamada**, que fornece o endereço da rotina a ser chamada. Assim não é possível saltar para o meio de um segmento de código arbitrário em um nível diferente. Somente pontos oficiais de entradas podem ser usados. Os conceitos de níveis de proteção e portas de chamadas foram pioneiramente usados no MULTICS, no qual eram vistos como **anéis de proteção**.

Um uso típico para esse mecanismo é sugerido na Figura 3.40. No nível 0, encontramos o núcleo do sistema operacional, o qual trata de E/S, gerenciamento de memória e outros assuntos críticos. No nível 1, o manipulador de chamadas ao sistema está presente. Os programas do usuário podem chamar rotinas nesse nível para que as chamadas de sistema sejam realizadas, mas somente

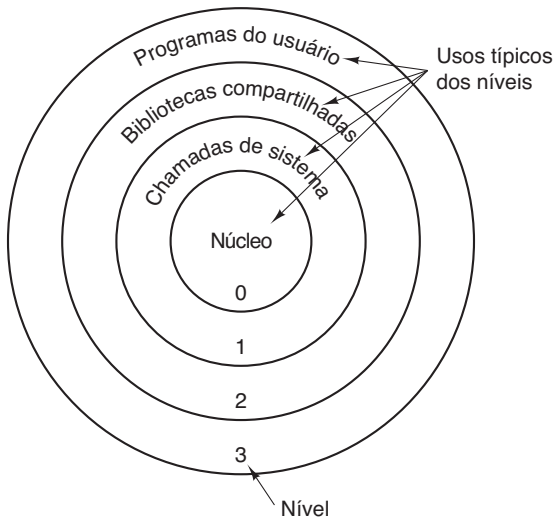

Figura 3.40 Proteção no Pentium.

uma lista de rotinas específicas e protegidas pode ser chamada. O nível 2 contém rotinas de bibliotecas, passíveis de serem compartilhadas entre muitos programas em execução. Os programas do usuário podem chamar essas rotinas e ler seus dados, mas estes não podem ser modificados. Por fim, os programas do usuário executam em nível 3, que apresenta a menor proteção.

Interrupções de software e de hardware empregam um mecanismo similar àquele das portas de chamadas. Elas também referenciam descritores, em vez de endereços absolutos, e os descritores apontam para rotinas específicas a serem executadas. O campo *Tipo* na Figura 3.37 diferencia segmentos de códigos, segmentos de dados e os vários tipos de portas.

3.8 Pesquisas em gerenciamento de memória

O gerenciamento de memória — especialmente os algoritmos de paginação — outrora foi uma área fértil de pesquisa, mas hoje em dia é escassa, pelo menos quanto a sistemas de propósito geral. A maioria dos sistemas reais tende a usar alguma variação do algoritmo do relógio, em razão da facilidade de implementação e efetividade relativa. No entanto, uma exceção recente é o reprojeto do sistema de memória virtual BSD 4.4 (Cranor e Parulkar, 1999).

De qualquer modo, ainda existem pesquisas em andamento relacionadas à paginação em novos tipos de sistemas. Por exemplo, telefones celulares e PDAs se transformaram em pequenos PCs, e muitos deles paginam a RAM para o 'disco', com a diferença de que um disco em um telefone celular é a memória flash, que tem propriedades diferentes das do disco magnético rotativo. Parte do trabalho recente é relatada por (In et al., 2007; Joo et al., 2006; Park et al., 2004a). Park et al. (2004b) também examinaram paginação por demanda consciente de energia em dispositivos móveis.

Além do mais, há pesquisas sobre a modelação do desempenho da paginação (Albers et al., 2002; Burton e Kelly, 2003; Cascaval et al., 2005; Panagiotou e Souza, 2006; Peserico, 2003). Interessa também o gerenciamento de memória para sistemas multimídia (Dasigenis et al., 2001; Hand, 1999) e sistemas de tempo real (Pizlo e Vitek, 2006).

3.9 Resumo

Neste capítulo examinamos o gerenciamento de memória. Vimos que os sistemas mais simples não realizam qualquer tipo de troca de processos entre a memória e o disco ou paginação. Uma vez que um programa é carregado para a memória, ele permanece nela até sua finalização. Alguns sistemas operacionais permitem somente um processo por vez na memória, enquanto outros suportam multiprogramação.

O próximo passo é a troca de processos entre a memória e o disco. Quando ela é usada, o sistema pode tratar processos em maior número do que a quantidade de memória de que dispõe e permitiria. Os processos para os quais não existe memória disponível são levados para o disco. O espaço livre na memória ou no disco é controlado com o uso de mapa de bits ou lista de lacunas.

Os computadores modernos muitas vezes apresentam alguma forma de memória virtual. Em sua configuração mais simples, o espaço de endereçamento de cada processo é dividido em blocos de tamanho uniforme chamados de páginas, que podem ser colocadas em qualquer moldura de página disponível na memória. Existem muitos algoritmos de substituição de página; dois dos melhores são o algoritmo do envelhecimento (*aging*) e o WSClock.

Para fazer com que os sistemas de paginação trabalhem bem, a escolha do algoritmo não é suficiente; é necessário também observar questões como a determinação do conjunto de trabalho, a política de alocação de memória e o tamanho da página.

A segmentação ajuda no tratamento de estruturas de dados que alteram seus tamanhos durante a execução e simplifica a ligação e o compartilhamento. Ela facilita, ainda, o provimento de proteção diferenciada para segmentos diferentes. Muitas vezes a segmentação e a paginação são combinadas para fornecer uma memória virtual bidimensional. O sistema MULTICS e o Pentium da Intel suportam segmentação e paginação.

Problemas

1. Na Figura 3.3, o registrador-base e o registrador-limite contêm o mesmo valor, 16.384. Isso é apenas um aciden-

te ou eles são sempre iguais? Se for apenas um acidente, por que eles são iguais nesse exemplo?

2. Um sistema de troca de processos elimina lacunas na memória via compactação. Ao supor uma distribuição aleatória de muitas lacunas e diversos segmentos de dados e um tempo de leitura/escrita de 10 ns para uma palavra de memória de 32 bits, quanto tempo ele levará para compactar 128 MB? Para simplificar, presuma que a palavra 0 é parte de uma lacuna e que a palavra da parte mais alta da memória contenha dados válidos.
3. Neste problema, você deve comparar o armazenamento necessário para manter o controle da memória disponível usando um mapa de bits *versus* uma lista encadeada. A memória de 128 MB é alocada em unidades de *n* bytes. Para a lista encadeada, presuma que a memória seja constituída de uma sequência alternada de segmentos e lacunas, cada um de 64 KB. Além disso, suponha que cada nó da lista encadeada precise de um endereçamento de memória de 32 bits — 16 bits para o comprimento e 16 bits para o campo *Próximo nó*. Quantos bytes de armazenamento são necessários para cada método? Qual é o melhor?
4. Considere um sistema de troca de processos entre a memória e o disco no qual a memória é constituída dos seguintes tamanhos de lacunas em ordem na memória: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB e 15 KB. Qual lacuna é tomada pelas solicitações sucessivas do segmento de:
   (a) 12 KB.
   (b) 10 KB.
   (c) 9 KB.

   para o *first fit*? Repita agora a questão para o *best fit*, o *worst fit* e o *next fit*.
5. Para cada um dos seguintes endereços virtuais decimais, calcule o número da página virtual e o deslocamento para uma página de 4 KB e para uma página de 8 KB: 20000, 32768, 60000.
6. O processador Intel 8086 não suporta memória virtual. Apesar disso, antigamente, algumas empresas venderam sistemas que continham uma CPU 8086 original e faziam paginação. Suponha como eles faziam isso. *Dica*: pense na localização lógica da MMU.
7. Considere o programa em C seguinte:

   ```
   int X[N];
   int step = M; // M é alguma constante predefinida
   for (int i=0; i < N; i += step) X[i] + 1;
   ```

   (a) Se esse programa for executado em uma máquina com um tamanho de página de 4 KB e uma TLB de entrada de 64 bits, que valores de M e N causarão uma ausência de página na TLB para cada execução do *laço* interno?
   (b) Sua resposta na parte (a) seria diferente se o *laço* fosse repetido muitas vezes? Explique.
8. A quantidade de espaço em disco que precisa estar disponível para armazenamento de página é relacionada com o número máximo de processos (*n*), o número de bytes no espaço de endereçamento virtual (*v*) e o número de bytes de RAM (*r*). Elabore uma expressão matemática para os requisitos de espaço de disco, considerando a pior das hipóteses. Até que ponto essa quantidade é realista?
9. Uma máquina tem um espaço de endereçamento de 32 bits e uma página de 8 KB. A tabela de páginas está totalmente em hardware, com uma palavra de 32 bits para cada entrada. Quando um processo tem início, a tabela de páginas é copiada para o hardware a partir da memória, no ritmo de uma palavra a cada 100 ns. Se cada processo executa durante 100 ms (incluindo o tempo para carregar a tabela de páginas), qual a fração do tempo de CPU que é dedicada ao carregamento das tabelas de páginas?
10. Suponha que uma máquina tenha endereços virtuais de 48 bits e endereços físicos de 32 bits.
    (a) Se as páginas são de 4 KB, quantas entradas estão na tabela de páginas se ela tiver apenas um único nível? Explique.
    (b) Suponha que esse mesmo sistema tenha uma TLB (*translation lookaside buffer* — tabela de tradução de endereços) com 32 entradas. Além disso, suponha que um programa contenha instruções que se encaixem em uma página e leiam sequencialmente elementos de números inteiros longos de um arranjo de milhares de páginas. O quanto a TLB será eficiente para esse caso?
11. Suponha que uma máquina tenha endereços virtuais de 38 bits e endereços físicos de 32 bits.
    (a) Qual é a principal vantagem de uma tabela de páginas em múltiplos níveis em relação a uma de nível único?
    (b) Com uma tabela de páginas de dois níveis, páginas de 16 KB e entradas de 4 bytes, quantos bits deveriam ser alocados para o campo da tabela no topo da página e quantos para o campo da tabela de páginas do próximo nível? Explique.
12. Um computador com um endereçamento de 32 bits usa uma tabela de páginas de dois níveis. Os endereços são quebrados em um campo de 9 bits para a tabela de páginas de nível 1, um campo de 11 bits para a tabela de páginas de nível 2 e um deslocamento. Qual o tamanho das páginas e quantas existem no espaço de endereçamento citado?
13. Suponha que um endereço virtual de 32 bits seja quebrado em quatro campos: *a*, *b*, *c* e *d*. Os três primeiros são usados para um sistema de tabela de páginas de três níveis. O quarto campo, *d*, é o deslocamento. O número de páginas depende dos tamanhos de todos os quatro campos? Se não, quais campos influenciam nessa questão e quais não?
14. Um determinado computador tem endereços virtuais de 32 bits e páginas de 4 KB. O programa e os dados, juntos, cabem na página de mais baixa ordem (0–4095). A pilha cabe na página de mais alta ordem. Quantas entradas são necessárias na tabela de páginas se a paginação tradicional (de um nível) é usada? E quantas entradas na tabela de páginas são necessárias para uma paginação de dois níveis, com 10 bits para cada parte?

15. Um computador cujos processos têm 1.024 páginas em seus espaços de endereçamento mantém suas tabelas na memória. A sobrecarga necessária para a leitura de uma palavra da tabela de páginas é de 5 ns. Para reduzir esse custo, o computador tem uma TLB, que contém 32 pares (página virtual, moldura de página) e assim pode fazer uma varredura nas entradas em 1 ns. Qual é a taxa de acerto necessária para reduzir a sobrecarga média para 2 ns?
16. A TLB no VAX não contém o bit *R*. Por quê?
17. Como pode uma memória associativa, necessária para uma TLB, ser implementada em hardware, e quais são as implicações quanto à capacidade de expansão para esse projeto?
18. Uma máquina tem um endereçamento virtual de 48 bits e um endereçamento físico de 32 bits. As páginas são de 8 KB. Quantas entradas são necessárias para a tabela de páginas?
19. Um computador com uma página de 8 KB, uma memória de 256 KB e um espaço de endereçamento de 64 GB usa uma tabela de páginas invertidas para implementar sua memória virtual. Qual tamanho deve ter a tabela de espalhamento para garantir um tamanho médio da cadeia de espalhamento menor que 1? Presuma que o tamanho da tabela de espalhamento é uma potência de dois.
20. Um estudante da disciplina projeto de compiladores propõe ao professor um projeto de um compilador que produzirá uma lista de referências de páginas, a qual poderá ser usada para implementar o algoritmo de substituição ótimo. Isso é possível? Por quê? Existe algo que poderia ser feito para melhorar a eficiência da paginação em tempo de execução?
21. Suponha que o fluxo de referência de página virtual contenha repetições de sequências longas de referências de páginas seguidas ocasionalmente por uma referência de página aleatória. Por exemplo, a sequência: 0, 1, ..., 511, 431, 0, 1, ..., 511, 332, 0, 1, ... consiste em repetições da sequência 0, 1, ..., 511 seguidas por uma referência aleatória às páginas 431 e 332.
    (a) Por que os algoritmos de substituição de página (LRU, FIFO, relógio) não serão eficazes no tratamento da carga de trabalho para uma alocação de página que seja menor que o comprimento da sequência?
    (b) Se nesse programa foram alocadas 500 molduras de páginas, descreva um método de substituição de página que tenha um desempenho melhor que os algoritmos LRU, FIFO ou relógio.
22. Se o algoritmo de substituição FIFO é usado com quatro molduras de página e oito páginas virtuais, quantas faltas de página ocorrerão com a cadeia de referências 0172327103 se os quatro quadros estão inicialmente vazios? Agora repita este problema para LRU.
23. Observe a sequência de páginas da Figura 3.14(b). Suponha que os bits *R*, para as páginas de *B* a *A*, sejam 11011011, respectivamente. Quais páginas serão removidas pelo algoritmo segunda chance?
24. Um computador pequeno tem quatro molduras de página. No primeiro tique de relógio, os bits *R* são 0111 (página 0 é 0, as demais são 1). Nos tiques subsequentes os valores são 1011, 1010, 1101, 0010, 1010, 1100 e 0001. Se o algoritmo do envelhecimento (*aging*) é usado com um contador de 8 bits, quais os valores dos quatro contadores após o último tique?
25. Dê um exemplo simples de uma sequência de referência de páginas onde a primeira página selecionada para substituição seja diferente para os algoritmos de substituição de página relógio e LRU. Suponha que em um processo sejam alocadas três molduras e que a cadeia de referência contenha números de página do conjunto 0, 1, 2, 3.
26. No algoritmo WSClock da Figura 3.20(c), o ponteiro do relógio aponta para a página com $R = 0$. Se $\tau = 400$, essa página será removida? O que acontecerá se $\tau = 1000$?
27. Quanto tempo leva para carregar um programa de 64 KB de um disco cujo tempo de posicionamento médio seja de 10 ms, o tempo de rotação seja de 10 ms e cujas trilhas contenham 32 KB, considerando:
    (a) tamanho de página de 2 KB?
    (b) tamanho de página de 4 KB?

    As páginas são espalhadas aleatoriamente ao redor do disco, e o número de cilindros é tão grande que a probabilidade de duas páginas estarem no mesmo cilindro é desprezível.
28. Um computador tem quatro molduras de página. O tempo de carregamento de página na memória, o instante do último acesso e os bits *R* e *M* para cada página são mostrados a seguir (os tempos estão em tiques de relógio):

Página	Carregado	Última ref.	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

    (a) Qual página será trocada pelo NRU?
    (b) Qual página será trocada pelo FIFO?
    (c) Qual página será trocada pelo LRU?
    (d) Qual página será trocada pelo segunda chance?
29. Considere o seguinte arranjo bidimensional:

    ```
    int X[64][64];
    ```

    Suponha que um sistema tenha quatro molduras de página e que cada moldura seja de 128 palavras (um número inteiro ocupa uma palavra). Os programas que manipulam o arranjo *X* se ajustam exatamente a uma página e sempre ocupam a página 0. Os dados são trocados para dentro e para fora das outras três molduras. O arranjo *X* é armazenado segundo a ordem da fila maior (isto é, *X*[0][1] segue *X*[0][0] na memória). Qual dos dois fragmentos de código mostrados a seguir gerará o menor número de faltas de página? Explique e calcule o número total de faltas de página.

    Fragmento A

    ```
    for (int j = 0; j < 64; j++)
    for (int i = 0; i < 64; i++) X[i][j] = 0;
    ```

Fragmento B

```
for (int i = 0; i < 64; i++)
for (int j = 0; j < 64; j++) X[i][j] = 0;
```

30. Uma das primeiras máquinas de compartilhamento de tempo, o PDP-1, tinha uma memória de 4 K palavras de 18 bits. Ele mantinha um processo por vez na memória. Quando o escalonador decidia executar outro processo, o processo da memória era escrito em um disco (tambor) de páginas, com 4 K palavras de 18 bits, ao redor da circunferência do disco. O disco poderia começar a escrever (ou ler) em qualquer palavra, em vez de somente na palavra 0. Por que você acha que esse disco foi escolhido?

31. Um computador provê, a cada processo, 65.536 bytes de espaço de endereçamento dividido em páginas de 4.096 bytes. Um determinado programa tem um tamanho de texto de 32.768 bytes, um tamanho de dados de 16.386 bytes e um tamanho de pilha de 15.870 bytes. Esse programa se encaixa no referido espaço de endereçamento? Se o tamanho da página fosse de 521 bytes, ele se encaixaria? (Lembre-se de que uma página não pode conter partes de dois segmentos diferentes.)

32. Uma página pode estar em dois conjuntos de trabalho (*working sets*) ao mesmo tempo? Explique.

33. Tem-se observado que o número de instruções executadas entre faltas de página é diretamente proporcional ao número de molduras de página alocadas para um programa. Se a memória disponível for duplicada, o intervalo médio entre faltas de página será duplicado. Suponha que uma instrução normal leve um microssegundo, mas, se uma falta de página ocorrer, ela levará 2.001 microssegundos (isto é, 2 ms para tratar a falta). Se um programa leva 60 s para executar — período em que ele terá 15 mil faltas de página —, quanto tempo ele levaria para executar se existissem duas vezes mais memória disponível?

34. Um grupo de projetistas de sistemas operacionais da empresa Frugal Computer está tentando encontrar meios de reduzir a quantidade de área de troca necessária em seus sistemas operacionais. O líder do grupo sugeriu não perder tempo, de modo algum, com o salvamento do texto do programa na área de troca, mas, simplesmente, paginá-lo diretamente do arquivo binário quando necessário. Se é que isso é possível, sob quais condições essa ideia funciona para o código do programa? E sob quais condições ela funciona para os dados?

35. Uma instrução de linguagem de máquina para carregar uma palavra de 32 bits para dentro de um registrador contém o endereço de 32 bits da própria palavra a ser carregada. Qual é o número máximo de faltas de página que essa instrução pode causar?

36. Quando a segmentação e a paginação são usadas em conjunto, como no MULTICS, primeiro o descritor de segmentos deve ser procurado e, então, o descritor da página. A TLB também trabalha dessa maneira, com dois níveis de procura?

37. Consideremos um programa que tenha os dois segmentos mostrados a seguir, consistindo de instruções no segmento 0 e de dados de leitura/escrita no segmento 1. O segmento 0 tem proteção leitura/execução e o segmento 1 tem proteção leitura/escrita. O sistema de memória é um sistema de memória virtual paginado por demanda com endereços virtuais que têm um número de página de 4 bits e um deslocamento de 10 bits. As tabelas de páginas e proteção são as seguintes (todos os números na tabela são decimais):

Segmento 0		**Segmento 1**	
Leitura/execução		**Leitura/escrita**	
Página virtual #	**Moldura da página #**	**Página virtual #**	**Moldura da página #**
0	2	0	Em disco
1	Em disco	1	14
2	11	2	9
3	5	3	6
4	Em disco	4	Em disco
5	Em disco	5	13
6	4	6	8
7	3	7	12

Para cada um dos seguintes casos, elas dão o endereço de memória real (efetiva) que resulta da tradução de endereço dinâmica ou identificam o tipo de erro que ocorre (seja erro de página ou de proteção).

(a) Buscar do segmento 1, página 1, deslocamento 3.

(b) Armazenar no segmento 0, página 0, deslocamento 16.

(c) Buscar do segmento 1, página 4, deslocamento 28.

(d) Saltar para localização no segmento 1, página 3, deslocamento 32.

38. Você consegue imaginar alguma situação em que dar suporte à memória virtual seria uma má ideia e o que se ganha quando não é necessário o suporte de memória virtual? Explique.

39. Desenhe um histograma e calcule a média e a mediana dos tamanhos de arquivos binários executáveis em um computador a que você tem acesso. Em um sistema Windows, olhe todos os arquivos .exe e .dll; em um sistema UNIX, verifique todos os arquivos executáveis no */bin*, */usr/bin* e */local/bin* que não sejam scripts (ou use o utilitário *file* para encontrar todos os executáveis). Determine o tamanho ótimo da página para esse computador, levando em conta apenas o código (nenhum dado). Considere a fragmentação interna e o tamanho da tabela de páginas, fazendo uma suposição razoável sobre o tamanho da entrada na tabela de páginas. Presuma que todos os progra-

mas podem ser executados com igual probabilidade e que, portanto, é possível atribuir-lhes o mesmo peso.

40. Programas pequenos para MS-DOS podem ser compilados como arquivos *.COM*. Esses arquivos são sempre carregados no endereço 0x100 em um segmento único de memória que é usado para código, dados e pilha. As instruções que transferem o controle da execução, como JMP e CALL, ou que acessam dados estáticos, de endereços fixos, têm o endereço compilado no código-objeto. Escreva um programa capaz de realocar esse arquivo de programa para executar em um endereço arbitrário. Seu programa deve varrer o código e procurar códigos-objeto de instruções que referenciam endereços fixos de memória e então modificar os endereços que apontam para as posições de memória dentro da faixa a ser realocada. Você pode encontrar os códigos-objeto em um texto de linguagem de programação assembly. Note que fazer isso com perfeição, sem utilizar informação adicional, em geral é impossível, pois algumas palavras de dados podem ter valores que imitem códigos-objeto de instruções.

41. Escreva um programa que simule um sistema de paginação usando o algoritmo de envelhecimento. O número de molduras de página é um parâmetro. A sequência de referências de páginas deve ser lida de um arquivo. Para um dado arquivo de entrada, represente o número de faltas de página por 1.000 referências de memória como função do número de molduras de página disponíveis.

42. Escreva um programa que demonstre o efeito de ausências de página na TLB sobre o tempo de acesso à memória efetivo medindo o tempo por acesso necessário para percorrer um arranjo grande.

(a) Explique os principais conceitos por trás do programa e descreva o que você espera que a saída mostre a alguma arquitetura de memória virtual prática.

(b) Execute o programa em algum computador e explique o quanto os dados se ajustam a suas expectativas.

(c) Repita a parte (b), mas para um computador mais antigo, com uma arquitetura diferente, e explique as principais diferenças na saída.

43. Escreva um programa que demonstre a diferença entre o uso de uma política de substituição de página local e uma global para o caso simples de dois processos. Você precisará de uma rotina que possa gerar uma cadeia de referência de página baseada em um modelo estatístico. Esse modelo tem *N* estados numerados de 0 a N-1, representando cada uma das referências de página possíveis, e uma probabilidade p_1 associada com cada estado *i*, representando a chance de que a próxima referência seja à mesma página. Em outras circunstâncias, a próxima referência de página será uma das outras páginas com igual probabilidade.

(a) Demonstre que a rotina de geração de cadeia de caracteres de referência de página se comporta apropriadamente para algum *N* com valor pequeno.

(b) Calcule a taxa de falta de página para um pequeno exemplo no qual há um processo e um número fixo de molduras de página. Explique por que o comportamento é correto.

(c) Repita a parte (b) com dois processos com sequências de referência de página independentes e duas vezes mais molduras de página que na parte (b).

(d) Repita a parte (c), mas usando uma política global em vez de uma local. Além disso, compare a taxa de falta de página por processo com a do método de política local.