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
```

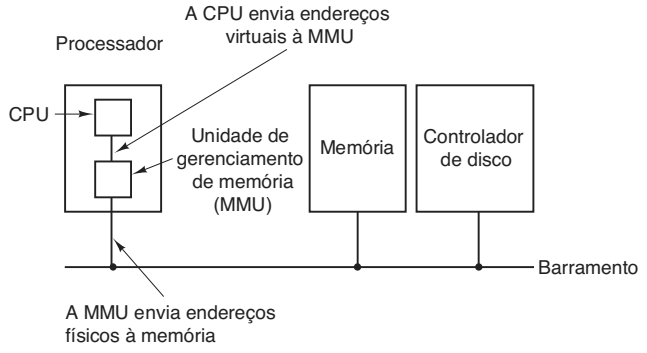
ele o faz para copiar o conteúdo do endereço de memória 1000 para REG (ou vice-versa, dependendo do computador). Endereços podem ser gerados usando indexação, registradores base, registradores de segmento e outras maneiras.

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.

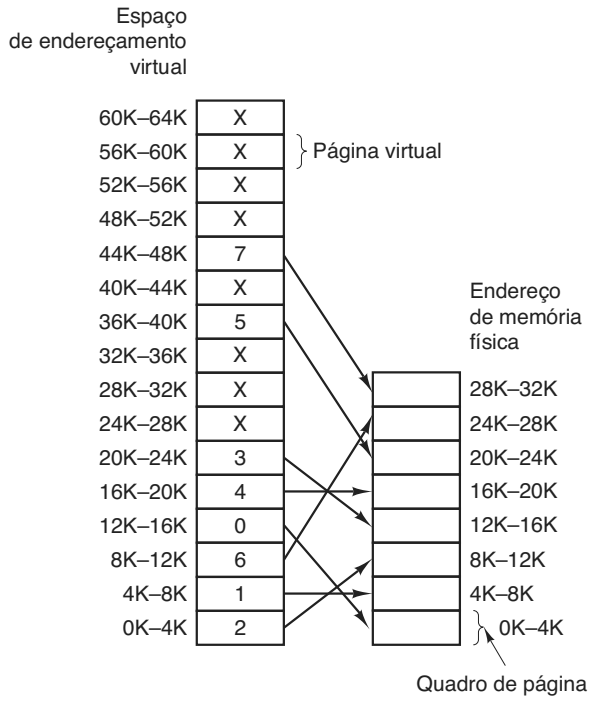


FIGURA 3.9 A relação entre endereços virtuais e endereços de memória física é dada pela **tabela de páginas**. Cada página começa com um múltiplo de 4096 e termina 4095 endereços acima; assim, 4K a 8K na verdade significa 4096-8191 e 8K a 12K significa 8192-12287.

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.

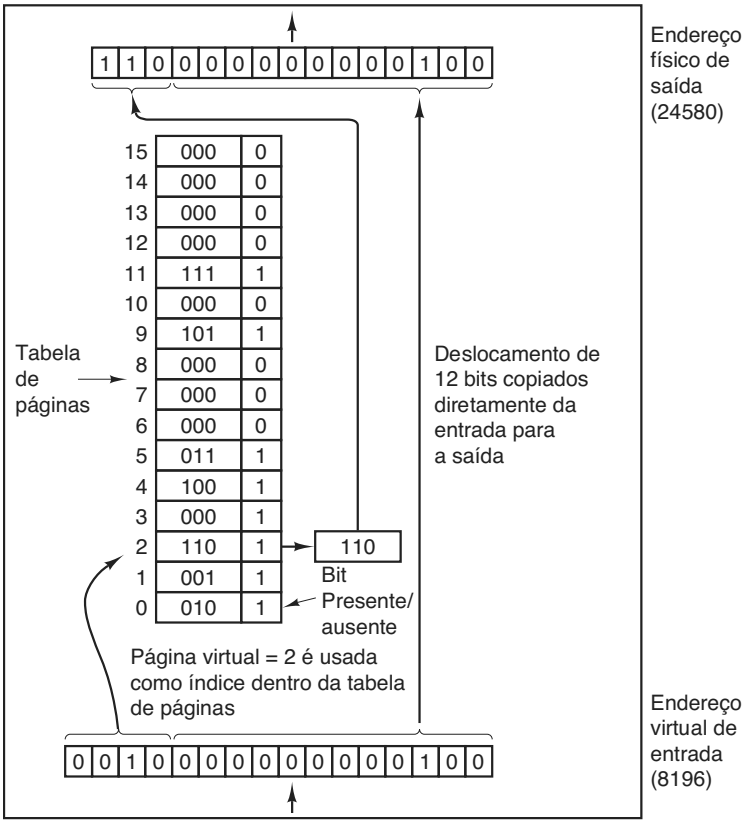
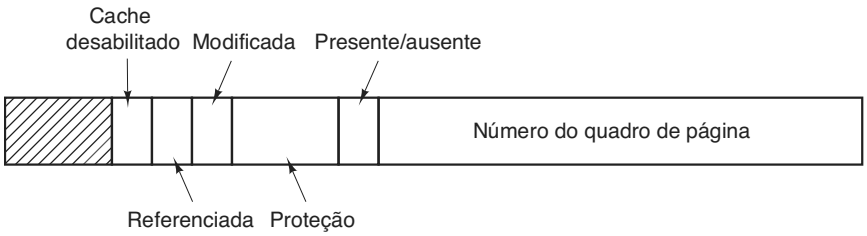


FIGURA 3.11 Uma entrada típica de uma tabela de páginas.



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.

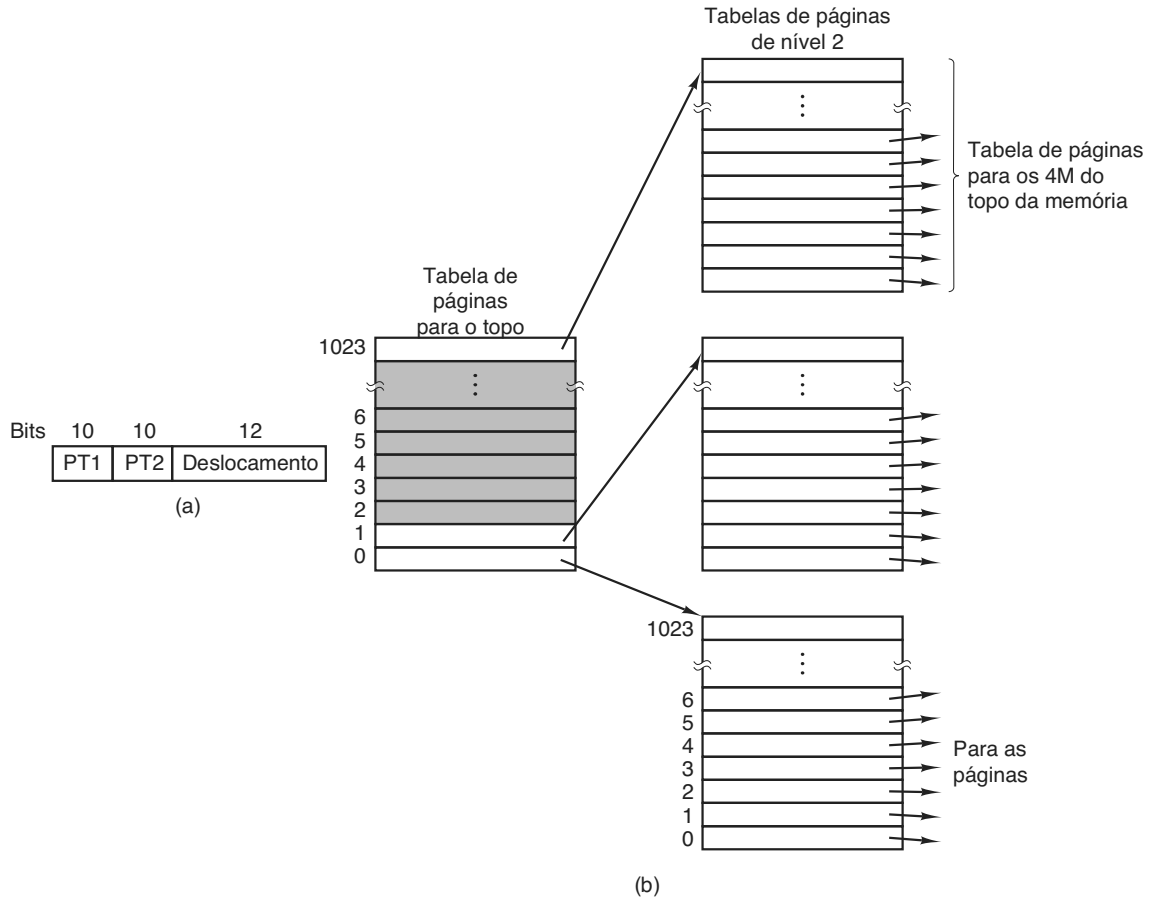
A entrada da tabela de páginas de nível 1, localizada através do campo *PT1* do endereço virtual, aponta para o endereço ou o número do quadro de página de uma tabela de páginas de nível 2. A entrada 0 da tabela de páginas de nível 1 aponta para a tabela de páginas relativa ao código do programa, a entrada 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.

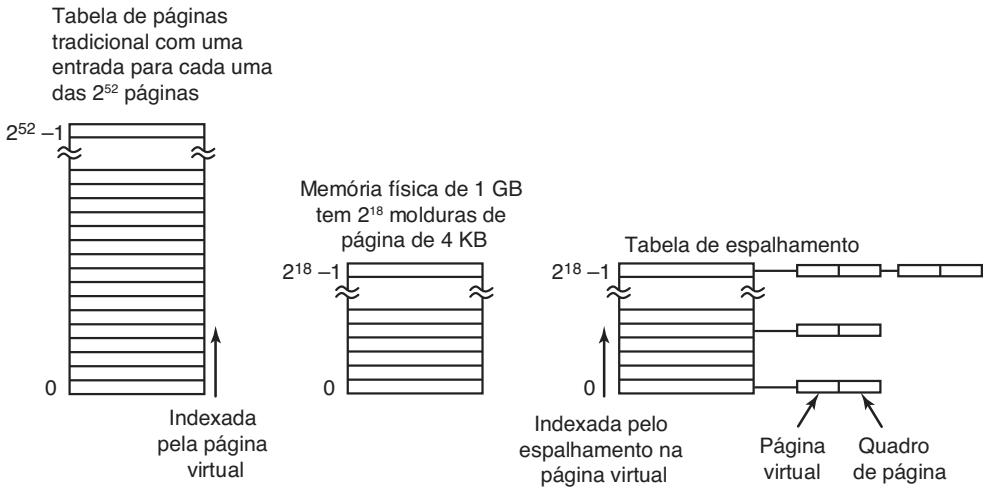
Tabelas de páginas invertidas

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.

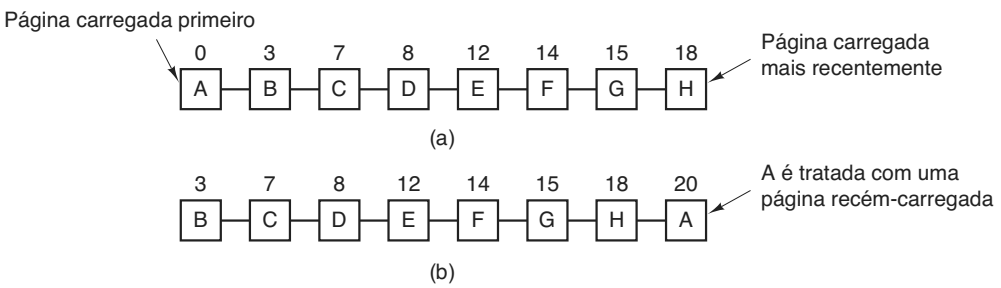
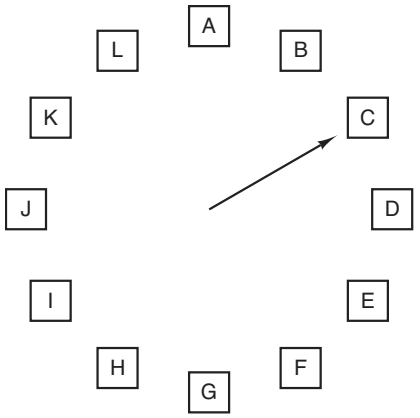


FIGURA 3.16 O algoritmo de substituição de páginas do relógio.



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.

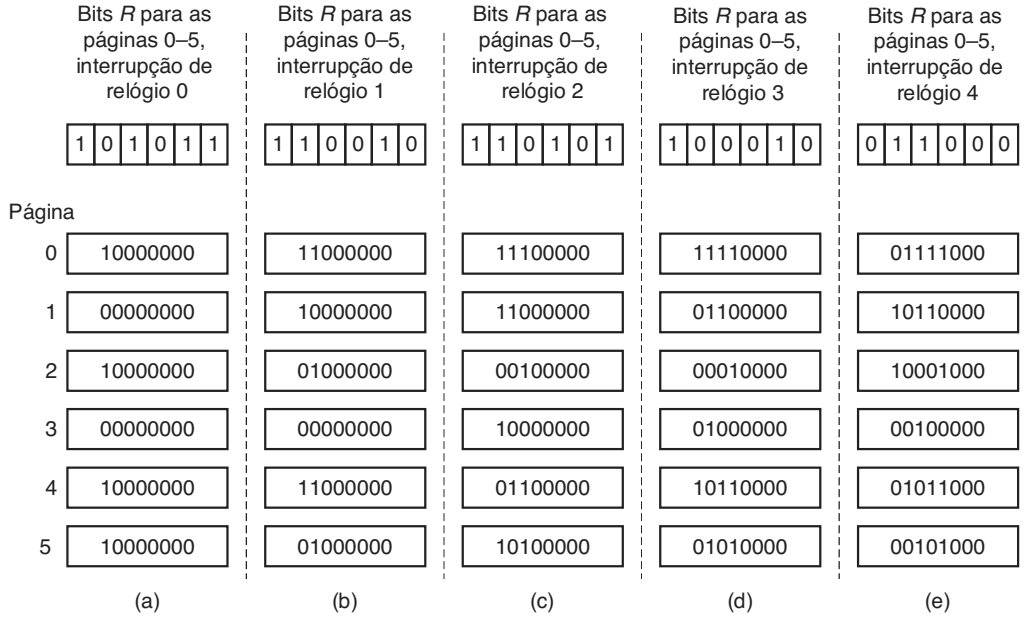
Esse algoritmo difere do LRU de duas maneiras importantes. Considere as páginas 3 e 5 na Figura 3.17(e). Nenhuma delas foi referenciada por duas interrupções de relógio; ambas foram referenciadas na interrupção anterior a elas. De acordo com o LRU, se uma página precisa ser substituída, devemos escolher 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).

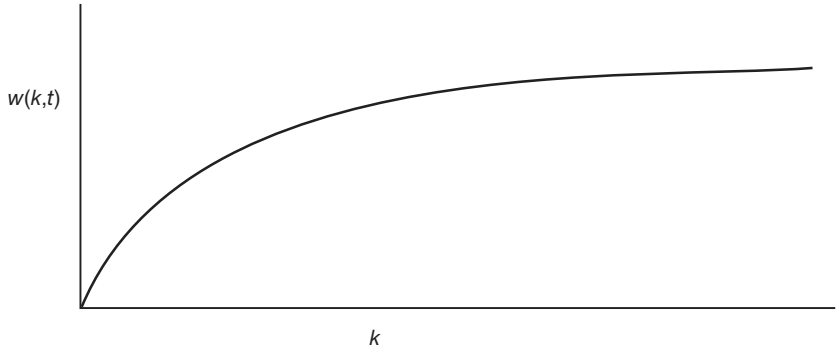
Em um sistema de multiprogramação, os processos muitas vezes são movidos para o disco (isto é, todas suas páginas são removidas da memória) para deixar que os outros tenham sua vez na CPU. A questão surge do que fazer quando um processo é trazido de volta outra vez. Tecnicamente, nada precisa ser feito. 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.

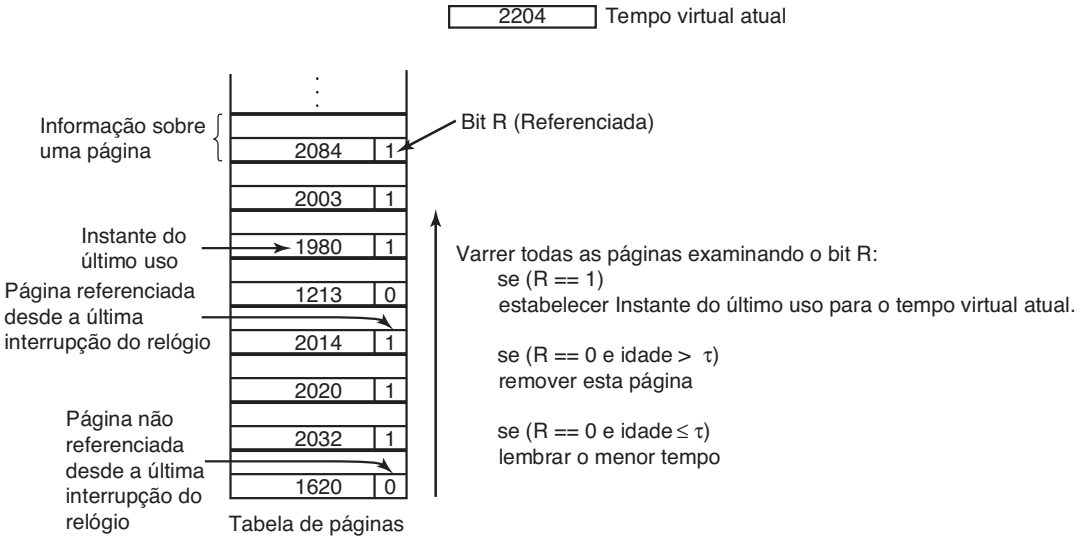
Em vez disso, várias aproximações são usadas. Uma delas é abandonar a ideia da contagem das últimas *k* referências de memória e em vez disso usar o tempo de execução. Por exemplo, em vez de definir o conjunto de trabalho como aquelas páginas usadas durante as últimas 10 milhões de referências de memória, 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

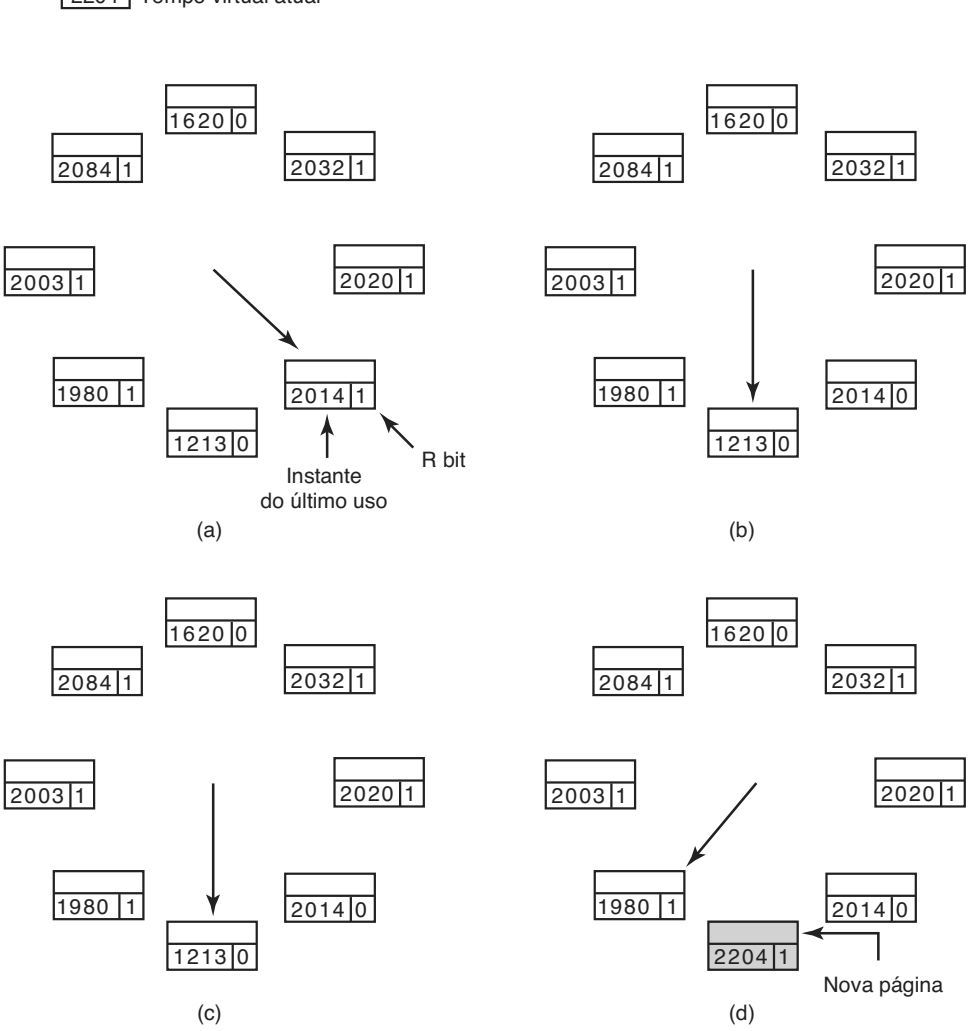
O algoritmo básico do conjunto de trabalho é enfadonho, já que a tabela de páginas inteira precisa ser varrida a cada falta de página até que uma candidata adequada seja localizada. Um algoritmo melhorado, que é baseado no algoritmo de relógio mas também usa a informação do conjunto de trabalho, é chamado 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