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.

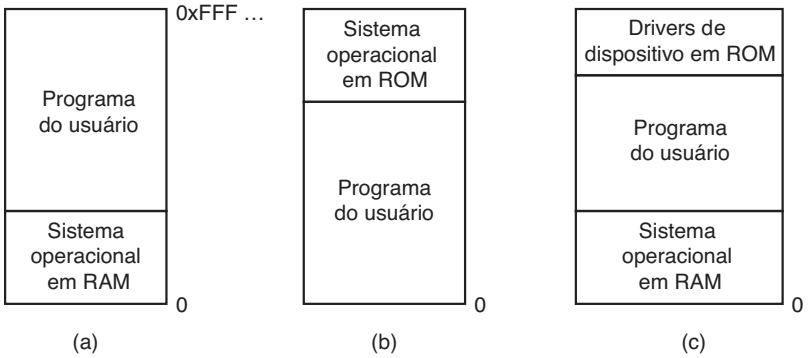
Uma maneira de se conseguir algum paralelismo em um sistema sem abstração de memória é programá-lo com múltiplos threads. Como todos os threads em um processo devem ver a mesma imagem da memória, o fato de eles serem forçados a fazê-lo não é um problema. Embora essa ideia funcione, ela é de uso limitado, 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.

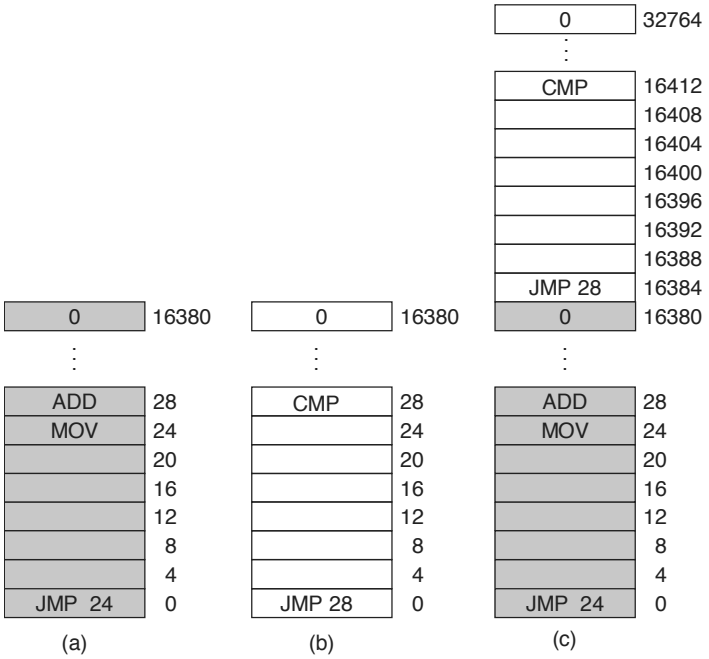
No entanto, essa solução tinha um problema importante, descrito na Figura 3.2. Aqui temos dois programas, cada um com 16 KB de tamanho, como mostrado nas figuras 3.2(a) e (b). O primeiro está sombreado para indicar que ele tem uma chave de memória diferente da do segundo. O primeiro programa começa com 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

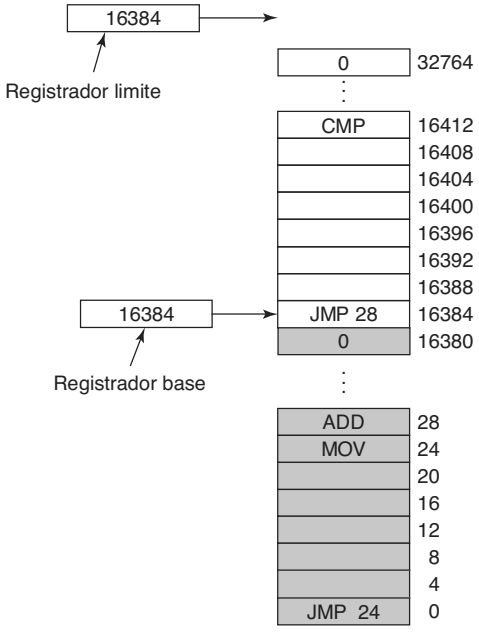
```
JMP 16412
```

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.

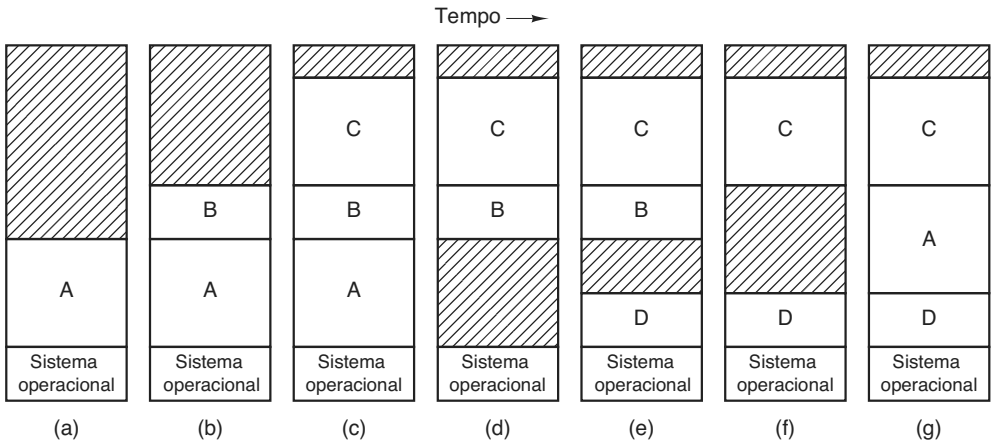
A operação de um sistema de troca de processos está ilustrada na Figura 3.4. De início, somente o processo *A* está na memória. Então os processos *B* e *C* são criados ou trazidos do disco. Na Figura 3.4(d) o processo *A* é devolvido ao disco. Então o processo *D* é inserido e o processo *B* tirado. 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.