CAPÍTULO 3

GERENCIAMENTO DE MEMÓRIA

A memória principal (RAM) é um recurso importante que deve ser cuidadosamente gerenciado. Apesar de o computador pessoal médio hoje em dia ter 10.000 vezes mais memória do que o IBM 7094, o maior computador no mundo no início da década de 1960, os programas estão ficando maiores mais rápido do que as memórias. Parafraseando a Lei de Parkinson, "programas tendem a expandir-se a fim de preencher a memória disponível para contê-los". Neste capítulo, estudaremos como os sistemas operacionais criam abstrações a partir da memória e como eles as gerenciam.

O que todo programador gostaria é de uma memória privada, infinitamente grande e rápida, que fosse não volátil também, isto é, não perdesse seus conteúdos quando faltasse energia elétrica. Aproveitando o ensejo, por que não torná-la barata, também? Infelizmente, a tecnologia ainda não produz essas memórias no momento. Talvez você descubra como fazê-lo.

Qual é a segunda escolha? Ao longo dos anos, as pessoas descobriram o conceito de **hierarquia de memórias**, em que os computadores têm alguns megabytes de memória cache volátil, cara e muito rápida, alguns gigabytes de memória principal volátil de velocidade e custo médios, e alguns terabytes de armazenamento em disco em estado sólido ou magnético não volátil, barato e lento, sem mencionar o armazenamento removível, com DVDs e dispositivos USB. É função do sistema operacional abstrair essa hierarquia em um modelo útil e então gerenciar a abstração.

A parte do sistema operacional que gerencia (parte da) hierarquia de memórias é chamada de **gerenciador de memória**. Sua função é gerenciar eficientemente a memória: controlar quais partes estão sendo usadas, alocar memória para processos quando eles precisam dela e liberá-la quando tiverem terminado.

Neste capítulo investigaremos vários modelos diferentes de gerenciamento de memória, desde os muito simples aos altamente sofisticados. Dado que gerenciar o nível mais baixo de memória cache é feito normalmente pelo hardware, o foco deste capítulo estará no modelo de memória principal do programador e como ela pode ser gerenciada. As abstrações para — e o gerenciamento do — armazenamento permanente (o disco), serão tratados no próximo capítulo. Examinaremos primeiro os esquemas mais simples possíveis e então gradualmente avançaremos para os esquemas cada vez mais elaborados.

3.1 Sem abstração de memória

A abstração de memória mais simples é não ter abstração alguma. Os primeiros computadores de grande porte (antes de 1960), os primeiros minicomputadores (antes de 1970) e os primeiros computadores pessoais (antes de 1980) não tinham abstração de memória. Cada programa apenas via 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 apenas a memória física, um conjunto de endereços de 0 a algum máximo, cada endereço correspondendo a uma célula contendo algum número de bits, normalmente oito.

Nessas condições, não era possível ter dois programas em execução na memória ao mesmo tempo. Se o primeiro programa escrevesse um novo valor para, digamos, a posição 2000, esse valor apagaria qualquer valor que o segundo programa estivesse armazenando ali. Nada funcionaria e ambos os programas entrariam em colapso quase que imediatamente.

Mesmo com o modelo de memória sendo apenas da memória física, várias opções são possíveis. Três variações são mostradas na Figura 3.1. O sistema operacional pode estar na parte inferior da memória em RAM (Random Access Memory — memória de acesso aleatório), como mostrado na Figura 3.1(a), ou pode estar em ROM (Read-Only Memory — memória apenas para leitura) no topo da memória, como mostrado na Figura 3.1(b), ou os drivers do dispositivo talvez estejam no topo da memória em um ROM e o resto do sistema em RAM bem abaixo, como mostrado na Figura 3.1(c). O primeiro modelo foi usado antes em computadores de grande porte e minicomputadores, mas raramente é usado. O segundo modelo é usado em alguns computadores portáteis e sistemas embarcados. O terceiro modelo foi usado pelos primeiros computadores pessoais (por exemplo, executando o MS-DOS), onde a porção do sistema no ROM é chamada de **BIOS** (**Basic Input Output System** — sistema básico de E/S). Os modelos (a) e (c) têm a desvantagem de que um erro no programa do usuário pode apagar por completo o sistema operacional, possivelmente com resultados desastrosos.

Quando o sistema está organizado dessa maneira, geralmente apenas um processo de cada vez pode estar executando. Tão logo 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 de comando e espera por um novo comando do usuário. Quando o sistema operacional recebe o comando, ele carrega um programa novo para a memória, sobrescrevendo o primeiro.

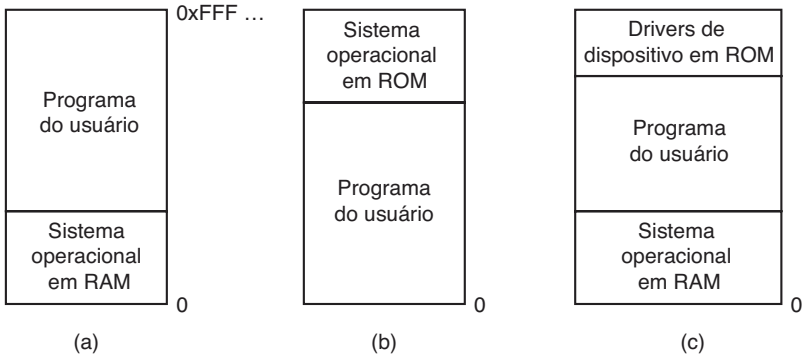
Uma maneira de se conseguir algum paralelismo em um sistema sem abstração de memória é programá-lo com múltiplos threads. Como todos os threads em um processo devem ver a mesma imagem da memória, o fato de eles serem forçados a fazê-lo não é um problema. Embora essa ideia funcione, ela é de uso limitado, pois o que muitas vezes as pessoas querem é que programas *não relacionados* estejam executando ao mesmo tempo, algo que a abstração de threads não realiza. Além disso, qualquer sistema que seja tão primitivo a ponto de não proporcionar qualquer abstração de memória é improvável que proporcione uma abstração de threads.

Executando múltiplos programas sem uma abstração de memória

No entanto, mesmo sem uma abstração de memória, é possível executar múltiplos programas ao mesmo tempo. O que um sistema operacional precisa fazer é salvar o conteúdo inteiro da memória em um arquivo de disco, então introduzir e executar o programa seguinte. Desde que exista apenas um programa de cada vez na memória, não há conflitos. Esse conceito (*swapping* — troca de processos) será discutido a seguir.

Com a adição de algum hardware especial, é possível executar múltiplos programas simultaneamente, mesmo sem swapping. Os primeiros modelos da IBM 360 solucionaram o problema como a seguir. A memória foi dividida em blocos de 2 KB e a cada um foi designada uma chave de proteção de 4 bits armazenada em registradores especiais dentro da CPU. Uma máquina com uma memória de 1 MB necessitava de apenas 512 desses registradores de 4 bits para um total de 256 bytes de armazenamento de chaves. A PSW (Program

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



Status Word — palavra de estado do programa) também continha uma chave de 4 bits. O hardware do 360 impedia qualquer tentativa de um processo em execução de acessar a memória com um código de proteção diferente do da chave PSW. Visto que apenas o sistema operacional podia mudar as chaves de proteção, os processos do usuário eram impedidos de interferir uns com os outros e com o sistema operacional em si.

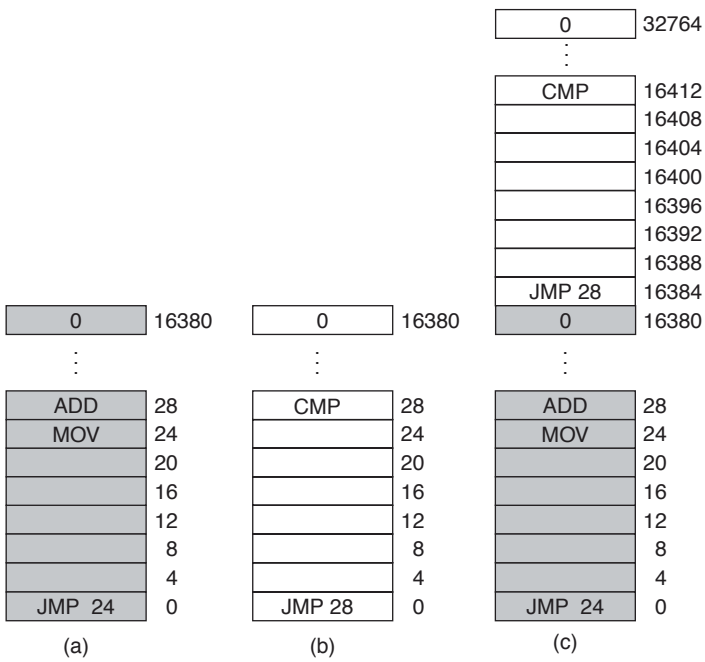
No entanto, essa solução tinha um problema importante, descrito na Figura 3.2. Aqui temos dois programas, cada um com 16 KB de tamanho, como mostrado nas figuras 3.2(a) e (b). O primeiro está sombreado para indicar que ele tem uma chave de memória diferente da do segundo. O primeiro programa começa com um salto para o endereço 24, que contém uma instrução MOV. O segundo inicia saltando para o endereço 28, que contém uma instrução CMP. As instruções que não são relevantes para essa discussão não são mostradas. Quando os dois programas são carregados consecutivamente na memória, começando no endereço 0, temos a situação da Figura 3.2(c). Para esse exemplo, presumimos que o sistema operacional está na região alta da memória e assim não é mostrado.

Após os programas terem sido carregados, eles podem ser executados. Dado que eles têm chaves de memória diferentes, nenhum dos dois pode danificar o outro. Mas o problema é de uma natureza diferente. Quando o primeiro programa inicializa, ele executa a instrução JMP 24, que salta para a instrução, como esperado. Esse programa funciona normalmente.

No entanto, após o primeiro programa ter executado 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 no primeiro programa, em vez da instrução CMP esperada. É muito provável que o programa entre em colapso bem antes de 1 s.

O problema fundamental aqui é que ambos os programas referenciam a memória física absoluta, e não é isso que queremos, de forma alguma. O que queremos é cada programa possa referenciar um conjunto privado de endereços local a ele. Mostraremos como isso pode ser conseguido. O que o IBM 360 utilizou como solução temporária foi modificar o segundo programa dinamicamente enquanto o carregava na memória, usando uma técnica conhecida como **realocação estática**. Ela funcionava da seguinte forma: quando um programa estava carregado no endereço 16.384, a constante 16.384 era acrescentada a cada endereço de programa durante o processo de carregamento (de maneira que "JMP 28" tornou-se "JMP 16.412" etc.). Conquanto esse mecanismo funcione se feito de maneira correta, ele não é uma solução muito geral e torna lento o carregamento. Além disso, exige informações adicionais em todos os programas executáveis cujas palavras contenham ou não

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



endereços (realocáveis). Afinal, o "28" na Figura 3.2(b) deve ser realocado, mas uma instrução como

```
MOV REGISTER1, 28
```

que move o número 28 para *REGISTER1* não deve ser realocada. O carregador precisa de alguma maneira dizer o que é um endereço e o que é uma constante.

Por fim, como destacamos no Capítulo 1, a história tende a repetir-se no mundo dos computadores. Embora o endereçamento direto de memória física seja apenas uma memória distante nos computadores de grande porte, minicomputadores, computadores de mesa, notebooks e smartphones, a falta de uma abstração de memória ainda é comum em sistemas embarcados e de cartões inteligentes. Dispositivos como rádios, máquinas de lavar roupas e fornos de micro-ondas estão todos cheios de software (em 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 o seu próprio software na sua torradeira.

Enquanto sistemas embarcados sofisticados (como smartphones) têm sistemas operacionais elaborados, os mais simples não os têm. Em alguns casos, há um sistema operacional, mas é apenas uma biblioteca que está vinculada ao programa de aplicação e fornece chamadas de sistema para desempenhar E/S e outras tarefas comuns. O sistema operacional **e-Cos** é um exemplo comum de um sistema operacional como biblioteca.

3.2 Uma abstração de memória: espaços de endereçamento

Como um todo, expor a memória física a processos tem várias desvantagens importantes. Primeiro, se os programas do usuário podem endereçar cada byte de memória, eles podem facilmente derrubar o sistema operacional, intencionalmente ou por acidente, provocando uma parada total no sistema (a não ser que exista um hardware especial como o esquema de bloqueio e chave do IBM 360). Esse problema existe mesmo que só um programa do usuário (aplicação) esteja executando. Segundo, com esse modelo, é difícil ter múltiplos programas executando ao mesmo tempo (revezando-se, se houver apenas uma CPU). Em computadores pessoais, é comum haver vários programas abertos ao mesmo tempo (um processador de texto, um programa de e-mail, um navegador da web), um deles tendo o foco atual, mas os outros sendo reativados ao clique de um mouse. Como essa situação é difícil de ser atingida quando não há abstração da memória física, algo tinha de ser feito.

3.2.1 A noção de um espaço de endereçamento

Dois problemas têm de ser solucionados para permitir que múltiplas aplicações estejam na memória ao mesmo tempo sem interferir umas com as outras: proteção e realocação. Examinamos uma solução primitiva para a primeira usada no IBM 360: rotular blocos de memória com uma chave de proteção e comparar a chave do processo em execução com aquele de toda palavra de memória buscada. No entanto, essa abordagem em si não soluciona o segundo problema, embora ele possa ser resolvido realocando programas à medida que eles são carregados, mas essa é uma solução lenta e complicada.

Uma solução melhor é inventar uma nova abstração para a memória: o espaço de endereçamento. Da mesma forma que o conceito de processo cria uma espécie de CPU abstrata para executar os programas, o espaço de endereçamento cria uma espécie 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 daqueles pertencentes a outros processos (exceto em algumas circunstâncias especiais onde os processos querem compartilhar seus espaços de endereçamento).

O conceito de um espaço de endereçamento é muito geral e ocorre em muitos contextos. Considere os números de telefones. Nos Estados Unidos e em muitos outros países, um número de telefone local costuma ter 7 dígitos. Desse modo, o espaço de endereçamento para números de telefone vai de 0.000.000 a 9.999.999, embora alguns números, como aqueles começando com 000, não sejam usados. Com o crescimento dos smartphones, modems e máquinas de fax, esse espaço está se tornando pequeno demais, e mais dígitos precisam ser usados. O espaço de endereçamento para portas de E/S no x86 varia de 0 a 16.383. Endereços de IPv4 são números de 32 bits, de maneira que seu espaço de endereçamento varia de 0 a $2^{32} - 1$ (de novo, com alguns números reservados).

Espaços de endereçamento não precisam ser numéricos. O conjunto de domínios da internet *.com* também é um espaço de endereçamento. Ele consiste em todas as cadeias de comprimento 2 a 63 caracteres que podem ser feitas usando letras, números e hífens, seguidas por *.com*. A essa altura você deve ter compreendido. É algo relativamente simples.

Algo um tanto mais difícil é como dar a cada programa seu próprio espaço de endereçamento, de maneira que o endereço 28 em um programa significa uma localização física diferente do endereço 28 em outro programa. A seguir discutiremos uma maneira simples que costumava ser comum, mas caiu em desuso por causa da capacidade de se inserirem esquemas muito mais complicados (e melhores) em chips de CPUs modernos.

Registradores base e registradores limite

Essa solução simples usa uma versão particularmente simples da **realocação dinâmica**. O que ela faz é mapear o espaço de endereçamento de cada processo em uma parte diferente da memória física de uma maneira 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 esses registradores são usados, os programas são carregados em posições de memória consecutivas sempre que 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), os valores base e limite que seriam carregados nesses registradores de 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, os registradores base e limite seriam 32.768 e 16.384.

Toda vez que um processo referencia a memória, seja para buscar uma instrução ou ler ou escrever uma palavra de dados, o hardware da CPU automaticamente adiciona o valor base ao endereço gerado pelo processo antes de enviá-lo para o barramento de memória. Ao mesmo tempo, ele confere se o endereço oferecido é igual ou maior do que o valor no registrador limite, caso em que uma falta é gerada 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 ela fosse

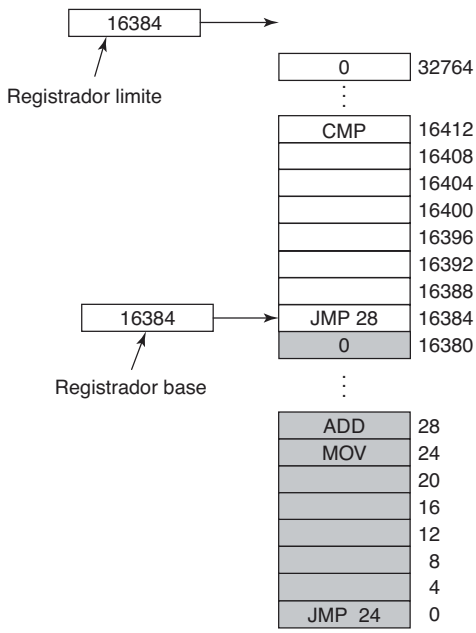
```
JMP 16412
```

portanto ela chega à instrução CMP como esperado. As configurações dos registradores base e limite durante a execução do segundo programa da Figura 3.2(c) são mostradas na Figura 3.3.

Usar registradores base e limite é uma maneira fácil de dar a cada processo seu próprio espaço de endereçamento privado, pois cada endereço de memória gerado automaticamente tem o conteúdo do registrador base adicionado a ele antes de ser enviado para a memória. Em muitas implementações, os registradores base e limite são protegidos de tal maneira que apenas o sistema operacional pode modificá-los. Esse foi o caso do CDC 6600, mas não no Intel 8088, que não tinha nem um registrador limite. Ele tinha múltiplos registradores base, permitindo programar textos e dados, por exemplo, para serem realocados independentemente, mas não oferecia proteção contra referências à memória além da capacidade.

Uma desvantagem da realocação usando registradores base e limite é a necessidade de realizar uma adição e uma comparação em cada referência de memória. Comparações podem ser feitas rapidamente, mas adições são lentas por causa do tempo de propagação do transporte (carry-propagation time), a não ser que circuitos de adição especiais sejam usados.

FIGURA 3.3 Registradores base ou limite podem ser usados para dar a cada processo um espaço de endereçamento em separado.



3.2.2 Troca de processos (Swapping)

Se a memória física do computador for grande o suficiente para armazenar todos os processos, os esquemas descritos até aqui bastarão de certa forma. Mas na prática, o montante total de RAM demandado por todos os processos é muitas vezes bem maior do que pode ser colocado na memória. Em sistemas típicos Windows, OS X ou Linux, algo como 50-100 processos ou mais podem ser iniciados tão logo o computador for ligado. Por exemplo, quando uma aplicação do Windows é instalada, ela muitas vezes emite comandos de tal forma que em inicializações subsequentes do sistema, um processo será iniciado somente para conferir se existem atualizações para as aplicações. Um processo desses pode facilmente ocupar 5-10 MB de memória. Outros processos de segundo plano conferem se há e-mails, conexões de rede chegando e muitas outras coisas. E tudo isso antes de o primeiro programa do usuário ter sido iniciado. Programas sérios de aplicação do usuário, como o Photoshop, podem facilmente exigir 500 MB apenas para serem inicializados e muitos gigabytes assim que começam a processar dados. Em consequência, manter todos os processos na memória o tempo inteiro exige um montante enorme de memória e é algo que não pode ser feito se ela for insuficiente.

Duas abordagens gerais para lidar com a sobrecarga de memória foram desenvolvidas ao longo dos anos. A estratégia mais simples, chamada de **swapping** (troca de processos), consiste em trazer cada processo em sua totalidade, executá-lo por um tempo e então colocá-lo de volta no disco. Processos ociosos estão armazenados em disco em sua maior parte, portanto não ocupam qualquer memória quando não estão sendo executados (embora alguns "despertem" periodicamente para fazer seu trabalho, e então voltam a "dormir"). A outra estratégia, chamada de **memória virtual**, permite que os programas possam ser executados mesmo quando estão apenas parcialmente na memória principal. A seguir estudaremos a troca de processos; na Seção 3.3 examinaremos a memória virtual.

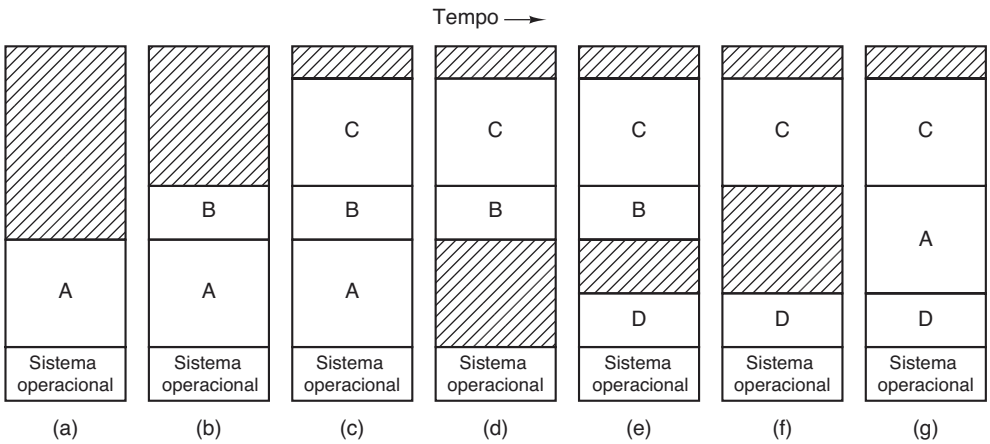
A operação de um sistema de troca de processos está ilustrada na Figura 3.4. De início, somente o processo *A* está na memória. Então os processos *B* e *C* são criados ou trazidos do disco. Na Figura 3.4(d) o processo *A* é devolvido ao disco. Então o processo *D* é inserido e o processo *B* tirado. Por fim, o processo *A* volta novamente. Como *A* está agora em uma posição diferente, os endereços contidos nele devem ser realocados, seja pelo software quando ele é trazido ou (mais provável) pelo hardware durante a execução do programa. Por exemplo, registradores base e limite funcionariam bem aqui.

Quando as trocas de processos criam múltiplos espaços na memória, é possível combiná-los em um grande espaço movendo todos os processos para baixo, o máximo possível. Essa técnica é conhecida como **compactação de memória**. Em geral ela não é feita porque exige muito tempo da CPU. Por exemplo, em uma máquina de 16 GB que pode copiar 8 bytes em 8 ns, ela levaria em torno de 16 s para compactar toda a memória.

Um ponto que vale a pena considerar diz respeito a quanta memória deve ser alocada para um processo quando ele é criado ou trocado. Se os processos são criados com um tamanho fixo que nunca muda, então a alocação é simples: o sistema operacional aloca exatamente o que é necessário, nem mais nem menos.

Se, no entanto, os segmentos de dados dos processos podem crescer, alocando dinamicamente memória

FIGURA 3.4 Mudanças na alocação de memória à medida que processos entram nela e saem dela. As regiões sombreadas são regiões não utilizadas da memória.



de uma área temporária, como em muitas linguagens de programação, um problema ocorre sempre que um processo tenta crescer. Se houver um espaço adjacente ao processo, ele poderá ser alocado e o processo será autorizado a crescer naquele espaço. Por outro lado, se o processo for adjacente a outro, aquele que cresce terá de ser movido para um espaço na memória grande o suficiente para ele, ou um ou mais processos terão de ser trocados para criar um espaço grande o suficiente. Se um processo não puder crescer em memória e a área de troca no disco estiver cheia, ele terá de ser suspenso até que algum espaço seja liberado (ou ele pode ser morto).

Se o esperado for que a maioria dos processos cresça à medida que são executados, provavelmente seja uma boa ideia alocar um pouco de memória extra sempre que um processo for trocado ou movido, para reduzir a sobrecarga associada com a troca e movimentação dos processos que não cabem mais em sua memória alocada. No entanto, ao transferir processos para o disco, apenas a memória realmente em uso deve ser transferida; é um desperdício levar a memória extra também. Na Figura 3.5(a) vemos uma configuração de memória na qual o espaço para o crescimento foi alocado para dois processos.

Se os processos podem ter dois segmentos em expansão — por exemplo, os segmentos de dados usados como uma área temporária para variáveis que são dinamicamente alocadas e liberadas e uma área de pilha para as variáveis locais normais e endereços de retorno — uma solução alternativa se apresenta, a saber, aquela da Figura 3.5(b). Nessa figura vemos que cada processo ilustrado tem uma pilha no topo da sua memória alocada, que cresce para baixo, e um segmento de dados logo além do programa de texto, que cresce para cima. A memória entre eles pode ser usada por qualquer segmento. Se ela acabar, o processo poderá ser transferido para outra área com espaço suficiente, ser transferido para o disco até que um espaço de tamanho suficiente possa ser criado, ou ser morto.

3.2.3 Gerenciando a memória livre

Quando a memória é designada dinamicamente, o sistema operacional deve gerenciá-la. Em termos gerais, há duas maneiras de se rastrear o uso de memória: mapas de bits e listas livres. Nesta seção e na próxima, examinaremos esses dois métodos. No Capítulo 10, estudaremos alguns alocadores de memória específicos no Linux [como os alocadores companheiros e de fatias (slab)] com mais detalhes.

Gerenciamento de memória com mapas de bits

Com um mapa de bits, a memória é dividida em unidades de alocação tão pequenas quanto umas poucas palavras e tão grandes quanto vários quilobytes. Correspondendo a cada unidade de alocação há um bit no mapa de bits, que é 0 se a unidade estiver livre e 1 se ela estiver ocupada (ou vice-versa). A Figura 3.6 mostra parte da memória e o mapa de bits correspondente.

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 expansão.

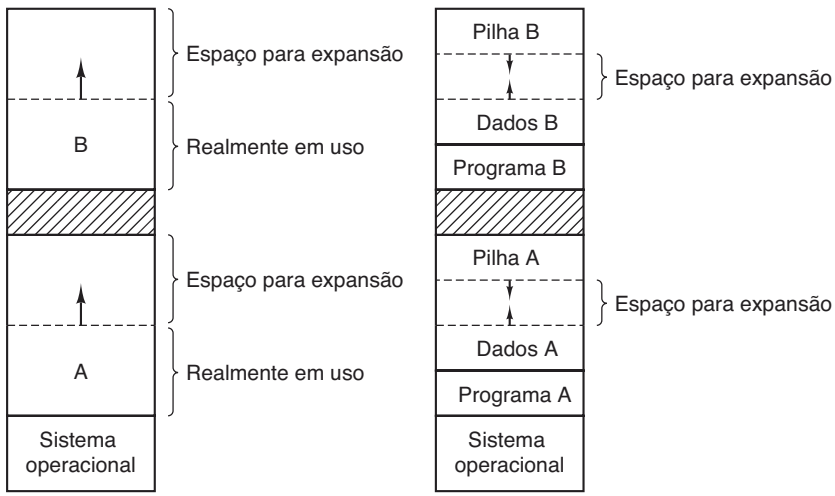
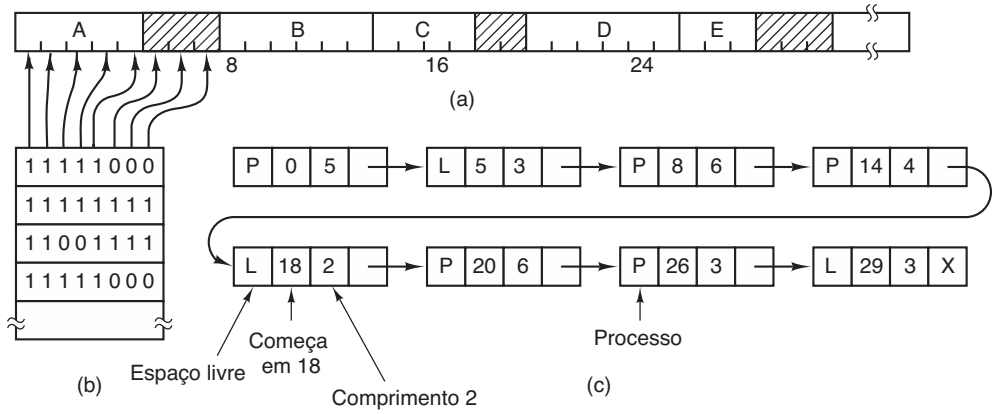


FIGURA 3.6 (a) Uma parte da memória com cinco processos e três espaços. As marcas indicam as unidades de alocação de memória. As regiões sombreadas (0 no mapa de bits) estão livres. (b) Mapa de bits correspondente. (c) A mesma informação como lista.



O tamanho da unidade de alocação é uma importante questão de projeto. Quanto menor a unidade de alocação, maior o mapa de bits. No entanto, mesmo com uma unidade de alocação tão pequena quanto 4 bytes, 32 bits de memória exigirão apenas 1 bit do mapa. Uma memória de 32*n* bits usará um mapa de *n* bits, então o mapa de bits ocupará apenas 1/32 da memória. Se a unidade de alocação for definida como grande, o mapa de bits será menor, mas uma quantidade considerável de memória será desperdiçada na última unidade do processo se o tamanho dele não for um múltiplo exato da unidade de alocação.

Um mapa de bits proporciona uma maneira simples de controlar as palavras na memória em uma quantidade fixa dela, porque seu tamanho depende somente dos tamanhos da memória e da unidade de alocação. O principal problema é que, quando fica decidido carregar um processo com tamanho de *k* unidades, o gerenciador de memória deve procurar o mapa de bits para encontrar uma sequência de *k* bits 0 consecutivos. Procurar em um mapa de bits por uma sequência de um comprimento determinado é uma operação lenta (pois a sequência pode ultrapassar limites de palavras no mapa); este é um argumento contrário aos mapas de bits.

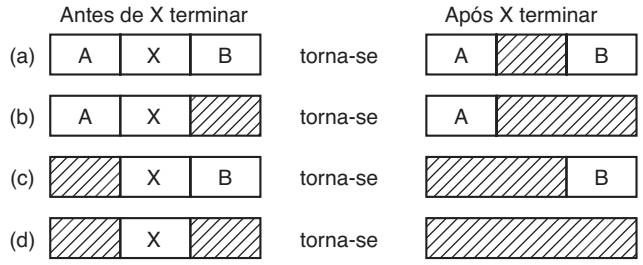
Gerenciamento de memória com listas encadeadas

Outra maneira de controlar o uso da memória é manter uma lista encadeada de espaços livres e de segmentos de memória alocados, onde um segmento contém um processo ou é um espaço vazio entre dois processos. A memória da Figura 3.6(a) é representada na Figura 3.6(c) como uma lista encadeada de segmentos. Cada entrada na lista especifica se é um espaço livre (L) ou alocado a um processo (P), o endereço no qual se inicia esse segmento, o comprimento e um ponteiro para o item seguinte.

Nesse exemplo, a lista de segmentos é mantida ordenada pelos endereços. Essa ordenação tem a vantagem de que, quando um processo é terminado ou transferido, atualizar a lista é algo simples de se fazer. Um processo que termina a sua execução tem dois vizinhos (exceto quando espaços no início ou no fim da memória). Eles podem ser tanto processos quanto espaços livres, levando às quatro combinações mostradas na Figura 3.7. Na Figura 3.7(a) a atualização da lista exige substituir um P por um L. Nas figuras 3.7(b) e 3.7(c), duas entradas são fundidas em uma, e a lista fica uma entrada mais curta. Na Figura 3.7(d), três entradas são fundidas e dois itens são removidos da lista.

Como a vaga da tabela de processos para o que está sendo concluído geralmente aponta para a entrada da lista do próprio processo, talvez seja mais conveniente ter a lista como uma lista duplamente encadeada, em vez daquela com encadeamento simples da Figura 3.6(c). Essa estrutura torna mais fácil encontrar a entrada anterior e ver se a fusão é possível.

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



Quando processos e espaços livres são mantidos em uma lista ordenada por endereço, vários algoritmos podem ser usados para alocar memória para um processo criado (ou um existente em disco sendo transferido para a memória). Presumimos que o gerenciador de memória sabe quanta memória alocar. O algoritmo mais simples é **first fit** (primeiro encaixe). O gerenciador de memória examina a lista de segmentos até encontrar um espaço livre que seja grande o suficiente. O espaço livre é então dividido em duas partes, uma para o processo e outra para a memória não utilizada, exceto no caso estatisticamente improvável de um encaixe exato. First fit é um algoritmo rápido, pois ele procura fazer a menor busca possível.

Uma pequena variação do first fit é o **next fit**. Ele funciona da mesma maneira que o *first fit*, exceto por memorizar a posição que se encontra um espaço livre adequado sempre que o encontra. Da vez seguinte que for chamado para encontrar um espaço livre, ele começa procurando na lista do ponto onde havia parado, em vez de sempre do princípio, como faz o first fit. Simulações realizadas por Bays (1977) mostram que o next fit tem um desempenho ligeiramente pior do que o do first fit.

Outro algoritmo bem conhecido e amplamente usado é o **best fit**. O best fit faz uma busca em toda a lista, do início ao fim, e escolhe o menor espaço livre que seja adequado. Em vez de escolher um espaço livre grande demais que talvez seja necessário mais tarde, o best fit tenta encontrar um que seja de um tamanho próximo do tamanho real necessário, para casar da melhor maneira possível a solicitação com os segmentos disponíveis.

Como um exemplo do first fit e best fit, considere a Figura 3.6 novamente. Se um bloco de tamanho 2 for necessário, first fit alocará o espaço livre em 5, mas o best fit o alocará em 18.

O best fit é mais lento do que o first fit, pois ele tem de procurar na lista inteira toda vez que é chamado. De uma maneira um tanto surpreendente, ele também resulta em um desperdício maior de memória do que o first fit ou next fit, pois tende a preencher a memória com segmentos minúsculos e inúteis. O first fit gera espaços livres maiores em média.

Para contornar o problema de quebrar um espaço livre em um processo e um trecho livre minúsculo, a solução poderia ser o **worst fit**, isto é, sempre escolher o maior espaço livre, de maneira que o novo segmento livre gerado seja grande o bastante para ser útil. No entanto, simulações demonstraram que o *worst fit* tampouco é uma grande ideia.

Todos os quatro algoritmos podem ser acelerados mantendo-se listas em separado para os processos e os espaços livres. Dessa maneira, todos eles devotam toda a sua energia para inspecionar espaços livres, não processos. O preço inevitável que é pago por essa aceleração na alocação é a complexidade e lentidão adicionais ao remover a memória, já que um segmento liberado precisa ser removido da lista de processos e inserido na lista de espaços livres.

Se listas distintas são mantidas para processos e espaços livres, a lista de espaços livres deve ser mantida ordenada por tamanho, a fim de tornar o best fit mais rápido. Quando o best fit procura em uma lista de segmentos de memória livre do menor para o maior, tão logo encontra um que se encaixe, ele sabe que esse segmento é o menor que funcionará, daí o nome. Não são necessárias mais buscas, como ocorre com o esquema de uma lista única. Com uma lista de espaços livres ordenada por tamanho, o first fit e o best fit são igualmente rápidos, e o next fit sem sentido.

Quando os espaços livres são mantidos em listas separadas dos processos, uma pequena otimização é possível. Em vez de ter um conjunto separado de estruturas de dados para manter a lista de espaços livres, como mostrado na Figura 3.6(c), a informação pode ser armazenada nos espaços livres. A primeira palavra de cada espaço livre pode ser seu tamanho e a segunda palavra um ponteiro para a entrada a seguir. Os nós da lista da Figura 3.6(c), que exigem três palavras e um bit (P/L), não são mais necessários.

Outro algoritmo de alocação é o **quick fit**, que mantém listas em separado para alguns dos tamanhos mais comuns solicitados. Por exemplo, ele pode ter uma tabela com n entradas, na qual a primeira é um ponteiro para o início de uma lista espaços livres de 4 KB, a segunda é um ponteiro para uma lista de espaços livres de 8 KB, a terceira de 12 KB e assim por diante. Espaços livres de, digamos, 21 KB, poderiam ser colocados na lista de 20 KB ou em uma lista de espaços livres de tamanhos especiais.

Com o quick fit, encontrar um espaço livre do tamanho exigido é algo extremamente rápido, mas tem as mesmas desvantagens de todos os esquemas que ordenam por tamanho do espaço livre, a saber, quando um processo termina sua execução ou é transferido da memória, descobrir seus vizinhos para ver se uma fusão com eles é possível é algo bastante caro. Se a fusão não for feita, a memória logo se fragmentará em um grande número de pequenos segmentos livres nos quais nenhum processo se encaixará.

3.3 Memória virtual

Embora os registradores base e os registradores limite possam ser usados para criar a abstração de espaços de endereçamento, há outro problema que precisa ser solucionado: gerenciar o bloatware.[1] Apesar de os tamanhos das memórias aumentarem depressa, os tamanhos dos softwares estão crescendo muito mais rapidamente. Nos anos 1980, muitas universidades executavam um sistema de compartilhamento de tempo com dúzias de usuários (mais ou menos satisfeitos) executando simultaneamente em um VAX de 4 MB. Agora a Microsoft recomenda no mínimo 2 GB para o Windows 8 de 64 bits. A tendência à multimídia coloca ainda mais demandas sobre a memória.

Como consequência desses desenvolvimentos, há uma necessidade de executar programas que são grandes demais para se encaixar na memória e há certamente uma necessidade de ter sistemas que possam dar suporte a múltiplos programas executando em simultâneo, cada um deles encaixando-se na memória, mas com todos coletivamente excedendo-a. A troca de processos não é uma opção atraente, visto que o disco SATA típico tem um pico de taxa de transferência de várias centenas de MB/s, o que significa que demora segundos para retirar um programa de 1 GB e o mesmo para carregar um programa de 1 GB.

O problema dos programas maiores do que a memória existe desde o início da computação, embora em áreas limitadas, como a ciência e a engenharia (simular a criação do universo, ou mesmo um avião novo, exige muita memória). Uma solução adotada nos anos 1960 foi dividir os programas em módulos pequenos, chamados de **sobreposições**. Quando um programa inicializava, tudo o que era carregado na memória era o gerenciador de sobreposições, que imediatamente carregava e executava a sobreposição 0. Quando terminava, ele dizia ao gerenciador de sobreposições para carregar a sobreposição 1, acima da sobreposição 0 na memória (se houvesse espaço para isso), ou em cima da sobreposição 0 (se não houvesse). Alguns sistemas de sobreposições eram altamente complexos, permitindo muitas sobreposições na memória ao mesmo tempo. As sobreposições eram mantidas no disco e transferidas para dentro ou para fora da memória pelo gerenciador de sobreposições.

Embora o trabalho real de troca de sobreposições do disco para a memória e vice-versa fosse feito pelo sistema operacional, o trabalho da divisão do programa em módulos tinha de ser feito manualmente pelo programador. Dividir programas grandes em módulos pequenos era uma tarefa cansativa, chata e propensa a erros. Poucos programadores eram bons nisso. Não levou muito tempo para alguém pensar em passar todo o trabalho para o computador.

O método encontrado (FOTHERINGHAM, 1961) ficou conhecido como **memória virtual**. A ideia básica é que cada programa tem seu próprio espaço de endereçamento, o qual é dividido em blocos chamados de **páginas**. Cada página é uma série contígua de endereços. Elas são mapeadas na memória física, mas nem todas precisam estar na memória física ao mesmo tempo para executar o programa. Quando o programa referencia uma parte do espaço de endereçamento que está na memória física, o hardware realiza o mapeamento necessário rapidamente. Quando o programa referencia uma parte de seu espaço de endereçamento que *não* está na memória física, o sistema operacional é alertado para ir buscar a parte que falta e reexecuta a instrução que falhou.

De certa maneira, a memória virtual é uma generalização da ideia do registrador base e registrador limite. O 8088 tinha registradores base separados (mas não registradores limite) para texto e dados. Com a memória virtual, em vez de ter realocações separadas apenas para os segmentos de texto e dados, todo o espaço de endereçamento pode ser mapeado na memória física em unidades razoavelmente pequenas. Mostraremos a seguir como a memória virtual é implementada.

A memória virtual funciona bem em um sistema de multiprogramação, com pedaços e partes de muitos programas na memória simultaneamente. Enquanto um programa está esperando que partes de si mesmo sejam lidas, a CPU pode ser dada para outro processo.

3.3.1 Paginação

A maioria dos sistemas de memória virtual usa uma técnica chamada de **paginação**, que descreveremos agora. Em qualquer computador, programas referenciam um conjunto de endereços de memória. Quando um programa executa uma instrução como

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

```
MOV REG,1000
```

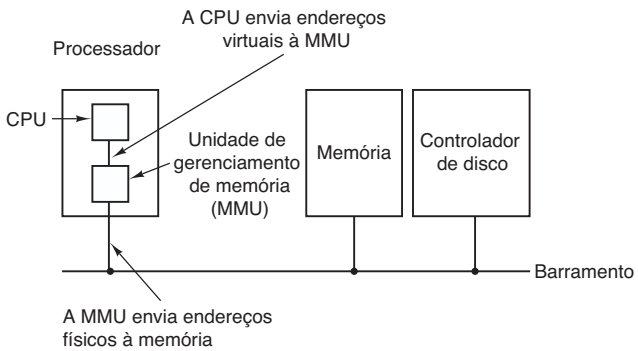
ele o faz para copiar o conteúdo do endereço de memória 1000 para REG (ou vice-versa, dependendo do computador). Endereços podem ser gerados usando indexação, registradores base, registradores de segmento e outras maneiras.

Esses endereços gerados por computadores são chamados de **endereços virtuais** e formam o **espaço de endereçamento virtual**. Em computadores sem memória virtual, o endereço virtual é colocado diretamente no barramento de memória e faz que a palavra de memória física com o mesmo endereço seja lida ou escrita. Quando a memória virtual é usada, os endereços virtuais não vão diretamente para o barramento da memória. Em vez disso, eles vão para uma **MMU** (**Memory Management Unit** — unidade de gerenciamento de memória) que mapeia os endereços virtuais em endereços de memória física, como ilustrado na Figura 3.8.

Um exemplo muito simples de como esse mapeamento funciona é mostrado na Figura 3.9. Nesse exemplo, temos um computador que gera endereços de 16 bits, de 0 a 64 K − 1. Esses são endereços virtuais. Esse computador, no entanto, tem apenas 32 KB de memória física. Então, embora programas de 64 KB possam ser escritos, eles não podem ser totalmente carregados na memória e executados. Uma cópia completa da imagem de núcleo de um programa, de até 64 KB, deve estar presente no disco, entretanto, de maneira que partes possam ser carregadas quando necessário.

O espaço de endereçamento virtual consiste em unidades de tamanho fixo chamadas de páginas. As unidades correspondentes na memória física são chamadas de **quadros de página**. As páginas e os quadros de página são geralmente do mesmo tamanho.

FIGURA 3.8 A posição e função da MMU. Aqui a MMU é mostrada como parte do chip da CPU porque isso é comum hoje. No entanto, logicamente, poderia ser um chip separado, como era no passado.



Nesse exemplo, elas têm 4 KB, mas tamanhos de página de 512 bytes a um gigabyte foram usadas em sistemas reais. Com 64 KB de espaço de endereçamento virtual e 32 KB de memória física, podemos ter 16 páginas virtuais e 8 quadros de páginas. Transferências entre a memória RAM e o disco são sempre em páginas inteiras. Muitos processadores dão suporte a múltiplos tamanhos de páginas que podem ser combinados e casados como o sistema operacional preferir. Por exemplo, a arquitetura x86-64 dá suporte a páginas de 4 KB, 2 MB e 1 GB, então poderíamos usar páginas de 4 KB para aplicações do usuário e uma única página de 1 GB para o núcleo. Veremos mais tarde por que às vezes é melhor usar uma única página maior do que um grande número de páginas pequenas.

A notação na Figura 3.9 é a seguinte: a série marcada 0K−4K significa que os endereços virtuais ou físicos naquela 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 o programa tenta acessar o endereço 0, por exemplo, usando a instrução

```
MOV REG,0
```

o endereço virtual 0 é enviado para a MMU. A MMU detecta que esse endereço virtual situa-se na página 0 (0 a 4095), que, de acordo com seu mapeamento, corresponde ao quadro de página 2 (8192 a 12287). Ele então transforma o endereço para 8192 e envia o endereço 8192 para o barramento. A memória desconhece completamente a MMU e apenas vê uma solicitação para leitura ou escrita do endereço 8192, a qual ela executa. Desse modo, a MMU mapeou efetivamente todos os endereços virtuais de 0 a 4095 em endereços físicos localizados de 8192 a 12287.

De modo 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 (no quadro de página física 6). Como um terceiro exemplo, o endereço virtual 20500 está localizado a 20 bytes do início da página virtual 5 (endereços virtuais 20480 a 24575) e é mapeado no endereço físico 12288 + 20 = 12308.

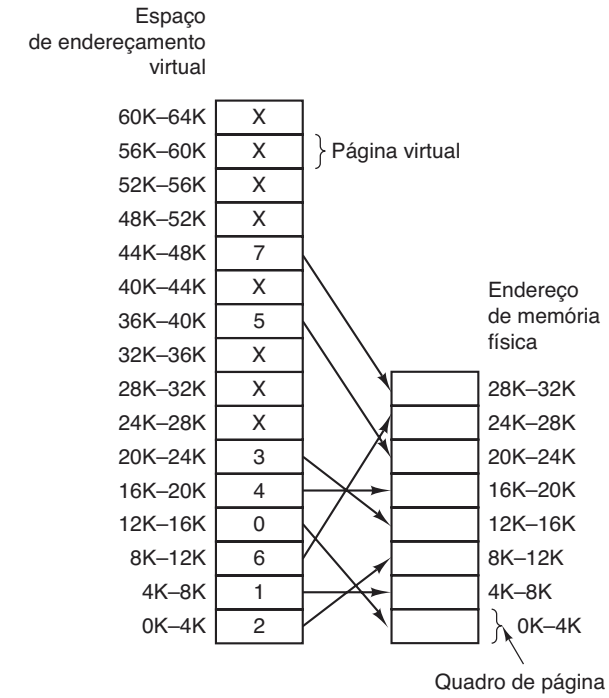


FIGURA 3.9 A relação entre endereços virtuais e endereços de memória física é dada pela **tabela de páginas**. Cada página começa com um múltiplo de 4096 e termina 4095 endereços acima; assim, 4K a 8K na verdade significa 4096-8191 e 8K a 12K significa 8192-12287.

Por si só, essa habilidade de mapear as 16 páginas virtuais em qualquer um dos oito quadros de páginas por meio da configuração adequada do mapa das MMU não soluciona o problema de que o espaço de endereçamento virtual é maior do que a memória física. Como temos apenas oito quadros de páginas físicas, apenas oito das páginas virtuais na Figura 3.9 estão mapeadas na memória física. As outras, mostradas com um X na figura, não estão mapeadas. No hardware real, um **bit Presente/ausente** controla quais páginas estão fisicamente presentes na memória.

O que acontece se o programa referencia um endereço não mapeado, por exemplo, usando a instrução

```
MOV REG,32780
```

a qual é o byte 12 dentro da página virtual 8 (começando em 32768)? A MMU observa que a página não está mapeada (o que é indicado por um X na figura) e faz a CPU desviar para o sistema operacional. Essa interrupção é chamada de **falta de página** (*page fault*). O sistema operacional escolhe um quadro de página pouco usado e escreve seu conteúdo de volta para o disco (se já não estiver ali). Ele então carrega (também do disco) a página recém-referenciada no quadro de página recém-liberado, muda o mapa e reinicia a instrução que causou a interrupção.

Por exemplo, se o sistema operacional decidiu escolher o quadro da página 1 para ser substituído, ele carregará a página virtual 8 no endereço físico 4096 e fará duas mudanças para o mapa da MMU. Primeiro, ele marcará a entrada da página 1 virtual como não mapeada, a fim de impedir quaisquer acessos futuros aos endereços virtuais entre 4096 e 8191. Então substituirá o X na entrada da página virtual 8 com um 1, assim, quando a instrução causadora da interrupção for reexecutada, ele mapeará os endereços virtuais 32780 para os endereços físicos 4108 (4096 + 12).

Agora vamos olhar dentro da MMU para ver como ela funciona e por que escolhemos usar um tamanho de página que é uma potência de 2. Na Figura 3.10 vimos um exemplo de um endereço virtual, 8196 (0010000000000100 em binário), sendo mapeado usando o mapa da MMU da Figura 3.9. O endereço virtual de 16 bits que chega à MMU está dividido em um número de página de 4 bits e um deslocamento de 12 bits. Com 4 bits para o número da página, podemos ter 16 páginas, e com 12 bits para o deslocamento, podemos endereçar todos os 4096 bytes dessa página.

O número da página é usado como um índice para a **tabela de páginas**, resultando no número do quadro de página correspondente àquela página virtual. Se o bit *Presente/ausente* for 0, ocorrerá uma interrupção para o sistema operacional. Se o bit for 1, o número do quadro de página encontrado na tabela de páginas é copiado para os três bits mais significativos para o registrador de saída, junto com o deslocamento de 12 bits, que é copiado sem modificações do endereço virtual de entrada. Juntos eles formam um endereço físico de 15 bits. O registrador de saída é então colocado no barramento de memória como o endereço de memória físico.

3.3.2 Tabelas de páginas

Em uma implementação simples, o mapeamento de endereços virtuais em endereços físicos pode ser resumido como a seguir: 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 das 16 páginas virtuais e os 12 bits inferiores especificariam então o deslocamento de bytes (0 a 4095) dentro da página selecionada. No entanto, uma divisão com 3 ou 5 ou algum outro número de bits para

a página também é possível. Divisões diferentes implicam tamanhos de páginas diferentes.

O número da página virtual é usado como um índice dentro da tabela de páginas para encontrar a entrada para essa página virtual. A partir da entrada da tabela de páginas, chega-se ao número do quadro (se ele existir). O número do quadro de página é colocado com os bits mais significativos do deslocamento, substituindo o número de página virtual, a fim de formar um endereço físico que pode ser enviado para a memória.

Assim, o propósito da tabela de páginas é mapear as páginas virtuais em quadros de páginas. Matematicamente falando, a tabela de páginas é uma função, com o número da página virtual como argumento e o número do quadro físico como resultado. Usando o resultado dessa função, o campo da página virtual em um endereço virtual pode ser substituído por um campo de quadro de página, desse modo formando um endereço de memória física.

Neste capítulo, nós nos preocupamos somente com a memória virtual e não com a virtualização completa. Em outras palavras: nada de máquinas virtuais ainda. Veremos no Capítulo 7 que cada máquina virtual exige sua própria memória virtual e, como resultado, a organização da tabela de páginas torna-se muito mais complicada, envolvendo tabelas de páginas sombreadas ou aninhadas e mais. Mesmo sem tais configurações arcanas, a paginação e a memória virtual são bastante sofisticadas, como veremos.

Estrutura de uma entrada da tabela de páginas

Vamos passar então da análise da estrutura das tabelas de páginas como um todo para os detalhes de uma única entrada da tabela de páginas. O desenho exato de uma entrada na tabela de páginas é altamente dependente da máquina, mas o tipo de informação presente é mais ou menos o mesmo de máquina para máquina. Na Figura 3.11 apresentamos uma amostra de entrada na tabela de páginas. O tamanho varia de computador para computador, mas 32 bits é um tamanho comum. O campo mais importante é o *Número do quadro de página*. Afinal, a meta do mapeamento de páginas é localizar esse valor. Próximo a ele, temos o bit *Presente/ausente*. Se esse bit for 1, a entrada é válida e pode ser usada. Se ele for 0, a página virtual à qual a entrada pertence não está atualmente na memória. Acessar uma entrada da tabela de páginas com esse bit em 0 causa uma falta de página.

FIGURA 3.10 A operação interna da MMU com 16 páginas de 4 KB.

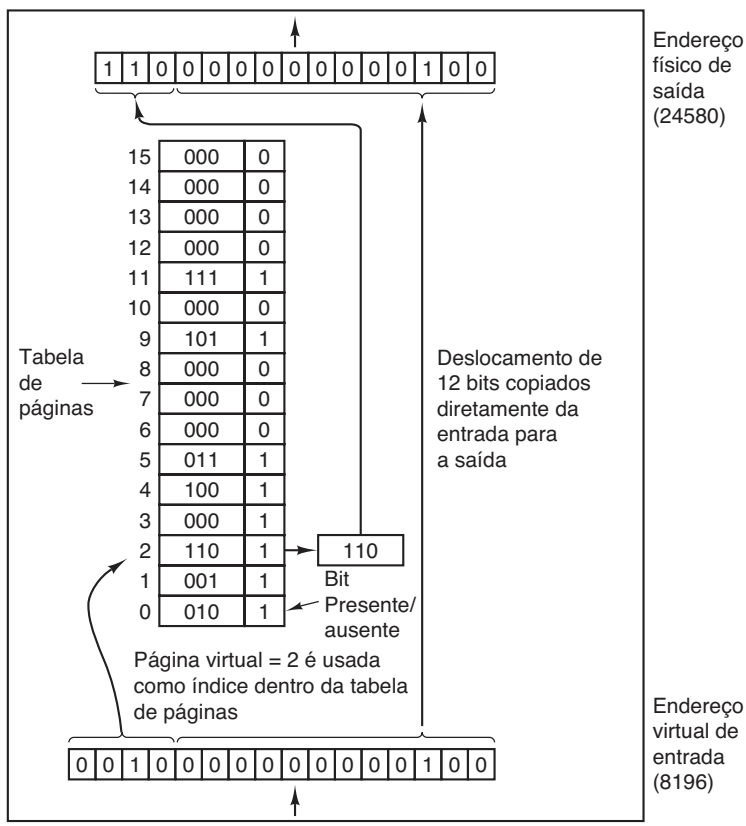
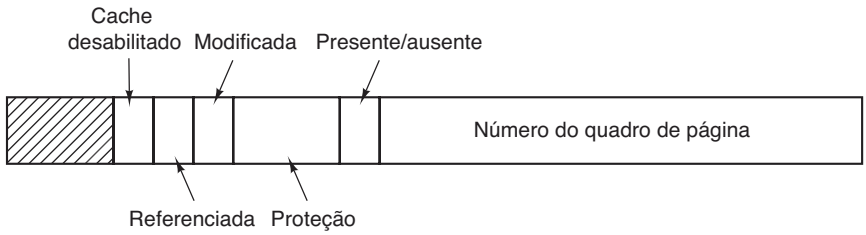


FIGURA 3.11 Uma entrada típica de uma tabela de páginas.



Os bits *Proteção* dizem quais tipos de acesso são permitidos. Na forma mais simples, esse campo contém 1 bit, com 0 para ler/escrever e 1 para ler somente. Um arranjo mais sofisticado é ter 3 bits, para habilitar a leitura, escrita e execução da página.

Os bits *Modificada* e *Referenciada* controlam o uso da página. Ao escrever na página, o hardware automaticamente configura o bit *Modificada*. Esse bit é importante quando o sistema operacional decide recuperar um quadro de página. Se a página dentro do quadro foi modificada (isto é, está "suja"), ela também deve ser atualizada no disco. Se ela não foi modificada (isto é, está "limpa"), ela pode ser abandonada, tendo em vista que a cópia em disco ainda é válida. O bit às vezes é chamado de **bit sujo**, já que ele reflete o estado da página.

O bit *Referenciada* é configurado sempre que uma página é referenciada, seja para leitura ou para escrita. Seu valor é usado para ajudar o sistema operacional a escolher uma página a ser substituída quando uma falta de página ocorrer. Páginas que não estão sendo usadas são candidatas muito melhores do que as páginas que estão sendo, e esse bit desempenha um papel importante em vários dos algoritmos de substituição de páginas que estudaremos posteriormente neste capítulo.

Por fim, o último bit permite que o mecanismo de cache seja desabilitado para a página. Essa propriedade é importante para páginas que mapeiam em registradores de dispositivos em vez da memória. Se o sistema operacional está parado em um laço estreito esperando que algum dispositivo de E/S responda a um comando que lhe foi dado, é fundamental que o hardware continue buscando a palavra do dispositivo, e não use uma cópia antiga da cache. Com esse bit, o mecanismo da cache pode ser desabilitado. Máquinas com espaços para E/S separados e que não usam E/S mapeada em memória não precisam desse bit.

Observe que o endereço de disco usado para armazenar a página quando ela não está na memória não faz parte da tabela de páginas. A razão é simples. A tabela de páginas armazena apenas aquelas informações de que o hardware precisa para traduzir um endereço virtual para um endereço físico. As informações que o sistema operacional precisa para lidar com faltas de páginas são mantidas em tabelas de software dentro do sistema operacional. O hardware não precisa dessas informações.

Antes de entrarmos em mais questões de implementação, vale a pena apontar de novo que o que a memória virtual faz em essência é criar uma nova abstração — o espaço de endereçamento — que é uma abstração da memória física, da mesma maneira que um processo é uma abstração do processador físico (CPU). A memória virtual pode ser implementada dividindo o espaço do endereço virtual em páginas e mapeando cada uma delas em algum quadro de página da memória física ou não as mapeando (temporariamente). Desse modo, ela diz respeito basicamente à abstração criada pelo sistema operacional e como essa abstração é gerenciada.

3.3.3 Acelerando a paginação

Acabamos de ver os princípios básicos da memória virtual e da paginação. É chegado o momento agora de entrar em maiores detalhes a respeito de possíveis implementações. Em qualquer sistema de paginação, duas questões fundamentais precisam ser abordadas:

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

O primeiro ponto é uma consequência do fato de que o mapeamento virtual-físico precisa ser feito em cada referência de memória. Todas as instruções devem em última análise vir da memória e muitas delas referenciam operandos na memória também. Em consequência, é preciso que se faça uma, duas, ou às vezes mais referências à tabela de páginas por instrução. Se a execução de uma instrução leva, digamos, 1 ns, a procura na tabela de páginas precisa 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 tornando-se a norma para computadores de mesa e laptops. Com um tamanho de página, digamos, de 4 KB, um espaço de endereço de 32 bits tem 1 milhão de páginas e um espaço de endereço de 64 bits tem mais do que você gostaria de contemplar. Com 1 milhão de páginas no espaço de endereço virtual, a tabela de página precisa ter 1 milhão de entradas. E lembre-se de que cada processo precisa da sua própria tabela de páginas (porque ele tem seu próprio espaço de endereço virtual).

A necessidade de mapeamentos extensos e rápidos é uma limitação muito significativa sobre como os computadores são construídos. O projeto mais simples (pelo menos conceitualmente) é ter uma única tabela de página consistindo de uma série de registradores de hardware rápidos, com uma entrada para cada página virtual, indexada pelo número da página virtual, como mostrado na Figura 3.10. Quando um processo é inicializado, o sistema operacional carrega os registradores com a tabela de páginas do processo, tirada de uma cópia mantida na memória principal. Durante a execução do processo, não são necessárias mais referências de memória para a tabela de páginas. As vantagens desse método são que ele é direto e não exige referências de memória durante o mapeamento. Uma desvantagem é que ele é terrivelmente caro se a tabela de páginas for grande; ele simplesmente não é prático na maioria das vezes. Outra desvantagem é que ter de carregar a tabela de páginas inteira em cada troca de contexto mataria completamente o desempenho.

No outro extremo, a tabela de página pode estar inteiramente na memória principal. Tudo o que o hardware precisa então é de um único registrador que aponte para o início da tabela de páginas. Esse projeto permite que o mapa virtual-físico seja modificado em uma troca de contexto através do carregamento de um registrador. É claro, ele tem a desvantagem de exigir uma ou mais referências de memória para ler as entradas na tabela de páginas durante a execução de cada instrução, tornando-a muito lenta.

TLB (Translation Lookaside Buffers) ou memória associativa

Vamos examinar agora esquemas amplamente implementados para acelerar a paginação e lidar com grandes espaços de endereços virtuais, começando com o primeiro tipo. O ponto de partida da maioria das técnicas de otimização é o fato de a tabela de páginas estar na memória. Potencialmente, esse esquema tem um impacto enorme sobre o desempenho. Considere, por exemplo, uma instrução de 1 byte que copia um registrador para outro. Na ausência da paginação, essa instrução faz apenas uma referência de memória, para buscar a instrução. Com a paginação, pelo menos uma referência de memória adicional será necessária, a fim de acessar a tabela de páginas. Dado que a velocidade de execução é geralmente limitada pela taxa na qual a CPU pode retirar instruções e dados da memória, ter de fazer duas referências de memória por cada uma reduz o desempenho pela metade. Sob essas condições, ninguém usaria a paginação.

Projetistas de computadores sabem desse problema há anos e chegaram a uma solução. Ela se baseia na observação de que a maioria dos programas tende a fazer um grande número de referências a um pequeno número de páginas, e não o contrário. Assim, apenas uma pequena fração das entradas da tabela de páginas é intensamente lida; o resto mal é usado.

A solução que foi concebida é equipar os computadores com um pequeno dispositivo de hardware para mapear endereços virtuais para endereços físicos sem ter de passar pela tabela de páginas. O dispositivo, chamado de **TLB (Translation Lookaside Buffer)** ou às vezes de **memória associativa**, está ilustrado na Figura 3.12. Ele normalmente está dentro da MMU e consiste em um pequeno número de entradas, oito neste exemplo, mas raramente mais do que 256. Cada entrada contém informações sobre uma página, incluindo o número da página virtual, um bit que é configurado quando a página é modificada, o código de proteção (ler/escrever/permissões de execução) e o quadro de página física na qual a página está localizada. Esses campos têm uma correspondência de um para um com os campos na tabela de páginas, exceto pelo número da página virtual, que não é necessário na tabela de páginas. Outro bit indica se a entrada é válida (isto é, em uso) ou não.

Um exemplo que poderia gerar a TLB da Figura 3.12 é um processo em um laço que abarque as páginas virtuais 19, 20 e 21, de maneira que essas entradas na TLB tenham códigos de proteção para leitura e execução. Os principais dados atualmente usados (digamos, um arranjo sendo processado) estão nas páginas 129 e 130. A página 140 contém os índices usados nos cálculos desse arranjo. Por fim, a pilha encontra-se nas páginas 860 e 861.

FIGURA 3.12 Uma TLB para acelerar a paginação.

Válida	Página virtual	Modificada	Proteção	Quadro de 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

Vamos ver agora como a TLB funciona. Quando um endereço virtual é apresentado para a MMU para tradução, o hardware primeiro confere para ver se o seu número de página virtual está presente na TLB comparando-o com todas as entradas simultaneamente (isto é, em paralelo). É necessário um hardware especial para realizar isso, que todas as MMUs com TLBs têm. Se uma correspondência válida é encontrada e o acesso não viola os bits de proteção, o quadro da página é tirado diretamente da TLB, sem ir à tabela de páginas. Se o número da página virtual estiver presente na TLB, mas a instrução estiver tentando escrever em uma página somente de leitura, uma falha de proteção é gerada.

O interessante é o que acontece quando o número da página virtual não está na TLB. A MMU detecta a ausência e realiza uma busca na tabela de páginas comum. Ela então destitui uma das entradas da TLB e a substitui pela entrada de tabela de páginas que acabou de ser buscada. Portanto, se a mesma página é usada novamente em seguida, da segunda vez ela resultará em uma presença de página em vez de uma ausência. Quando uma entrada é retirada da TLB, o bit modificado é copiado de volta na entrada correspondente da tabela de páginas na memória. Os outros valores já estão ali, exceto o bit de referência. Quando a TLB é carregada da tabela de páginas, todos os campos são trazidos da memória.

Gerenciamento da TLB por software

Até o momento, presumimos que todas as máquinas com memória virtual paginada têm tabelas de página reconhecidas pelo hardware, mais uma TLB. Nesse esquema, o gerenciamento e o tratamento das faltas de TLB são feitos inteiramente pelo hardware da MMU. Interrupções para o sistema operacional ocorrem apenas quando uma página não está na memória.

No passado, esse pressuposto era verdadeiro. No entanto, muitas máquinas RISC, incluindo o SPARC, MIPS e o HP PA (já abandonado), realizam todo esse gerenciamento de página em software. Nessas máquinas, as entradas de TLB são explicitamente carregadas pelo sistema operacional. Quando ocorre uma ausência de TLB, em vez de a MMU ir às tabelas de páginas para encontrar e buscar a referência de página necessária, ela apenas gera uma falha de TLB e joga o problema no colo do sistema operacional. O sistema deve encontrar a página, remover uma entrada da TLB, inserir uma nova e reiniciar a instrução que falhou. E, é claro, tudo isso deve ser feito em um punhado de instruções, pois ausências de TLB ocorrem com muito mais frequência do que faltas de páginas.

De maneira bastante surpreendente, se a TLB for moderadamente grande (digamos, 64 entradas) para reduzir a taxa de ausências, o gerenciamento de software da TLB acaba sendo aceitavelmente eficiente. O principal ganho aqui é uma MMU muito mais simples, o que libera uma área considerável no chip da CPU para caches e outros recursos que podem melhorar o desempenho. O gerenciamento da TLB por software é discutido por Uhlig et al. (1994).

Várias estratégias foram desenvolvidas muito tempo atrás para melhorar o desempenho em máquinas que realizam gerenciamento de TLB em software. Uma abordagem ataca tanto a redução de ausências de TLB quanto a redução do custo de uma ausência de TLB quando ela ocorre (BALA et al., 1994). Para reduzir as ausências de TLB, às vezes o sistema operacional pode usar sua intuição para descobrir quais páginas têm mais chance de serem usadas em seguida e para pré-carregar entradas para elas na TLB. Por exemplo, quando um processo cliente envia uma mensagem a um processo servidor na mesma máquina, é muito provável que o processo servidor terá de ser executado logo. Sabendo disso, enquanto processa a interrupção para realizar o send, o sistema também pode conferir para ver onde o código, os dados e as páginas da pilha do servidor estão e mapeá-los antes que tenham uma chance de causar falhas na TLB.

A maneira normal para processar uma ausência de TLB, seja em hardware ou em software, é ir até a tabela de páginas e realizar as operações de indexação para localizar a página referenciada. O problema em realizar essa busca em software é que as páginas que armazenam

a tabela de páginas podem não estar na TLB, o que causará faltas de TLB adicionais durante o processamento. Essas faltas podem ser reduzidas mantendo uma cache de software grande (por exemplo, 4 KB) de entradas em uma localização fixa cuja página seja sempre mantida na TLB. Ao conferir a primeira cache do software, o sistema operacional pode reduzir substancialmente as ausências de TLB.

Quando o gerenciamento da TLB por software é usado, é essencial compreender a diferença entre diversos tipos de ausências. Uma **ausência leve** (soft miss) ocorre quando a página referenciada não se encontra na TLB, mas está na memória. Tudo o que é necessário aqui é que a TLB seja atualizada. Não é necessário realizar E/S em um disco. Tipicamente uma ausência leve necessita de 10-20 instruções de máquina para lidar e pode ser concluída em alguns nanossegundos. Em comparação, uma **ausência completa** (hard miss) ocorre quando a página em si não está na memória (e, é claro, também não está na TLB). Um acesso de disco é necessário para trazer a página, o que pode levar vários milissegundos, dependendo do disco usado. Uma ausência completa é facilmente um milhão de vezes mais lenta que uma suave. Procurar o mapeamento na hierarquia da tabela de páginas é conhecido como um **passeio na tabela de páginas** (page table walk).

Na realidade, a questão é mais complicada ainda. Uma ausência não é somente leve ou completa. Algumas ausências são ligeiramente leves (ou mais completas) do que outras. Por exemplo, suponha que o passeio de página não encontre a página na tabela de páginas do processo e o programa incorra, portanto, em uma falta de página. Há três possibilidades. Primeiro, a página pode estar na realidade na memória, mas não na tabela de páginas do processo. Por exemplo, a página pode ter sido trazida do disco por outro processo. Nesse caso, não precisamos acessar o disco novamente, mas basta mapear a página de maneira apropriada nas tabelas de páginas. Essa é uma ausência bastante leve chamada **falta de página menor** (minor page fault). Segundo, uma **falta de página maior** (major page fault) ocorre se ela precisar ser trazida do disco. Terceiro, é possível que o programa apenas tenha acessado um endereço inválido e nenhum mapeamento precisa ser acrescentado à TLB. Nesse caso, o sistema operacional tipicamente mata o programa com uma **falta de segmentação**. Apenas nesse caso o programa fez algo errado. Todos os outros casos são automaticamente corrigidos pelo hardware e/ou o sistema operacional — ao custo de algum desempenho.

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 de tabela de páginas na memória original. Mas esse não é o único problema que precisamos combater. Outro problema é como lidar com espaços de endereços virtuais muito grandes. A seguir discutiremos duas maneiras de lidar com eles.

Tabelas de páginas multinível

Como uma primeira abordagem, considere o uso de uma **tabela de páginas multinível**. Um exemplo simples é mostrado na Figura 3.13. Na Figura 3.13(a) temos um endereço virtual de 32 bits que é dividido em um campo *PT1* de 10 bits, um campo *PT2* de 10 bits e um campo de *Deslocamento* de 12 bits. Dado que os deslocamentos são de 12 bits, as páginas são de 4 KB e há um total de 2^{20} delas.

O segredo para o uso do método da tabela de páginas multinível é evitar manter todas as tabelas de páginas na memória o tempo inteiro. Em particular, aquelas que não são necessárias não devem ser mantidas. Suponha, por exemplo, que um processo precise de 12 megabytes: os 4 megabytes da base da memória para o código do programa, os próximos 4 megabytes para os dados e os 4 megabytes do topo da memória para a pilha. Entre o topo dos dados e a parte de baixo da pilha há um espaço gigante que não é usado.

Na Figura 3.13(b) vemos como a tabela de páginas de dois níveis funciona. À esquerda vemos a tabela de páginas de nível 1, com 1024 entradas, correspondendo ao campo *PT1* de 10 bits. Quando um endereço virtual é apresentado à MMU, ele primeiro extrai o campo *PT1* e usa esse valor como um índice na tabela de páginas de nível 1. Cada uma dessas 1024 entradas representa 4M, pois todo o espaço de endereço virtual de 4 gigabytes (isto é, 32 bits) foi dividido em segmentos de 4096 bytes.

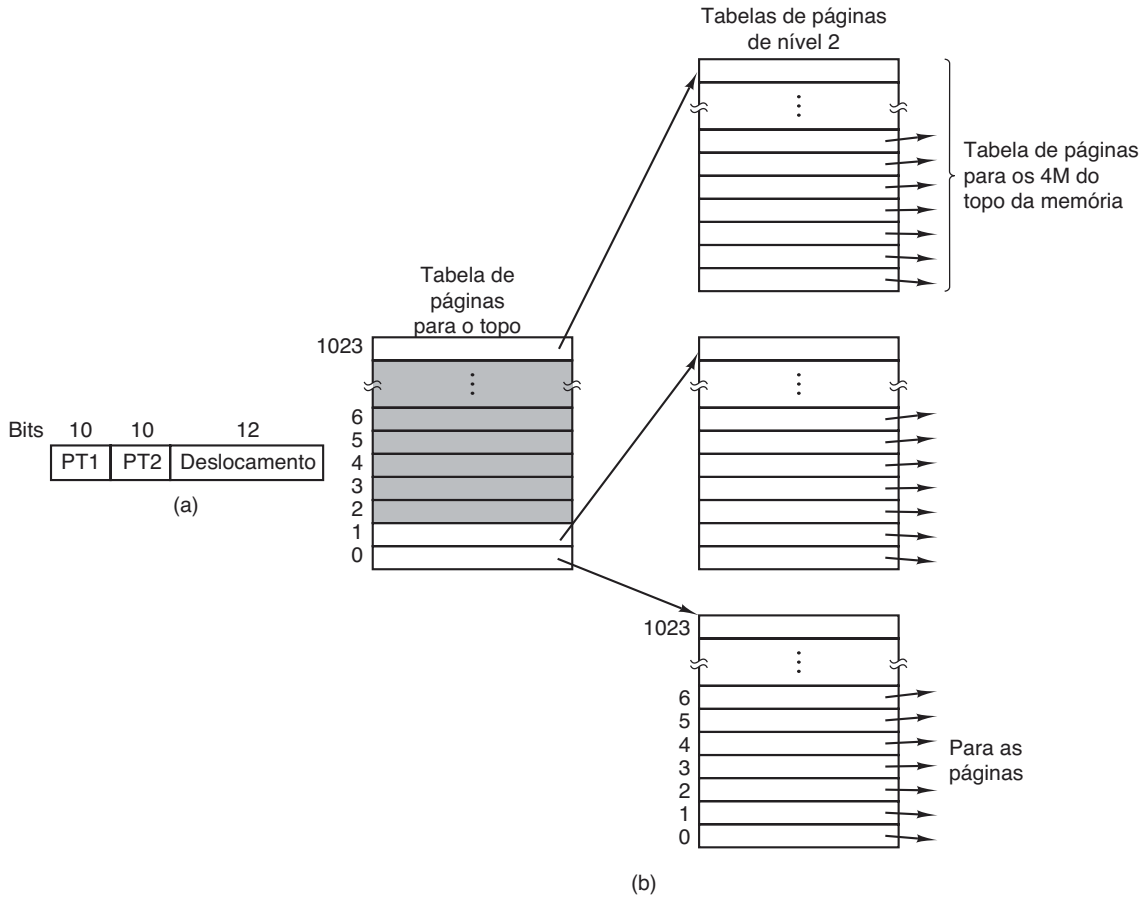
A entrada da tabela de páginas de nível 1, localizada através do campo *PT1* do endereço virtual, aponta para o endereço ou o número do quadro de página de uma tabela de páginas de nível 2. A entrada 0 da tabela de páginas de nível 1 aponta para a tabela de páginas relativa ao código do programa, a entrada 1 aponta para a tabela de páginas relativa aos dados e a entrada 1023 aponta para a tabela de páginas relativa à pilha. As outras entradas (sombreadas) não são usadas. O campo *PT2* é agora usado como um índice na tabela de páginas de nível 2 escolhida para encontrar o número do quadro de página correspondente.

Como exemplo, considere o endereço virtual de 32 bits 0x00403004 (4.206.596 em decimal), que corresponde a 12.292 bytes dentro do trecho dos dados. Esse endereço virtual corresponde a *PT1* = 1, *PT2* = 3 e *Deslocamento* = 4. A MMU primeiro usa o *PT1* com índice da tabela de páginas de nível 1 e obtém a entrada 1, que corresponde aos endereços de 4M a 8M − 1. Ela então usa *PT2* como índice para a tabela de páginas de nível 2 recém-encontrada e extrai a entrada 3, que corresponde aos endereços 12.228 a 16.383 dentro de seu pedaço de 4M (isto é, endereços absolutos 4.206.592 a 4.210.687). Essa entrada contém o número do quadro de página contendo o endereço virtual 0x00403004. Se essa página não está na memória, o bit *Presente/ausente* na entrada da tabela de páginas terá o valor zero, o que causará uma falta de página. Se a página estiver presente na memória, o número do quadro 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 para a memória.

O interessante a ser observado a respeito da Figura 3.13 é que, embora o espaço de endereço contenha mais de um milhão de páginas, apenas quatro tabelas de páginas são necessárias: a tabela de nível 1 e as três tabelas de nível 2 relativas aos endereços de 0 a 4M (para o código do programa), 4M a 8M (para os dados) e aos 4M do topo (para a pilha). Os bits *Presente/ausente* nas 1021 entradas restantes da página do nível superior são configurados para 0, forçando uma falta de página se um dia forem acessados. Se isso ocorrer, o sistema operacional notará que o processo está tentando referenciar uma memória que ele não deveria e tomará as medidas apropriadas, como enviar-lhe um sinal ou derrubá-lo. Nesse exemplo, escolhemos números arredondados para os vários tamanhos e escolhemos *PT1* igual a *PT2*, mas na prática outros valores também são possíveis, é claro.

O sistema de tabelas de páginas de dois níveis da Figura 3.13 pode ser expandido para três, quatro, ou mais níveis. Níveis adicionais proporcionam mais flexibilidade. Por exemplo, o processador 80.386 de 32 bits da Intel (lançado em 1985) era capaz de lidar com até 4 GB de memória, usando uma tabela de páginas de dois níveis, que consistia de um **diretório de páginas** cujas entradas apontavam para as tabelas de páginas, que, por sua

FIGURA 3.13 (a) Um endereço de 32 bits com dois campos de tabela de páginas. (b) Tabelas de páginas de dois níveis.



vez, apontavam para os quadros de página de 4 KB reais. Tanto o diretório de páginas quanto as tabelas de páginas continham 1024 entradas cada, dando um total de $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$ bytes endereçáveis, como desejado.

Dez anos mais tarde, o Pentium Pro introduziu outro nível: a **tabela de apontadores de diretórios de página** (page directory pointer table). Além disso, ele ampliou cada entrada em cada nível da hierarquia da tabela de páginas de 32 para 64 bits, então ele poderia endereçar memórias acima do limite de 4 GB. Como ele tinha apenas 4 entradas na tabela do apontador do diretório de páginas, 512 em cada diretório de páginas e 512 em cada tabela de páginas, o montante total de memória que ele podia endereçar ainda era limitado a um máximo de 4 GB. Quando o suporte de 64 bits apropriado foi acrescentado à família x86 (originalmente pelo AMD), o nível adicional *poderia* ter sido chamado de "apontador de tabelas de apontadores de diretórios de página" ou algo tão horrível quanto. Isso estaria perfeitamente de acordo com a maneira como os produtores de chips tendem a nomear as coisas. Ainda bem que não fizeram isso. A alternativa que apresentaram, "**mapa de página nível 4**", pode não ser um nome especialmente prático, mas pelo menos é mais curto e um pouco mais claro. De qualquer maneira, esses processadores agora usam todas as 512 entradas em todas as tabelas, resultando em uma quantidade de memória endereçável de $2^9 \times 2^9 \times 2^9 \times 2^9 \times 2^{12} = 2^{48}$ bytes. Eles poderiam ter adicionado outro nível, mas provavelmente acharam que 256 TB seriam suficientes por um tempo.

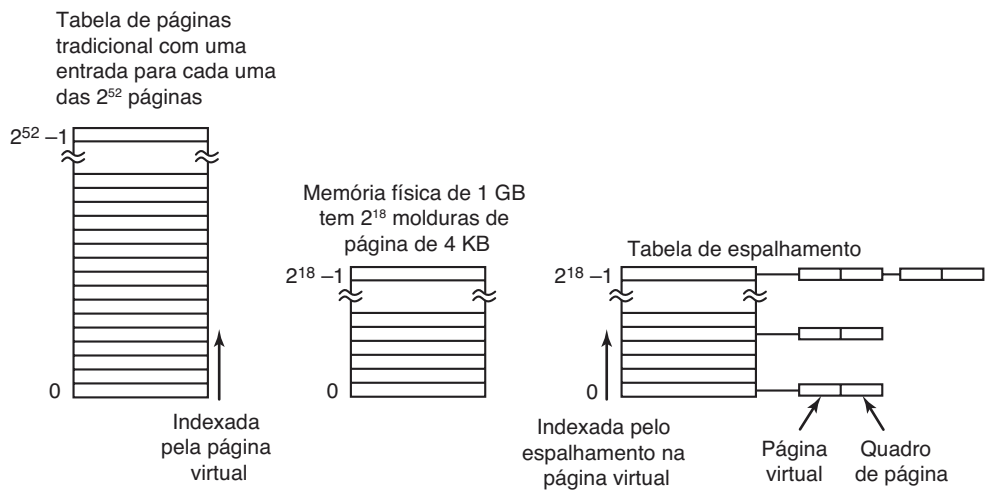
Tabelas de páginas invertidas

Uma alternativa para os níveis cada vez maiores em uma hierarquia de paginação é conhecida como **tabela de páginas invertidas**. Elas foram usadas pela primeira vez por processadores como o PowerPC, o UltraSPARC e o Itanium (às vezes referido como "Itanic", já que não foi realmente o sucesso que a Intel esperava). Nesse projeto, há apenas uma entrada por quadro de página na memória real, em vez de uma entrada por página de espaço de endereço virtual. Por exemplo, com os endereços virtuais de 64 bits, um tamanho de página de 4 KB e 4 GB de RAM, uma tabela de página invertida exige apenas 1.048.576 entradas. A entrada controla qual (processo, página virtual) está localizado na moldura da página.

Embora tabelas de páginas invertidas poupem muito espaço, pelo menos quando o espaço de endereço virtual é muito maior do que a memória física, elas têm um sério problema: a tradução virtual-física torna-se muito mais difícil. Quando o processo *n* referencia a página virtual *p*, o hardware não consegue mais encontrar a página física usando *p* como um índice para a tabela de páginas. Em vez disso, ele deve pesquisar a tabela de páginas invertidas inteira para uma entrada (*n, p*). Além disso, essa pesquisa deve ser feita em cada referência de memória, não apenas em faltas de páginas. Pesquisar uma tabela de 256K a cada referência de memória não é a melhor maneira de tornar sua máquina realmente rápida.

A saída desse dilema é fazer uso da TLB. Se ela conseguir conter todas as páginas intensamente usadas, a tradução pode acontecer tão rápido quanto com as tabelas de páginas regulares. Em uma ausência na TLB, no entanto, a tabela de página invertida tem de ser pesquisada em software. Uma maneira de realizar essa pesquisa é ter uma tabela de espalhamento (hash) nos endereços virtuais. Todas as páginas virtuais atualmente na memória que têm o mesmo valor de espalhamento são encadeadas juntas, como mostra a Figura 3.14. Se

FIGURA 3.14 Comparação de uma tabela de página tradicional com uma tabela de página invertida.



a tabela de encadeamento tiver o mesmo número de entradas que o número de páginas físicas da máquina, o encadeamento médio será de apenas uma entrada de comprimento, acelerando muito o mapeamento. Assim que o número do quadro de página for encontrado, a nova dupla (virtual, física) é inserida na TLB.

As 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 tabela de páginas é gigantesco. Por exemplo, com páginas de 4 MB e endereços virtuais de 64 bits, são necessárias 2^{42} entradas de tabelas de páginas. Outras abordagens para lidar com grandes memórias virtuais podem ser encontradas em Talluri et al. (1995).

3.4 Algoritmos de substituição de páginas

Quando ocorre uma falta de página, o sistema operacional tem de escolher uma página para remover da memória a fim de abrir espaço para a que está chegando. Se a página a ser removida foi modificada enquanto estava na memória, ela precisa ser reescrita para o disco a fim de atualizar a cópia em disco. Se, no entanto, ela não tiver sido modificada (por exemplo, ela contém uma página de código), a cópia em disco já está atualizada, portanto não é preciso reescrevê-la. A página a ser lida simplesmente sobrescreve a página que está sendo removida.

Embora seja possível escolher uma página ao acaso para ser descartada a cada falta de página, o desempenho do sistema será muito melhor se for escolhida uma página que não é intensamente usada. Se uma página intensamente usada for removida, ela provavelmente terá de ser trazida logo de volta, resultando em um custo extra. Muitos trabalhos, tanto teóricos quanto experimentais, têm sido feitos sobre o assunto dos algoritmos de substituição de páginas. A seguir descreveremos alguns dos mais importantes.

Vale a pena observar que o problema da "substituição de páginas" ocorre em outras áreas do projeto de computadores também. Por exemplo, a maioria dos computadores tem um ou mais caches de memória consistindo de blocos de memória de 32 ou 64 bytes. Quando a cache está cheia, algum bloco precisa ser escolhido para ser removido. Esse problema é precisamente o mesmo que ocorre na substituição de páginas, exceto em uma escala de tempo mais curta (ele precisa ser feito em alguns nanossegundos, não milissegundos como com a substituição de páginas). A razão para a escala de tempo mais curta é que as ausências do bloco na cache são satisfeitas a partir da memória principal, que não tem atrasos devido ao tempo de busca e de latência rotacional do disco.

Um segundo exemplo ocorre em um servidor da web. O servidor pode manter um determinado número de páginas da web intensamente usadas em sua cache de memória. No entanto, quando ela está cheia e uma nova página é referenciada, uma decisão precisa ser tomada a respeito de qual página na web remover. As considerações são similares a páginas de memória virtual, exceto que as da web jamais são modificadas na cache, então sempre há uma cópia atualizada "no disco". Em um sistema de memória virtual, as páginas na memória principal podem estar limpas ou sujas.

Em todos os algoritmos de substituição de páginas a serem estudados a seguir, surge a seguinte questão: quando uma página será removida da memória, ela deve ser uma das páginas do próprio processo que causou a falta ou pode ser uma pertencente a outro processo? No primeiro caso, estamos efetivamente limitando cada processo a um número fixo de páginas; no segundo, não. Ambas são possibilidades. Voltaremos a esse ponto na Seção 3.5.1.

3.4.1 O algoritmo ótimo de substituição de página

O algoritmo de substituição de página melhor possível é fácil de descrever, mas impossível de implementar de fato. Ele funciona deste modo: no momento em que ocorre uma falta de página, há um determinado conjunto de páginas na memória. Uma dessas páginas será referenciada na próxima instrução (a página contendo essa instrução). Outras páginas talvez não sejam referenciadas até 10, 100 ou talvez 1.000 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 que a página com o maior rótulo deve ser removida. Se uma página não vai ser usada para 8 milhões de instruções e outra página não vai ser usada para 6 milhões de instruções, remover a primeira adia ao máximo a próxima falta de página. Computadores, como as pessoas, tentam adiar ao máximo a ocorrência de eventos desagradáveis.

O único problema com esse algoritmo é que ele é irrealizável. No momento da falta de página, o sistema operacional não tem como saber quando cada uma das páginas será referenciada em seguida. (Vimos uma situação similar anteriormente com o algoritmo de escalonamento "tarefa mais curta primeiro"

— como o sistema pode dizer qual tarefa é a mais curta?) Mesmo assim, ao executar um programa em um simulador e manter um controle sobre todas as referências de páginas, é possível implementar o algoritmo ótimo na *segunda* execução usando as informações de referência da página colhidas durante a *primeira* execução.

Dessa maneira, é possível comparar o desempenho de algoritmos realizáveis com o do melhor possível. Se um sistema operacional atinge um desempenho de, digamos, apenas 1% pior do que o do algoritmo ótimo, o esforço investido em procurar por um algoritmo melhor resultará em uma melhora de no máximo 1%.

Para evitar qualquer confusão possível, é preciso deixar claro que esse registro de referências às páginas trata somente do programa recém-mensurado e então com apenas uma entrada específica. O algoritmo de substituição de página derivado dele é, então, específico àquele programa e dados de entrada. Embora esse método seja útil para avaliar algoritmos de substituição de página, ele não tem uso para sistemas práticos. A seguir, estudaremos algoritmos que *são* úteis em sistemas reais.

3.4.2 O algoritmo de substituição de páginas não usadas recentemente (NRU)

A fim de permitir que o sistema operacional colete estatísticas de uso de páginas úteis, a maioria dos computadores com memória virtual tem dois bits de status, *R* e *M*, associados com cada página. *R* é colocado sempre que a página é referenciada (lida ou escrita). *M* é colocado quando a página é escrita (isto é, modificada). Os bits estão contidos em cada entrada de tabela de página, como mostrado na Figura 3.11. É importante perceber que esses bits precisam ser atualizados em cada referência de memória, então é essencial que eles sejam atualizados pelo hardware. Assim que um bit tenha sido modificado para 1, ele fica em 1 até o sistema operacional reinicializá-lo em 0.

Se o hardware não tem esses bits, eles podem ser simulados usando os mecanismos de interrupção de relógio e falta de página do sistema operacional. Quando um processo é inicializado, todas as entradas de tabela de páginas são marcadas como não presentes na memória. Tão logo qualquer página é referenciada, uma falta de página vai ocorrer. O sistema operacional então coloca o bit *R* em 1 (em suas tabelas internas), muda a entrada da tabela de páginas para apontar para a página correta, com o modo SOMENTE LEITURA, e reinicializa a instrução. Se a página for subsequentemente modificada, outra falta de página vai ocorrer, permitindo que o sistema operacional coloque o bit *M* e mude o modo da página para LEITURA/ESCRITA.

Os bits *R* e *M* podem ser usados para construir um algoritmo de paginação simples como a seguir. Quando um processo é inicializado, ambos os bits de páginas para todas as suas páginas são definidos como 0 pelo sistema operacional. Periodicamente (por exemplo, em cada interrupção de relógio), o bit *R* é limpo, a fim de distinguir as páginas não referenciadas recentemente daquelas que foram.

Quando ocorre uma falta de página, o sistema operacional inspeciona todas as páginas e as divide em quatro categorias baseadas nos valores atuais de seus 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 de classe 1 pareçam, em um primeiro olhar, impossíveis, elas ocorrem quando uma página de classe 3 tem o seu bit *R* limpo por uma interrupção de relógio. Interrupções de relógio não limpam o bit *M* porque essa informação é necessária para saber se a página precisa ser reescrita para o disco ou não. Limpar *R*, mas não *M*, leva a uma página de classe 1.

O algoritmo **NRU** (**Not Recently Used** — não usada recentemente) remove uma página ao acaso de sua classe de ordem mais baixa que não esteja vazia. Implícito nesse algoritmo está a ideia de que é melhor remover uma página modificada, mas não referenciada, a pelo menos um tique do relógio (em geral em torno de 20 ms) do que uma página não modificada que está sendo intensamente usada. A principal atração do NRU é que ele é fácil de compreender, moderadamente eficiente de implementar e proporciona um desempenho que, embora não ótimo, pode ser adequado.

3.4.3 O algoritmo de substituição de páginas primeiro a entrar, primeiro a sair

Outro algoritmo de paginação de baixo custo é o **primeiro a entrar, primeiro a sair** (**first in, first out** — FIFO). Para ilustrar como isso funciona, considere um supermercado que tem prateleiras suficientes para exibir exatamente *k* produtos diferentes. Um dia, uma empresa introduz um novo alimento de conveniência — um iogurte orgânico, seco e congelado, de reconstituição instantânea em um forno de micro-ondas. É um sucesso imediato, então nosso supermercado finito tem de se livrar do produto antigo para estocá-lo.

Uma possibilidade é descobrir qual produto o supermercado tem estocado há mais tempo (isto é, algo que ele começou a vender 120 anos atrás) e se livrar dele supondo que ninguém mais se interessa. Na realidade, o supermercado mantém uma lista encadeada de todos os produtos que ele vende atualmente na ordem em que foram introduzidos. O produto novo vai para o fim da lista; o que está em primeiro na lista é removido.

Com um algoritmo de substituição de página, pode-se aplicar a mesma ideia. O sistema operacional mantém uma lista de todas as páginas atualmente na memória, com a chegada mais recente no fim e a mais antiga na frente. Em uma falta de página, a página da frente é removida e a nova página acrescentada ao fim da lista. Quando aplicado a lojas, FIFO pode remover a cera para bigodes, mas também pode remover a farinha, sal ou manteiga. Quando aplicado aos computadores, surge o mesmo problema: a página mais antiga ainda pode ser útil. Por essa razão, FIFO na sua forma mais pura raramente é usado.

3.4.4 O algoritmo de substituição de páginas segunda chance

Uma modificação simples para o FIFO que evita o problema de jogar fora uma página intensamente usada é inspecionar o bit *R* da página mais antiga. Se ele for 0, a página é velha e pouco utilizada, portanto é substituída imediatamente. Se o bit *R* for 1, o bit é limpo, e a página é colocada no fim da lista de páginas, e seu tempo de carregamento é atualizado como se ela tivesse recém-chegado na memória. Então a pesquisa continua.

A operação desse algoritmo, chamada de **segunda chance**, é mostrada na Figura 3.15. Na Figura 3.15(a) vemos as páginas *A* até *H* mantidas em uma lista encadeada e divididas pelo tempo que elas chegaram na memória.

Suponha que uma falta de página ocorra no instante 20. A página mais antiga é *A*, que chegou no instante 0, quando o processo foi inicializado. Se o bit *R* da página *A* for 0, ele será removido da memória, seja sendo escrito para o disco (se ele for sujo), ou simplesmente abandonado (se ele for limpo). Por outro lado, se o bit *R* for 1, *A* será colocado no fim da lista e seu "tempo de carregamento" será atualizado para o momento atual (20). O bit *R* é também colocado em 0. A busca por uma página adequada continua com *B*.

O que o algoritmo segunda chance faz é procurar por uma página antiga que não esteja referenciada no intervalo de relógio mais recente. Se todas as páginas foram referenciadas, a segunda chance degenera-se em um FIFO puro. Especificamente, imagine que todas as páginas na Figura 3.15(a) têm seus bits *R* em 1. Uma a uma, o sistema operacional as move para o fim da lista, zerando o bit *R* cada vez que ele anexa uma página ao fim da lista. Por fim, a lista volta à página *A*, que agora tem seu bit *R* zerado. Nesse ponto *A* é removida. Assim, o algoritmo sempre termina.

3.4.5 O algoritmo de substituição de páginas do relógio

Embora segunda chance seja um algoritmo razoável, ele é desnecessariamente ineficiente, pois ele está sempre movendo páginas em torno de sua lista. Uma abordagem melhor é manter todos os quadros de páginas em uma lista circular na forma de um relógio, como mostrado na Figura 3.16. Um ponteiro aponta para a página mais antiga.

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

FIGURA 3.15 Operação de segunda chance. (a) Páginas na ordem FIFO. (b) Lista de páginas se uma falta de página ocorrer no tempo 20 e o bit *R* de *A* possuir o valor 1. Os números acima das páginas são seus tempos de carregamento.

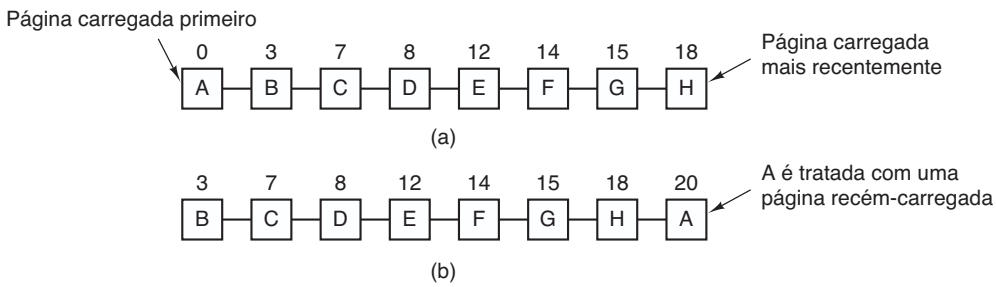
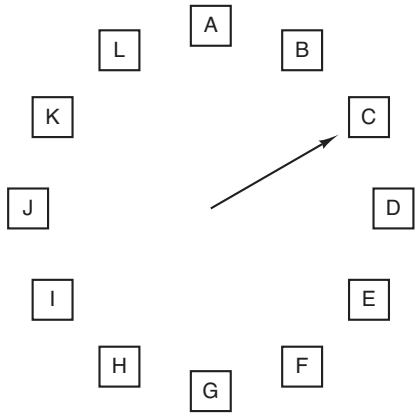


FIGURA 3.16 O algoritmo de substituição de páginas do relógio.



Quando ocorre uma falta de página, a página indicada pelo ponteiro é inspecionada. A ação executada depende do bit R:

R = 0: Remover a página
R = 1: Zerar R e avançar o ponteiro

3.4.6 Algoritmo de substituição de páginas usadas menos recentemente (LRU)

Uma boa aproximação para o algoritmo ótimo é baseada na observação de que as páginas que foram usadas intensamente nas últimas instruções provavelmente o serão em seguida de novo. De maneira contrária, páginas que não foram usadas há eras provavelmente seguirão sem ser utilizadas por um longo tempo. Essa ideia sugere um algoritmo realizável: quando ocorre uma falta de página, jogue fora aquela que não tem sido usada há mais tempo. Essa estratégia é chamada de paginação **LRU** (**Least Recently Used** — usada menos recentemente).

Embora o LRU seja teoricamente realizável, ele não é nem um pouco barato. Para se implementar por completo o LRU, é necessário que seja mantida uma lista encadeada de todas as páginas na memória, com a página mais recentemente usada na frente e a menos recentemente usada na parte de trás. A dificuldade é que a lista precisa ser atualizada a cada referência de memória. Encontrar uma página na lista, deletá-la e então movê-la para a frente é uma operação que demanda muito tempo, mesmo em hardware (presumindo que um hardware assim possa ser construído).

No entanto, há outras maneiras de se implementar o LRU com hardwares especiais. Primeiro, vamos considerar a maneira mais simples. Esse método exige equipar o hardware com um contador de 64 bits, *C*, que é automaticamente incrementado após cada instrução. Além disso, cada entrada da tabela de páginas também deve ter um campo grande o suficiente para conter o contador. Após cada referência de memória, o valor atual de *C* é armazenado na entrada da tabela de páginas para a página recém-referenciada. Quando ocorre uma falta de página, o sistema operacional examina todos os contadores na tabela de página para encontrar a mais baixa. Essa página é a usada menos recentemente.

3.4.7 Simulação do LRU em software

Embora o algoritmo de LRU anterior seja (em princípio) realizável, poucas máquinas, se é que existe alguma, têm o hardware necessário. Em vez disso, é necessária uma solução que possa ser implementada em software. Uma possibilidade é o algoritmo de substituição de páginas não usadas frequentemente (**NFU** — **Not Frequently Used**). A implementação exige um contador de software associado com cada página, de início zero. A cada interrupção de relógio, o sistema operacional percorre todas as páginas na memória. Para cada página, o bit *R*, que é 0 ou 1, é adicionado ao contador. Os contadores controlam mais ou menos quão frequentemente cada página foi referenciada. Quando ocorre uma falta de página, aquela com o contador mais baixo é escolhida para substituição.

O principal problema com o NFU é que ele lembra um elefante: jamais esquece nada. Por exemplo, em um compilador de múltiplos passos, as páginas que foram intensamente usadas durante o passo 1 podem ainda ter um contador alto bem adiante. Na realidade, se o passo 1 possuir o tempo de execução mais longo de todos os passos, as páginas contendo o código para os passos subsequentes poderão ter sempre contadores menores do que as páginas do passo 1. Em consequência, o sistema operacional removerá as páginas úteis em vez das que não estão mais sendo usadas.

Felizmente, uma pequena modificação no algoritmo NFU possibilita uma boa simulação do LRU. A modificação tem duas partes. Primeiro, os contadores são deslocados um bit à direita antes que o bit *R* seja acrescentado. Segundo, o bit *R* é adicionado ao bit mais à esquerda em vez do bit mais à direita.

A Figura 3.17 ilustra como o algoritmo modificado, conhecido como **algoritmo de envelhecimento**, funciona. 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, configurando seus bits *R* para 1, enquanto os outros seguiram em 0. Após os seis contadores correspondentes terem sido deslocados e o bit *R* inserido à esquerda, eles têm 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, é removida a página cujo contador é o mais baixo. É claro que a página que não tiver sido referenciada por, digamos, quatro interrupções de relógio, terá quatro zeros no seu contador e, desse modo, terá um valor mais baixo do que um contador que não foi referenciado por três interrupções de relógio.

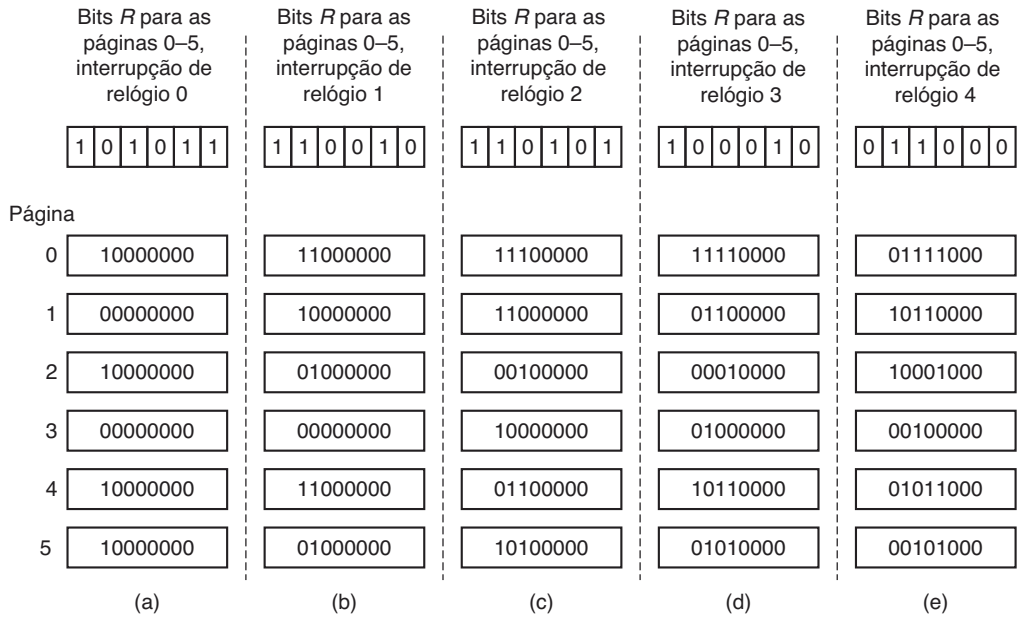
Esse algoritmo difere do LRU de duas maneiras importantes. Considere as páginas 3 e 5 na Figura 3.17(e). Nenhuma delas foi referenciada por duas interrupções de relógio; ambas foram referenciadas na interrupção anterior a elas. De acordo com o LRU, se uma página precisa ser substituída, devemos escolher uma dessas duas. O problema é que não sabemos qual delas foi referenciada por último no intervalo entre a interrupção 1 e a interrupção 2. Ao registrar apenas 1 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 também foi referenciada duas interrupções antes e a 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 nesse exemplo), o que limita seu horizonte passado. Suponha que duas páginas cada tenham um valor de contador de 0. Tudo o que podemos fazer é escolher uma delas ao acaso. Na realidade, é bem provável que uma das páginas tenha sido referenciada nove intervalos atrás e a outra, há 1.000 intervalos. Não temos como ver isso. Na prática, no entanto, 8 bits geralmente é o suficiente se uma interrupção de relógio for de em torno de 20 ms. Se uma página não foi referenciada em 160 ms, ela provavelmente não é importante.

3.4.8 O algoritmo de substituição de páginas do conjunto de trabalho

Na forma mais pura de paginação, os processos são inicializados sem nenhuma de suas páginas na memória. Tão logo a CPU tenta buscar a primeira instrução, ela detecta uma falta de página, fazendo que o sistema operacional traga a página contendo a primeira instrução. Outras faltas de páginas para variáveis globais e a pilha geralmente ocorrem logo em seguida. Após um tempo, o processo tem a maior parte das páginas que ele precisa para ser executado com relativamente poucas faltas de páginas. Essa estratégia é chamada de **paginação por demanda**, pois

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 por (a) a (e).



as páginas são carregadas apenas sob demanda, não antecipadamente.

É claro, é bastante fácil escrever um programa de teste que sistematicamente leia todas as páginas em um grande espaço de endereçamento, causando tantas faltas de páginas que não há memória suficiente para conter todas elas. Felizmente, a maioria dos processos não funciona desse jeito. Eles apresentam uma **localidade de referência**, significando que durante qualquer fase de execução o processo referencia apenas uma fração relativamente pequena das suas páginas. Cada passo de um compilador de múltiplos passos, por exemplo, referencia apenas uma fração de todas as páginas, e a cada passo essa fração é diferente.

O conjunto de páginas que um processo está atualmente usando é o seu **conjunto de trabalho** (DENNING, 1968a; DENNING, 1980). Se todo o conjunto de trabalho está na memória, o processo será executado sem causar muitas faltas até passar para outra fase de execução (por exemplo, o próximo passo do compilador). Se a memória disponível é pequena demais para conter todo o conjunto de trabalho, o processo causará muitas faltas de páginas e será executado lentamente, já que executar uma instrução leva alguns nanossegundos e ler em uma página a partir do disco costuma levar 10 ms. A um ritmo de uma ou duas instruções por 10 ms, seria necessária uma eternidade para terminar. Um programa causando faltas de páginas a todo o momento está **ultrapaginando** (**thrashing**) (DENNING, 1968b).

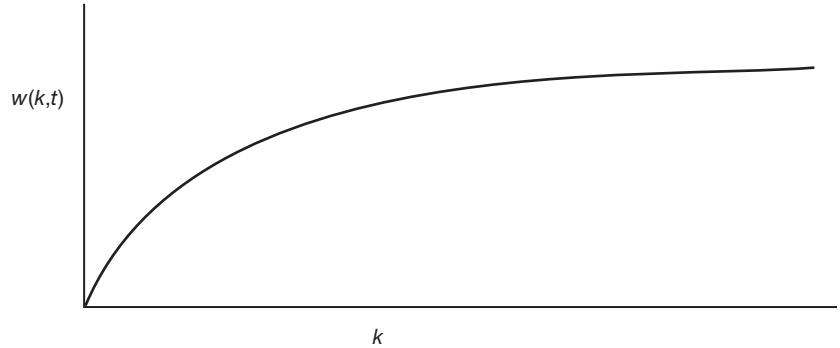
Em um sistema de multiprogramação, os processos muitas vezes são movidos para o disco (isto é, todas suas páginas são removidas da memória) para deixar que os outros tenham sua vez na CPU. A questão surge do que fazer quando um processo é trazido de volta outra vez. Tecnicamente, nada precisa ser feito. O processo simplesmente causará faltas de páginas até que seu conjunto de trabalho tenha sido carregado. O problema é que ter inúmeras faltas de páginas toda vez que um processo é carregado é algo lento, e também desperdiça um tempo considerável de CPU, visto que o sistema operacional leva alguns milissegundos de tempo da CPU para processar uma falta de página.

Portanto, muitos sistemas de paginação tentam controlar o conjunto de trabalho de cada processo e certificar-se de que ele está na memória antes de deixar o processo ser executado. Essa abordagem é chamada de **modelo do conjunto de trabalho** (DENNING, 1970). Ele foi projetado para reduzir substancialmente o índice de faltas de páginas. Carregar as páginas *antes* de deixar um processo ser executado também é chamado de **pré-paginação**. Observe que o conjunto de trabalho muda com o passar do tempo.

Há muito tempo se sabe que os programas raramente referenciam seu espaço de endereçamento de modo uniforme, mas que as referências tendem a agrupar-se em um pequeno número de páginas. Uma referência de memória pode buscar uma instrução ou dado, ou ela pode armazenar dados. Em qualquer instante de tempo, t, existe um conjunto consistindo de todas as páginas usadas pelas k referências de memória mais recentes. Esse conjunto, $w(k, t)$, é o conjunto de trabalho. Como todas as $k = 1$ referências mais recentes precisam ter utilizado páginas que tenham sido usadas pelas $k > 1$ referências mais recentes, e possivelmente outras, $w(k, t)$ é uma função monoliticamente não decrescente como função de k. À medida que k torna-se grande, o limite de $w(k, t)$ é finito, pois um programa não pode referenciar mais páginas do que o seu espaço de endereçamento contém, e poucos programas usarão todas as páginas. A Figura 3.18 descreve o tamanho do conjunto de trabalho como uma função de k.

O fato de que a maioria dos programas acessa aleatoriamente um pequeno número de páginas, mas que

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



esse conjunto muda lentamente com o tempo, explica o rápido crescimento inicial da curva e então o crescimento muito mais lento para o k maior. Por exemplo, um programa que está executando um laço ocupando duas páginas e acessando dados de quatro páginas pode referenciar todas as seis páginas a cada 1.000 instruções, mas a referência mais recente a alguma outra página pode ter sido um milhão de instruções antes, durante a fase de inicialização. Por esse comportamento assintótico, o conteúdo do conjunto de trabalho não é sensível ao valor de k escolhido. Colocando a questão de maneira diferente, existe uma ampla gama de valores de k para os quais o conjunto de trabalho não é alterado. Como o conjunto de trabalho varia lentamente com o tempo, é possível fazer uma estimativa razoável sobre quais páginas serão necessárias quando o programa for reiniciado com base em seu conjunto de trabalho quando foi parado pela última vez. A pré-paginação consiste em carregar essas páginas antes de reiniciar o processo.

Para implementar o modelo do conjunto de trabalho, é necessário que o sistema operacional controle quais páginas estão nesse conjunto. Ter essa informação leva imediatamente também a um algoritmo de substituição de página possível: quando ocorre uma falta de página, ele encontra uma página que não esteja no conjunto de trabalho e a remove. Para implementar esse algoritmo, precisamos de uma maneira precisa de determinar quais páginas estão no conjunto de trabalho. Por definição, o conjunto de trabalho é o conjunto de páginas usadas nas k mais recentes referências à memória (alguns autores usam as k mais recentes referências às páginas, mas a escolha é arbitrária). A fim de implementar qualquer algoritmo do conjunto de trabalho, algum valor de k deve ser escolhido antecipadamente. Então, após cada referência de memória, o conjunto de páginas usado pelas k mais recentes referências à memória é determinado de modo único.

É claro, ter uma definição operacional do conjunto de trabalho não significa que há uma maneira eficiente de calculá-lo durante a execução do programa. Seria possível se imaginar um registrador de deslocamento de comprimento k, com cada referência de memória deslocando esse registrador de uma posição à esquerda e inserindo à direita o número da página referenciada mais recentemente. O conjunto de todos os k números no registrador de deslocamento seria o conjunto de trabalho. Na teoria, em uma falta de página, o conteúdo de um registrador de deslocamento poderia ser lido e ordenado. Páginas duplicadas poderiam, então, ser removidas. O resultado seria o conjunto de trabalho. No entanto, manter o registrador de deslocamento e processá-lo em uma falta de página teria um custo proibitivo, então essa técnica nunca é usada.

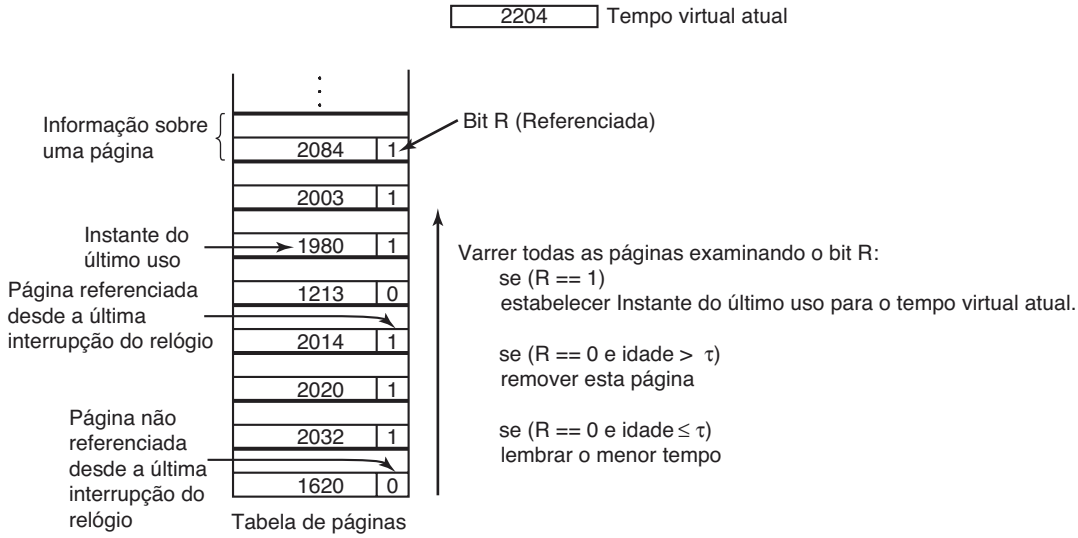
Em vez disso, várias aproximações são usadas. Uma delas é abandonar a ideia da contagem das últimas k referências de memória e em vez disso usar o tempo de execução. Por exemplo, em vez de definir o conjunto de trabalho como aquelas páginas usadas durante as últimas 10 milhões de referências de memória, podemos defini-lo como o conjunto de páginas usado durante os últimos 100 ms do tempo de execução. Na prática, tal definição é tão boa quanto e muito mais fácil de usar. Observe que para cada processo apenas seu próprio tempo de execução conta. Desse modo, se um processo começa a ser executado no tempo T e teve 40 ms de tempo de CPU no tempo real $T + 100$ ms, para fins de conjunto de trabalho, seu tempo é 40 ms. A quantidade de tempo de CPU que um processo realmente usou desde que foi inicializado é muitas vezes chamada de seu **tempo virtual atual**. Com essa aproximação, o conjunto de trabalho de um processo é o conjunto de páginas que ele referenciou durante os últimos τ segundos de tempo virtual.

Agora vamos examinar um algoritmo de substituição de página com base no conjunto de trabalho. A ideia básica é encontrar uma página que não esteja no conjunto de trabalho e removê-la. Na Figura 3.19 vemos um trecho de uma tabela de páginas para alguma máquina. Como somente as páginas localizadas na memória são consideradas candidatas à remoção, as que estão ausentes da memória são ignoradas por esse algoritmo. Cada entrada contém (ao menos) dois itens fundamentais de informação: o tempo (aproximado) que a página foi usada pela última vez e o bit R (Referenciada). Um retângulo branco vazio simboliza os outros campos que não são necessários para esse algoritmo, como o número do quadro de página, os bits de proteção e o bit M (modificada).

O algoritmo funciona da seguinte maneira: supõe-se que o hardware inicializa os bits R e M, como já discutido. De modo similar, presume-se que uma interrupção periódica de relógio ative a execução do software que limpa o bit *Referenciada* em cada tique do relógio. A cada falta de página, a tabela de páginas é varrida à procura de uma página adequada para ser removida.

À medida que cada entrada é processada, o bit R é examinado. Se ele for 1, o tempo virtual atual é escrito no campo *Instante de último uso* na tabela de páginas, indicando que a página estava sendo usada no momento em que a falta ocorreu. Tendo em vista que a página foi referenciada durante a interrupção de relógio atual, ela claramente está no conjunto de trabalho e não é candidata a ser removida (supõe-se que τ corresponda a múltiplas interrupções de relógio).

FIGURA 3.19 Algoritmo do conjunto de trabalho.



Se *R* é 0, a página não foi referenciada durante a interrupção de relógio atual e pode ser candidata à remoção. Para ver se ela deve ou não ser removida, sua idade (o tempo virtual atual menos seu *Instante de último uso*) é calculada e comparada a τ. Se a idade for maior que τ, a página não está mais no conjunto de trabalho e a página nova a substitui. A atualização das entradas restantes é continuada.

No entanto, se *R* é 0 mas a idade é menor do que ou igual a τ, a página ainda está no conjunto de trabalho. A página é temporariamente poupada, mas a página com a maior idade (menor valor de *Instante do último uso*) é marcada. Se a tabela inteira for varrida sem encontrar uma candidata para remover, isso significa que todas as páginas estão no conjunto de trabalho. Nesse caso, se uma ou mais páginas com $R = 0$ forem encontradas, a que tiver a maior idade será removida. Na pior das hipóteses, todas as páginas foram referenciadas durante a interrupção de relógio atual (e, portanto, todas com $R = 1$), então uma é escolhida ao acaso para ser removida, preferivelmente uma página limpa, se houver uma.

3.4.9 O algoritmo de substituição de página WSClock

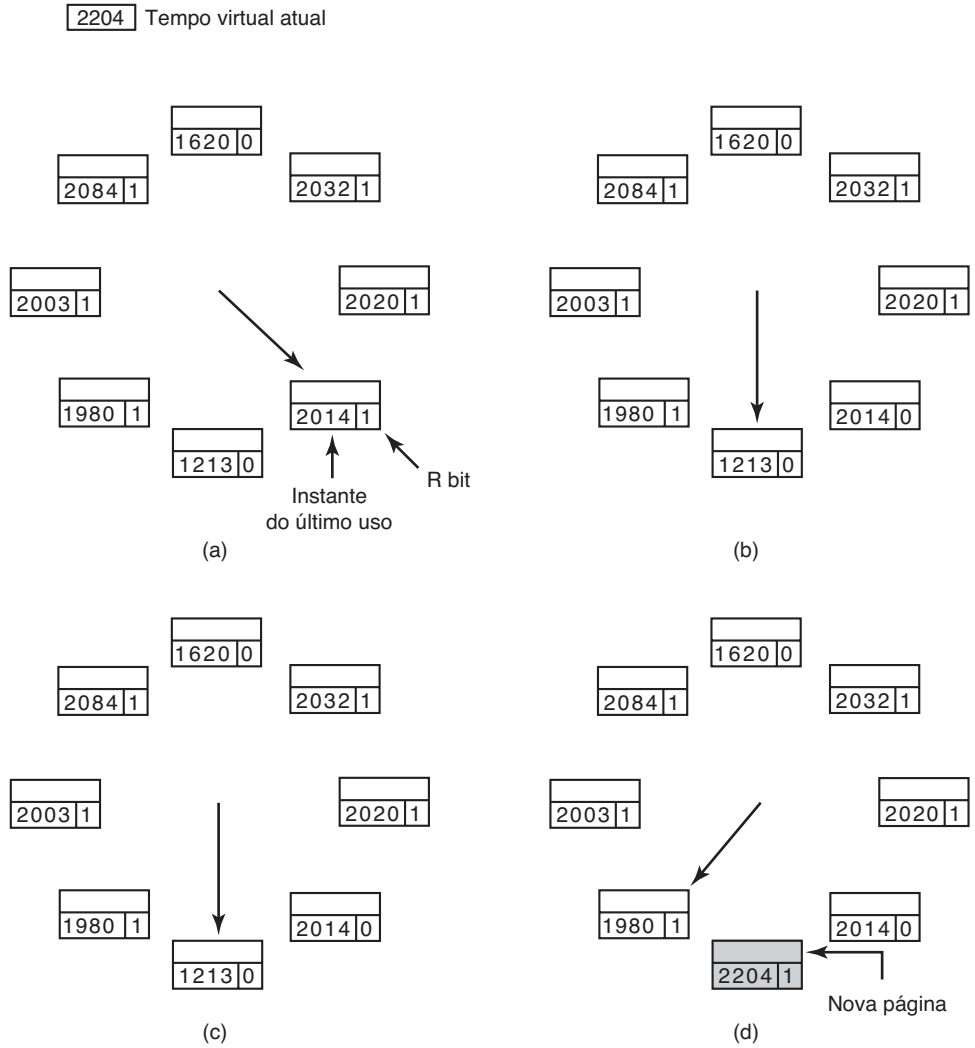
O algoritmo básico do conjunto de trabalho é enfadonho, já que a tabela de páginas inteira precisa ser varrida a cada falta de página até que uma candidata adequada seja localizada. Um algoritmo melhorado, que é baseado no algoritmo de relógio mas também usa a informação do conjunto de trabalho, é chamado de **WSClock** (CARR e HENNESSEY, 1981). Por sua simplicidade de implementação e bom desempenho, ele é amplamente usado na prática.

A estrutura de dados necessária é uma lista circular de quadros de páginas, como no algoritmo do relógio, e como mostrado na Figura 3.20(a). De início, essa lista está vazia. Quando a primeira página é carregada, ela é adicionada à lista. À medida que mais páginas são adicionadas, elas vão para a lista para formar um anel. Cada entrada contém o campo do *Instante do último uso* do algoritmo do conjunto de trabalho básico, assim 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 que estiver sendo apontada é examinada primeiro. Se o bit *R* for 1, a página foi usada durante a interrupção de relógio atual, então ela não é uma candidata ideal para ser removida. O bit *R* é então colocado em 0, o ponteiro avança para a próxima página, e o algoritmo é repetido para aquela página. O estado após essa sequência de eventos é mostrado na Figura 3.20(b).

Agora considere o que acontece se a página apontada tem $R = 0$, como mostrado na Figura 3.20(c). Se a idade é maior do que τ e a página está limpa, ela não está no conjunto de trabalho e uma cópia válida existe no disco. O quadro da página é simplesmente reivindicado e a nova página colocada lá, como mostrado na Figura 3.20(d). Por outro lado, se a página está suja, ela não pode ser reivindicada imediatamente, pois nenhuma cópia válida está presente no disco. Para evitar um chaveamento de processo, a escrita em disco é escalonada, mas o ponteiro é avançado e o algoritmo continua com a página seguinte. Afinal, pode haver uma página velha e limpa mais adiante e que pode ser usada imediatamente.

FIGURA 3.20 Operação 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 ciclo do relógio. Para reduzir o tráfego de disco, um limite pode ser estabelecido, permitindo que um máximo de *n* páginas sejam reescritas. Uma vez que esse limite tenha sido alcançado, não serão escalonadas mais escritas novas.

O que acontece se o ponteiro deu uma volta completa e voltou ao seu ponto de partida? Há dois casos que precisamos considerar:

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

No primeiro caso, o ponteiro apenas continua a se mover, procurando por uma página limpa. Dado que uma ou mais escritas foram escalonadas, eventualmente alguma escrita será completada e a sua página marcada como limpa. A primeira página limpa encontrada é removida. Essa página não é necessariamente a primeira escrita escalonada porque o driver do disco pode reordenar escritas a fim de otimizar o desempenho do disco.

No segundo caso, todas as páginas estão no conjunto de trabalho, de outra maneira pelo menos uma escrita teria sido escalonada. Por 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 não existir nenhuma, então a página atual é escolhida como a vítima e será reescrita em disco.

3.4.10 Resumo dos algoritmos de substituição de página

Examinamos até agora uma variedade de algoritmos de substituição de página. Agora iremos resumi-los brevemente. A lista de algoritmos discutidos está na Figura 3.21.

FIGURA 3.21 Algoritmos de substituição de páginas discutidos no texto.

Algoritmo	Comentário
Ótimo	Não implementável, mas útil como um padrão de desempenho
NRU (não usado 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 usado)	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

O algoritmo ótimo remove a página que será referenciada por último. Infelizmente, não há uma maneira para determinar qual página será essa, então, na prática, esse algoritmo não pode ser usado. No entanto, ele é útil como uma medida-padrão pela qual outros algoritmos podem ser mensurados.

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. Esse algoritmo é fácil de implementar, mas é muito rudimentar. Há outros melhores.

O algoritmo FIFO controla a ordem pela qual as páginas são carregadas na memória mantendo-as em uma lista encadeada. Remover a página mais antiga, então, torna-se trivial, mas essa página ainda pode estar sendo usada, de maneira que o FIFO é uma má escolha.

O algoritmo segunda chance é uma modificação do FIFO que confere se uma página está sendo usada antes de removê-la. Se ela estiver, a página é poupada. Essa modificação melhora muito o desempenho. O algoritmo do relógio é simplesmente uma implementação diferente do algoritmo segunda chance. Ele tem as mesmas propriedades de desempenho, mas leva um pouco menos de tempo para executar o algoritmo.

O LRU é um algoritmo excelente, mas não pode ser implementado sem um hardware especial. Se o hardware não estiver disponível, ele não pode ser usado. O NFU é uma tentativa rudimentar, não muito boa, de aproximação do LRU. No entanto, o algoritmo do envelhecimento é uma aproximação muito melhor do LRU e pode ser implementado de maneira eficiente. Trata-se de uma boa escolha.

Os últimos dois algoritmos usam o conjunto de trabalho. O algoritmo do conjunto de trabalho proporciona um desempenho razoável, mas é de certa maneira caro de ser implementado. O WSClock é uma variante que não apenas proporciona um bom desempenho, como também é eficiente de ser implementado.

Como um todo, os dois melhores algoritmos são o do envelhecimento e o WSClock. Eles são baseados no LRU e no conjunto de trabalho, respectivamente. Ambos proporcionam um bom desempenho de paginação e podem ser implementados eficientemente. Alguns outros bons algoritmos existem, mas esses dois provavelmente são os mais importantes na prática.

3.5 Questões de projeto para sistemas de paginação

Nas seções anteriores explicamos como a paginação funciona e introduzimos alguns algoritmos de substituição de página básicos. Mas conhecer os mecanismos básicos não é o suficiente. Para projetar um sistema e fazê-lo funcionar bem, você precisa saber bem mais. É como a diferença entre saber como mover a torre, o cavalo e o bispo, e outras peças do xadrez, e ser um bom jogador. Nas seções seguintes, examinaremos outras questões que os projetistas de sistemas operacionais têm de considerar cuidadosamente a fim de obter um bom desempenho de um sistema de paginação.

3.5.1 Políticas de alocação local *versus* global

Nas seções anteriores discutimos vários algoritmos de escolha da página a ser substituída quando ocorresse

uma falta. Uma questão importante associada com essa escolha (cuja discussão varremos cuidadosamente para baixo do tapete até agora) é sobre como a memória deve ser alocada entre os processos concorrentes em execução.

Dê uma olhada na Figura 3.22(a). Nessa figura, três processos, *A, B* e *C*, compõem o conjunto dos processos executáveis. Suponha que *A* tenha uma falta de página. O algoritmo de substituição de página deve tentar encontrar a página usada menos recentemente considerando apenas as seis páginas atualmente alocadas para *A*, ou ele deve considerar todas as páginas na memória? Se ele considerar somente as páginas de *A*, a página com o menor valor de idade será *A5*, de modo que obteremos a situação da Figura 3.22(b).

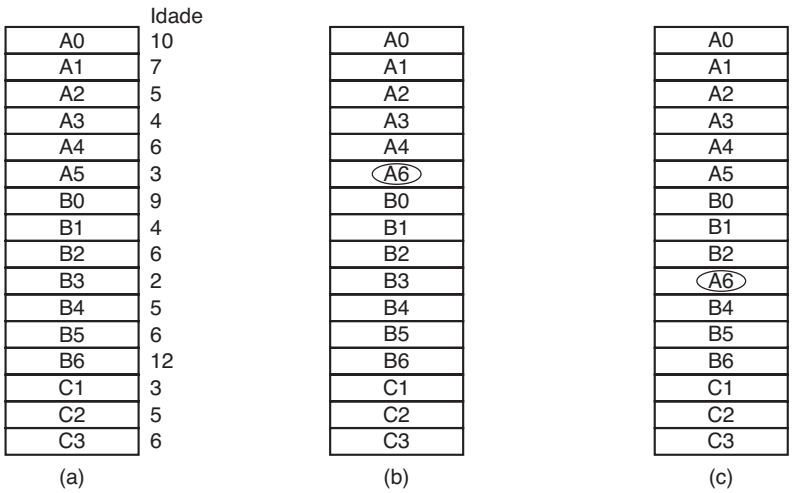
Por outro lado, se a página com o menor valor de idade for removida sem levar em conta a quem pertence, a página *B3* será escolhida e teremos a situação da Figura 3.22(c). O algoritmo da Figura 3.22(b) é um algoritmo de substituição de página **local**, enquanto o da Figura 3.22(c) é um algoritmo **global**. Algoritmos locais efetivamente correspondem a alocar a todo processo uma fração fixa da memória. Algoritmos globais alocam dinamicamente quadros de páginas entre os processos executáveis. Desse modo, o número de quadros de páginas designadas a cada processo varia com o tempo.

Em geral, algoritmos globais funcionam melhor, especialmente quando o tamanho do conjunto de trabalho puder variar muito através do tempo de vida de um processo. Se um algoritmo local for usado e o conjunto de trabalho crescer, resultará em ultrapaginação, mesmo se houver um número suficiente de quadros de páginas disponíveis. Se o conjunto de trabalho diminuir, os algoritmos locais vão desperdiçar memória. Se um algoritmo global for usado, o sistema terá de decidir continuamente quantos quadros de páginas designar para cada processo. Uma maneira é monitorar o tamanho do conjunto de trabalho como indicado pelos bits de envelhecimento, mas essa abordagem não evita necessariamente a ultrapaginação. O conjunto de trabalho pode mudar de tamanho em milissegundos, enquanto os bits de envelhecimento são uma medida muito rudimentar estendida a um número de interrupções de relógio.

Outra abordagem é ter um algoritmo para alocar quadros de páginas para processos. Uma maneira é determinar periodicamente o número de processos em execução e alocar a cada processo uma porção igual. Desse modo, com 12.416 quadros de páginas disponíveis (isto é, sistema não operacional) e 10 processos, cada processo recebe 1.241 quadros. Os seis restantes vão para uma área comum a ser usada quando ocorrer a falta de página.

Embora esse método possa parecer justo, faz pouco sentido conceder porções iguais de memória a um processo de 10 KB e a um processo de 300 KB. Em vez disso, as páginas podem ser alocadas em proporção ao tamanho total de cada processo, com um processo de 300 KB recebendo 30 vezes a quantidade alocada a um processo de 10 KB. Parece razoável dar a cada processo algum número mínimo, de maneira que ele possa ser executado por menor que seja. Em algumas máquinas, por exemplo, uma única instrução de dois operandos pode precisar de até seis páginas, pois a instrução em si, o operando fonte e o operando destino podem todos extrapolar os limites da página. Com uma alocação de apenas cinco páginas, os programas contendo tais instruções não poderão ser executados.

FIGURA 3.22 Substituição de página local *versus* global. (a) Configuração original. (b) Substituição de página local. (c) Substituição de página global.



Se um algoritmo global for usado, talvez seja possível começar cada processo com algum número de páginas proporcional ao tamanho do processo, mas a alocação precisa ser atualizada dinamicamente à medida que ele é executado. Uma maneira de gerenciar a alocação é usar o algoritmo **PFF (Page Fault Frequency** — frequência de faltas de página). Ele diz quando aumentar ou diminuir a alocação de páginas de um processo, mas não diz nada sobre qual página substituir em uma falta. Ele apenas controla o tamanho do conjunto de alocação.

Para uma grande classe de algoritmos de substituição de páginas, incluindo LRU, sabe-se que a taxa de faltas diminui à medida que mais páginas são designadas, como discutimos. Esse é o pressuposto por trás da PFF. Essa propriedade está ilustrada na Figura 3.23.

Medir a frequência de faltas de página é algo direto: apenas conte o número de faltas por segundo, possivelmente tomando a média de execução através dos últimos segundos também. Uma maneira fácil de fazer isso é somar o número de faltas durante o segundo imediatamente anterior à média de execução atual e dividir por dois. A linha tracejada *A* corresponde a uma frequência de faltas de página que é inaceitavelmente alta, portanto o processo que gerou as faltas de páginas recebe mais quadros de páginas para reduzir a frequência de faltas. A linha tracejada *B* corresponde a uma frequência de faltas de página tão baixa que podemos presumir que o processo tem memória demais. Nesse caso, molduras de páginas podem ser retiradas. Assim, PFF tentará manter a frequência de paginação para cada processo dentro de limites aceitáveis.

É importante observar que alguns algoritmos de substituição de página podem funcionar com uma política de substituição local ou uma global. Por exemplo, FIFO pode substituir a página mais antiga em toda a memória (algoritmo global) ou a página mais antiga possuída pelo processo atual (algoritmo local). De modo similar, 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, em alguns casos, é independente do algoritmo.

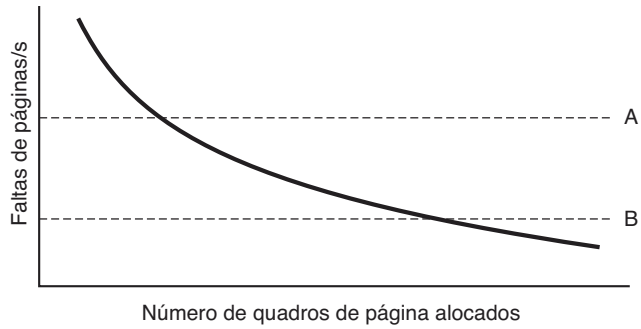
Por outro lado, para outros algoritmos de substituição de página, apenas uma estratégia local faz sentido. Em particular, o conjunto de trabalho e os algoritmos WSClock referem-se a algum processo específico e devem ser aplicados nesse contexto. Na realidade, não há um conjunto de trabalho para a máquina como um todo, e tentar usar a união de todos os conjuntos de trabalho perderia a propriedade de localidade e não funcionaria bem.

3.5.2 Controle de carga

Mesmo com o melhor algoritmo de substituição de páginas e uma ótima alocação global de quadros de páginas para processar, pode ocorrer a ultrapaginação. Na realidade, sempre que os conjuntos de trabalho combinados de todos os processos excedem a capacidade da memória, a ultrapaginação pode ser esperada. Um sintoma dessa situação é que o algoritmo PFF indica que alguns processos precisam de mais memória, mas nenhum processo precisa de menos memória. Nesse caso, não há maneira de dar mais memória àqueles processos que precisam dela sem prejudicar alguns outros. A única solução real é livrar-se temporariamente de alguns processos.

Uma boa maneira de reduzir o número de processos competindo pela memória é levar alguns deles para o disco e liberar todas as páginas que eles estão segurando. Por exemplo, um processo pode ser levado para o disco e seus quadros de páginas divididos entre outros processos que estão ultrapaginando. Se a ultrapaginação parar, o sistema pode executar por um tempo dessa

FIGURA 3.23 Frequência de faltas de página como função do número de quadros de página designados.



maneira. Se ela não parar, outro processo tem de ser levado para o disco e assim por diante, até a ultrapaginação cessar. Desse modo, mesmo com a paginação, a troca de processos entre a memória e o disco talvez ainda possa ser necessária, apenas agora ela será usada para reduzir a demanda potencial por memória, em vez de reivindicar páginas.

A ideia de trocar processos para o disco para aliviar a carga sobre a memória é reminiscente do escalonamento de dois níveis, no qual alguns processos são colocados em disco e um escalonador de curto prazo é usado para escalonar os processos restantes. Claramente, as duas ideias podem ser combinadas, de modo que se remova apenas um número suficiente de processos para o disco com o intuito de tornar aceitável a frequência de faltas de páginas. Periodicamente, alguns processos são trazidos do disco para a memória e outros são levados para ele.

No entanto, outro fator a ser considerado é o grau de multiprogramação. Quando o número de processos na memória principal é baixo demais, a CPU pode ficar ociosa por períodos substanciais. Esse fator recomenda considerar não somente o tamanho dos processos e frequência da paginação ao decidir qual processo deve ser trocado, mas também características, como se o processo seria do tipo limitado pela CPU ou por E/S, e quais características os processos restantes têm.

3.5.3 Tamanho de página

O tamanho de página é um parâmetro que pode ser escolhido pelo sistema operacional. Mesmo que o hardware tenha sido projetado com, por exemplo, páginas de 4096 bytes, o sistema operacional pode facilmente considerar os pares de página 0 e 1, 2 e 3, 4 e 5, e assim por diante, como páginas de 8 KB sempre alocando dois quadros de páginas de 8192 bytes consecutivas para eles.

Determinar o melhor tamanho de página exige equilibrar vários fatores competindo entre si. Como resultado, não há um tamanho ótimo geral. Para começo de conversa, dois fatores pedem um tamanho de página pequeno. Um segmento de código, dados, ou pilha escolhido ao acaso não ocupará um número inteiro de páginas. Na média, metade da página final estará vazia. O espaço extra nessa página é desperdiçado. Esse desperdício é chamado de **fragmentação interna**. Com n segmentos na memória e um tamanho de página de p bytes, $np/2$ bytes serão desperdiçados em fragmentação interna. Esse raciocínio defende um tamanho de página pequeno.

Outro argumento em defesa de um tamanho de página pequeno torna-se aparente quando pensamos sobre um programa consistindo em oito fases sequenciais de 4 KB cada. Com um tamanho de página de 32 KB, esse programa demandará 32 KB durante o tempo inteiro de execução. Com um tamanho de página de 16 KB, ele precisará de apenas 16 KB. Com um tamanho de página de 4 KB ou menor, ele exigirá apenas 4 KB a qualquer instante. Em geral, um tamanho de página grande causará mais desperdício de espaço na memória.

Por outro lado, páginas pequenas implicam que os programas precisarão de muitas páginas e, desse modo, uma tabela grande de páginas. Um programa de 32 KB precisa apenas de quatro páginas de 8 KB, mas 64 páginas de 512 bytes. Transferências para e do disco são geralmente uma página de cada vez, com a maior parte do tempo sendo gasta no posicionamento da cabeça de leitura/gravação e no tempo de rotação necessário para que a cabeça de leitura/gravação atinja o setor correto, então a transferência de uma página pequena leva praticamente o mesmo tempo que a de uma página grande. Podem ser necessários 64 × 10 ms para carregar 64 páginas de 512 bytes, mas somente 4 × 12 ms para carregar quatro páginas de 8 KB.

Além disso, páginas pequenas ocupam muito espaço no **TLB**. Digamos que seu programa use 1 MB de memória com um conjunto de trabalho de 64 KB. Com páginas de 4 KB, o programa ocuparia pelo menos 16 entradas no TLB. Com páginas de 2 MB, uma única entrada de TLB seria suficiente (na teoria, mas talvez você queira separar dados de instruções). Como entradas de TLB são escassas e críticas para o desempenho, vale a pena usar páginas grandes sempre que possível. Para equilibrar essas escolhas, sistemas operacionais às vezes usam tamanhos diferentes de páginas para partes diferentes do sistema. Por exemplo, páginas grandes para o núcleo e menores para os processos do usuário.

Em algumas máquinas, a tabela de páginas deve ser carregada (pelo sistema operacional) em registradores de hardware toda vez que a CPU trocar de um processo para outro. Nessas máquinas, ter um tamanho de página pequeno significa que o tempo exigido para carregar os registradores de página fica mais longo à medida que o tamanho da página fica menor. Além disso, o espaço ocupado pela tabela de páginas aumenta à medida que o tamanho da página diminui.

Esse último ponto pode ser analisado matematicamente. Seja de s bytes o tamanho médio do processo e de p bytes o tamanho de página. Além disso, presuma que cada entrada de página exija e bytes. O número aproximado de páginas necessário por processo é então

de s/p, ocupando se/p bytes de espaço de tabela de página. A memória desperdiçada na última página do processo por causa da fragmentação interna é $p/2$. Desse modo, o custo adicional total decorrente da tabela de páginas e da perda pela fragmentação interna é dado pela soma desses dois termos:

$$custo\ adicional = se / p + p/2$$

O primeiro termo (tamanho da tabela de páginas) é grande quando o tamanho da página é pequeno. O segundo termo (fragmentação interna) é grande quando o tamanho da página é grande. O valor ótimo precisa encontrar-se em algum ponto intermediário. Calculando a derivada primeira com relação a p e equacionando-a a zero, chegamos à equação

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

A partir dessa equação podemos derivar uma fórmula que dá o tamanho de página ótimo (considerando apenas a memória desperdiçada na fragmentação e o 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 disponíveis comercialmente têm usado tamanhos de páginas que variam de 512 bytes a 64 KB. Um valor típico costumava ser 1 KB, mas hoje 4 KB é mais comum.

3.5.4 Espaços separados de instruções e dados

A maioria dos computadores tem um único espaço de endereçamento tanto para programas quanto para dados, como mostrado na Figura 3.24(a). Se esse espaço de endereçamento for grande o suficiente, tudo mais funcionará bem. No entanto, se for pequeno demais, ele força os programadores a encontrar uma saída para fazer caber tudo no espaço de endereçamento.

Uma solução, apresentada pioneiramente no PDP-11 (16 bits), é ter dois espaços de endereçamento diferentes para instruções (código do programa) e dados, chamados de **espaço I** e **espaço D**, respectivamente, como ilustrado na Figura 3.24(b). Cada espaço de endereçamento se situa entre 0 e um valor máximo, em geral $2^{16} - 1$ ou $2^{32} - 1$. O ligador (linker) precisa saber quando endereços I e D separados estão sendo usados, pois quando eles estão, os dados são realocados para o endereço virtual 0, em vez de começarem após o programa.

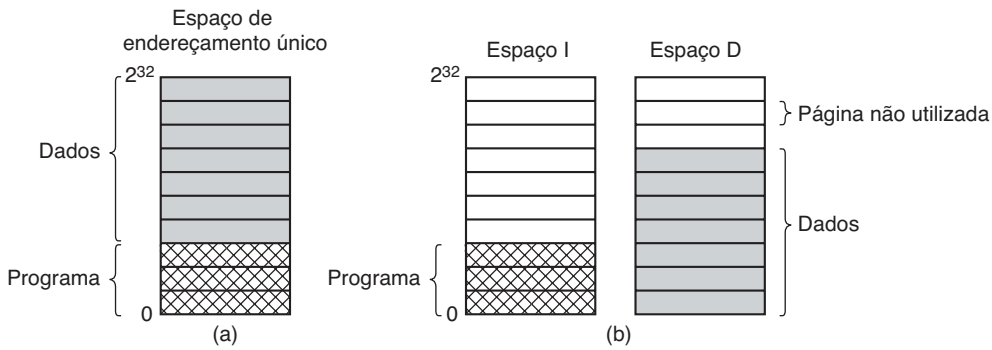
Em um computador com esse tipo de projeto, ambos os espaços de endereçamento podem ser paginados, independentemente um do outro. Cada um tem sua própria tabela de páginas, com seu próprio mapeamento de páginas virtuais para quadros de página física. Quando o hardware quer buscar uma instrução, ele sabe que deve usar o espaço I e a tabela de páginas do espaço I. De modo similar, dados precisam passar pela tabela de páginas do espaço D. Fora essa distinção, ter espaços de I e D separados não apresenta quaisquer complicações especiais para o sistema operacional e duplica o espaço de endereçamento disponível.

Embora os espaços de endereçamento sejam grandes, seu tamanho costumava ser um problema sério. Mesmo hoje, no entanto, espaços de I e D separados são comuns. No entanto, em vez de serem usados para os espaços de endereçamento normais, eles são usados agora para dividir a cache L1. Afinal de contas, na cache L1, a memória ainda é bastante escassa.

3.5.5 Páginas compartilhadas

Outra questão de projeto importante é o compartilhamento. Em um grande sistema de multiprogramação, é comum que vários usuários estejam executando o mesmo programa ao mesmo tempo. Mesmo um único usuário pode estar executando vários programas que

FIGURA 3.24 (a) Um espaço de endereçamento. (b) Espaços I e D independentes.

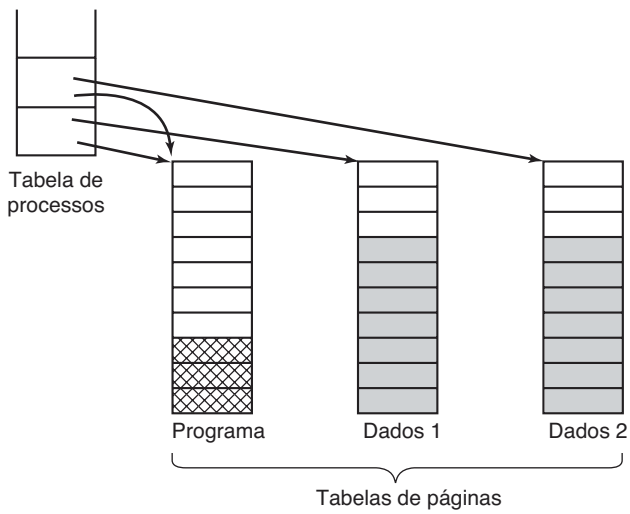


usam a mesma biblioteca. É claramente mais eficiente compartilhar as páginas, para evitar ter duas cópias da mesma página na memória ao mesmo tempo. Um problema é que nem todas as páginas são compartilháveis. Em particular, as que são somente de leitura, como um código de programa, podem sê-lo, mas o compartilhamento de páginas com dados é mais complicado.

Se o sistema der suporte aos espaços I e D, o compartilhamento de programas é algo relativamente direto, fazendo que dois ou mais processos usem a mesma tabela de páginas para seu espaço I, mas diferentes tabelas de páginas para seus espaços D. Tipicamente, em uma implementação que dá suporte ao compartilhamento dessa forma, as tabelas de páginas são estruturas de dados independentes da tabela de processos. Cada processo tem então dois ponteiros em sua tabela: um para a tabela de páginas do espaço I e outro para a de páginas do espaço D, como mostrado na Figura 3.25. Quando o escalonador escolhe um processo para ser executado, ele usa esses ponteiros para localizar as tabelas de páginas apropriadas e ativa a MMU usando-as. Mesmo sem espaços I e D separados, os processos podem compartilhar programas (ou, às vezes, bibliotecas), mas o mecanismo é mais complicado.

Quando dois ou mais processos compartilham algum 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 decidir remover *A* da memória, removendo todas as suas páginas e preenchendo os quadros das páginas vazias com algum outro programa, isso fará que *B* gere um grande número de faltas de páginas para trazê-las de volta outra vez.

FIGURA 3.25 Dois processos que compartilham o mesmo programa compartilhando sua tabela de páginas.



De modo semelhante, quando *A* termina a sua execução, é essencial que o sistema operacional saiba que as páginas ainda estão em uso e, então, seu espaço de disco não será liberado por acidente. Pesquisar todas as tabelas de páginas para ver se uma página está sendo compartilhada normalmente é muito caro, portanto estruturas de dados especiais são necessárias para controlar as páginas compartilhadas, em especial se a unidade de compartilhamento for a página individual (ou conjunto de páginas), em vez de uma tabela de páginas inteira.

Compartilhar dados é mais complicado do que compartilhar códigos, mas não é impossível. Em particular, em UNIX, após uma chamada de sistema fork, o processo pai e o processo filho são solicitados a compartilhar tanto o código do programa quanto os dados. Em um sistema de paginação, o que é feito muitas vezes é dar a cada um desses processos sua própria tabela de páginas e fazer que ambos apontem para o mesmo conjunto de dados. Assim, nenhuma cópia de páginas é realizada no instante fork. No entanto, todas as páginas de dados são mapeadas em ambos os processos como SOMENTE PARA LEITURA (read-only).

Enquanto ambos os processos apenas lerem os seus dados, sem modificá-los, essa situação pode continuar. Tão logo qualquer um dos processos atualize uma palavra da memória, a violação da proteção somente para leitura causa uma interrupção no sistema operacional. Uma cópia dessa página então é feita e assim cada processo tem agora sua própria cópia particular. Ambas as cópias estão agora configuradas para LER/ESCREVER, portanto operações de escrita subsequentes para qualquer uma delas procedem sem interrupções. Essa estratégia significa que aquelas páginas que jamais são modificadas (incluindo todas as páginas do programa) não precisam ser copiadas. Apenas as páginas de dados que são realmente modificadas precisam ser copiadas. Essa abordagem, chamada de **copiar na escrita** (*copy on write*), melhora o desempenho ao reduzir o número de cópias.

3.5.6 Bibliotecas compartilhadas

O compartilhamento pode ser feito em outras granularidades além das páginas individuais. Se um programa for inicializado duas vezes, a maioria dos sistemas operacionais vai compartilhar automaticamente todas as páginas de texto de maneira que apenas uma cópia esteja na memória. Páginas de texto são sempre de leitura somente, portanto não há problema

aqui. Dependendo do sistema operacional, cada processo pode ficar com sua própria cópia privada das páginas de dados, ou elas podem ser compartilhadas e marcadas somente de leitura. Se qualquer processo modificar uma página de dados, será feita uma cópia privada para ele, ou seja, o método copiar na escrita (copy on write) será aplicado.

Nos sistemas modernos, há muitas bibliotecas grandes usadas por muitos processos, por exemplo, múltiplas bibliotecas gráficas e de E/S. Ligar estaticamente todas essas bibliotecas a todo 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 **DLLs** ou **Dynamic Link Libraries** — 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 nomeadas no comando para o vinculador, como o comando do UNIX

```
ld *.o –lc –lm
```

que liga todos os arquivos (do objeto) *.o* no diretório atual e então 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*) são chamadas de **externas indefinidas** e buscadas nas bibliotecas. Se forem encontradas, elas são incluídas no arquivo binário executável. Quaisquer funções que elas chamam, mas que ainda não estão presentes também se tornam externas indefinidas. Por exemplo, *printf* precisa de *write*, então se *write* ainda não foi incluída, o vinculador procurará por ela e a incluirá quando for encontrada. Quando o vinculador tiver terminado, um arquivo binário é escrito para o disco contendo todas as funções necessárias. Funções presentes nas bibliotecas, mas não chamadas, não são incluídas. Quando o programa é carregado na memória e executado, todas as funções de que ele precisa 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 desperdiçaria uma quantidade tremenda de espaço no disco, assim como desperdiçaria espaço em RAM quando eles fossem carregados, já que o sistema não teria como saber como ele poderia compartilhá-los. É aí que entram as bibliotecas compartilhadas. Quando um programa está ligado a bibliotecas compartilhadas (que são ligeiramente diferentes das estáticas), em vez de incluir a função efetiva chamada, o vinculador inclui uma pequena rotina de *stub* que liga à função chamada no momento da execução. Dependendo do sistema e dos detalhes de configuração, bibliotecas compartilhadas são carregadas seja quando o programa é carregado ou quando as funções nelas são chamadas pela primeira vez. É claro, se outro programa já carregou a biblioteca compartilhada, não há necessidade de fazê-lo novamente — esse é o ponto da questão. Observe que, quando uma biblioteca compartilhada é carregada ou usada, toda a biblioteca não é lida na memória de uma única vez. As páginas entram uma a uma, na medida do necessário; assim, as funções que não são chamadas não serão trazidas à RAM.

Além de tornar arquivos executáveis menores e também salvar espaço na memória, bibliotecas compartilhadas têm outra vantagem importante: se uma função em uma biblioteca compartilhada for atualizada para remover um erro, não será necessário recompilar os programas que a chamam. Os antigos arquivos binários continuam a funcionar. Essa característica é de especial importância para softwares comerciais, em que o código-fonte não é distribuído ao cliente. Por exemplo, se a Microsoft encontrar e consertar 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 iniciados.

Bibliotecas compartilhadas vêm com um pequeno problema, no entanto, que tem de ser solucionado, como mostra a Figura 3.26. Aqui vemos dois processos compartilhando uma biblioteca de 20 KB de tamanho (presumindo que cada caixa tenha 4 KB). No entanto, a biblioteca está localizada em endereços diferentes em cada processo, presumivelmente porque os programas em si não são do mesmo tamanho. No processo 1, a biblioteca começa no endereço 36K; no processo 2, em 12K. Suponha que a primeira coisa que a primeira função na biblioteca tem de fazer é saltar para o endereço 16 na biblioteca. Se a biblioteca não fosse compartilhada, poderia ser realocada dinamicamente quando carregada, então o salto (no processo 1) poderia ser para o endereço virtual 36K + 16. Observe que o endereço físico na RAM onde a biblioteca está localizada não importa, já que todas as páginas são mapeadas de endereços físicos pela MMU no hardware.