Sistemas Operacionais

Inclui JAVA

Conceitos e Aplicaqõe:

Abraham Sílberschatz
Peter Galvin
Greg Gagne

CAMPUS

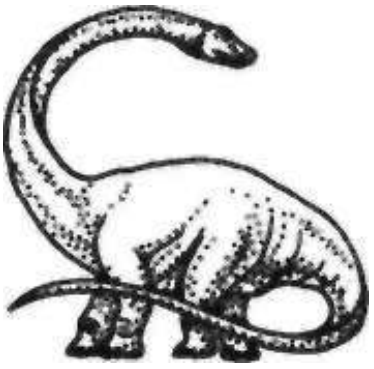
SUMÁRIO

PARTE UM VISÃO GERAL

PARTE DOIS GERÊNCIA DE PROCESSOS

PARTE TRÊS GERÊNCIA DE MEMÓRIA

Capítulo 1



INTRODUÇÃO

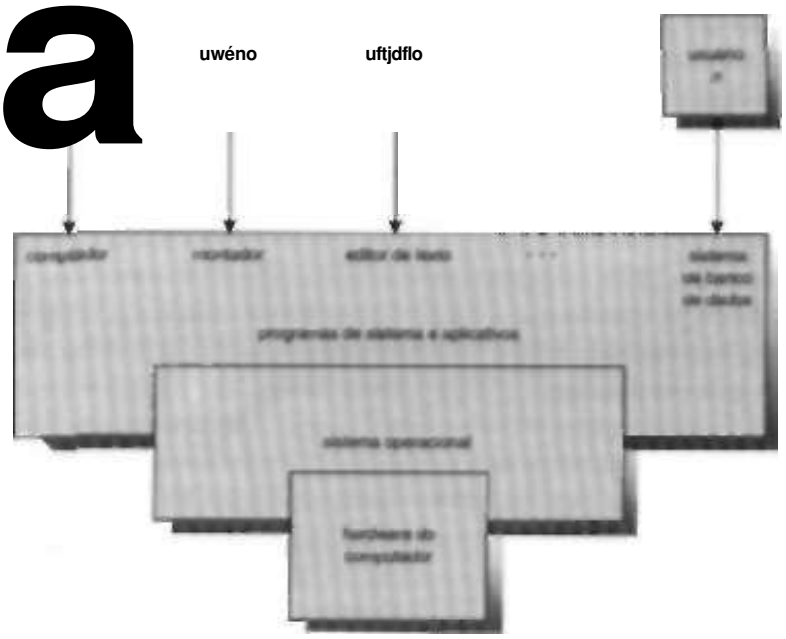
Um sistema operacional é um programa que atua como intermediário entre o usuário e o hardware de um computador. O propósito de um sistema operacional c fornecer um ambiente no qual o usuário possa executar programas. O principal objetivo de um sistema operacional é portanto tornar o uso do sistema de computação conveniente. Uma meta secundária é usar o hardware do computador de forma eficiente.

Para entender o que são sistemas operacionais, primeiro precisamos compreender como eles se desenvolveram. Neste capítulo, fazemos um apanhado do desenvolvimento dos sistemas operacionais desde os primeiros sistemas aos atuais sistemas de muItiprogramação e de tempo compartilhado. Ao avançarmos pelos diferentes estágios, vemos como os componentes dos sistemas operacionais evoluíram como soluções naturais para os problemas dos primeiros sistemas de computação. Compreender as razões por trás do desenvolvimento dos sistemas operacionais permite observar as tarefas que eles executam e como o fazem.

1.1 • O que é um sistema operacional?

Um sistema operacional é um componente importante de praticamente todo sistema de computação. Um sistema de computação pode ser dividido em basicamente quatro componentes: o hardware, o sistema operacional, os programas aplicativos e os usuários (Figura 1.1).

Figura 1.1 Visão abstrata dos componentes de um sistema de computação.



O hardware - a unidade central de processamento (CPU, *central processing Uttit),* a memória e os dispositivos de entrada/saída (I/O, Input/Output) - fornece os recursos básicos de computação. Os programas aplicativos - processadores de texto, planilhas eletrõnicas, compiladores c navegadores Web - definem as maneiras em que esses recursos são usados para resolver os problemas de computação dos usuários. Pode haver muitos usuários diferentes (pessoas, máquinas, outros computadores) tentando resolver problemas diferentes. Da mesma forma, pode haver muitos programas aplicativos diferentes. O sistema operacional controla e coordena o uso do hardware entre os vários programas aplicativos para os vários usuários.

Um sistema operacional í semelhante a *um governo.* Os componentes de um sistema de computação são seu hardware, software e dados. O sistema operacional fornece o meio para o uso adequado desses recursos na operação do sistema de computador. Como um governo, o sistema operacional não executa nenhuma função útil por si mesma. Simplesmente fornece um *ambiente* no qual outros programas podem realizar tarefas úteis.

Podemos considerar um sistema operacional como um alocador de recursos. Um sistema de computação possuí muitos recursos (hardware e software) que podem ser necessários para resolver um problema: tempo de CPU, espaço na memória, espaço de armazenamento de arquivos, dispositivos de entrada/saída (l/O), entre outros. O sistema operacional atua como gerente desses recursos e os aloca a programas e usuários específicos, conforme necessário, para a execução das tarefas. Como pode haver muitos pedidos de recursos, possivelmente conflitantes entre si, o sistema operacional deve decidir em que pedidos serão alceados recursos para que ele possa operar o sistema de computação de forma eficiente c justa.

Uma visão ligeiramente diferente de um sistema operacional enfatiza a necessidade de controlar os vários dispositivos de 1/0 e programas de usuário. Um sistema operacional é um programa de controle. Um programa de controle controla a execução dos programas de usuário para evitar erros e o uso indevido do computador. Preocupa-se especialmente com a operação e o controle de dispositivos de l/O.

Em geral, no entanto, não existe uma definição completamente adequada de um sistema operacional. Os sistemas operacionais existem porque são uma forma razoável de resolver o problema de criar um sistema de computação que possa ser usado. O objetivo primordial dos sistemas de computação é executar programas de usuário e tornar fácil a resolução dos problemas de usuário. Para atingir essa meta, 0 hardware é construído. Como o hardware por si só não *é* particularmente fácil de usar, programas aplicativos são desenvolvidos. Esses vários programas exigem certas operações comuns, como aquelas que controlam os dispositivos de I/O. As funções comuns de controle e alocação de recursos são então reunidas em um único software: o sistema operacional.

Não existe uma definição universalmente aceita do que faz e do que não faz parte do sistema operacional. Um ponto de vista simples *é* que tudo o que o fornecedor entrega quando você solicita"o sistema operacional" deve ser considerado. Os requisitos de memória c os recursos incluídos, no entanto, variam muito de sistema para sistema. Alguns usam menos de 1 megabyte de espaço e não têm nem um editor de tela inteira, enquanto outros exigem centenas de megabytes de espaço e baseiam-se inteiramente em sistemas gráficos de janelas. Uma definição mais comum *é* que o sistema operacional é um programa que está sempre executando no computador (geralmente chamado núcleo ou *kernel),* todo o resto consistindo em programas aplicativos. Normalmente, adotamos essa última definição. A questão em torno do que constitui um sistema operacional está se tornando importante. Em 1998, o Departamento de justiça norte-americano entrou com um processo contra a Microsoft basicamente alegando que a empresa incluía um número excessivo de funcionalidades em seus sistemas operacionais, impedindo, assim, qualquer concorrência por parte de outros fabricantes.

E mais fácil definir um sistema operacional pelo que ele *faz* do que pelo q u e d e i . O principal objetivo de um sistema operacional *è* a *conveniência do usuário.* Os sistemas operacionais existem porque têm como missão tornar a tarefa computacional mais fácil. Essa visão fica particularmente nítida quando analisamos sistemas operacionais para computadores pessoais de pequeno porte (PCs).

Uma meta secundária é a operação *eficiente* do sistema de computação. Essa meta é particularmente importante para sistemas multiusuário compartilhados e de grande porte. Esses sistemas são geralmente caros, por isso devem ser o mais eficientes possível. Essas duas metas - conveniência e eficiência - às vezes são contraditórias. No passado, a eficiência era frequentemente mais importante que a conveniência. Assim, boa parte da teoria dos sistemas operacionais concentra-se no uso otimizado dos recursos computacionais.

Para entender o que são sistemas operacionais e o que eles fazem, vamos considerar o seu desenvolvimento nos últimos 35 anos. Ao acompanhar essa evolução, poderemos identificar quais são os elementos comuns dos sistemas operacionais e verificar como e por que esses sistemas se desenvolveram dessa maneira.

Os sistemas operacionais e a arquiterura dos computadores tiveram grande influência mútua. Para facilitar o uso do hardware, os pesquisadores desenvolveram sistemas operacionais. A medida que os sistemas operacionais foram criados e utilizados, ficou claro que mudanças no projeto do hardware poderiam simplificá-los. Nessa breve revisão histórica, observe como a identificação dos problemas de sistemas operacionais levou ã introdução de novos recursos de hardware.

1.2 • Sistemas em lote (batch)

Os primeiros computadores eram máquinas exageradamente grandes (em termos físicos), operadas a partir de um console. Os dispositivos de entrada comuns eram leitoras de cartões e unidades de fita. Os dispositivos de saída comuns eram impressoras de linhas, unidades de fita e perfuradoras de cartões. O usuário não interagia dirctamente com os sistemas de computação. Km vez disso, ele preparava um job (tarefa), que consistia no programa, dados e algumas informações de controle sobre a natureza da tarefa (cartões de controle), c o submetia ao operador do computador. A tarefa geralmente tinha a forma de cartões perfurados. Algum tempo depois (minutos, horas ou dias), a saída aparecia. A saída consistia no resultado do programa, um dump de memória e registradores no caso de erro de programa.

O sistema operacional nesses primeiros computadores era bem simples. Sua principal tarefa era transferir controle automaticamente de um job para o próximo. O sistema operacional estava sempre residente na memória (Figura 1.2).

Para acelerar o processamento, os operadores reuniam os jobs em lotes com necessidades semelhantes e os executavam no computador como um grupo. Assim, os programadores deixavam seus programas com o operador. O operador classificava os programas em lotes com requisitos semelhantes e, à medida que o computador ficava disponível, executava cada lote ou batch. A saída de cada job seria enviada de volta ao programador apropriado.

Neste ambiente de execução, a CPU muitas vezes fica ociosa, porque as velocidades dos dispositivos mecânicos de I/O são intrinsecamente mais lentas do que as dos dispositivos eletrônicos. Mesmo uma CPU lenta funciona na faixa de microssegundos, executando milhares de instruções por segundo. Uma leitora de cartões rápida, por outro lado, pode ler 1.200 cartões por minuto (20 cartões por segundo). Assim, a diferença em velocidade entre a CPU e seus dispositivos de I/O pode ser de três ordens de grandeza ou mais. Com o tempo, é claro, melhorias na tecnologia e a introdução de discos resultaram em dispositivos de I/O mais rápidos. No entanto, as velocidades de CPU aumentaram ainda mais, por isso o problema não só não foi resolvido mas também exacerbado.

Figura 1.2. Layout da memória para um sistema em batch simples.

A introdução da tecnologia de disco permitiu que o sistema operacional mantivesse todos os jobs cm um disco, em vez de cm uma leitora de cartões serial. Com acesso direto a vários jobs, o escalonamento de jobs poderia ser executado para usar recursos e realizar tarefas de forma eficiente. O Capítulo 6 aborda em detalhes jobs e escalonamento de CPU; alguns aspectos importantes são discutidos aqui.

O aspecto mais importante do escalonamento de jobs é a capacidade de multiprogramação. Um único usuário não pode, em geral, manter a CPU ou os dispositivos de *VQ* ocupados em todos os momentos. A multiprogramação aumenta a utilização de CPU organizando jobs de forma que a CPU sempre tenha um job para executar.

A ideia é explicada a seguir. O sistema operacional mantém vários jobs na memória ao mesmo tempo (Figura 1,3). Esse conjunto de jobs é um subconjunto dos jobs mantidos no pool de jobs (já que o número de jobs que pode ser mantido simultaneamente na memória geralmente é muito menor do que o número de jobs que pode estar no pool de jobs). 0 sistema operacional escolhe e começa a executar um dos jobs na memória. Em alguns momentos, o job terá de esperar a conclusão de alguma tarefa, como uma operação de I/O. Em um sistema não-multiprogramado, a CPU ficaria ociosa. Em um sistema de multiprogramação, o sistema operacional simplesmente passa para outro job e o executa. Quando *esse job* precisa esperar, a CPU passa para outro job e assim por diante. Por fim, o primeiro job termina a espera e tem a CPU de volta. Desde que haja pelo menos um job para executar, a CPU nunca fica ociosa.

Essa ideia é comum cm outras situações da vida. Um advogado não trabalha apenas para um cliente de cada vez. Em vez disso, vários clientes podem estar sendo atendidos ao mesmo tempo. Enquanto um caso está aguardando julgamento ou preparação de documentos, o advogado pode trabalhar em outros casos. Se o advogado tiver um número suficiente de clientes, ele nunca ficará ocioso por falta de trabalho. (Advogados ociosos tendem a se tornar políticos, por isso existe um certo valor social cm manter os advogados ocupados.)

A multiprogramação é a primeira instância em que o sistema operacional precisa tomar decisões pelos usuários. Os sistemas operacionais multiprogramados são, portanto, bastante sofisticados. Todos os jobs que entram no sistema são mantidos no pool de jobs. Esse pool consiste cm todos os processos residentes no disco aguardando alocação da memória principal. Se vários jobs estiverem prontos para serem carregados na memória e, se não houver espaço suficiente para todos, o sistema deverá fazer a escolha. Essa tomada de decisão é chamada *escalonamento de jobs* c será discutida no Capítulo 6. Quando o sistema operacional seleciona um job do pool de jobs, ele o carrega na memória para execução. Ter vários programas na memória ao mesmo tempo requer alguma forma de gerência de memória, tema que será abordado nos Capítulos *9* e 10. Além disso, se vários jobs estiverem prontos para executar ao mesmo tempo, o sistema deverá escolher um deles. Essa tomada de decisão *é* chamada escalonamento de CPU e será discutida no Capítulo 6. Finalmente, múltiplos jobs sendo executados ao mesmo tempo vão exigir que haja pouca interferência mútua em todas as fases do sistema operacional, incluindo escalonamento de processos, armazenamento em disco e gerência de memória. Essas considerações são discutidas em todo o texto.

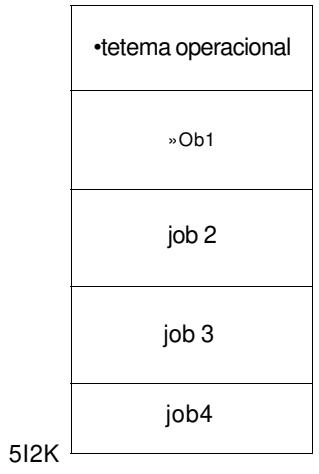


Figura 1.3 Layout de memória para um sistema de multiprogramação.

1.3 • Sistemas de tempo compartilhado

Os sistemas em batch multiprogramados forneceram um ambiente no qual os vários recursos do sistema (por exemplo, CPU, memória, dispositivos periféricos) eram utilizados de forma eficaz, mas não permitiam a interação do usuário com o sistema de computação. Tempo compartilhado, ou multitarefa, *c* uma extensão lógica da muItiprogramação. A CPU executa vários jobs alternando entre eles, mas as trocas ocorrem com tanta frequência que os usuários podem interagir com cada programa durante sua execução.

Um sistema de computação interativo permite a comunicação direta entre o usuário e o sistema. O usuário passa instruções ao sistema operacional ou a um programa diretamente, usando um teclado ou um mouse, e espera por resultados imediatos. Da mesma forma, o tempo de resposta deve ser curto - geralmente em torno de 1 segundo.

Um sistema operacional de tempo compartilhado permite aos muitos usuários compartilharem o computador ao mesmo tempo. Como cada ação ou comando em um sistema de tempo compartilhado tende a ser curto, apenas um pequeno tempo de CPU é necessário para cada usuário. Como o sistema alterna rapidamente de um usuário para outro, cada usuário tem a impressão de que todo o sistema de computação está dedicado ao seu uso, enquanto, na verdade, um computador está sendo compartilhado por muitos usuários.

Um sistema operacional de tempo compartilhado utiliza o escalonamento de CPU e a mu Iti programação para fornecer a cada usuário uma pequena parte de um computador de tempo compartilhado. Cada usuário tem pelo menos um programa separado na memória. Um programa carregado na memória e em execução é normalmente chamado de processo. Quando um processo executa, geralmente executa durante um curto espaço de tempo antes de terminar ou de precisar realizar uma operação de I/O- A operação de entrada/saída pode ser interativa, ou seja, a saída para o usuário é feita em um monitor c a entrada é a partir de um teclado, mouse ou outro dispositivo. Como a I/O interativa geralmente tem velocidade humana, pode levar muito tempo para terminar. A entrada, por exemplo, pode ser limitada pela velocidade de digitação do usuário; sete caracteres por segundo é rápido para pessoas, mas é incrivelmente lento para computadores. Hm vez de deixar a CPU inativa durante essa operação de entrada interativa, o sistema operacional rapidamente passará a CPU para o programa de algum outro usuário.

Os sistemas operacionais de tempo compartilhado são ainda mais complexos do que os sistemas operacionais multiprogramados. Em ambos, vários jobs devem ser mantidos na memória ao mesmo tempo, por isso o sistema deve contar com recursos de gerência de memória e proteção (Capítulo 9). Para alcançar um tempo de resposta razoável, os jobs talvez precisem ser passados rapidamente da memória principal para o disco que agora serve como extensão da memória principal. Um método comum de alcançar essa meta é a *memória virtual,* uma técnica que permite a execução de um job que talvez não esteja completamente na memória (Capítulo 10). A principal vantagem óbvia desse esquema é que os programas podem ser maiores do que a memória física. Além disso, ele abstrai a memória principal em um vetor grande e uniforme de armazenamento, separando a memória lógica conforme vista pelo usuário da memória física. Esse arranjo libera os programadores da preocupação relativa aos limites da memória. Os sistemas de tempo compartilhado também fornecem um sistema de arquivos (Capítulo 11). O sistema de arquivos reside cm uma coleção de discos; portanto, o gerenciamnto de disco deve ser realizado (Capítulo 13). Além disso, os sistemas de tempo compartilhado fornecem um mecanismo para execução concorrente, que requer esquemas sofisticados de escalonamento de CPU (Capítulo 6). Para garantir a execução correra, o sistema deve fornecer mecanismos para a comunicação e sincronização de jobs, e pode garantir que os jobs não fiquem presos em deadlocks, eternamente esperando uns pelos outros (Capítulo 7).

A ideia do tempo compartilhado foi demonstrada já em 1960, mas como os sistemas de tempo compartilhado são difíceis e caros de construir, só se tornaram comuns no início dos anos 70. Embora algum processamento em batch ainda ocorra, a maioria dos sistemas hoje é de tempo compartilhado. Por essa razão, a multiprogramação e o tempo compartilhado são os temas centrais dos sistemas operacionais modernos e também deste livro.

1.4 • Sistemas de computadores pessoais

Os computadores pessoais (PCs) apareceram nos anos 70. Durante sua primeira década, a CPU nos PCs não tinha os recursos necessários para proteger um sistema operacional dos programas de usuário. Os sistemas operacionais para PC, portanto, não eram nem multiusuário nem multitarefa. No entanto, as metas desses sistemas operacionais mudaram com o tempo; em vez de maximizar a utilização de CPUe periféricos, os sistemas optam por maximizar a conveniência e a capacidade de resposta ao usuário. Esses sistemas incluem PCs com Microsoft Windows e Apple Macintosh. O sistema operacional MS-DOS da Microsoft foi substituído por inúmeras variações do Microsoft Windows c a IBM atualizou o MS-DOS para o sistema multitarefa OS/2. O sistema operacional Apple Macintosh foi portado para um hardware mais avançado e agora inclui novos recursos, como memória virtual e multitarefa. O Linux, um sistema operacional semelhante ao UNIX disponível para PCs, também tornou-se popular recentemente.

Os sistemas operacionais para esses computadores se beneficiaram de várias formas do desenvolvimento de sistemas operacionais para mainframes. Os microcomputadores foram imediatamente capazes de adotar parte da tecnologia desenvolvida para os sistemas de grande porte. Por outro lado, os custos de hardware para microcomputadores são suficientemente baixos para que as pessoas utilizem apenas um computador, c a utilização de CPU não é mais uma preocupação importante. Assim, algumas das decisões de projeto tomadas em sistemas operacionais para mainframes talvez são sejam apropriadas para sistemas menores. Outras decisões de projeto ainda se aplicam. Por exemplo, a proteção de arquivos em princípio não era necessária cm uma máquina pessoal. No entanto, esses computadores muitas vezes estão ligados a outros computadores em redes tocais ou conexões Internet. Quando outros computadores e usuários podem acessar os arquivos no PC, a proteção de arquivos torna-sc novamente um recurso necessário de um sistema operacional. A falta de tal proteção tornou fácil a destruição de dados por programas maliciosos em sistemas como o MS-DOS ou Macintosh. Esses programas podem ser auto-reproduzíveis, podendo se espalhar por mecanismos *wortn* ou *vírus,* causando danos a empresas inteiras ou até mesmo redes mundiais.

No geral, um exame dos sistemas operacionais para mainframes e microcomputadores mostra que os recursos que estavam disponíveis em determinado momento apenas para mainframes foram adotados pelos microcomputadores. Os mesmos conceitos são apropriados para as várias classes diferentes de computadores: mainframes, minicomputadores c microcomputadores (Figura 1.4).

Um bom exemplo desse movimento ocorreu com o sistema operacional MULTICS. O MULTICS foi desenvolvido de 1965 a 1970 no Massachusetts Institute of Technology (MIT) como um *utilitário.* Executava

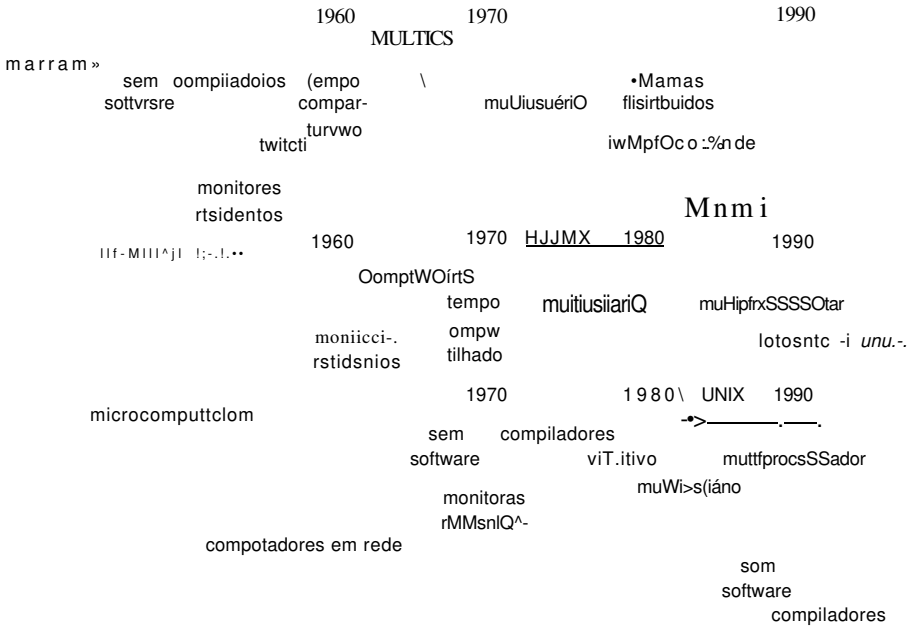


Figura 1.4 Migração dos conceitos e recursos dos sistemas operacionais.

em um computador mainframe grande e complexo (o GE 645). Muitas das ideias que foram desenvolvidas para o MULTICS foram subsequentemente usadas no Bell Laboratories (um dos parceiros originais no desenvolvimento do MULTICS) na criação do UNIX. O sistema operacional UNIX foi criado por volta de 1970 para um minicomputador PDP-1Í. Por volta de 1980, os recursos do UNIX tornaram-se a base para os sistemas operacionais semelhantes ao UNIX em sistemas de microcomputador e estão sendo incluídos nos sistemas operacionais mais recentes, tais como Microsoft Windows NT, IBM OS/2 e o Sistema Operacional Macintosh. Assim, os recursos desenvolvidos para um grande sistema de mainframe passaram para os microcomputadores com o tempo.

A medida que os recursos de sistemas operacionais grandes tinham sua escala reduzida para se ajustar aos PCs, sistemas de hardware mais poderosos, rápidos e sofisticados iam sendo desenvolvidos. A estação de trabalho pessoal é um PC grande - por exemplo, SUN SPARCstation, HP/Apollo, o computador IBM RS/6000 ou um sistema de classe Intel Pentium executando o Windows NT ou um derivado do UNIX. Muitas universidades e empresas tem um grande número de estações de trabalho interconectadas por redes locais. A medida que os PCs ganham hardware e software mais sofisticados, a linha que separa as duas categorias está desaparecendo.

1.5 • Sistemas paralelos

A maioria dos sistemas até hoje são sistemas de um único processador, ou seja, só têm uma CPU principal. No entanto, existe uma tendência em direção aos sistemas multiprocessador. Tais sistemas têm mais de um processador em comunicação ativa, compartilhando o barramento, o clock e, âs vezes, a memória e os dispositivos periféricos. Esses sistemas são chamados sistemas fortemente acoplados *(tightly coupled).*

Existem vários motivos para construir cisternas assim. Uma vantagem é a maior produção *(ihroughput).* Aumentando o número de processadores, espera-se realizar mais trabalho em menos tempo. A taxa de aumento de velocidade com *n* processadores, entretanto, não é *n,* mas menor que ». Quando vários processadores cooperam em uma tarefa, determinada quantidade de esforço é necessária para manter todas as partes trabalhando corretamente. Esse esforço, mais a disputa por recursos compartilhados, diminui o ganho esperado dos processadores adicionais. Da mesma forma, um grupo de *n* programadores trabalhando em conjunto não resulta em // vezes a quantidade de trabalho sendo realizada.

Os sistemas com múltiplos processadores também podem economizar dinheiro cm comparação com vários sistemas de um único processador, porque os processadores podem compartilhar periféricos, armazenamento de massa e fontes de alimentação. Sc vários programas forem processar o mesmo conjunto de dados, fica mais barato armazenar esses dados em um disco e fazer com que todos os processadores os compartilhem, cm vez de ter muitos computadores com discos locais e muitas cópias dos dados.

Outro motivo para a existência de sistemas com múltiplos processadores é o aumento da confiabilidade. Se as funções puderem ser distribuídas adequadamente entre vários processadores, a falha de um processador não vai interromper o sistema, apenas reduzirá sua velocidade. Se houver dez processadores e um falhar, cada um dos nove processadores restantes deverá pegar uma parte do trabalho do processador que falhou. Assim, o sistema inteiro executa apenas 109ó mais lentamente, cm vez de parar por completo. Essa capacidade de continuar a fornecer serviço proporcional ao nível do hardware sobrevivente é chamada de degradação normal.Os sistemas p.ojetados para degradação normal também são chamados de tolerantes a falhas.

A operação contínua na presença de falhas requer um mecanismo para permitir que a falha seja detectada, diagnosticada e corrigida (se possível). O sistema Tandem utiliza duplicação de hardware e software para garantir a operação continuada apesar das falhas. O sistema consiste em dois processadores idênticos, cada qual com sua própria memória local. Os processadores são conectados por um barramento. Um processador é o principal e o outro *é* o reserva. Duas cópias são mantidas de cada processo: uma no processador principal e outra no reserva. Em pontos de verificação fixos na execução do sistema, as informações de estado de cada job (incluindo uma cópia da imagem da memória) são copiadas da máquina principal para a reserva. Sc uma falha for detectada, a cópia reserva é ativad;» e reiniciada a partir do ponto de verificação mais recente. Essa solução é obviamente cara, pois existe muita duplicação de hardware.

Os sistemas de múltiplos processadores mais comuns agora usam multiprocessamento simétrico, no qual cada processador executa uma cópia idêntica do sistema operacional, e essas cópias se comunicam entre si

conforme necessário. Alguns sistemas utilizam multiprocessamento assimétrico, no qual a cada processador é atribuída uma tarefa específica. Um processador mestre controla o sistema; os outros processadores procuram o mestre para receber instruções ou têm tarefas predefinidas. Esse esquema define uma relação mestre-escravo. O processador mestre escalona c aloca trabalho para os processadores escravos.

Multiprocessamento simétrico (SMP, *sytumetric multiprocesshig)* significa que todos os processadores são iguais; não existe relação de mestre-escravo entre os processadores. Cada processador executa uma cópia do sistema operacional de forma concorrente. A Figura 1.5 ilustra uma arquitetura SMP típica. Um exemplo de sistema SMP é a versão Encore do UNIX para o computador Multimax. Esse computador pode ser configurado de modo a empregar dezenas de processadores, todos executando cópias do UNIX. O benefício desse modelo é que muitos processos podem executar simultaneamente - (N processos podem executar se houver N CPUs) - sem causar deterioração significativa de desempenho. No entanto, devemos controlar com cuidado as operações de I/O para garantir que os dados cheguem ao processador adequado. Alem disso, como as CPUs são separadas, uma pode estar ociosa enquanto outra está sobrecarregada, resultando em ineficiências. Essas ineficiências podem ser evitadas se os processadores compartilharem determinadas estruturas de dados. Um sistema multiproccssado desse tipo permitirá que processos e recursos, como memória, sejam compartilhados de forma dinâmica entre vários processadores e pode diminuir a variância entre os processadores. Um sistema desse tipo deve ser cuidadosamente elaborado, como será visto no Capítulo 7. Praticamente todos os sistemas operacionais modernos - incluindo o Windows NT, Solaris, Digital UNIX, OS/2 e Linux - agora fornecem suporte a SMP.

A diferença entre multiprocessamento simétrico e assimétrico pode ser o resultado de hardware ou software. Hardware especial pode diferenciar os múltiplos processadores, ou o software pode ser escrito para permitir apenas um mestre e vários escravos. Por exemplo, o sistema operacional SunOS Versão 4 da Sun, fornece multiprocessamento assimétrico, enquanto a Versão 5 (Solaris 2) é simétrica no mesmo hardware.

A medida que os microprocessadores se tornam menos caros e mais poderosos, funções adicionais de sistemas operacionais são passadas aos processadores escravos ou *back-ends.* Por exemplo, é relativamente fácil adicionar um microprocessador com sua própria memória para gerenciar um sistema de disco. O microprocessador poderia receber uma sequência de pedidos da CPU principal e implementar seu próprio algoritmo de escalonamento e fila de disco. Esse arranjo libera a CPU principal do custo de escalonamento de disco. Os PCs contêm um microprocessador no teclado para converter as sequências de teclas em códigos a serem enviados para a CPU. Na verdade, esse uso dos microprocessadores tornou-se tão comum que não é mais considerado multiprocessamento.

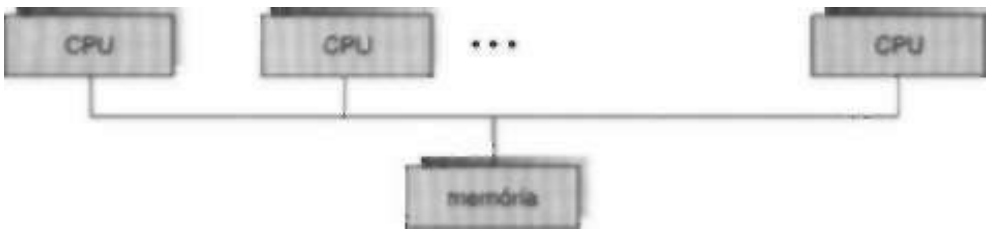


Figura 1.5 Arquitetura de multiprocessamento simétrico.

1.6 • Sistemas de tempo real

Outra forma de sistema operacional de propósito especial é o sistema de tempo real. Um sistema de tempo real é usado quando existem requisitos rígidos de tempo na operação de um processador ou no fluxo de dados; assim, ele geralmente é usado como um dispositivo de controle em uma aplicação dedicada. Os sensores levam dados ao computador. O computador deve analisar os dados e talvez ajustar os controles para modificar as entradas dos sensores. Os sistemas que controlam experimentos científicos, sistemas de imagens médicas, sistemas de controle industrial e determinados sistemas de exibição são de tempo real. Também estão incluídos alguns sistemas de injeção de combustível em motores de veículos, controladores de eletrodomésticos e sistemas de armas. Um sistema de tempo real tem limitações de tempo bem definidas e fixas. O processamento *tem* de ser feito dentro dos limites definidos ou o sistema falhará. Por exemplo, de nada adiantaria

para um braço mecânico de um robô ser instruído a parar *depois* de ter batido no carro que ele estava construindo. Um sistema de tempo real é considerado como estando cm bom estado de funcionamento somente se retornar o resultado correto dentro de qualquer limite de tempo. Compare esse requisito com um sistema de tempo compartilhado, cm que é desejável (mas não obrigatório) responder rapidamente, ou com um sistema em batch, em que pode não haver limitação de tempo.

Existem dois tipos de sistema de tempo real. Um sistema de tempo real crítico garante que as tarefas críticas sejam executadas a tempo. Essa meta requer que todos os atrasos no sistema sejam limitados, desde a rccjperação de dados armazenados até o tempo que o sistema operacional precisa para terminar qualquer solicitação realizada. Tais limitações de tempo determinam os recursos disponíveis nos sistemas desse tipo. O armazenamento secundário de qualquer tipo geralmente é limitado ou ausente, sendo os dados armazenados em memória temporária ou em memória de leitura (ROM). A ROM está localizada em dispositivos de armazenamento não-volátil que retêm seu conteúdo mesmo nos casos de queda de energia; a maioria dosoutros tipos de memória é volátil. Os recursos mais avançados dos sistemas operacionais também estão ausentes, pois tendem a separar o usuário do hardware c essa separação resulta cm im_ci tc^a sobre o tempo que determinada operação levará. Por exemplo, a memória virtual (discutida no Capítulo 10) quase nunca é encontrada nos sistemas de tempo real. Portanto, os sistemas de tempo real crítico entram em conflito com a operação dos sistemas de tempo compartilhado, e os dois não podem ser combinados. Como nenhum dos sistemas operacionais de uso geral existentes fornece suporte à funcionalidade de tempo real crítico, não nos preocuparemos com esse tipo de sistema neste livro.

Um tipo menos restritivo de sistema de tempo real é o sistema de tempo real não-crítico, em que uma tarefa crítica de tempo real recebe prioridade sobre as demais tarefas e retém essa prioridade até ser concluída. Como ocorre com os sistema de tempo real crítico, os atrasos de kernel precisam ser limitados: uma tarefa de tempo real não pode ficar esperando indefinidamente para execução pelo kernel. O tempo real não-crítico é uma meta alcançável que pode ser combinada com outros tipos de sistemas. Os sistemas de tempo real não-crítico, no entanto, têm utilidade mais limitada do que os sistema de tempo real crítico. Considerando sua falta de suporte de prazo, são arriscados de usar para controle industrial e robótica. No entanto, existem várias áreas nas quais são úteis, incluindo multimídia, realidade virtual e projetos científicos avançados, tais como exploração submarina e naves de exploração planetária. Esses sistemas precisam *ât* recursos avançados de sistemas operacionais que não podem ser suportados por sistema de tempo real crítico. Devido aos usos expandidos da funcionalidade de tempo real não-crítico, ela está presente na maioria dos sistemas operacionais atuais, incluindo as principais versões do UNIX.

No Capítulo 6, consideramos os recursos de escalonamento necessários para implementar a funcionalidade de tempo real não-crítico em um sistema operacional. No Capítulo 10, descrevemos o projeto de gerência de memória para a computação de tempo real. Finalmente, no Capítulo 22, descrevemos os componentes de tempo real do sistema operacional Windows NT.

1.7 • Sistemas distribuídos

O crescimento das redes de computadores - especialmente a Internet c a World Widc Web (WWW) - teve um impacto profundo no desenvolvimento recente dos sistemas operacionais. Quando os PCs surgiram nos anos 70, foram projetados para uso"pessoar e eram geralmente considerados computadores independentes. Com o início do uso generalizado da Internet nos anos 80 para correio eletrônico, ftp e gopher, muitos PCs passaram a se conectar a redes de computadores. Com a introdução da Web em meados da década de 1990, a conectividade de rede passou a ser considerada um componente essencial de um sistema de computação.

Praticamente todos os PCs e estações de trabalho modernos são capazes de executar um navegador Web para acessar documentos hipertexto na Web. Sistemas operacionais como o Windows, OS/2, MacOS e UNIX agora também incluem o software de sistema (como TCP/IP e PPP) que permite a um computador acessar a Internet via uma rede local ou conexão telefónica. Vários sistemas incluem o navegador Web propriamente dito, assim como clientes e servidores de correio eletrônico, de login remoto e de transferência de arquivos.

Em contraste com os sistemas fortemente acoplados discutidos na Seção 1.5, as redes de computadores usadas nesses tipos de aplicações consistem em uma coleçâo de processadores que não compartilham memória ou clock. Em vez disso, cada processador tem sua própria memória local. Os processadores se comunicam entre si através de várias linhas de comunicação, tais como barramentos ou linhas telefónicas de alta velocidade. Esses sistemas geralmente são chamados sistemas fracamente acoplados *(loosely coupled systems)* ou sistemas *distribuídos.*

Alguns sistemas operacionais levaram o conceito de redes e sistemas distribuídos um passo além da noção de fornecer conectividade de rede. Um sistema operacional de rede é um sistema operacional que fornece recursos como compartilhamento de arquivos através da rede e isso inclui um esquema de comunicação que permite a processos diferentes em computadores diferentes trocarem mensagens. Um computador que executa um sistema operacional de rede atua independentemente de todos os outros computadores na rede, embora esteja ciente da rede e seja capaz de se comunicar com outros computadores ligados na rede. Um sistema operacional distribuído é um ambiente menos autónomo: os diferentes sistemas operacionais interagem o suficiente para dar a impressão de que existe um único sistema operacional controlando a rede. Redes e sistemas distribuídos são discutidos nos Capítulos 14 a 16.

1.8 • Resumo

Os sistemas operacionais foram desenvolvidos nos últimos 40 anos para alcançar dois objetivos principais. km primeiro lugar, o sistema operacional tenta escalonar as atividades computacionais para garantir um bom desempenho do sistema de computação. Em segundo lugar, fornece um ambiente conveniente para O desenvolvimento e a execução de programas.

Inicialmente, os sistemas de computação eram usados a partir da console principal. Software montadores, utilitários de carga, linkeditores e compiladores melhoraram a conveniência da programação do sistema, mas também exigiam tempo substancial de configuração. Para reduzir o tempo de configuração, as empresas contrataram operadores e reuniram em lotes jobs semelhantes.

Os sistemas em batch permitiram o seqiiençiamento automático de jobs por um sistema operacional residente e melhoraram consideravelmente a utilização geral do computador. O computador não tinha mais de esperar a operação humana. A utilização da CPU era ainda baixa, no entanto, devido à baixa velocidade dos dispositivos de l/O em relação à velocidade da CPU. A operação offline de dispositivos lentos fornece um meio de utilizar vários sistemas do tipo leitora-fita e fita-impressora para uma única CPU.

Para melhorar o desempenho geral do sistema de computação, os desenvolvedores introduziram o conceito de *multiprogramação.* Com a multiprogramação, vários jobs são mantidos na memória ao mesmo tempo-, a CPU alterna entre eles para aumentar a utilização de CPU e diminuir o tempo total necessário para executar os jobs.

A multiprogramação, que foi desenvolvida para melhorar o desempenho, também permite o compartilhamento de tempo. Os sistemas operacionais de tempo compartilhado permitem que muitos usuários (de um a centenas) utilizem um sistema de computação de forma interativa ao mesmo tempo.

Os PCs são microcomputadores consideravelmente menores e mais baratos do que os sistemas mainframe. Os sistemas operacionais para esses computadores se beneficiaram do desenvolvimento dos sistemas operacionais para mainframes de diversas maneiras. No entanto, como as pessoas têm computadores para uso exclusivo, a utilização de CPU não é mais uma preocupação importante. Portanto, parte das decisões de projeto realizadas em sistemas operacionais para mainframes talvez não seja adequada para sistemas menores.

Os sistemas paralelos têm mais de uma CPU em comunicação direta; as CPUs compartilham o barramento e, às vezes, compartilham memória e dispositivos periféricos. Tais sistemas podem fornecer maior produtividade e melhor confiabilidade.

Um sistema de tempo real crítico geralmente é usado como um dispositivo de controle em uma aplicação dedicada. Um sistema operacional de tempo real crítico tem limitações de tempo fixas e bem definidas. O processamento *precisa* ser feito dentro dos limites definidos, ou o sistema falhará. Os sistemas de tempo real não-crítico têm limitações de tempo menos rigorosas e não suportam escalonamento de prazos.

Recentemente, a influência da Internet c da World Wide Web encorajou 0 desenvolvimento de sistemas operacionais modernos que incluem navegadores Web e software de comunicação e rede como recursos incorporados. Paralelo ao crescimento da Web está o uso crescente de Java, uma combinação de linguagem de programação e tecnologia que fornece muitos recursos tornando-a um exemplo adequado a ser apresentado como um estudo aplicado de muitos conceitos de sistemas operacionais.

Mostramos a progressão lógica do desenvolvimento dos sistemas operacionais, impulsionada pela inclusão de recursos no hardware da CPU que são necessários para funcionalidade avançada. Kssa tendência pode ser vista hoje na evolução dos PCs, com hardware barato sendo suficientemente aprimorado para permitir, por sua vez, melhores características.

• Exercícios

1.1 Quais são os três principais objetivos de um sistema operacional?

1.2 Liste as quatro etapas necessárias para executar um programa em uma máquina completamente dedicada.

1.3 Qual a principal vantagem da multiprogramação?

1.4 Quais as principais diferenças entre os sistemas operacionais para mainframes e para computadores pessoais?

1.5 Em um ambiente de multiprogramação e de tempo compartilhado, vários usuários compartilham o sistema ao mesmo tempo. Essa situação pode resultar em vários problemas de segurança.

a. Quais são dois desses problemas?

b. Podemos garantir o mesmo grau de segurança em uma máquina de tempo compartilhado que temos em uma máquina dedicada? Explique.

1.6 Defina as propriedades essenciais dos seguintes tipos de sistemas operacionais:

a. Batch

b. Interativo

c. Tempo compartilhado

d. Tempo real

c. Rede

f. Distribuído

1.7 Enfatizamos a necessidade do sistema operacional fazer uso eficiente do hardware do computador. Quando c apropriado para o sistema operacional abandonar esse princípio e"despcrdiçar*' recursos? Por que esse sistema na verdade não é um desperdício?

1.8 Em quais circunstâncias um usuário ficaria melhor se estivesse usando um sistema de tempo compartilhado em vez de um PC ou uma estação de trabalho exclusiva?

1.9 Descreva as diferenças entre multiprocessamento simétrico e assimétrico. Quais as três vantagens e uma desvantagem de sistemas com múltiplos processadores?

1.10 Qual é a principal dificuldade que um programador deve superar quando estiver escrevendo um sistema operacional para um ambiente de tempo real?

1.11 Considere as várias definições de *sistema operacional.* Considere se o sistema operacional deveria incluir aplicações como navegadores Web e programas de e-mail. Apresente argumentos para as duas possibilidades e justifique sua resposta.

Notas bibliográficas

Os sistemas de tempo compartilhado foram propostos pela primeira vez por Strachcy 11959|. Os primeiros sistemas de tempo compartilhado foram o Compatible Time-Sharing System (CTSS) desenvolvido no MIT [Corbato et ai. 1962] e o sistema SDC Q-.Í2, construído pelo System Dcvelopment Corporation [Schwartz et ai. 1964, Schwarrz e Weissman 1967]. Outros sistemas iniciais, porém mais sofisticados, incluem o sistema MULTIplexed Information and Computing Services (MULTICS) desenvolvido no MIT [Corbato e

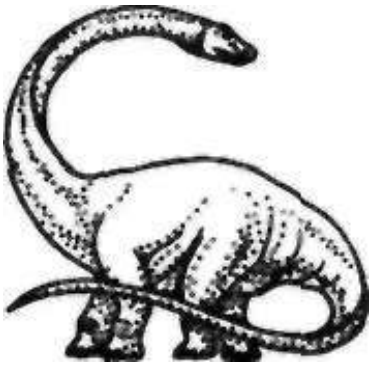
Vyssotsky 1965], o sistema XDS-940 desenvolvido na Universidade da Califórnia em Berkelcy [Lichtenbcrger e Pirtle 1965] e o sistema IBM TSS/360 [Lett e Konigsford 1968J.

Um levantamento dos sistemas operacionais distribuídos foi apresentado por Tanenbaum c van Renesse 11985J. Os sistemas operacionais de tempo real são discutidos por Stankovic e Ramamrithan |1989J. Uma edição especial de *Operating System Keview* sobre sistemas operacionais de tempo real foi feita por Zhao J19891.

O MS-DOS e PCs são descritos por Norton 11986) e por Norton e Wilton |1988|- Uma visão geral do hardware e software do Apple Macintosh é apresentada em lApple 1987]. O sistema operacional OS/2 é tratado em [Microsoft 1989J. Mais informações sobre o OS/2 podem ser encontradas em Letwin [1988] e em Deitei e Kogan [1992J. Solomon [1998] discute a estrutura do sistema operacional Microsoft Windows NT.

Existem vários livros didáticos gerais atualizados sobre sistemas operacionais [Finkel 1988, Krakowiak 1988, Pinkert e Wear 1989, Deitei 1990, Stallings 1992, Tanenbaum 19921.

Capítulo 2



ESTRUTURAS DE SISTEMAS DE COMPUTAÇÃO

Precisamos ter um conhecimento geral da estrutura de um sistema de computação antes de podermos explicar os detalhes da operação do sistema. Neste capítulo, analisamos as várias partes distintas dessa estrutura para embasar nosso conhecimento. Kste capítulo trata basicamente da arquitetura de sistemas de computação, por isso você pode folheá-lo ou saltá-lo se já conhece os conceitos. Como um sistema operacional está i ntimamente ligado aos mecanismos de entrada e saída (I/O) de um computador, esse tópico será discutido em primeiro lugar. As seções seguintes examinam a estrutura de armazenamento de dados.

O sistema operacional precisa garantir a operação correta do sistema de computação. Para que os programas de usuário não interfiram na operação adequada do sistema, o hardware deve fornecer os mecanismos apropriados para garantir o comportamento correio. Mais adiante neste capítulo, descrevemos a arquitetura de computação básica que torna possível a criação de um sistema operacional funcional,

2.1 • Operação dos sistemas de computação

Um sistema de computação de uso geral moderno consiste em uma CPU e em uma série de controladoras de dispositivos que são conectadas através de um barramento comum que fornece acesso à memória compartilhada (Figura 2.1). Cada controladora de dispositivo está encarregada de um tipo específico de dispositivo (por exemplo, unidades de disco, dispositivos de áudio e monitores de vídeo). A CPU e as controladoras de dispositivo podem executar de modo concorrente, competindo pelos ciclos de memória. Para garantir o

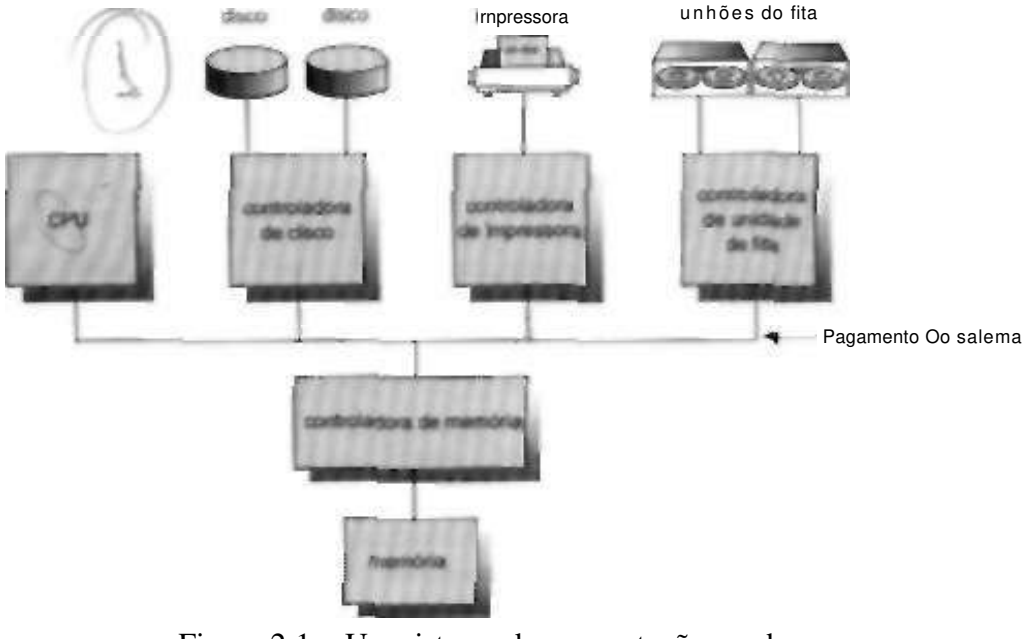


Figura 2.1 Um sistema de computação moderno.

acesso correto ã memória compartilhada, uma controladora de memória é fornecida e sua função é sincronio acesso à memória.

Para que um computador comece a funcionar- por exemplo, quando é ligado ou reinicializado - ele precisa ter um programa inicial para executar. Esse programa inicial, ou *programa de partida (bootstrap program),* tende a ser simples. Inicializa todos os aspectos do sistema, desde registradores de CPU a controladoras de dispositivos, passando pelo conteúdo da memória! O programa de partida deve saber como carregar o sistema operacional e iniciar a execução desse sistema. Para alcançar essa meta, o programa deve localizar e carregar na memória o kernel do sistema operacional. O sistema operacional em seguida inicia a execução do primeiro processo, como "init **, e espera que algum evento ocorniM ocorrência de um evento é geralmente assinalada por uma interrupção de hardware ou software. O hardware pode disparar uma interrupção a qualquer momento enviando um sinal para a CPU; geralmente por meio do barramento do sistema. O software pode disparar uma interrupção executando uma operação especial denominada chamada ao sistema *(system call)* ou *chamada ao monitor {monitor cal!).*

Existem muitos tipos diferentes de eventos que podem disparar uma interrupção, por exemplo, a conclusão de uma operação de I/O, divisão por zero, acesso inválido à memória e um pedido por algum serviço de sistema operacional. Para cada interrupção, uma rotina de serviço é designada responsável para tratar a interrupção.

Quando a CPU é interrompida, ela pára o que está fazendo e imediatamente transfere a execução para um local fixo. Esse local fixo geralmente contém o endereço de início onde está localizada a rotina de serviço para a interrupção. A rotina de serviço de interrupção executa; quando c concluída, a CPU retoma a computação interrompida. Um diagrama de tempo dessa operação é apresentada na Figura 2.2.

/~As interrupções são uma parte importante de uma arquitetura de computador. Cada projeto de computador tem seu próprio mecanismo de interrupção, mas várias funções são comuns. A interrupção deve transferir o controle para a rotina de serviço adequada. O método dircto para lidar com essa transferência seria chamar uma rotina genérica para examinar as informações de interrupção; a rotina, por sua vez, chamaria a rotina de processamento daquela interrupção específica. No entanto, as interrupções devem ser tratadas rapidamente c, considerando que existe um número predefinido de interrupções possíveis, uma tabela de ponteiros às rotinas de interrupção pode ser usada cm vez disso. A rotina de interrupção é chamada de forma indireta através da tabela, sem necessidade de uma rotina intermediária. Em geral, a tabela de ponteiros é armazenada na memória baixa (as primeiras 100 posições, aproximadamente). Essas posições guardam os endereços das rotinas de serviço de interrupção para os vários dispositivos. Esse vetor de endereços, ou vetor de interrupção, é então indexado por um número único de dispositivo, fornecido com o pedido de interrupção, para passar o endereço da rotina de serviço de interrupção para o dispositivo solicitante. Sistemas operacionais tão diferentes quanto o MS-DOS e o UNIX tratam interrupções dessa maneira.J

A arquitetura de interrupção também deve salvar o endereço da instrução interrompida. Muitos projetos anteriores simplesmente armazenavam o endereço de interrupção em uma posição fixa ou em uma posição indexada pelo número do dispositivo. Arquiteturas mais recentes armazenam o endereço de retorno na pilha do sistema. Se a rotina de interrupção precisar modificar o estado do processador, por exemplo, modificando os valores do registrador, deverá explicitamente salvar o estado aiual e depois restaurar esse estado antes do retorno. Depois que a interrupção tiver sido atendida, o endereço de retorno salvo é carregado no contador de programa e a computação interrompida é retomada, como se a interrupção não tivesse acontecido.

J_/ 'O s sistemas operacionais modernos são baseados em interrupções. Se não houver processos para executar, nenhum dispositivo de I/O ao qual fornecer serviço e nenhum usuário a ser atendido, um sistema operacional ficará parado, esperando que algo aconteça. Os eventos são quase sempre sinalizados pela ocorrência de uma interrupção ou um *trap.* Trap, ou exceção, é uma interrupção gerada por software causada por um erro (por exemplo, a divisão por zero, ou acesso inválido à memória), ou por um pedido específico de um programa de usuário para que um serviço de sistema operacional seja executado. O fato de um sistema operacional ser baseado em interrupções define a estrutura geral do sistema. Para cada tipo de interrupção, segmentos separados de código no sistema operacional determinam que ação deve ser rcalizaday

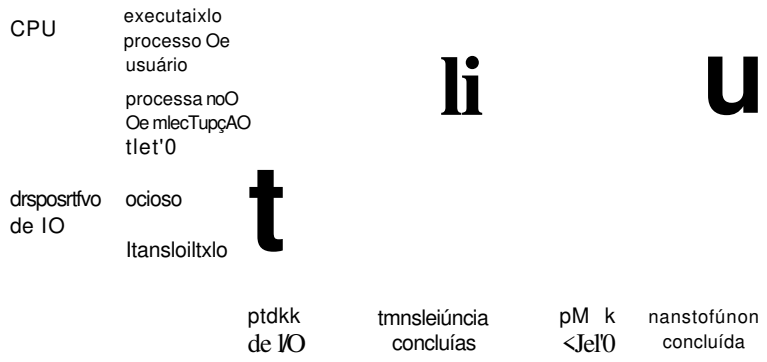


Figura 2.2 Diagrama de tempo de interrupção para um único processo gerando saída.

2.2 • Estrutura de I/O

Como discutido na Seção 2.1, um sistema de computação de uso geral consiste em uma CPU e em múltiplas controladoras de dispositivo que são conectadas através de um barramento comum. Cada controladora de dispositivo está encarregada de um tipo específico de dispositivo. Dependendo da controladora, pode haver mais de um dispositivo anexado. Por exemplo, a controladora SCSI (Small Computer-Systems Interface), encontrada em muitos computadores de pequeno e médio portes, pode ter sete ou mais dispositivos conectados a ela. Uma controladora de dispositivo mantém algum armazenamento em buffer local e um conjunto de registradores de propósito especial. A controladora de dispositivo é responsável pela passagem dos dados entre os dispositivos periféricos que ela controla e o buffer local. O tamanho do buffer local em uma controladora de dispositivo varia de uma controladora a outra, dependendo do dispositivo específico sendo controlado. Por exemplo, o tamanho do buffer de uma controladora de disco é igual ou um múltiplo do tamanho da menor parte endereçável de um disco, denominada setor, que geralmente tem 512 bytes.

2.2.1 Interrupções de I/O

Para começar uma operação de I/O, a CPU carrega os registradores adequados dentro da controladora de dispositivo. A controladora de dispositivo, por sua vez, examina o conteúdo desses registradores para determinar que ação deve ser tomada. Por exemplo, se encontrar um pedido de leitura, a controladora começará a transferir dados do dispositivo para o seu buffer local. Uma vez concluída a transferência, a controladora de dispositivo informa a CPU que terminou a operação. Essa comunicação é feita disparando uma interrupção.

Essa situação ocorrerá, cm geral, como resultado de um processo de usuário que solicita I/O. Uma vez iniciada a operação de I/O, dois roteiros são possíveis. No caso mais simples, a I/O é iniciada; em seguida, quando tiver sido concluída, o controle é devolvido para o processo de usuário. Esse caso é denominado I/O síncrona. A outra possibilidade, chamada I/O assíncrona, devolve o controle ao programa de usuário sem esperar que a I/O termine. A I/O continua enquanto outras operações de sistema ocorrem (Figura 2.3).

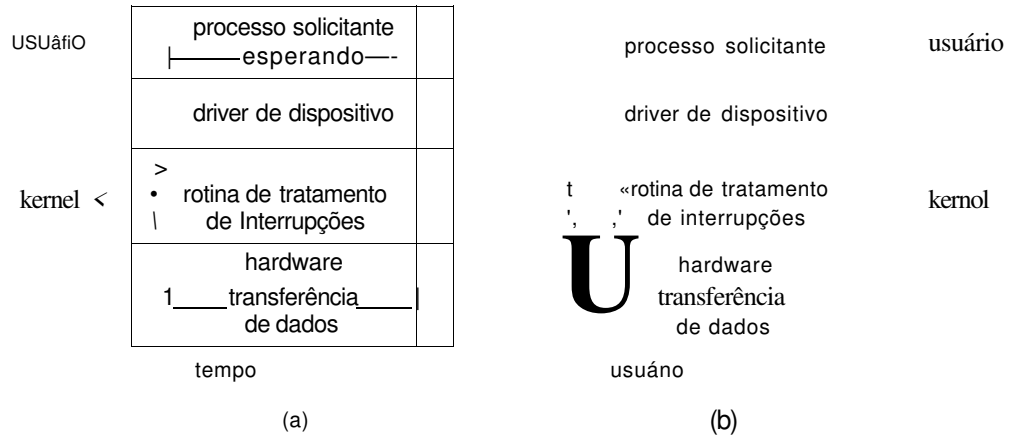


Figura 2.3 Dois métodos de I/O: (a) síncrona e (b) assíncrona.

A espera pela conclusão de l/O pode ser realizada de duas formas. Alguns computadores possuem uma instrução especial wait que torna a CPU inativa até a próxima interrupção. As máquinas que não possuem uma instrução desse tipo podem ter um laço de espera:

Loop: jmp *Loop*

Esse laço simplesmente continua até a ocorrência de uma interrupção, transferindo controle para outra parte do sistema operacional. Tal laço pode também incluir consultas *{potlmg)* a dispositivos de I/O que não forneçam suporte ã estrutura de interrupção; em vez disso, esses dispositivos simplesmente definem vim *flag* em um dos seus registradores e esperam que o sistema operacional o perceba.

Se a CPU sempre esperar pela conclusão de l/O, no máximo um pedido de 1/0 fica pendente em determinado momento. Assim, sempre que ocorre uma interrupção de l/O, o sistema operacional sabe exatamente que dispositivo está causando a interrupção. Por outro lado, essa abordagem exclui as operações de l/O concorrentes para vários dispositivos e exclui a possibilidade de sobreposição de computação útil com l/O.

Uma alternativa melhor é iniciar a operação de I/O e, em seguida, continuar o processamento de outro código de sistema operacional ou programa de usuário. Uma chamada ao sistema (um pedido ao sistema operacional) é então necessária para permitir que o programa de usuário espere pela conclusão de I/O, se desejado. Se nenhum programa de usuário estiver pronto para executar, e o sistema operacional não tiver outras tarefas a realizar, a instrução wai t ou o laço de espera ainda são necessários, como antes. Também é necessário manter um registro dos muitos pedidos de I/O simultâneos. Para isso, o sistema operacional utiliza uma tabela contendo uma entrada para cada dispositivo de I/O: a tabela de status de dispositivo (Figura 2.4). Cada entrada na tabela indica o tipo, endereço e estado do dispositivo (não funciona, ocioso ou ocupado). Se o dispositivo estiver ocupado com um pedido, o tipo de pedido e outros parâmetros serão armazenados na entrada da tabela para aquele dispositivo. Como é possível que outros processos emitam pedidos ao mesmo dispositivo, o sistema operacional também manterá uma fila de espera - uma lista de pedidos em espera - para cada dispositivo de I/O.

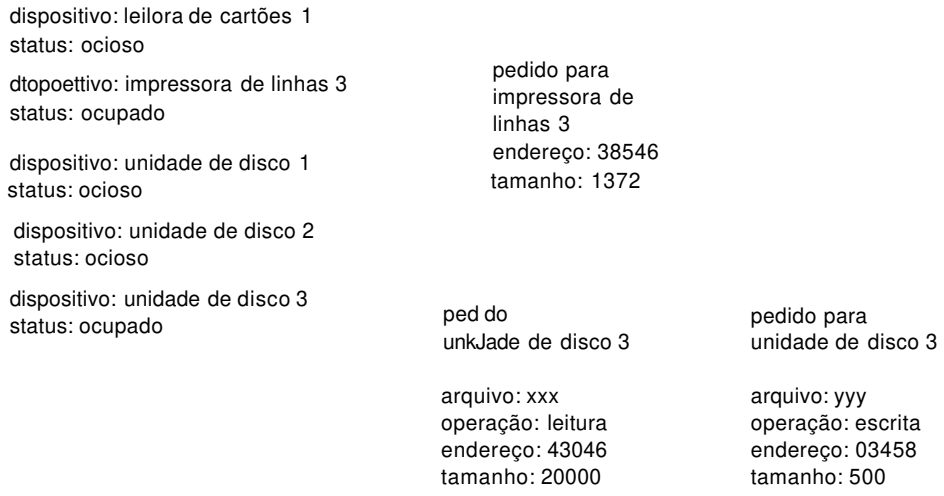


Figura 2.4 Tabela de status de dispositivo.

Um dispositivo de l/O interrompe quando precisa de serviço. Quando ocorre uma interrupção, o sistema operacional primeiro determina que dispositivo causou a interrupção. Em seguida, consulta a tabela de dispositivo de I/O para determinar o status desse dispositivo e modifica a entrada na tabela para rçfletir a ocorrência da interrupção. Para a maioria dos dispositivos, uma interrupção indica a conclusão de um pedido de I/O. Sc houver pedidos adicionais esperando na fila para este dispositivo, o sistema operacional começa a processar o próximo pedido.

Finalmente, o controle é devolvido da interrupção de I/O. Sc um processo estava esperando a conclusão desse pedido (como registrado na tabela de status de dispositivo), podemos devolver o controle a ele. Caso contrário, voltamos para o que quer que estivéssemos fazendo antes da interrupção de I/O: a execução do

programa de usuário (o programa iniciou uma operação de l/O e essa operação agora terminou, mas o programa ainda não esperou pela conclusão da operação) ou o laço de espera (o programa iniciou duas ou mais operações de I/O e está esperando que determinada operação termine, mas essa interrupção foi dc uma das outras operações). Em um sistema de tempo compartilhado, o sistema operacional poderia alternar para outro processo pronto para execução.

Os esquemas usados por dispositivos de entrada específicos podem variar. Muitos sistemas interativos permitem que os usuários antecipem a digitação - insiram dados antes que eles sejam solicitados - nos terminais. Nesse caso, podem ocorrer interrupções, indicando a chegada dc caracteres do terminal, embora o bloco de status de dispositivo indique que nenhum programa solicitou entrada desse dispositivo. Se a digitação antecipada for permitida, um buffer deverá ser fornecido para armazenar os caracteres até que algum programa os solicite. Km geral, pode ser necessário um buffer para cada terminal de entrada.

A principal vantagem da I/O assíncrona é maior eficiência do sistema. Enquanto ocorre a operação de /O, a CPU do sistema pode ser usada para processamento ou início de I/O para outros dispositivos. Como a I/O pode ser lenta comparada com a velocidade do processador, o sistema faz uso eficiente de seus recursos. Na Scção 2.2.2, outro mecanismo para melhorar o desempenho do sistema é apresentado.

2.2.2 Estrutura de DMA

Considere um driver simples de entrada no terminal. Quando uma linha vai ser lida do terminal, o primeiro caractere digitado é enviado ao computador. Quando esse caractere é recebido, o dispositivo de comunicação assíncrona (ou porta serial) ao qual a linha do terminal está conectada interrompe a CPU. Quando o pedido de interrupção do terminal chega, a CPU está pronta para executar alguma instrução. (Se a CPU estiver no meio da execução de uma instrução, a interrupção normalmente fica pendente até a conclusão da execução daquela instrução.) O endereço dessa instrução interrompida *é* salvo, e o controle é transferido à rotina de serviço de interrupção para o dispositivo adequado.

A rotina de serviço de interrupção salva o conteúdo dc todos os registradores da CPU que precisarão ser utilizados. Verifica se há condições de erro que possam ter resultado da operação dc entrada mais recente. Em seguida, pega o caractere do dispositivo e armazena esse caractere em um buffer. A rotina de interrupção também deve ajustar os contadores e ponteiros para ter certeza de que o próximo caractere de entrada será armazenado na próxima posição no buffer. Em seguida, a rotina de interrupção ativa um flag na memória indicando às outras partes do sistema operacional que uma nova entrada foi recebida. As outras partes são responsáveis por processar os dados no buffer c por transferir os caracteres ao programa que está solicitando a entrada (ver a Scção 2.5). Depois, a rotina dc serviço de interrupção restaura o conteúdo de todos os registradores salvos e transfere o controle de volta para a instrução interrompida.

Se os caracteres estiverem sendo digitados em um terminal de 9600 baud, o terminal pode aceitar e transferir um caractere aproximadamente a cada I milissegundo, ou 1000 microssegundos. Uma rotina de serviço de interrupção bem escrita pode precisar de 2 microssegundos por caractere para inserir caracteres em um buffer, deixando 998 microssegundos de cada 1000 para computação da CPU (e para o serviço dc outras interrupções). Considerando essa disparidade, a I/O assíncrona geralmente recebe baixa prioridade de interrupção, permitindo que outras interrupções mais importantes sejam processadas primeiro ou até mesmo que haja a preempção da interrupção atual. Um dispositivo de alta velocidade, no entanto, como uma fita, disco ou rede de comunicação, pode ser capaz de transmitir informações praticamente a velocidades de memória; a CPU precisaria de 2 microssegundos para responder a cada interrupção, com cada interrupção chegando a cada 4 microssegundos (por exemplo). Isso não deixaria muito tempo para a execução de processos.

Para resolver esse problema, o acesso direto à memória (DMA - Direct Memory Access) é usado para dispositivos dc l/O de alta velocidade. Depois de configurar os buffers, ponteiros e contadores para o dispositivo de l/O, a controladora de dispositivo transfere um bloco inteiro de dados diretamente entre seu próprio buffer de armazenamento e a memória, sem intervenção da CPU. Apenas uma interrupção é gerada por bloco, cm vez de uma interrupção por byte (ou palavra) gerada para dispositivos de baixa velocidade.

A operação básica da CPU é a mesma. Um programa de usuário, ou o próprio sistema operacional, pode solicitar a transferência de dados. O sistema operacional encontra um buffer (um buffer vazio para entrada ou um buffer cheio para saída) de um pool de buffers para a transferência. (Um buffer tem geralmente de 128

a 4.096 bytes, dependendo do tipo de dispositivo.) Em seguida, uma parte do sistema operacional chamada driver de dispositivo *(device driver)* configura os registradores da controladora de DMA para usar os endereços de origem e destino adequado c o tamanho da transferência. A controladora de DMA é então instruída a começar a operação de I/O. Enquanto a controladora de DMA está executando a transferência de dados, a CPU está livre para realizar outras tarefas. Como a memória geralmente pode transferir apenas uma palavra de cada vez, a controladora de DMA "rouba" ciclos de memória da CPU. Esse roubo de ciclos pode reduzir a velocidade de execução da CPU durante a transferência de DMA. A controladora de DMA interrompe a CPU quando a transferência estiver completa.

2.3 • Estrutura de armazenamento

Os programas de computador precisam estar na memória principal (também chamada de Memória de acesso aleatório, ou **RAM)** para serem executados. A memória principal é a única área de armazenamento (milhões a bilhões de bytes) que o processador pode acessar dirctamente. É implementada em uma tecnologia de semicondutores chamada RAM dinâmica ou DRAM, que forma um vetor de palavras na memória. Cada palavra tem seu próprio endereço. A interação é obtida através de uma sequência de instruções 1 oad ou store para endereços específicos na memória. A instrução 1 oad move uma palavra da memória principal para um registrador interno na CPU, enquanto uma instrução store move o conteúdo de um registrador para a memória principal. Além dessas instruções explícitas, a CPU carrega automaticamente instruções da memória principal para execução.

Um ciclo típico de execução de instrução, como realizado em um sistema com arquítetura von Ncumann, inicia buscando uma instrução na memória e armazenando essa instrução no registrador de instrução. A instrução é então decodificada e poderá fazer com que os operandos sejam buscados na memória e armazenados cm algum registrador interno. Depois que a instrução nos operandos tiver sido executada, o resultado pode ser armazenado de volta na memória. Observe que a unidade de memória vê apenas um fluxo de endereços de memória; não sabe como eles são gerados (o contador de instruções, indexação, indireções, endereços literais etc.) ou para que são usados (instruções ou dados). Da mesma forma, podemos ignorar *como* um endereço de memória é gerado por um programa. Estamos interessados apenas na sequência de endereços de memória gerados pelo programa em execução.

Idealmente, teríamos programas e dados residentes na memória principal de forma permanente. Isso não é possível pelos dois motivos seguintes:

1. A memória principal geralmente é pequena demais para armazenar todos os programas e dados necessários de forma permanente.
2. A memória principal é um dispositivo de armazenamento *volátil* que perde seu conteúdo quando há falta de energia ou esta é desligada.

Assim, a maioria dos sistemas de computação fornece armazenamento secundário como uma extensão da memória principal. O requisito básico do armazenamento secundário é que ele seja capaz de armazenar grandes quantidades de dados de forma permanente.

O dispositivo de armazenamento secundário mais comum é um disco magnético, que permite armazenar programas e dados. A maior parte dos programas (navegadores Web, compiladores, processadores de textos, planilhas eletrônicas, entre outros) é armazenada em disco até ser carregada na memória. Muitos programas usam o disco como origem e destino das informações para seu processamento. Portanto, a gerência adequada do armazenamento em disco é de fundamental importância para um sistema de computação, como discutido no Capítulo 13.

Em um sentido mais amplo, no entanto, a estrutura do armazenamento que descrevemos - consistindo em registradores, memória principal e discos magnéticos - é apenas um dos muitos sistemas de armazenamento possíveis. Existem também memória cache, CD-ROM, fitas magnéticas e assim por diante. Cada sistema de armazenamento fornece as funções básicas de armazenar um dado c de manter esse dado até que ele seja recuperado posteriormente. As principais diferenças entre os vários sistemas de armazenamento estão na velocidade, custo, tamanho e volatilidade. Nas Seções 2.3.1 a 2.3.3, há uma descrição da memória principal, discos magnéticos e fitas magnéticas, porque eles ilustram as propriedades gerais de todos os dispositivos de

armazenamento importantes em termos comerciais. No Capítulo 13, discutimos as propriedades específicas de muitos dispositivos em particular, tais como discos flexíveis, discos rígidos, CD-ROMs e DVDs.

2.3.1 Memória principal

A memória principal e os registradores incorporados ao próprio processador são o único tipo de memória que a CPU pode acessar diretamente. (Considere que existem instruções de máquina que usam endereços de memória como argumentos, mas não há nenhuma que use endereços de disco.) Portanto, quaisquer instruções em execução, e quaisquer dados sendo usados pelas instruções, devem estar em um desses dispositivos de armazenamento de acesso direto. Se os dados não estiverem na memória, deverão ser passados para lá antes que a CPU possa realizar qualquer operação com eles.

No caso de l/O, como mencionado na Seção 2.1, cada controladora de I/O inclui registradores para armazenar comandos c os dados sendo transferidos. Geralmente, instruções de I/O especiais permitem transferências de dados entre esses registradores e a memória do sistema. Para permitir acesso mais conveniente aos dispositivos de l/O, muitas arquiteturas de computador fornecem I/O mapcada em memória. Nesse caso, faixas de endereços de memória são reservadas e mapeadas nos registradores de dispositivos. Leituras e escritas nesses endereços de memória fazem com que os dados sejam transferidos de e para os registradores de dispositivos. Esse método é adequado para dispositivos que têm tempos de resposta rápidos, tais como controladoras de vídeo. No IBM PC, cada posição na tela é mapcada em uma posição na memória. Exibir um texto na tela é quase tão fácil quanto escrever o texto nas posições apropriadas mapeadas em memória.

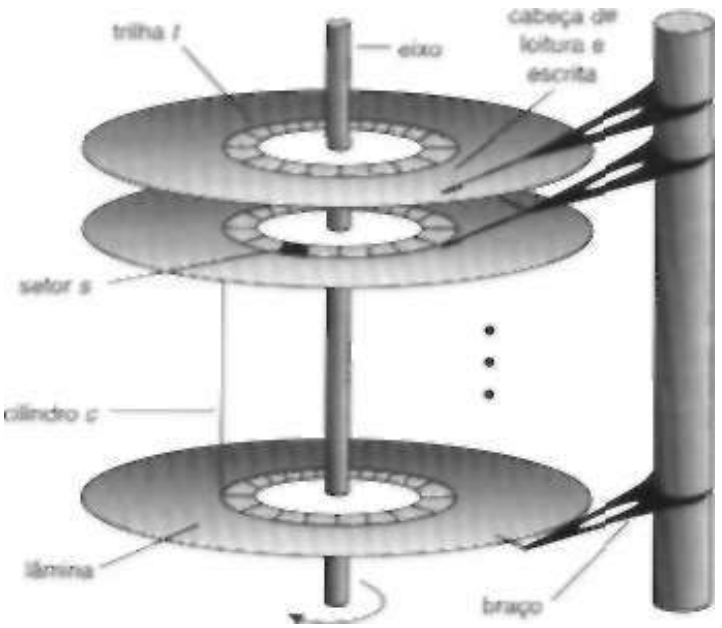
I/O mapeada em memória também é conveniente para outros dispositivos, tais como as portas seriais e paralelas usadas para conectar modems e impressoras a um computador. A CPU transfere dados através desses tipos de dispositivos fazendo a leitura e escrita de alguns registradores do dispositivo, chamados porta de I/O. Para enviar uma longa sequência de bytes através de uma porta serial mapeada em memória, a CPU escreve um byte de dados no registrador de dados e ativa um bit no registrador de controle para indicar que o byte está disponível. O dispositivo pega o byte de dados e limpa o bit no registrador de controle para indicar que está pronto para o próximo byte. Em seguida, a CPU pode transferir o próximo byte. Se a CPU utilizar *polling* para monitorar o bit de controle, constantemente retornando para ver se o dispositivo está pronto, esse método de operação é chamado de I/O programada (PIO). Se a CPU não faz *polling* do bit de controle, mas recebe uma interrupção quando o dispositivo está pronto para o próximo byte, a transferência de dados é considerada baseada em interrupção.

Os registradores internos à CPU geralmente são acessíveis em um ciclo do clock da CPU. A maioria das CPUs pode decodificar instruções e realizar operações simples sobre o conteúdo do registrador à taxa de uma ou mais operações por pulso do clock. O mesmo não se aplica à memória principal, que é acessada via uma transação no barramento de memória. O acesso à memória pode levar muitos ciclos até ser concluído e, nesse caso, o processador normalmente precisa aguardar já que não possui os dados necessários para concluir a instrução executada. Essa situação é intolerável, devido à frequência dos acessos à memória. A solução é acrescentar memória rápida entre a CPU e a memória principal. Um buffer de memória usado para acomodar essa diferença de velocidades, chamado cache, será descrito na Seção 2.4.1.

2.3.2 Discos magnéticos

Os discos magnéticos fornecem boa parte do armazenamento secundário para os sistemas de computação modernos. Em termos conceituais, os discos são relativamente simples (Figura 2.5). Cada lâmina de disco tem uma forma plana circular, como um CD. Seus diâmetros variam de 1,8 a 5,25 polegadas (4,6 a 13,34 cm). As duas superfícies de uma lâmina são cobertas por um material magnético semelhante a uma fita magnética. As informações são armazenadas por meio da escrita magnética nas lâminas.

Uma cabeça de leitura e escrita flutua logo acima de cada superfície da lâmina. As cabeças são acopladas a um braço de disco, que move todas as cabeças como uma unidade. A superfície de uma lâmina é logicamente dividida em trilhas circulares, que são subdivididas em setores. O conjunto de trilhas que está em uma posição do braço forma um cilindro. Pode haver milhares de cilindros concêntricos em uma unidade de disco, e



tolaçào

Figura 2.5 Mecanismo de movimentação de cabeça.

cada trilha pode conter centenas de setores. A capacidade de armazenamento de unidades de disco comuns é medida em gigabytes. (Um kilobyte é 1.024 bytes, um megabyte é 1.024^2 bytes e um gigabyte, 1.024* bytes, mas os fabricantes de disco geralmente arredondam esses números e dizem que I megabyte é 1 milhão de bytes c 1 gigabyte é 1 bilhão de bytes.)

Quando o disco está em uso, o motor da unidade gira em alta velocidade. A maioria das unidades gira de 60 a 150 vezes por segundo. A velocidade de disco tem duas partes. A taxa de transferência é a taxa na qual ocorre o fluxo de dados entre a unidade e o computador. O tempo de posicionamento, às vezes chamado tempo de acesso aleatório, consiste no tempo usado para mover o braço do disco até o cilindro desejado, denominado tempo de busca, e no tempo usado para que o setor desejado gire até a cabeça do disco, denominado latência rotacional- Os discos típicos transferem vários megabytes de dados por segundo e podem apresentar tempos de busca c latências rotacionais de vários milissegundos.

Como a cabeça de disco é sustentada sobre um colchão de ar extremamente fino (medido em micra), existe o risco de a cabeça entrar em conta to com a superfície do disco. Emboni as lâminas de disco sejam revestidas com uma fina camada protetora, às vezes a cabeça danifica a superfície magnética. Esse acidente é chamado choque da cabeça. Esse tipo de falha normalmente não pode ser reparado; o disco inteiro precisa ser substituído.

Um disco pode ser removível, permitindo que diferentes discos sejam montados conforme necessário. Discos magnéticos removíveis geralmente consistem em uma lâmina, mantida em um invólucro plástico para evitar danos quando não estiver na unidade de disco. Os discos flexíveis, ou disquetes, são discos magnéticos removíveis e baratos que têm um invólucro de plástico contendo uma lâmina flexível. A cabeça da unidade de disquete em geral fica diretamente sobre a superfície do disco, de modo que a unidade ê* projetada para girar mais lentamente do que uma unidade de disco rígido, o que reduz o desgaste na superfície do disco. A capacidade de armazenamento de um disco flexível geralmente fica em torno de apenas 1 megabyte. Existem discos removíveis disponíveis que funcionam de forma muito semelhante a discos rígidos normais e que têm sua capacidade medida em gigabytes.

Uma unidade de disco é conectada a um computador por uma série de cabos denominados barramento de I/O. Vários tipos de barramento estão disponíveis, incluindo EIDE c SCSI. As transferências de dados em um barramento são executadas por processadores eletrônicos especiais chamados controladoras. A controladora do host é a controladora na extremidade do barramento junto ao computador. Uma controladora de disco é incorporada em cada unidade de disco. Para realizar uma operação de I/O de disco, o computador emite um comando na controladora de host, geralmente usando portas de l/O mapeadas em memória, con-

forme descrito na Seção 2.3.1. A controladora do host envia então o comando via mensagens para a controladora de disco, e a controladora de disco, opera o hardware da unidade de disco para executar o comando. As controladoras de disco geralmente têm um cache incorporado. A transferência de dados na unidade de disco ocorre entre o cache e a superfície do disco e a transferência de dados para o host, em rápidas velocidades eletrônicas, ocorre entre o cache e a controladora do host.

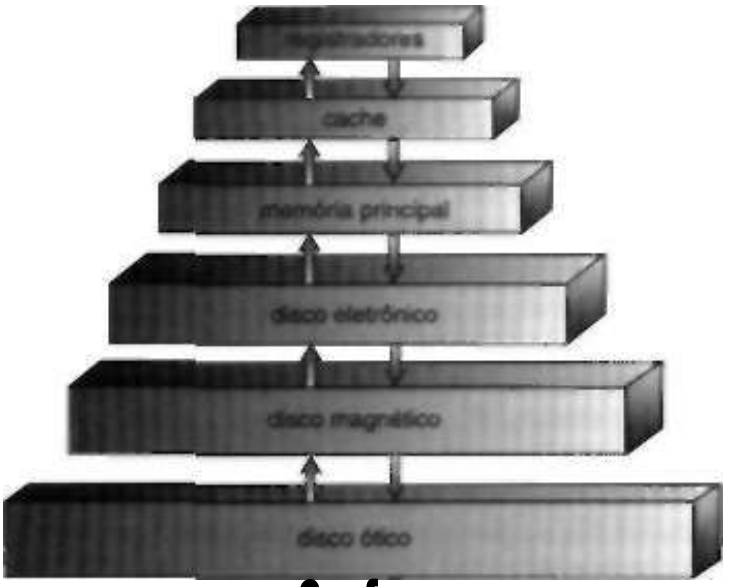
2.3.3 Fitas magnéticas

A fita magnética foi usada como um meio inicial do armazenamento secundário. Embora seja relativamente permanente, e possa armazenar grandes quantidades de dados, seu tempo de acesso é lento quando comparado àquele da memória principal. O acesso aleatório a fitas magnéticas c praticamente mil vezes mais lento do que o acesso aleatório a discos magnéticos, por isso as fitas não são úteis como armazenamento secundário. As fitas são basicamente usadas para backup, para o armazenamento de informações usadas com pouca frequência e como um meio de transferir informações de um sistema para outro.

Uma fita é mantida em um carretel c é rebobinada após passar por uma cabeça de leitura/escrita. Chegar ao ponto correto na fita pode levar alguns minutos, mas, uma vez posicionadas, as unidades de fita podem gravar dados em velocidades comparáveis às unidades de disco. A capacidade das fitas varia muito, dependendo do tipo específico de unidade de fita. Algumas fitas armazenam 20 vezes mais dados do que uma unidade de disco grande. As fitas e seus drivers geralmente são categorizados por largura, incluindo 2,4,e 19 milímetros, *VA* e *Vi* polegada.

2.4 • Hierarquia de armazenamento

A grande variedade de sistemas de armazenamento em um sistema de computação pode ser organizada cm uma hierarquia (Figura 2.6) de acordo com velocidade e custo. Os níveis mais altos são caros, mas rápidos. Ao descermos na hierarquia, o custo por bit geralmente diminui, enquanto o tempo de acesso em geral aumenta. Essa compensação é razoável; se determinado sistema de armazenamento fosse ao mesmo tempo mais rápido e mais barato do que outro - tendo outras propriedades iguais - não haveria motivo para usar memória mais lenta e mais cara. Na verdade, muitos dos primeiros dispositivos de armazenamento, incluindo fita de papel c memórias de núcleos, estão relegados a museus, agora que a fita magnética e a memória semtcondutora se tornaram mais rápidas e baratas.



&=fc

Mas magnéticas

1

Figura 2.6 Hierarquia de dispositivos de armazenamento.

Além de ter diferentes velocidades e custos, os vários sistemas de armazenamento são voláteis ou não-voláteis. O armazenamento volátil perde seu conteúdo quando é interrompida a energia para o dispositivo. Na falta de sistemas geradores e baterias de emergência caras, os dados devem ser gravados no armazenamento não-volátil para segurança. Na hierarquia apresentada na Figura 2.6, os sistemas de armazenamento acima dos vários discos são voláteis, enquanto os abaixo do disco cletrônico são não-voláteis. Um disco cletrônico pode ser volátil ou não-volátil. Durante a operação normal, o disco cletrônico armazena dados em um grande vetor DRAM, que é volátil. No entanto, muitos dispositivos de disco cletrônico contêm um disco rígido magnético oculto e uma bateria para energia reserva. Se a alimentação externa for interrompida, a controladora do disco cletrônico copia os dados da RAM para o disco magnético. Quando a força volta, a controladora copia os dados de volta na RAM.

O projeto de um sistema de memória completo deve balancear todos esses fatores: ele só vai usar memória cara quando necessário, fornecendo ao mesmo tempo a maior quantidade possível de memória não-volátil barata. Caches podem ser instalados para melhorar os déficits de desempenho onde houver uma grande disparidade de tempo de acesso ou taxa de transferência entre dois componentes.

2.4.1 Uso de cache

Caching é um princípio importante dos sistemas de computação. As informações são normalmente mantidas em algum sistema de armazenamento (como a memória principal). A medida que são usadas, as informações são copiadas para um sistema de armazenamento mais rápido - o cache - temporariamente. Quando precisamos de determinada informação, primeiro vamos verificar se ela está no cache. Se estiver, usamos os dados dirctamente do cache; se não estiver, usamos os dados do sistema de memória principal, colocando uma cópia no cache partindo do pressuposto de que existe uma alta probabilidade de que esses dados serão necessários novamente em breve.

Ampliando essa visão, registradores programáveis internos, como os registradores de índice, fornecem um cache de alta velocidade para a memória principal. O programador (ou o compilador) implementa os algoritmos de aloçação e substituição de registradores para decidir quais informações devem ser mantidas nos registradores e quais devem ser passadas para a memória principal. Existem também caches que são totalmente implementados em hardware. Por exemplo, a maioria dos sistemas tem um cache de instruções para armazenar as próximas instruções a serem executadas. Sem esse cache, a CPU teria de esperar vários ciclos enquanto uma instrução é buscada na memória principal. Por motivos semelhantes, a maioria dos sistemas tem um ou mais caches de dados de alta velocidade na hierarquia de memória. Não estamos preocupados com esses caches somente em hardware neste livro, pois estão fora do controle do sistema operacional.

Como os caches têm tamanho limitado, a gerência de caches é um importante problema de projeto. A Seleção cuidadosa do tamanho e de uma política de substituição de caches pode resultar em 80 a 99% de todos os acessos feitos no cache, causando um desempenho excepcionalmente alto. Vários algoritmos de substituição para caches controlados por software são discutidos no Capítulo 10.

A memória principal pode ser considerada um cache rápido para memória secundária, já que os dados no armazenamento secundário devem ser copiados para a memória principal para uso e os dados devem estar na memória principal antes de serem movidos para o armazenamento secundário por segurança. Os dados do sistema de arquivos podem aparecer cm vários níveis na hierarquia de armazenamento. No nível mais alto, o sistema operacional poderá manter um cache de dados do sistema de arquivos na memória principal. Além disso, discos de RAM eletrônicos (também chamados de discos de estado sólido) podem ser usados para armazenamento de alta velocidade, sendo acessados através da interface do sistema de arquivos. A maior parte do armazenamento secundário está em discos magnéticos. O armazenamento em disco magnético, por sua vez, frequentemente é copiado em fita ou discos removíveis para proteger contra perda de dados em caso de falha de disco rígido. Alguns sistemas arquivam automaticamente dados de arquivos antigos do armazenamento secundário no terciário, tais como *jukeboxes* de fita, para diminuir os custos de armazenamento (ver a Seção 13.7.2).

O movimento de informação entre os níveis de uma hierarquia de armazenamento pode ser explícito ou implícito, dependendo do projeto de hardware e do software de controle do sistema operacional. Por exemplo, a transferência de dados do cache para a CPU e registradores geralmente é uma função de hardware, sem interven-

ção do sistema operacional. Por outro lado, a transferência de dados do disco para a memória geralmente é controlada pelo sistema operacional.

2.4.2 Coerência e consistência

Em uma estrutura de armazenamento hierárquico, os mesmos dados podem aparecer em diferentes níveis do sistema de armazenamento. Por exemplo, considere um inteiro A localizado no arquivo B que deverá ser incrementado de 1. Vamos supor que o arquivo B resida em um disco magnético. A operação de incremento começa emitindo uma operação de l/O para copiar o bloco de disco no qual A reside para a memória principal. Essa operação c seguida por uma possível cópia de A para o cache e pela cópia de A para um registrador interno. Assim, a cópia de A aparece em vários locais. Depois que ocorre o incremento no registrador interno, o valor de A difere nos vários sistemas de armazenamento. O valor de A torna-se igual somente depois que o novo valor de A é gravado novamente no disco magnético.

Em um ambiente no qual só existe um processo executando de cada vez, esse arranjo não apresenta dificuldades, já que o acesso a um inteiro A sempre será a cópia no nível mais alto da hierarquia. No entanto, em ambientes multitarefa, nos quais a CPU alterna entre vários processos, deve-se tomar muito cuidado para garantir que, se vários processos desejam acessar A, cada um desses processos obterá o valor atualizado de A mais recente.

A situação fica mais complicada em ambientes com múltiplos processadores onde, além de manter registradores internos, a CPU também contém um cache local. Em ambientes desse ripo, uma cópia de A pode existir simultaneamente em vários caches. Como as várias CPUs podem executar concorrentemente, devemos garantir que uma atualização no valor de A cm um cache seja imediatamente refletida em todos os outros caches nos quais A reside. Essa situação é chamada coerência de cache e geralmente é um problema de hardware (tratado abaixo do nível do sistema operacional).

Em ambientes distribuídos, a situação fica ainda mais complexa. Em tais ambientes, várias cópias (réplicas) do mesmo arquivo podem ser mantidas em diferentes computadores distribuídos no espaço. Como as várias réplicas podem ser acessadas c atualizadas de forma concorrente, é preciso garantir que, quando uma réplica esteja sendo atualizada em um local, todas as demais também sejam atualizadas o quanto antes. Existem várias formas de conseguir isso, conforme discutido no Capítulo 17.

2.5 • Proteção de hardware

Os primeiros sistemas de computação eram sistemas operados por programador com usuário único. Quando os programadores operavam o computador da console, tinham controle completo sobre o sistema. A medida que os sistemas operacionais se desenvolveram, no entanto, esse controle foi passado ao sistema operacional. Começando com o monitor residente, o sistema operacional começou a executar muitas dessas funções, especialmente I/O, que eram responsabilidade do programador anteriormente.

Além disso, para melhorar a utilização do sistema, o sistema operacional começou a *compartilhar* recursos do sistema entre vários programas simultaneamente. Com o spooling, um programa poderia estar executando enquanto a l/O ocorria para outros processos; o disco mantinha dados simultaneamente para muitos processos. A multiprogramação colocou vários programas na memória ao mesmo tempo.

Esse compartilhamento melhorou a utilização c aumentou os problemas. Quando o sistema era executado sem compartilhamento, um erro em um programa poderia causar problemas apenas para o programa executado. Com o compartilhamento, muitos processos poderiam ser afetados negativamente por um bug em um dos programas.

Por exemplo, considere o sistema operacional cm batch simples (Seção 1.2), que fornece apenas o seqüenciamento automático de jobs. Vamos supor que um programa fique preso em um laço lendo cartões de entrada. O programa lerá todos os seus dados e, a menos que algo o interrompa, ele continuará lendo os carrões do próximo job, do seguinte, e assim por diante. Esse laço pode impedir a operação correta de muitos jobs.

Erros ainda mais sutis podem ocorrer em um sistema multiprogramado, em que um programa com erro pode modificar o programa ou dados de outro programa, ou mesmo do próprio monitor residente. O MS-DOS e o Macintosh OS permitem esse tipo de erro.

Sem proteção contra esses tipos de erros, o computador deve executar apenas um processo de cada vez, ou toda a saída deverá ficar sob suspeita. Um sistema operacional corretamente projetado deve garantir que um programa incorreto (ou malicioso) não cause a execução incorrera de outros programas.

Muitos erros de programação são detectados pelo hardware. Esses erros são normalmente tratados pelo sistema operacional. Se um programa de usuário talhar de algum modo - por exemplo, tentando executar uma instrução ilegal ou acessar memória que não esteja no espaço de endereçamento do usuário, o hardware vai gerar uma exceção, ou trap, para o sistema operacional. Essa exceção transfere controle por meio do vetor de interrupção para o sistema operacional, como uma interrupção. Sempre que ocorre um erro de programa, o sistema operacional deve encerrar o programa de forma anormal. Essa situação é tratada pelo mesmo código que um término anormal solicitado pelo usuário. Uma mensagem de erro adequada é exibida, e a memória do programa pode sofrer um dump. O dump de memória é geralmente gravado em um arquivo de modo que o usuário ou programador possam examiná-lo e talvez corrigir ou reiniciar o programa.

2.5.1 Operação em modo dual

Para garantir a operação adequada, devemos proteger o sistema operacional e todos os outros programas e seus dados de qualquer programa funcionando mal. A proteção é necessária para qualquer recurso compartilhado. A abordagem usada por muitos sistemas operacionais fornece suporte que permite diferenciar os vários modos de execução.

São necessários pelo menos dois modos de operação separados: o modo usuário e o modo monitor (também chamado de modo supervisor, modo sistema ou modo privilegiado). Um bit, chamado de bit de modo, é acrescentado ao hardware do computador para indicar o modo corretor monitor (0) ou usuário (1). Com o bit de modo, é possível distinguir entre uma tarefa que é executada em nome do sistema operacional e uma que é executada em nome do usuário. Como veremos, essa melhoria de arquitetura *á* útil para muitos outros aspectos da operação do sistema.

No momento da inicialização do sistema, o hardware inicia no modo monitor. O sistema operacional *é* então carregado c inicia os processos de usuário no modo usuário. Sempre que ocorrer uma exceção ou interrupção, o hardware alterna do modo usuário para o modo monitor (ou seja, muda o estado do bit de modo para 0). Assim, sempre que o sistema operacional obtiver controle do computador, ele estará no modo monitor. O sistema sempre alterna para o modo usuário (definindo o bit de modo para 1) antes de passar o controle a um programa de usuário.

O modo dual de operação fornece uma forma de proteger o sistema operacional contra usuários errantes, protegendo também os usuários errantes uns dos outros. Essa proteção é alcançada designando parte das instruções de máquina que podem causar dano como instruções privilegiadas. O hardware permite que as instruções privilegiadas sejam executadas apenas no modo monitor. Sc houver uma tentativa de executar uma instrução privilegiada no modo usuário, o hardware não executará a instrução, e a tratará como ilegal gerando uma exceção para o sistema operacional.

A falta de um modo dual com suporte de hardware pode causar sérias complicações em um sistema operacional. Por exemplo, o MS-DOS foi escrito para a arquitetura Intel 8088, que não possui bit de modo e, portanto, não tem modo dual. Um programa de usuário com problemas de execução pode apagar o sistema operacional gravando dados por cima dele, c múltiplos programas podem gravar em um dispositivo ao mesmo tempo, possivelmente com resultados desastrosos. Versões mais recentes e avançadas da CPU da Intel, tais como o Pentium, permitem a operação em modo dual. Consequentemente, sistemas operacionais mais recentes, tais como o Microsoft Windows NT e o IBM OS/2 podem aproveitar esse recurso e fornecer maior proteção para o sistema operacional.

2.5.2 Proteção de I/O

Um programa de usuário pode perturbar a operação normal do sistema emitindo instruções de I/O ilegais, acessando posições da memória no próprio sistema operacional ou recusando-se a liberar a CPU. Podemos usar vários mecanismos para evitar que tais problemas ocorram no sistema.

Para evitar que um usuário execute uma operação ilegal de l/O, definimos todas as instruções de I/O como instruções privilegiadas. Assim, os usuários não podem emitir instruções de I/O diretamente; devem fazê-lo por meio do sistema operacional. Para que a proreção de I/O esteja completa, é preciso ter certeza de que um programa de usuário nunca poderá obter controle do computador no modo monitor. Se pudesse, a proteçâo de I/O poderia estar comprometida.

Considere o computador executando em modo usuário. Ele alternará para o modo monitor sempre que ocorrer uma interrupção ou execção, passando para o endereço determinado pelo vetor de interrupção. Vamos supor que um programa de usuário, como parte de sua execução, armazena um novo endereço no vetor de interrupção. Esse novo endereço poderia sobrescrever o endereço anterior com um endereço no programa de usuário. Em seguida, quando ocorresse uma exceção ou interrupção, o hardware alternaria para o modo monitor e transferiria o controle por meio do vetor de interrupção (modificado) para o programa de usuário! O programa de usuário poderia obter controle do computador no modo monitor.

2.5.3 Protcção de memória

Para garantir a operação correta, c preciso proteger o vetor de interrupção contra qualquer modificação por parte de um programa de usuário. Além disso, também é preciso proteger as rotinas de serviço de interrupção no sistema operacional contra qualquer modificação. Caso contrário, um programa de usuário poderá sobrescrever as instruções na rotina de serviço de interrupção, introduzindo saltos para posições do seu programa c assim ganhando o controle das rotinas de serviço, que executam em modo privilegiado. Mesmo que o usuário não tenha obtido controle não-autorizado do computador, modificar as rotinas de serviço de interrupção provavelmente perturbaria a operação correta do sistema de computação e de seus mecanismos de spooling e buffering.

Vemos então que é preciso fornecer proteçâo de memória pelo menos para o vetor de interrupção e as rotinas de serviço de interrupção do sistema operacional. Em geral, é preciso proteger o sistema operacional do acesso por programas de usuário e, alem disso, proteger os programas de usuário uns dos outros. Essa proteçâo deve ser fornecida pelo hardware. Pode ser i mplemcntada de várias formas, como descrito no Capítulo 9. Aqui, fazemos um esboço de uma implementação possível.

Para separar o espaço de memória de cada programa é preciso ter a capacidade de determinar a faixa de endereços que o programa pode acessar e proteger a memória que estiver válida fora desse espaço. É possível fornecer essa proteçâo usando dois registradores: o de base e o de limite, conforme ilustrado na Figura 2.7. O registrador de base mantém o menor endereço de memória física válida; o registrador de limite contém o tamanho da faixa. Por exemplo, se o registrador de base contiver 300040 e o registrador de limite for 120900, o programa poderá acessar legalmente todos os endereços de 300040 a 420940, inclusive.

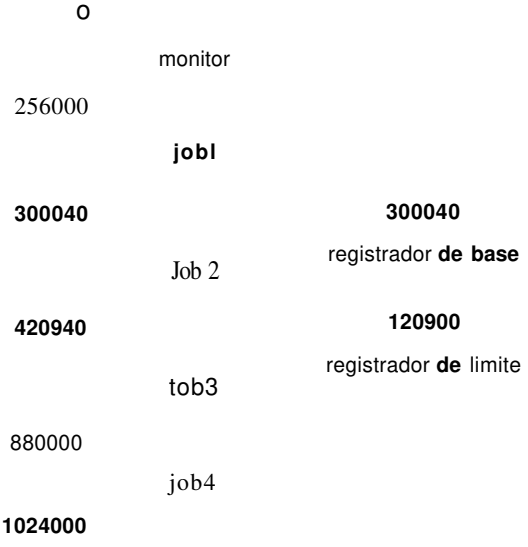


Figura 2.7 Um registrador de base e um registrador de limite definem um espaço de endereçamento lógico.

Essa proteção *é* alcançada pelo hardware da CPU, que compara *todo* endereço gerado em modo usuário com os registradores. Qualquer tentativa por parte de um programa executando em modo usuário de acessar a memória do monitor ou a memória de outros usuários resulta em uma exceção para o monitor, que trata a tentativa como um erro fatal (Figura 2.8). Esse esquema impede o programa de usuário de modificar (acidental ou deliberadamente) o código ou as estruturas de dados do sistema operacional ou de outros usuários.

Os registradores de base e limite podem ser carregados apenas pelo sistema operacional, que utiliza uma instrução privilegiada especial. Como as instruções privilegiadas podem ser executadas apenas em modo monitor e como só o sistema operacional executa em modo monitor, somente o sistema operacional pode carregar os registradores de base e limite. Esse esquema permite que o monitor altere o valor dos registradores, mas impede que os programas de usuário mudem o conteúdo dos registradores.

O sistema operacional, ao executar em modo monitor, recebe acesso irrestrito à memória do monitor e dos usuários. Isso permite que o sistema operacional carregue os programas de usuários na memória, faça o dump desses programas em caso de erro, acesse c modifique os parâmetros das chamadas ao sistema etc.

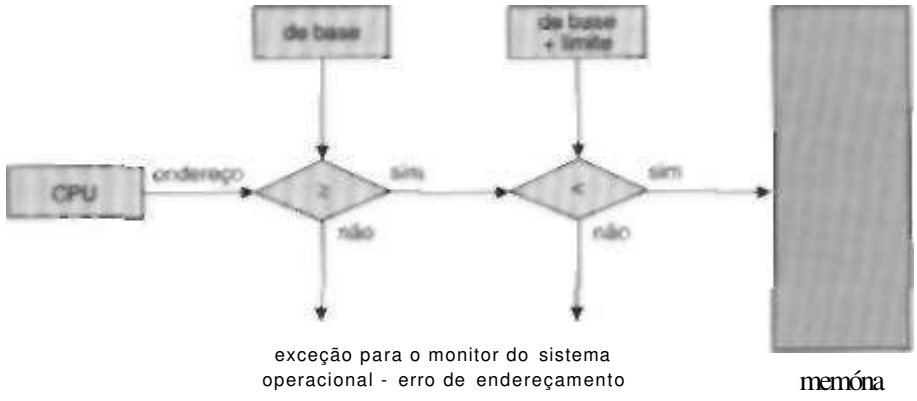


Figura 2.8 Proteção de endereços via hardware com registradores de base e limite.

2.5.4 Proteção de CPU

A terceira peça do quebra-eabeças de proteção é garantir que o sistema operacional mantenha o controle. É preciso evitar que um programa de usuário fique preso em um laço infinito sem nunca devolver o controle ao sistema operacional. Para alcançar essa meta, podemos usar um timer (temporizador). Um timer pode ser configurado para interromper o computador após um período específico. O período pode ser fixo (por exemplo, 1/60 de segundo) ou variável (por exemplo, de 1 milissegundo a 1 segundo, em incrementos de 1 milissegundo). Um timer variável geralmente *é* implementado por um clock de taxa fixa e um contador. O sistema operacional inicializa o contador. A cada pulso do clock, o contador é decrementado. Quando o contador chega a 0, ocorre uma interrupção. Por exemplo, um contador de 10 bits com um clock de 1 milissegundo permitiria interrupções em intervalos de I a 1.024 milissegundos em passos de I milissegundo.

Antes de passar o controle ao usuário, o sistema operacional garante que o timer esteja preparado para fazer *a* interrupção. Sc o timer interromper, o controle é automaticamente transferido para o sistema operacional, que poderá tratar a interrupção como um erro fatal ou dar mais tempo ao programa. As instruções que modificam a operação do timer são claramente privilegiadas.

Assim, podemos usar o timer para impedir que um programa de usuário execute por muito tempo. Uma técnica simples *é* inicializar o contador com o período que determinado programa pode executar. Um programa com um limite de tempo de 7 minutos, por exemplo, teria seu contador inicializado em 420. A cada segundo, o timer interrompe e o contador é diminuído em 1, Enquanto o contador estiver positivo, o controle é devolvido ao programa de usuário. Quando o contador fica negativo, o sistema operacional encerra o programa por exceder o limite de tempo atribuído.

Um uso mais comum do timer é implementar o compartilhamento de tempo. No caso mais simples, o timer poderia ser configurado para interromper a cada *N* milissegundos, onde N é a fatia de tempo que cada usuário pode executar antes do próximo usuário obter o controle da CPU. O sistema operacional é chamado no final de cada fatia de tempo para realizar as várias tarefas administrativas, tais como somar o valor N ao re-

gistro que especifica a quantidade de tempo que um programa de usuário executou até aquele momento (para fins de contabilização). O sistema operacional também reinicia registradores, variáveis internas e buffers, e muda vários outros parâmetros para preparar a execução do próximo programa. (Esse procedimento é chamado *troca de contexto* e será abordado no Capítulo 4.) Depois da troca de contexto, o próximo programa continua a execução no ponto em que parou (quando a fatia de tempo anterior expirou).

Outro uso do timet c computar a hora atual. Uma interrupção do timer indica a passagem de algum espaço de tempo, permitindo que o sistema operacional calcule a hora atual em referência a alguma hora inicial. Se tivermos interrupções a cada 1 segundo, e tivermos tido 1.427 interrupções desde as 13:00, podemos calcular que a hora atual é 13:23:47. Alguns computadores determinam a hora atual dessa forma, mas os cálculos devem ser feitos cuidadosamente para que a hora seja precisa, já que o tempo de processamento da interrupção (e outros tempos transcorridos enquanto as interrupções estiverem desabilitadas) tende a diminuir a velocidade do clock de software. Muitos computadores têm um relógio horário de hardware separado, independente do sistema operacional.

2.6 • Arquitetura geral do sistema

O desejo de melhorar a utilização do sistema de computação levou ao desenvolvimento da multiprogramaÇão e do tempo compartilhado, nos quais os recursos do sistema de computação são compartilhados entre vários programas e processos diferentes. O compartilhamento levou diretamente a modificações da arquitetura básica do computador para permitir ao sistema operacional manter controle sobre o sistema de computação, especialmente sobre I/O. O controle deve ser mantido a fim de fornecer operação contínua, consistente e correia.

Para manter controle, os desenvolvedores introduziram um modo dual de execução (modo usuário e modo monitor). Esse esquema dá suporte ao conceito de instruções privilegiadas, que somente podem ser executadas no modo monitor. As instruções de I/O e as instruções para modificar os registradores de gerência de memória ou o timer são instruções privilegiadas.

Como se pode imaginar, várias outras instruções também são classificadas como privilegiadas. Por exemplo, a instrução hal t é privilegiada; um programa de usuário nunca deve poder parar o computador. As instruções para ativar c desativar o sistema de interrupção também são privilegiadas, já que a operação adequada do timer e de l/O dependem da capacidade de responder a interrupções corrctatnente. A instrução para mudar do modo usuário para o modo monitor é privilegiada e, em muitas máquinas, qualquer mudança no bit de modo é privilegiada.

Como as instruções de l/O são privilegiadas, só podem ser executadas pelo sistema operacional. Então, como o programa de usuário realiza operações de I/O? Tornando as instruções de I/O privilegiadas, evitamos que os programas de usuário realizem operações de l/O, quer válidas ou inválidas. A solução a esse problema é que, como apenas o monitor pode realizar operações de I/O, o usuário deve *solicitar* ao monitor que faça I/O em nome do usuário.

Tal pedido é denominado chamada ao sistema (também denominado chamada ao monitor ou chamada de função do sistema operacional). Uma chamada ao sistema é feita de várias formas, dependendo da funcionalidade fornecida pelo processador subjacente. Em todas as formas, é o método usado por um processo para solicitar ação pelo sistema operacional. Uma chamada ao sistema geralmente assume a forma de uma exceção para uma posição específica no vetor de interrupção. Essa exceção pode ser executada por uma instrução genérica trap, embora alguns sistemas (tais como a família MIPS R2000) tenham uma instrução syscall específica.

Quando uma chamada ao sistema é executada, ela é tratada pelo hardware como uma interrupção de software. O controle passa pelo vetor de interrupção para uma rotina de serviço no sistema operacional, c o bit de modo é definido para o modo monitor. A rotina de serviço de chamada ao sistema é parte do sistema operacional. O monitor examina a instrução de interrupção para determinar qual chamada ao sistema ocorreu; um parâmetro indica que tipo de serviço o programa de usuário está solicitando. Informações adicionais necessárias para o pedido podem ser passadas em registradores, na pilha, ou na memória (com ponteiros para as posições da memória passados nos registradores). O monitor verifica se os parâmetros estão corretos e válidos, executa o pedido e devolve o controle para a instrução após a chamada ao sistema.

Assim, para realizar uma operação de l/O, um programa de usuário executa uma chamada ao sistema para solicitar que o sistema operacional realize l/O em seu nome (Figura 2.9). 0 sistema operacional, executando cm modo monitor, verifica se 0 pedido é válido e (se o pedido for válido) executa a operação de l/O solicitada. O sistema operacional retorna então para o usuário.

2.7 • Resumo

Os sistemas de mui ti programação e de tempo compartilhado melhoram o desempenho fazendo a sobreposição das operações de I/O c da CPU em uma única máquina. Tal sobreposição requer que a transferência de ciados entre a CPU e um dispositivo de I/O seja realizada por acesso baseado em interrupções ou *polling* a uma porta de l/O, ou por uma transferência de dados DMA.

Para que um computador realize sua tarefa de execução de programas, os programas devem estar na memória principal. A memória principal é a única área de armazenamento grande que o processador pode acessar dirctamente. E uma matriz de palavras ou bytes, variando cm tamanho de centenas de milhares a centenas de milhões. Cada palavra tem seu próprio endereço. A memória principal é um dispositivo de arma?.enamento volátil que perde seu conteúdo quando a fonte de alimentação é desligada ou perdida. A maioria dos sistemas de computação fornece armazenamento secundário como uma extensão da memória principal. O principal requisito do armazenamento secundário é ser capaz de armazenar grandes quantidades de dados de forma permanente. O dispositivo de armazenamento secundário mais comum é um disco magnético, que armazena programas c dados. Um disco magnético é um dispositivo de armazenamento não-volátil que também fornece acesso aleatório. As fitas magnéticas são usadas basicamente para backup, para armazenamento de informações usadas com pouca frequência e como meio para transferir informações de um sistema para outro.

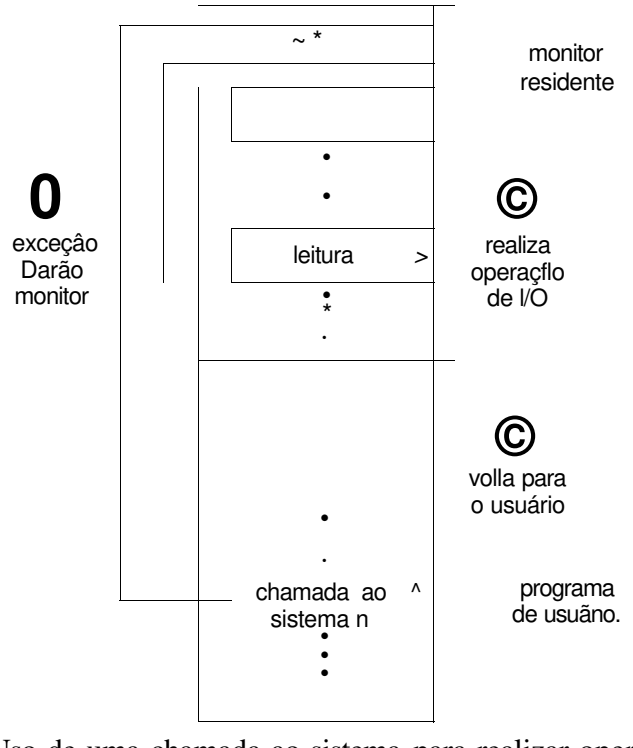


Figura 2.9 Uso de uma chamada ao sistema para realizar operações de l/O.

A grande variedade de sistemas de armazenamento em um sistema de computação pode ser organizada em uma hierarquia de acordo com sua velocidade e custo. Os níveis mais altos são caros, mas rápidos. À medida que descemos na hierarquia, o custo por bit geralmente diminui, enquanto o tempo de acesso geralmente aumenta.

O sistema operacional precisa garantir a operação correta do sistema de computação. Para evitar que os programas de usuário interfiram na operação correta do sistema, o hardware possui dois modos: o modo usuário e o modo monitor. Várias instruções (como as instruções de I/O e as instruções de parada) são privile-

gíadas c só podem ser executadas no modo monitor. A memória na qual o sistema operacional reside também deve estar protegida contra modificações pelo usuário. Um timer impede laços infinitos. Esses recursos (modo dual, instruções privilegiadas, proteção de memória, interrupção do timer) são os blocos básicos usados pelos sistemas operacionais para alcançar a operação correta. O Capítulo 3 continua essa discussão com detalhes dos recursos fornecidos pelos sistemas operacionais.

• Exercícios

2.1 *Prefetching* (busca prévia) é um método de sobrepor a I/O de um job com a computação do próprio job. A ideia é simples. Depois da conclusão de uma operação de leitura e que um job está prestes a começar a trabalhar com os dados, o dispositivo de entrada é instruído a começar a próxima leitura imediatamente. A CPU e o dispositivo de entrada ficam ocupados. Com sorte, quando o job estiver pronto para o próximo item de dados, o dispositivo de entrada terá terminado de ler aquele item de dados. A CPU pode então começar o processamento dos dados recém-lidos, enquanto o dispositivo de leitura começa a ler os dados seguintes. Uma ideia semelhante pode ser usada para saída. Nesse caso, o job cria dados que são colocados em um buffer até que um dispositivo de saída possa aceitá-los.

Compare o esquema de *prefetching* ao *spooling,* onde a CPU sobrepõe a entrada de um job com a computação e a saída de outros jobs.

2.2 Como a distinção entre o modo monitor e o modo usuário funciona como uma forma rudimentar de sistema de proteção (segurança)?

- 2.3 Quais são as diferenças entre uma exceção e uma interrupção? Qual é o uso de cada função?

2.4 Para que tipos de operações o DMA é útil? Justifique sua resposta.

2.5 Quais das seguintes instruções devem ser privilegiadas?

a. Definir o valor do timer
b. Ler o clock
c. Limpar a memória
d. Desligar as interrupções
e. Passar do modo usuário para o modo monitor

2.6 Alguns sistemas de computação não fornecem um modo privilegiado de operação em hardware. Considere se é possível construir um sistema operacional seguro para esses computadores. Apresente argumentos a favor e contra.

2.7 Alguns dos primeiros computadores protegiam o sistema operacional colocando-o em uma partição de memória que não podia ser modificada pelo job de usuário ou pelo próprio sistema operacional. Descreva duas dificuldades que você acha que podem surgir com um esquema desse tipo.

2.8 Proteger o sistema operacional é crucial para garantir que o sistema de computação opere corretamente. Fornecer essa proteção é o motivo da operação em modo dual, proteção de memória e o timer. Para permitir flexibilidade máxima, no entanto, também deveríamos ter limitações mínimas para o usuário.

A seguir está uma lista de instruções que normalmente são protegidas. Qual é o conjunto *mínimo* de instruções que devem ser protegidas?

a. Passar para o modo usuário
b. Mudar para o modo monitor
c. Ler a memória do monitor
d. Escrever na memória do monitor
e. Buscar uma instrução na memória do monitor
f. Ativar a interrupção do timer
g. Desativar a interrupção do timer

2.9 Apresente dois motivos pelos quais os caches são úteis. Que problemas eles resolvem? Que problemas causam? Se um cache puder ser tão grande quanto o dispositivo que ele representa (por exem-

pio, um cache tão grande quanto um disco), por que não deixar ele ficar desse ramanho e eliminar o dispositivo?

2.10 Escrever um sistema operacional que possa operar sem interferência de programas de usuário maliciosos ou sem depuração requer assistência do hardware. Cite três auxílios de hardware para escrever um sistema operacional e descreva como eles podem ser usados em conjunto para proteger o sistema operacional.

2.11 Algumas CPUs fornecem mais do que dois modos de operação. Quais são os dois possíveis usos desses múltiplos modos?

Notas bibliográficas

Hennessy c Patterson [1996] abordam sistemas e barramentos de I/O, e a arquitetura de sistema em geral. Tanenbaum 11990] descreve a arquitetura de microcomputadores, começando em um nível de hardware detalhado.

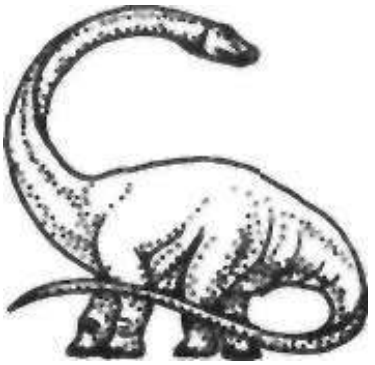
Discussões com relação à tecnologia de disco magnético são apresentadas por Frcedman [1983] e Harkcr C colegas 11981]. Discos óticos são abordados por Kenville 11982], Fujitani [19841, 0'l.eary e Kitts 11985], Gait [1988], c Olscn e Kenley 11989]. Discussões sobre discos flexíveis são apresentadas por Pechura e Schoefflcr [1983] c por Sarisky [1983].

Os caches de memória, incluindo a memória associativa, são descritos e analisados por Smith [19821. -Esse trabalho também inclui uma bibliografia extensa sobre o assunto. Hennessy e Patterson [1996] discutem os aspectos de hardware de TLBs, caches e MMUs.

Discussões gerais relativas à tecnologia do armazenamento de massa são oferecidas por Chi [1982] e por Hoagland[1985|.

Capítulo 3

ESTRUTURAS DE SISTEMAS OPERACIONAIS



Um sistema operacional fornece o ambiente no qual os programas são executados. Internamente, os sistemas operacionais variam muito em sua constituição, sendo organizados em muitas linhas diferentes. O projeto de um novo sistema operacional é uma tarefa importante. Portanto, é fundamental que os objetivos do sistema sejam bem definidos antes do início do projeto. O tipo de sistema desejado é a base para as escolhas entre os vários algoritmos e estratégias.

Existem vários pontos de vista usados para analisar um sistema operacional. Um deles é examinando os serviços que ele fornece. Outro é analisando a interface que ele disponibiliza a usuários e programadores. Um terceiro é desmontar o sistema em seus componentes c suas interconexões. Neste capítulo, exploramos todos os três aspectos dos sistemas operacionais, mostrando os pontos de vista dos usuários, programadores e projedstas de sistemas operacionais. Consideramos que serviços são fornecidos por um sistema operacional, como são fornecidos e quais são as várias metodologias para projetar tais sistemas.

3.1 • Componentes do sistema

E possível criar um sistema tão grande e complexo quanto um sistema operacional simplesmente dividindo-o em partes menores. Cada uma dessas partes deve ser uma porção bem delineada do sistema, com entradas, saídas e funções cuidadosamente definidas. Obviamente, nem todos os sistemas têm a mesma estrutura. No entanto, muitos sistemas modernos compartilham a meta de dar suporte aos componentes do sistema delineados nas Seções 3.1.1 a 3.1.8.

3.1.1 Gerência de processos

Um programa não faz nada a não ser que suas instruções sejam executadas por uma CPU. Um processo pode ser considerado um programa em execução, mas sua definição será ampliada à medida que explorarmos melhor o conceito. Um programa de usuário de tempo compartilhado, como um compilador, é um processo. Um processador de textos executado por um usuário individual em um PC é um processo. Uma tarefa de sistema, como enviar saída para uma impressora, também é um processo. Por enquanto, podemos considerar que um processo é um job ou um programa de tempo compartilhado, mas mais tarde você vai aprender que o conceito é mais geral. Como veremos no Capítulo 4, é possível fornecer chamadas ao sistema que permitem aos processos criar subprocessos para a execução concorrente.

Um processo precisa de determinados recursos - incluindo tempo de CPU, memória, arquivos e dispositivos de I/O - para realizar sua tarefa. Esses recursos são dados ao processo quando ele é criado ou alocados a ele durante sua execução. Além dos vários recursos físicos e lógicos que um processo obtém quando é criado, vários dados de inicialização (entrada) podem ser fornecidos. Por exemplo, considere um processo cuja função seja exibir o status de um arquivo na tela de um terminal. O processo receberá como entrada o nome do arquivo e executará as instruções e chamadas ao sistema adequadas para obter c exibir as informa-

ções desejadas no terminal. Quando o processo terminar, o sisrema operacional solicitará de volta quaisquer recursos reutilizáveis.

Enfatizamos que um programa por si só não é um processo; um programa é uma entidade *passiva,* como o conteúdo de um arquivo armazenado em disco, enquanto um processo é uma entidade *ativa,* com um contador do programa especificando a próxima instrução a ser executada. A execução de um processo deve ser sequencial. A CPU executa uma instrução do processo após a outra ate- o processo terminar. Além disso, a qualquer momento, no máximo uma instrução é executada em nome do processo. Assim, embora dois processos possam ser associados com o mesmo programa, eles são considerados duas sequências de execução separadas. E comum ter um programa que utilize muitos processos para sua execução.

Um processo é a unidade de trabalho em um sistema. Um sistema desse tipo consiste em uma coleção de processos, alguns dos quais são processos de sistema operacional (aqueles que executam código do sistema) e o resto são processos de usuário (aqueles que executam código de usuário). Todos esses processos podem executar concorrentemente, multiplexando a CPU entre eles.

O sistema operacional é responsável pelas seguintes atividades em relação à gerência de processos:

- Criar e excluir processos de usuário e de sistema
- Suspender e retomar processos
- Fornecer mecanismos para a sincronização de processos
- Fornecer mecanismos para a comunicação de processos
- Fornecer mecanismos para o tratamento de deadlocks

Os Capítulos 4 a 7 discutem as técnicas de gerência de processos

3.1.2 Gerência da memória principal

Como discutido no Capítulo 1, a memória principal é central à operação de um sistema de computação moderno. A memória principal é um grande vetor de palavras ou bytes, variando em tamanho de centenas de milhares a bilhões. Cada palavra ou byte tem seu próprio endereço. A memória principal é um repositório de dados rapidamente acessíveis compartilhados pela CPU e dispositivos de I/O. O processador central lê as instruções da memória principal durante o ciclo de busca de instruções, e lê e grava dados da memória principal durante o ciclo de busca de dados. As operações de I/O implementadas via DMA também fazem a leitura e escrita de dados na memória principal. A memória principal geralmente é o único dispositivo de armazenamento grande que a CPU pode endereçar e acessar diretamente. Por exemplo, para que a CPU processe os dados do disco, esses dados devem primeiro ser transferidos para a memória principal por chamadas de I/O geradas pela CPU. Do mesmo modo, as instruções devem estar na memória principal para que a CPU as execute.

Para que um programa seja executado, ele deve ser mapeado para endereços absolutos e carregados na memória. A medida que o programa executa, ele acessa instruções e dados do programa a partir da memória gerando esses endereços absolutos. Por fim, o programa termina, seu espaço de memória é declarado disponível e o próximo programa pode ser carregado e executado.

Para melhorar a utilização da CPU e a velocidade da resposta do computador aos seus usuários, é preciso manter vários programas na memória. Existem muitos esquemas diferentes de gerência de memória. Esses esquemas refletem as várias abordagens à gerência de memória, e a eficácia dos diferentes algoritmos depende de de cada situação específica. A Seleção de um esquema de gerência de memória para um sistema específico depende de muitos fatores - especialmente do projeto de *hardware* do sistema. Cada algoritmo requer seu próprio suporte de hardware.

O sistema operacional é responsável pelas seguintes atividades em relação à gerência de memória:

- Manter registro das partes da memória que estão sendo usadas no momento e por quem
- Decidir que processos deverão ser carregados na memória quando houver espaço disponível
- Alocar e dcsalocar espaço na memória, conforme necessário

As técnicas de gerência de memória serão discutidas nos Capítulos 9 e 10.

3.1.3 Gerência de arquivos

A gerência de arquivos é um dos componentes mais visíveis de um sistema operacional. Os computadores podem armazenar informações em vários tipos di ferentes de meios físicos. A fita magnética, o disco magnético e o disco ótico são os meios mais comuns. Cada um desses meios possui suas próprias características e organização física. Cada meio é controlado por um dispositivo, como uma unidade de disco ou fita, que também tem suas características exclusivas. Essas propriedades incluem velocidade de acesso, capacidade, taxa de transferência de dados e método de acesso (sequencial ou aleatório).

Para o uso conveniente do sistema de computação, o sistema operacional fornece uma visão lógica uniforme do armazenamento de informações. O sistema operacional abstrai as propriedades físicas de seus dispositivos de armazenamento para definir uma unidade lógica de armazenamento, o arquivo. O sistema operacional mapcia os arquivos nos meios físicos e acessa esses arquivos através dos dispositivos de armazenamento.

Um arquivo é uma coleção de informações relacionadas definidas por seu criador. Geralmente, os arquivos representam programas (fonte e objeto) e dados. Os arquivos de dados podem ser numéricos, alfabéticos ou alfanuméricos. Além disso, podem ter forma livre (por exemplo, arquivos de texto) ou podem ter uma formatação rígida (por exemplo, campos fixos). Um arquivo consiste em uma sequência de bits, bytes, linhas ou registros cujos significados são definidos por seus criadores. O conceito de arquivo é bastante geral.

O sistema operacional implementa o conceito abstraio de um arquivo gerenciando mídia de armazenamento de massa, como fitas e discos, c os dispositivos que os controlam. Além disso, os arquivos são normalmente organizados em diretórios para facilitar seu uso. Finalmente, quando vários usuários têm acesso aos arquivos, pode ser desejável controlar quem pode acessar os arquivos, e de que forma pode fazê-lo (por exemplo, para leitura, escrita, acréscimo).

O sistema operacional é responsável pelas seguintes atividades em relação à gerência de arquivos:

- Criar e excluir arquivos
- Criar e excluir diretórios
- Fornecer suporte a primitivas para manipular arquivos e diretórios
- Mapear arquivos no armazenamento secundário
- Fazer backup de arquivos em meios de armazenamento estáveis (não-voláteis)

As técnicas de gerência de arquivos serão discutidas no Capítulo 11.

3.1.4 Gerência do sistema de I/O

Um dos objetivos de um sistema operacional é ocultar as peculiaridades de dispositivos de hardware específicos do usuário. Por exemplo, no UNIX, as peculiaridades dos dispositivos de I/O são ocultadas do grosso do sistema operacional propriamente dito pelo subsistema de I/O. O subsistema de I/O consiste em:

- Um componente de gerência de memória que inclui *buffering,* armazenamento em cache e *spooting*
- Uma interface geral de driver de dispositivo
- Drivcrs para dispositivos de hardware específicos

Apenas o driver de dispositivo conhece as peculiaridades do dispositivo específico ao qual foi atribuído.

No Capítulo 2, discutimos como as rotinas de tratamento de interrupções e os drivers de dispositivo são usados na construção de subsistemas de I/O eficientes. No Capítulo 12, discutimos como o subsistema de I/O se relaciona com os outros componentes do sistema, gerência dispositivos, transfere dados e detecta a conclusão de operações de I/O.

3.1.5 Gerência de armazenamento secundário

O principal objetivo de um sistema de computação é executar programas. Esses programas, com os dados que acessam, devem estar na memória principal, ou armazenamento primário, durante a execução. Como a memória principal é pequena demais para acomodar todos os dados e programas, e como os dados que armazena são perdidos quando a energia é desconectada, o sistema de computação deve fornecer armazenamento

secundário para dar suporte ã memória principal. A maioria dos sistemas de computação modernos usam discos como o principal meio de armazenamento onlinc, para programas e dados. A maioria dos programas» incluindo compiladores, montadores, rotinas de classificação, editores e formatadores, são armazenados em um disco até serem carregados na memória e utilizam o disco como origem e destino de seu processamento. Portanto, a gerência adequada do armazenamento em disco é de importância crucial para um sistema de computação, O sistema operacional é responsável pelas seguintes atividades em relação à gerência de disco:

- Gerência de espaço livre
- Alocação de espaço (armazenamento)
- Escalonamento de disco

Como o armazenamento secundário é usado com frequência, ele deve ser utilizado com eficiência. A velocidade global de operação de um computador pode depender muito das velocidades do subsistema de disco e dos algoritmos que manipulam o subsistema. As técnicas para a gerência de armazenamento secundário serão discutidas no Capítulo 13.

^3.1.6 Redes

Um sistema distribuído é uma coleção de processadores que não compartilham memória, dispositivos periféricos_ou um clock. Em vez disso, cada processador tem sua própria memória local e clock, e os processadores se comunicam entre si através de várias linhas de comunicação, como barramentos ou redes de alta velocidade. Os processadores em um sistema distribuído variam em tamanho e função. Podem incluir pequenos microprocessadores, estações de trabalho, minicomputadores e grandes sistemas de computação de uso geral.

Os processadores no sistema são conectados através de uma rede de comunicação, que pode ser configurada de várias formas diferentes. A rede pode ser total ou parcialmente conectada. O projeto da rede de comunicação deve considerar as estratégias de conexão e roteamento de mensagens, e os problemas de disputa e segurança.

Um sistema distribuído reúne sistemas fisicamente separados e possivelmente heterogéneos em um único sistema coerente, fornecendo ao usuário acesso aos vários recursos mantidos pelo sistema. O acesso a um recurso compartilhado permite maior velocidade de computação, maior funcionalidade, maior disponibilidade de dados e melhor confiabilidade. Os sistemas operacionais geralmente generalizam o acesso à rede como uma forma de acesso a arquivos, com os detalhes da rede estando contidos no driver de dispositivo de interface da rede. Os protocolos que criam um sistema distribuído podem ter um grande efeito na utilidade e popularidade do sistema. A inovação da World Wide Web foi criar um novo método de acesso para compartilhamento de informações. Melhorou o protocolo de transferência de arquivos existente (FTP) e o protocolo de sistema de arquivos de rede (NFS) removendo a necessidade de um usuário efetuar logon antes de poder usar um recurso remoto. Definiu um novo protocolo, http, para uso na comunicação entre um servidor e um navegador Web. Um navegador Web só precisa enviar um pedido de informação ao servidor Web de uma máquina remota e as informações (texto, gráficos, links com outras informações) são devolvidas. Esse aumento na conveniência promoveu grande crescimento no uso de http e da Web em geral.

Os Capítulos 14 a 17 discutem redes e sistemas distribuídos, com ênfase especial em computação distribuída usando Java.

3.1.7 Sistema de proteção

Se um sistema de computação tiver vários usuários e permitir a execução concorrente de múltiplos processos, esses processos deverão ser protegidos das atividades uns dos outros. Para que isso aconteça, existem mecanismos que garantem que os arquivos, segmentos de memória, CPU c outros recursos possam ser operados apenas pelos processos que obtiveram autorização adequada do sistema operacional.

Por exemplo, o hardware de endereçamento de memória garante que um processo só pode executar dentro de seu próprio espaço de endereçamento. O timer garante que nenhum processo pode obter controle da CPU sem mais tarde ceder esse controle. Os registradores de controle de dispositivo não são acessíveis aos usuários, de modo que a integridade dos vários dispositivos periféricos é protegida.

A proteção *é* qualquer mecanismo para controlar o acesso de programas, processos ou usuários aos recursos definidos por um sistema de computação. Esse mecanismo deve fornecer meios para a especificação dos controles a serem impostos e os meios para seu cumprimento.

A proteção pode melhorar a confiabilidade detectando erros latentes nas interfaces entre os subsistemas. A detecção precoce dos erros dc interface muitas vezes pode evitar a contaminação de um subsistema saudável por outro subsistema com problemas de funcionamento. Um recurso desprotegido não pode se defender contra uso (ou mau uso) por um usuário não-autorizado ou incompetente. Um sistema orientado à proteção fornece um meio para distinguir entre uso autorizado e não-autorizado, como discutido no Capítulo 18.

N3.1.8 Sistema interpretador de comandos

' Um dos programas de sistema mais importantes para um sistema operacional é o interpretador de comandos, que é a interface entre o usuário e o sistema operacional. Alguns sistemas operacionais incluem o interpretador de comandos no kernel. Outros sistemas operacionais, tais como MS-DOS e UNIX, tratam o interpretador de comandos como um programa especial que fica executando quando um job *é* iniciado ou quando um usuário entra no sistema (em sistemas de tempo compartilhado). '

Muitos comandos são passados ao sistema operacional por instruções de controle. Quando um novo job é iniciado em um sistema em batch, ou quando um usuário efetua logon em um sistema de tempo compartilhado, um programa que lê e interpreta instruções de controle é executado automaticamente. Esse programa às vezes é chamado de interpretador de cartões dc controle, ou o interpretador da linha de comandos, e geralmente é chamado de shell. Sua função é simples: obter o comando seguinte e executá-lo. /

Os sistemas operacionais geralmente se diferenciam na área do shell, com um interpretador de comandos amigável ao usuário tornando o sistema mais agradável a alguns usuários. Um estilo de interface amigável ao usuário *è* o sistema de janelas e menus baseado em mouse usado no Macintosh e no Microsoft Windows. O mouse é movido para posicionar o ponteiro sobre imagens na tela, ou ícones, que representam programas, arquivos e funções do sistema. Dependendo da localização do ponteiro do mouse, clicar em um botão do mouse pode chamar um programa, selecionar um arquivo ou diretório - conhecido como pasta - ou abrir um menu que contém comandos. Shells mais poderosos, complexos e difíceis de aprender são apreciados por outros usuários. Em alguns desses shells, os comandos são digitados em um teclado e exibidos cm uma tela ou terminal de impressão, com a tecla Knter (ou Return) indicando que um comando está completo e pronto para ser executado. Os shells do MS-DOS e UNIX operam dessa forma.

As instruções de comando lidam com a criação e gerência de processos, tratamento de I/O. gerência de armazenamento secundário, gerência de memória principal, acesso a sistema de arquivos, proteção e redes.

3.2 • Serviços de sistemas operacionais

Um sistema operacional fornece um ambiente para a execução de programas. Fornece certos serviços a programas e aos usuários desses programas. Os serviços específicos fornecidos, é claro, diferem de um sistema operacional para outro, mas podemos identificar classes comuns. Esses serviços de sistemas operacionais são fornecidos para a conveniência do programador, a fim de tornar a tarefa de programação mais fácil.

- *Execução de programa:* O sistema deve ser capaz de carregar um programa na memória e executar esse programa. O programa deve ser capaz de encerrar a sua execução, quer de forma normal ou anormal (indicando erro).
- *Operações de l/O:* Um programa em execução pode precisar de l/O. Essa I/O pode envolver um arquivo ou dispositivo de l/O. Para dispositivos específicos, funções específicas podem ser desejadas (como rebobinar a unidade de fita, ou limpar uma tela de terminal. Para fins de eficiência e proteção, os usuários em geral não podem controlar os dispositivos de I/O diretamente. Portanto, o sistema operacional deve fornecer os meios para realizar as operações de entrada e saída.
- *Manipulação do sistema de arquivos:* O sistema de arquivos é particularmente interessante. Obviamente, os programas precisam ler e gravar arquivos. Também precisam criar e excluir arquivos por nome.

* *Comunicações:* Existem muitas circunstâncias nas quais um processo precisa trocar informações com outro processo. Existem duas maneiras principais nas quais tal comunicação pode ocorrer. A primeira ocorre entre processos que estão executando no mesmo computador; a segunda ocorre entre processos que estão executando em diferentes sistemas de computação ligados por uma rede/As comunicações podem ser implementadas via *memória compartilhada* ou pela técnica de troca de mensagens, na qual pacotes de informações são movidos entre processos pelo sistema operacional.

* *Detecção de erros: O* sistema operacional precisa estar constantemente ciente de possíveis erros. Os erros podem ocorrer no hardware da CPU e da memória (como um erro de memória ou falta de energia), em dispositivos de I/O {como um erro de paridade em fita, uma falha de conexão na rede ou falta de papel na impressora), e no programa de usuário (como um *overfiow*aritmético, uma tentativa de acessar uma posição ilegal na memória, ou uso excessivo do tempo da CPUj/tara cada tipo de erro, o sistema operacional deve tomar a medida adequada para garantir uma computação correta e consistente.

"* Além disso, existe um outro conjunto de funções de sistemas operacionais não para ajudar o usuário, mas para garantir a operação eficiente do sistema em si. Os sistemas com múltiplos usuários podem ganhar eficiência compartilhando os recursos do computador entre os usuários.

(• *Alocução de recursos:* Quando existem múltiplos usuários ou múltiplos jobs executando ao mesmo tempo, recursos devem ser alocados a cada um deles. Muitos tipos diferentes de recursos são gerenciados pelo sistema operacional/Alguns (como ciclos de CPU, memória principal e armazenamento de arquivos) podem ter código de alocação especial, enquanto outros (como dispositivos de I/O) podem ter código de pedido e liberação muito mais geral. Por exemplo, para determinar como usar melhora CPU, os sistemas operacionais possuem rotinas de escalonamento de CPU que levam em conta a velocidade da CPU, os jobs que devem ser executados, o número de registradores disponíveis, entre outros fatores. Também pode haver rotinas para alocar uma unidade de fita para uso por um job. Uma rotina desse tipo localiza uma unidade de fita não-utilizada e marca uma tabela interna para registrar o novo usuário da unidade. Outra rotina é usada para limpar essa tabela. Essas rotinas também podem alocar *plotters,* modems e outros dispositivos periféricos.

/• *Contabilização:* E preciso manter um registro dos usuários que utilizam os recursos do computador, em que quantidade e que tipos de recursos. Esse registro pode ser usado para contabilização (para que os usuários possam ser faturados) ou simplesmente para acumular estatísticas de uso/As estatísticas de uso podem ser uma ferramenta valiosa para os pesquisadores que desejam reconfigurar o sistema para melhorar os serviços de computação.

• *Proteçào:* Os proprietários das informações armazenadas em um sistema de computação multiusuário podem desejar controlar o uso dessas informações. Quando vários processos não-conexos independentes forem executados ao mesmo tempo, um processo não poderá interferir em outros ou no próprio sistema operacional/A proteção visa garantir que todo acesso aos recursos do sistema seja controlado. A *segurança* do sistema contra acesso por pessoas estranhas também é importante. Tal segurança começa com cada usuário tendo de se autenticar para o sistema, geralmente por meio de uma senha, para ter acesso aos recursos/Estende-se à defesa dos dispositivos de I/O externos, incluindo modems e placas de rede, de tentativas de acesso inválidas e ao registro de todas as conexões para detecção de invasões. Se determinado sistema precisar ser protegido e seguro, precauções devem ser tomadas em todo o sistema. A medida da força de uma corrente está no seu elo mais fraco.

3.3 • Chamadas ao sistema

As chamadas ao sistema *(system cfllls)* fornecem a interface entre um processo e o sistema operacional. Essas chamadas estão geralmente disponíveis como instruções em linguagem assembly c, cm geral, são listadas nos manuais usados por programadores cm linguagem assembly/

Certos sistemas permitem que as chamadas ao sistema sejam feitas diretamente de um programa de linguagem de nível mais alto e, nesse caso, as chamadas normalmente lembram chamadas de sub-rotinas ou de

funções predefinidas. Podem gerar uma chamada a uma rotina de execução especial que realiza a chamada ao sistema, ou a chamada ao sistema pode ser gerada dirctamente *in-lhie.*

Várias linguagens - como C, C+ + e Perl-foram definidas para substituir a linguagem assembly na programação de sistemas. Essas linguagens permitem que as chamadas ao sistema sejam feitas dirctamente. Por exemplo, as chamadas ao sistema do UNIX podem ser feitas diretamente a partir de um programa cm C ou C + + . As chamadas ao sistema para as plataformas Microsoft Windows modernas fazem parte da API Win32, que está disponível para uso por todos os compiladores escritos para o Microsoft Windows.

Java não permite que as chamadas ao sistema sejam feitas diretamente, porque uma chamada ao sistema é específica a um sistema operacional e resulta em código específico daquela plataforma. No entanto, se determinada aplicação exigir recursos específicos do sistema, um programa em Java pode chamar um método escrito em outra linguagem - geralmente C ou C++ - que, por sua vez> pode fazer a chamada ao sistema. Tais métodos são conhecidos como métodos "nativos".

/ Como um exemplo da forma em que as chamadas ao sistema são usadas, considere escrever um programa simples para ler dados de um arquivo e copiá-los para outro arquivo. A primeira entrada que o programa precisará são os nomes dos dois arquivos: o arquivo de entrada e o arquivo de saída. Esses nomes podem ser especificados de muitas maneiras, dependendo do projeto do sistema operacional. Uma abordagem é fazer o programa pedir ao usuário os nomes dos dois arquivos. Em um sistema interativo, essa abordagem exigirá uma sequência de chamadas ao sistema, primeiro para escrever uma mensagem solicitando os nomes na tela e depois para ler os caracteres que definem os dois arquivos. Em sistemas baseados em mouse e ícones, um menu de nomes de arquivos geralmente é exibido em uma janela. O usuário pode usar o mouse para sclecionar o nome de origem e uma janela pode ser aberta para que um nome de destino seja especificado./

Depois que os dois nomes de arquivos tiverem sido obtidos, o programa deve abrir o arquivo de entrada e criar o arquivo de saída. Cada uma dessas operações requer outra chamada ao sistema. Existem condições de erro possíveis para cada operação. Quando o programa tentar abrir o arquivo de entrada, poderá verificar que não existe um arquivo com aquele nome ou que o arquivo está protegido contra acesso. Nesses casos, o programa deverá imprimir uma mensagem na console (outra sequencia de chamadas ao sistema) e, em seguida, fazer o término anormal (outra chamada ao sistema). Se o arquivo de entrada existir, devemos criar um novo arquivo de saída. Talvez já exista outro arquivo de saída com este nome. Essa situação pode fazer o programa abortar (uma chamada ao sistema) ou podemos apagar o arquivo existente (outra chamada ao sistema) e criar um novo (outra chamada ao sistema). Outra opção, em um sistema interativo, é perguntar ao usuário (uma sequência de chamadas ao sistema para obter como saída a mensagem e ler a resposta do terminal) se o arquivo existente deve ser substituído ou se o programa deve ser abortado.

Agora que os dois arquivos estão prontos, entramos em um laço que lê dados do arquivo de entrada (uma chamada ao sistema) c os grava no arquivo de saída (outra chamada ao sistema). Cada operação de leitura e escrita deve retornar informações de status com relação às várias condições de erro possíveis. Na entrada, o programa poderá verificar que o final do arquivo foi alcançado, ou que aconteceu uma falha de hardware no processo de leitura (como um erro de paridade). A operação de escrita poderá encontrar vários erros, dependendo do dispositivo de saída (falta de espaço em disco, fim físico da fita, falta de papel na impressora etc).

Finalmente, depois que o arquivo todo tiver sido copiado, o programa poderá fechar os dois arquivos (outra chamada ao sistema), gravar uma mensagem na console (mais chamadas ao sistema) e finalmente terminar normalmente (a chamada final ao sistema). Como podemos ver, os programas fazem uso pesado do sistema operacional.

Entretanto, muitos usuários nunca chegam a ver esse nível de detalhe. O sistema de suporte à execução (o conjunto de funções incorporado em bibliotecas incluídas com um compilador) para a maioria das linguagens de programação fornece uma interface muito mais simples. Por exemplo, a instrução cout em C+ + provavelmente é compilada em uma chamada para uma rotina de suporte de execução que emite as chamadas ao sistema necessárias, verifica se há erros e, finalmente, volta ao programa de usuário. Assim, a maior parte dos detalhes da interface do sistema operacional é oculta do programador pelo compilador e pelo pacote de suporte à execução.

As chamadas ao sistema ocorrem de diferentes maneiras, dependendo do computador que está sendo usado. Geralmente, mais informações são necessárias além de simplesmente identificar a chamada ao sistema desejada. O tipo e a quantidade exata de informações variam de acordo com o sistema operacional e a chamada em questão. Por exemplo, para obter entrada, precisamos especificar o arquivo ou dispositivo a ser usado como origem, e o endereço e o tamanho do buffer de memória no qual a entrada deve ser lida. E claro, o dispositivo ou arquivo c o tamanho podem estar implícitos na chamada.

Três métodos gerais são usados para passar parâmetros para o sistema operacional. A abordagem mais simples é passar os parâmetros em *registradores.* Em alguns casos, no entanto, pode haver mais parâmetros do que registradores. Nesses casos, os parâmetros em geral são armazenados em um *bloco* ou tabela na memória, e 0 endereço do bloco é passado como um parâmetro em um registrador (Figura 3.1). Os parâmetros também podem ser colocados, ou *inseridos,* na *pilha* pelo programa e *lidos e retirados* da pilha pelo sistema operacional. Alguns sistemas operacionais preferem os métodos de bloco ou pilha, porque essas abordagens não limitam o número ou tamanho dos parâmetros sendo passados.

As chamadas ao sistema podem ser agrupadas basicamente em cinco categorias principais: controle de processos, manipulação de arquivos, manipulação de dispositivos, manutenção de informações e comunicações. Nas Seções 3.3.1 a 3.3.5, discutimos rapidamente os tipos de chamadas ao sistema que podem ser fornecidas por um sistema operacional. A maioria dessas chamadas ao sistema suportam, ou são suportadas por, conceitos e funções discutidos em outros capítulos deste livro. A Figura 3.2 resume os tipos de chamadas ao sistema normalmente fornecidas por um sistema operacional.

Figura 3.1 Passagem de parâmetros como uma tabela.

/3.3.1 Controle de processo

Um programa em execução precisa ser capaz de parar sua execução de forma normal (end) ou anormal (abort). Se uma chamada ao sistema for feita para terminar de forma anormal o programa cm execução no momento, ou se o programa tiver um problema e causar um erro *{hap)t* às vezes ocorre um dump de memória c uma mensagem de erro é gerada. O dump é gravado em disco e pode ser examinado por um *depurador* para determinar a causa do problema. Em circunstâncias normais ou anormais, o sistema operacional deve transferir o controle para o interpretador de comandos. O interpretador de comandos então lê o próximo comando. Em um sistema interativo, o interpretador de comandos simplesmente continua com o comando seguinte; parte-se do pressuposto de que o usuário emitirá um comando apropriado para reagir a qualquer erro/Em um sistema em batch, o interpretador de comandos geralmente encerra todo o job e continua com o job seguinte. Alguns sistemas permitem que cartões de controle indiquem ações especiais de recuperação em casos de erro. Sc um programa descobrir um erro na sua entrada e desejar fazer um término anormal, também poderá definir um nível de erro. Erros mais graves podem ser indicados por um parâmetro de erro de nível mais alto. E possível então combinar o término normal c anormal definindo um término normal como erro de nível 0 . 0 interpretador de comandos ou um programa posterior pode usar esse nível de erro para determinar a próxima ação automaticamente.

- Controle de processos
  - end, abort
  - load, execute
  - creatc process, terminate process
  - get process attributes, set process attributcs
  - wait for time
  - wait event {esperar evento), signal event (sinalizar evento)
  - allocate and free memory
- Gerência de arquivos
  - create file, delete file
  - open, dose
  - rcad (ler), write (escrever), reposition (reposicionar)
  - get file attributcs, set file attributes
- Gerência de dispositivos
  - request device, release device
  - read, write, reposition
  - get device attributcs, set device attributcs
  - logically attach or detach devices
- Manutenção de informações
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes
- Comunicações
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

Figura 3.2 Tipos de chamadas ao sistema.

Um processo ou job que está executando um programa pode querer carregar outro programa, por meio de chamadas 1 oad e execute. Esse recurso permite ao interpretador de comandos executar um programa conforme orientação, por exemplo, de um comando de usuário, de um clique no mouse ou de um comando batch. Uma questão interessante é a quem devolver o controle quando o programa carregado terminar sua execução. Essa questão está relacionada com o problema de o programa existente ser perdido, salvo ou ter permissão para continuar a execução de forma concorrente com o novo programa,

Se o controle voltar ao programa existente quando o novo programa terminar, devemos salvar a imagem na memória do programa existente; assim, criamos de fato um mecanismo para um programa chamar outro programa. Se ambos continuarem concorrentemente, teremos criado um novo job ou processo para ser nuiltiprogramado. Muitas vezes, existe uma chamada ao sistema especificamente para este objetivo (create process ou submit job).

Se criarmos um novo job ou processo, ou mesmo uma série de jobs ou processos, devemos ser capazes de controlar sua execução. Esse controle exige a capacidade de determinar e redefinir os atributos de um job ou processo, incluindo a prioridade do job, seu tempo de execução máximo permitido e assim por diante (get

process attrl butes e set process attributes). Também é possível terminar um job ou processo criado (terminate process) se verificarmos que ele está incorreto ou não é mais necessário.

Tendo criado novos jobs ou processos, talvez seja necessário esperar que eles concluam sua execução. Talvez seja preciso esperar determinado período de tempo (wai t time); mais provavelmente, podemos ter de esperar determinado evento ocorrer (wai t event). Os jobs e processos devem então indicar quando esse evento ocorreu (signa) event). As chamadas ao sistema desse tipo, que lidam com a coordenação de processos concorrentes, são discutidas em mais detalhes no Capítulo 7.

Outro conjunto de chamadas ao sistema é útil na depuração de um programa. Muitos sistemas fornecem chamadas ao sistema para fazer o dump de memória. Isso é útil para a depuração. Um trace de programa lista cada instrução conforme ela é executada; é fornecido por poucos sistemas. Até os microprocessadores fornecem um modo de CPU chamado *passo a passo,* no qual uma exceção é executada pela CPU após cada instrução. A exceção geralmente c capturada por um depurador, que é um programa de sistema projetado para auxiliar o programador a encontrar e corrigir bugs.

Muitos sistemas operacionais fornecem um perfil de tempo de um programa. Ele indica a quantidade de tempo que o programa executa em determinada posição ou grupo de posições. Um perfil de tempo requer um recurso de rastreio *(trace)* ou interrupções regulares do timer. A cada ocorrência de uma interrupção do timer, o valor do contador do programa é registrado. Com um número suficientemente frequente de interrupções do timer, um quadro estatístico do tempo gasto nas várias partes do programa pode ser obtido.

Existem tantos aspectos c variações no controle de processos c jobs que será preciso usar exemplos para esclarecer esses conceitos. O sistema operacional MS-DOS é um exemplo de um sistema monotarefa, que possui um interpretador de comandos chamado quando o computador é iniciado (Figura 3.3(a)). Como o MS-DOS é monotarefa, ele utiliza um método simples para executar um programa e não cria um novo processo. Ele carrega o programa na memória, gravando por cima de si mesmo para permitir ao programa a maior quantidade possível de memória (Figura 3.3 (b)). Em seguida, o sistema define o ponteiro de instruções para a primeira instrução do programa. O programa executa até um erro causar uma exceção, ou o programa executar uma chamada ao sistema para terminar. Em ambos os casos, um código de erro *è* salvo na memória para uso posterior. Depois disso, a pequena parte do interpretador de comandos que não foi sobreposta retoma a execução. Sua primeira tarefa é recarregar o resto do interpretador de comandos do disco. Quando esta tarefa tiver sido realizada, o interpretador de comandos torna o código de erro anterior disponível ao usuário ou ao próximo programa.

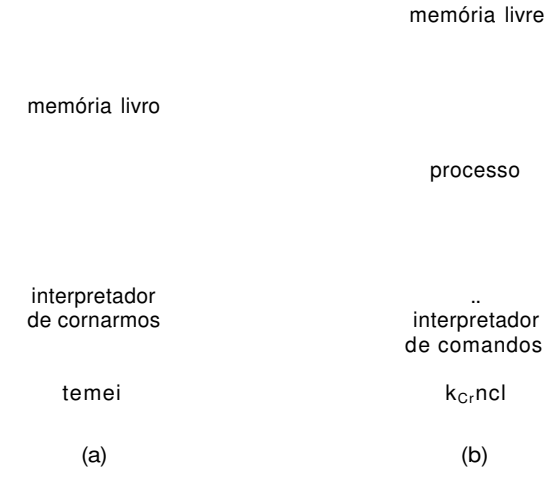


Figura 3.3 Execução do MS-DOS. (a) Na inicialização do sistema.
(b) Executando um programa.

Embora o sistema operacional MS-DOS não tenha capacidades gerais de multiiarcfa, ele fornece um método para execução concorrente limitada. Um programa TSR é um programa que "intercepta uma interrupção" e, em seguida, termina com a chamada ao sistema terminate and stay resident. Por exemplo, ele pode interceptar a interrupção do clock colocando o endereço de uma de suas sub-rotinas na lista de rotinas de in-

tcrrupçâo a serem chamadas quando o timer do sistema for acionado. Dessa forma, a rotina TSR será executada várias vezes por segundo, a cada pulso do clock. A chamada ao sistema terminate and stay resident faz com que o MS-DOS reserve o espaço ocupado pelo TSR, para que não seja sobreposto quando o interpretador de comandos for recarregado.

O UNIX de Bcrkeley é um exemplo de sistema multitarefa. Quando um usuário efetua logon no sistema, o *shell* (interpretador de comandos) escolhido pelo usuário é executado. Esse shell é semelhante ao shell do MS-DOS, no sentido de aceitar comandos e executar programas solicitados pelo usuário. No entanto, como o **UNIX** *é* um sistema multitarefa, o interpretador de comandos pode continuar a executar enquanto outro programa é executado (Figura 3.4). Para iniciar um novo processo, o shell executa uma chamada ao sistema fork. Em seguida, o programa selccionado é carregado na memória via uma chamada ao sistema exec, e o programa é então executado. Dependendo da forma em que o comando foi emitido, o shell espera que o processo seja finalizado ou executa o processo "em segundo plano". Neste caso, o Shell solicita imediatamente um novo comando. Quando um processo estiver executando em segundo plano, não poderá receber entrada diretamente do teclado, porque o shell está usando esse recurso. As operações de entrada e saída são, portanto, realizadas por meio de arquivos ou usando um mouse e uma interface de janelas. Enquanto isso, o usuário está livre para pedir ao shell que execute outros programas, monitore o andamento do processo de execução, mude a prioridade daquele programa etc. Quando o processo tiver acabado, ele executa uma chamada ao sistema exi t para encerrar, passando ao processo que fez a chamada um código de status zero ou um código de erro diferente de zero. Esse status ou código de erro fica então disponível para o shell ou outros programas. Os processos são discutidos no Capítulo 4.

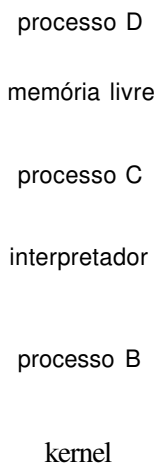


Figura 3.4 UNIX executando vários programas.

3.3.2 Gerência de arquivos

O sistema de arquivos será discutido em maiores detalhes no Capítulo 11. No entanto, é possível identificar várias chamadas ao sistema comuns que lidam com arquivos.

/Em primeiro lugar/é preciso criar e excluir arquivos, através das chamadas create e dei ete. Cada chamada ao sistema requer o nome do arquivo e talvez alguns atributos do arquivo. Uma vez criado o arquivo, é preciso abri-lo (open) e usá-lo. Podemos também realizar as operações de leitura, escrita ou reposicionar, respectivamente read, write ou reposition (voltar ou saltar até o fim do arquivo, por exemplo). Finalmente, precisamos fechar o arquivo, por meio de close, indicando que ele não está mais sendo usado. /

Talvez seja necessário realizar o mesmo conjunto de operações para os diretórios, se houver uma estrutura de diretórios para organizar os arquivos no sistema de arquivos. Além disso, para arquivos ou diretórios, é preciso ser capaz de determinar os valores dos vários atributos e talvez redefini-los, se necessário. Os atributos de arquivo incluem o nome do arquivo, o tipo do arquivo, os códigos de proreção, as informações de contabilização etc. Pelo menos duas chamadas ao sistema, get file attributeeset file attribute, são necessárias para esta função. Alguns sistemas operacionais fornecem um número muito maior de chamadas.

3.3.3 Gerência de dispositivos

Um programa, a medida que está sendo executado, talvez precise de recursos adicionais para prosseguir. Os recursos adicionais podem ser mais memória, unidades de fita, acesso a arquivos e assim por diante. Se os recursos estiverem disponíveis, poderão ser concedidos e o controle poderá ser devolvido ao programa de usuário; caso contrário, o programa terá de esperar até que recursos suficientes estejam disponíveis.

Os arquivos podem ser considerados dispositivos abstratos ou virtuais. Assim, muitas das chamadas ao sistema por arquivos também são necessárias para dispositivos. Se houver vários usuários do sistema, no entanto, devemos primeiro realizar uma operação de request, para obter o dispositivo e garantir seu uso exclusivo. Depois de terminar com o dispositivo, devemos realizar a operação de rei ease, para liberá-lo. Essas funções são semelhantes às chamadas open e close para arquivos./

Assim que o dispositivo tiver sido solicitado (e alocado), será possível realizar as operações de leitura, escrita c (possivelmente) reposicionamento do dispositivo, por meio de read, write e reposition, assim como ocorre com arquivos comuns. Na verdade, a similaridade entre dispositivos de l/O e arquivos é tão grande que muitos sistemas operacionais, incluindo o UNIX e o MS-DOS, combinam os dois cm uma estrutura de arquivo-dispositivo. Nesse caso, os dispositivos de I/O são identificados por nomes de arquivo especiais.

3.3.4 Manutenção de informações

/Muitas chamadas ao sistema existem simplesmente a fim de transferir informações entre o programa de usuário e o sistema operacional. Por exemplo, a maioria dos sistemas possuí uma chamada ao sistema para obter a data e hora atuais, por meio de time c date. Outras; chamadas ao sistema podem retornar informações sobre o sistema, tal como o número de usuários atuais, o número da versão do sistema operacional, a quantidade de memória ou espaço em disco livre e assim por diante./

Além disso, o sistema operacional mantém informações sobre todos os seus processos e existem chamadas ao sistema para acessar essas informações. Em geral, também existem chamadas para redefinir informações de processo (get process attnbutes e set process attributes). Na Seção 4.1.3, discutimos que informações são normalmente mantidas.

3 .3.5 Comunicação

Existem dois modelos comuns de comunicação. No modelo de troca de mensagens, as informações são trocadas através de um recurso de comunicação entre processos fornecido pelo sistema operacional. Antes da comunicação ocorrer, uma conexão deve ser estabelecida. O nome do outro comunicador deve ser conhecido, quer seja outro processo na mesma CPU ou um processo em outro computador conectado por uma rede de comunicação. Cada computador em uma rede tem um *nome de host,* como um número IP, pelo qual é conhecido. Da mesma forma, cada processo tem um *nome de processo,* que é traduzido em um identificador equivalente com o qual o sistema operacional pode fazer referência a elejfts chamadas ao sistema get hosti d e get processid efetuam essa tradução. Esses identificadores são então passados às chamadas de uso geral open e close fornecidas pelo sistema de arquivos ou às chamadas específicas open connection e close connection, dependendo do modelo de comunicação do sistema. O processo destinatário geralmente deve dar sua permissão para que a comunicação ocorra com uma chamada accept connection. A maioria dos processos que estarão recebendo conexões são *daemons* de uso especial, que são programas de sistema fornecidos para esse fim. Eles executam uma chamada watt for connection e são despertados quando uma chamada é feita. A origem da comunicação, chamada de *cliente,* e o daemon receptor, chamado de *servidor,* trocam mensagens por meio das chamadas read message e wri te message. A chamada cl ose connect i on encerra a comunicação.

No modelo de memória compartilhada, os processos utilizam as chamadas ao sistema map memory para obter acesso às regiões da memória de propriedade de outros processos. Lembre-se de que, normalmente, o sistema operacional tenta evitar que um processo acesse a memória de outro processo. A memória compartilhada requer que dois ou mais processos concordem em remover esta restrição. Eles podem então trocar informações lendo e gravando dados nessas áreas compartilhadas. A forma dos dados e sua localização são determinadas por esses processos e não estão sob controle do sistema operacional. Os processos também são responsáveis por garantir que não estão escrevendo na mesma posição simultaneamente. Tais me-

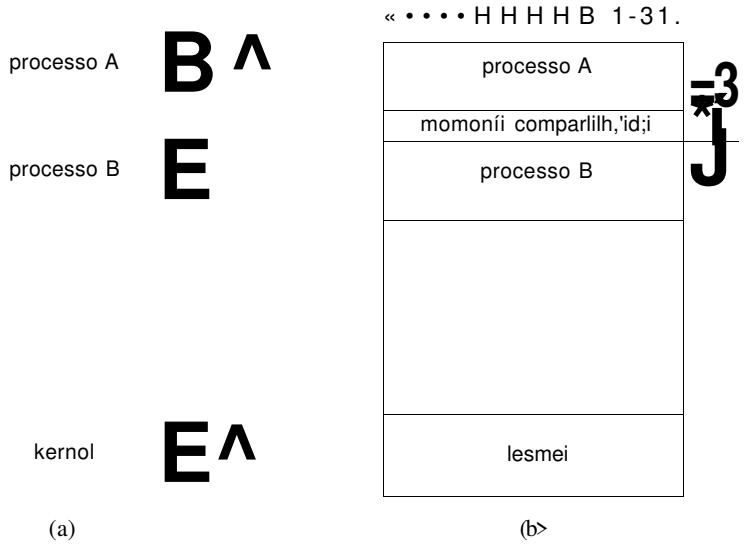


Figura 3.5 Modelos de comunicação, (a) Troca de mensagens, (b) Memória compartilhada.

canismos estão descritos no Capítulo 7. lambem vamos analisar uma variação do modelo de processo - threads - que compartilha memória por default. Os threads serão tratados no Capítulo 5.

Esses dois métodos são comuns nos sistemas operacionais e alguns sistemas até implementam ambos. A troca de mensagens é útil quando um número menor de dados precisa ser trocado, porque não há necessidade de evitar conflitos. Também é mais fácil implementar do que a memória compartilhada para a comunicação entre computadores. A memória compartilhada permite velocidade máxima c conveniência de comunicação, pois pode ser realizada a velocidades de memória em um computador. Existem, no entanto, problemas nas áreas de proteção e sincronização. Os dois modelos de comunicação são comparados na Figura 3.5. No Capítulo 4, vamos analisar a implementação Java de cada modelo.

3.4 • Programas de sistema

/Outro aspecto de um sistema moderno é a colcção de programas de sistema^Lembre-se da Figura 1.1, que representava a hierarquia lógica de um computador. No nível mais baixo está o hardware. Em seguida, está o sistema operacional, os programas de sistema e, finalmente, os programas aplicativos./!)s programas de sistema fornecem um ambiente conveniente para o desenvolvimento e a execução de programasJÁlguns deles são simplesmente interfaces de usuário às chamadas ao sistema; outros são consideravelmente mais complexos. Podem ser divididos nestas categorias: /

- *Gerência de arquivos:* Esses programas criam, excluem, copiam, renomeiam, imprimem, fazem dump, listam e geralmente manipulam arquivos e diretórios.
- *Informações de status:* Alguns programas simplesmente pedem ao sistema informações relativas à data, hora, quantidade de memória ou espaço em disco disponível, número de usuários ou informações de status semelhantes. Essas informações são, então, formatadas e impressas no terminal ou outro dispositivo ou arquivo de saída.
- *Modificação de arquivo:* Vários editores de texto podem estar disponíveis para criar c modificar o conteúdo dos arquivos armazenados em disco ou fita.
- *Suporte à linguagem de programação:* Compiladores, montadores e interpretadores para linguagens de programação comuns (tais como C, C+ +, Java, Visual Basic e PERL) são geralmente fornecidos ao usuário com o sistema operacional. A maioria desses programas agora são pagos c fornecidos à parte.
- *(Carregamento e execução de programas:* Depois que um programa é montado ou compilado, deve ser carregado na memória para ser executado. O sistema pode oferecer utilitários de carga absolutos, utilitários de carga relocável, linkeditores e utilitários de carga cm *overlay.* Sistemas de depuração para linguagens de nível mais alto ou linguagem de máquina também são necessários.

- *Comunicações:* Esses programas oferecem o mecanismo para criar conexões virtuais entre processos, usuários e diferentes sistemas de computação. Permitem aos usuários enviar mensagens às telas uns dos outros, navegar pelas páginas da Web, enviar mensagens de correio eletrônico, efetuar logon remotamente ou transferir arquivos de uma máquina para outra.

A maior parte dos sistemas operacionais possui programas úteis na resolução de problemas comuns ou na realização de operações comuns. Tais programas incluem navegadores da Web, processadores e formatadores de texto, planilhas eieirônicas, sistemas de bancos de dados, geradores de compiladores, pacotes de plotagem e análise estatística, e jogos. Esses programas são chamados utilitários do sistema ou programas aplicativos.

Talvez o programa de sistema mais importante para um sistema operacional seja o interpretador de comandos, sendo sua principal função obter e executar o próximo comando especificado pelo usuário.

Muitos dos comandos emitidos neste nível manipulam arquivos nas operações de criação, exclusão, listagem, impressão, cópia, execução, entre outras. Existem duas maneiras gerais nas quais esses comandos podem ser implementados. Em uma abordagem, *o* interpretador de comandos propriamente dito contém o código para executar o comando. Por exemplo, um comando para excluir um arquivo pode fazer com que o interpretador de comandos salte para uma seção do código que configura parâmetros e faz a chamada ao sistema adequada. Nesse caso, o número de comandos que podem ser emitidos determina o tamanho do interpretador de comandos, já que cada comando requer seu próprio código de implementação.

Uma abordagem alternativa - usada pelo UNIX, entre outros sistemas operacionais - implementa a maioria dos comandos por programas de sistema. Nesse caso, o interpretador de comandos não entende o comando de forma alguma; ele simplesmente usa o comando para identificar um arquivo a ser carregado na memória e executado. Assim, o comando UNIX para excluir um arquivo

rmG

procuraria um arquivo chamado *rtti,* carregaria o arquivo na memória c o executaria com o parâmetro *(i.* A função associada com o comando rm seria completamente definida pelo código no arquivo *mi.* Dessa forma, os programadores podem adicionar facilmente novos comandos ao sistema criando novos arquivos com nomes adequados. O programa interpretador de comandos, que pode ser pequeno, não precisa ser alterado para que os novos comandos sejam acrescentados.

Existem problemas com essa abordagem ao projeto de um interpretador de comandos. Observe primeiro que, como o código para executar um comando é um programa de sistema separado, o sistema operacional deverá fornecer um mecanismo para passar parâmetros do interpretador de comandos para o programa de sistema. Essa tarefa muitas vezes pode ser confusa, já que o interpretador de comandos e o programa de sistema talvez não estejam na memória ao mesmo tempo, e a lista de parâmetros pode ser longa. Além disso, é mais lento carregar um programa e executá-lo do que simplesmente saltar para outra seção do código dentro do programa ativo.

Outro problema é que a interpretação dos parâmetros cabe ao programador do programa de sistema. Assim, os parâmetros podem ser fornecidos de forma inconsistente entre programas parecendo semelhantes ao usuário, mas escritos em momentos diferentes por programadores diferentes.

O sistema operacional visto por muitos usuários é definido pelos programas de sistema, em vez das chamadas ao sistema em si. Vamos considerar os PCs. Quando o seu computador está executando o sistema operacional Microsoft Windows, um usuário pode ver um shell do MS-DOS de linha de comandos ou a interface gráfica de mouse e janelas. Ambos usam o mesmo conjunto de chamadas ao sistema, mas as chamadas têm aspecto diferente c atuam de formas diferentes. Consequentemente, a visão desse usuário pode ser substancialmente diferente da estrutura real do sistema. O projeto de uma interface de usuário amigável e útil, portanto, não é uma função direta do sistema operacional. Neste livro, vamos nos concentrar nos problemas fundamentais envolvidos no fornecimento de serviço adequado a programas de usuário. Do ponto de vista do sistema operacional, não existe distinção entre programas de usuário e programas de sistema.

3.5/" Estrutura do sistema

Um sistema tão grande e complexo quanto um sistema operacional moderno deve ser cuidadosamente construído para que funcione bem e possa ser facilmente modificado. Uma abordagem comum é dividir a tarefa

cm pequenos componentes cm vez de ter um sistema monolítico/Cada um desses módulos deve ser uma porção bem delineada do sistema, com entradas, saídas e funções cuidadosamente definidas. Já discutimos rapidamente os componentes comuns dos sistemas operacionais (Seção 3.1). Nesta seção, vamos discutir como esses componentes são interconectados e combinados no kernel.

3.5.1 Estrutura simples

Existem vários sistemas comerciais que não possuem uma estrutura bem definida. Frequentemente, tais sistemas operacionais começaram como sistemas pequenos, simples e limitados, crescendo alem do seu escopo original. O MS-DOS é um exemplo de um sistema assim. Foi originalmente projetado e implementado por algumas pessoas que não tinham ideia de que ele se tornaria tão popular. Foi escrito para fornecer funcionalidade máxima no menor espaço possível (por causa do hardware limitado no qual era executado), por isso ele não foi dividido cuidadosamente em módulos. A Figura 3.6 mostra sua estrutura.

No MS-DOS, as interfaces e níveis de funcionalidade não são bem separados. Por exemplo, os programas aplicativos podem acessar as rotinas básicas de l/O para escrever dirctamente na tela e nas unidades de disco. Tal liberdade deixa o MS-DOS vulnerável a programas errantes (ou maliciosos), causando falhas no sistema inteiro quando os programas de usuário falham. È claro, o MS-DOS também foi limitado pelo hardware da época. Como o Intel 8088 para o qual o sistema foi escrito não fornece modo dual nem proteção de hardware, os projetistas do MS-DOS não tiveram opção a não ser deixar o hardware de base acessível.

Outro exemplo de estruturação limitada é o sistema operacional UNIX original. O UNIX é outro sistema que inicialmente era limitado pela funcionalidade de hardware. Consiste em duas partes separáveis: o kernel e os programas de sistema. O kernel é separado ainda em uma série de interfaces e drivers de dispositivo, que foram adicionados e expandidos ao longo dos anos, à medida que o UNIX evoluiu. Podemos considerar o sistema operacional UNIX tradicional como sendo em camadas, como mostrado na Figura 3.7. Tudo o que está abaixo da interface de chamadas ao sistema e acima do hardware físico é o kernel. O kernel fornece sistema de arquivos, escalonamento de CPU, gerência de memória e outras funções de sistema operacional através de chamadas ao sistema. Somando tudo, significa muita funcionalidade a ser combinada em um nível. Os programas de sistema usam as chamadas de sistema suportadas pelo kernel para fornecer funções úteis, tais como compilação e manipulação de arquivos.

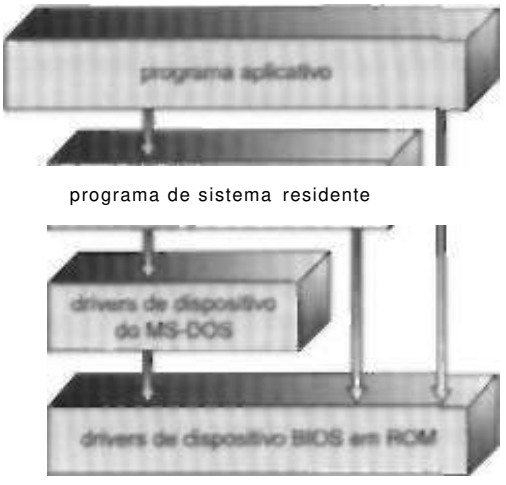


Figura 3.6 Estrutura cm camadas do MS-DOS.

As chamadas ao sistema definem a interface de programação de aplicações (Application Programing Interface- API) para o UNIX; o conjunto de programas de sistema normalmente disponíveis define a interface de usuário. As interfaces de programador e de usuário definem o contexto que o kernel deve suportar.

Novas versões do UNIX são projetadas para usar hardware mais avançado. Com suporte de hardware adequado, os sistemas operacionais podem ser divididos em partes menores e mais apropriadas do que aquelas permitidas pelos sistemas MS-DOS ou UNIX originais. O sistema operacional pode reter muito mais controle do computador e dos aplicativos que utilizam o computador. Os implementadores têm maior liberdade

para fazer alterações nas funções internas do sistema c na criação de sistemas operacionais modulares. Nessa abordagem *top-down,* a funcionalidade e os recursos gerais são determinados e separados em componentes. Ocultar informações também é importante, porque deixa os programadores livres para implementar rotinas de baixo nível conforme a necessidade, desde que a interface externa da rotina fique inalterada e que a rotina em si execute a tarefa como especificado.

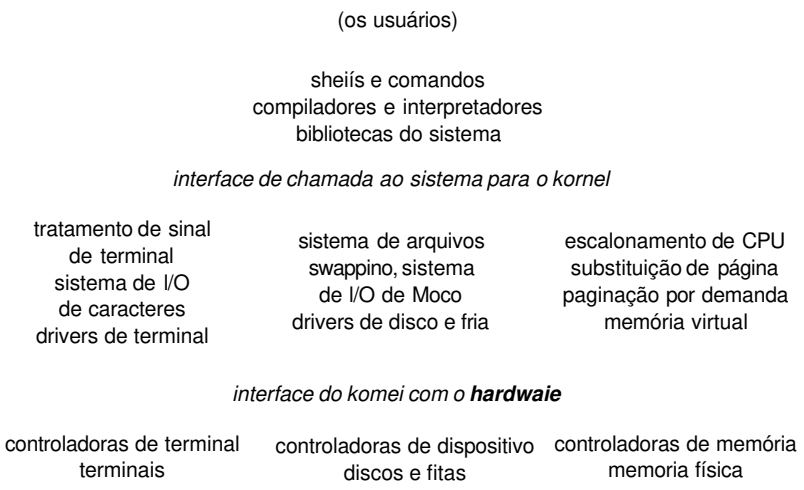


Figura 3.7 Estrutura do sistema UNIX.

3.5.2 Abordagem em camadas

A modularização de um sistema pode ser feita de muitas formas; um método é a abordagem em camadas, na qual o sistema operacional é dividido cm uma série de camadas (níveis), cada qual construída sobre camadas inferiores. A camada inferior (camada 0) é o hardware; a camada superior (camada N) é a interface de usuário.

Uma camada de sistema operacional e" uma implementação de um objeto abstraio que é o encapsulamento de dados e das operações que podem manipular esses dados. Uma camada típica de um sistema operacional - digamos a camada *M-é* representada na Figura 3.8. Ela consiste em estruturas de dados e um conjunto de rotinas que podem ser chamadas por camadas de nível mais alto. A camada M, por sua vez, pode chamar operações cm camadas de níveis mais baixos.

A principal vantagem da abordagem em camadas é a modularidade. As camadas são selecionadas de forma que cada uma utilize as funções (operações) e serviços apenas das camadas de nível mais baixo. Essa abordagem simplifica a depuração e a verificação do sistema. A primeira camada pode ser depurada sem preocupação com o resto do sistema, porque, por definição, ela só utiliza 0 hardware básico (que é considerado correio) para implementar suas funções. Depois que a primeira camada c depurada, pode-se presumir seu funcionamento correto enquanto a segunda camada é depurada e assim por diante. Se for encontrado um erro durante a depuração de determinada camada, o erro deve estar nessa camada, porque as camadas inferiores já foram depuradas. Assim, o projeto e a implementação do sistema são simplificados quando o sistema é dividido em camadas.

Cada camada é implementada apenas com as operações fornecidas pelas camadas de nível inferior. Uma camada não precisa saber como essas operações são implementadas; só precisa saber o que essas operações fazem. Portanto, cada camada oculta a existência de determinadas estruturas de dados, operações e hardware de camadas de nível mais alto.

A principal dificuldade da abordagem cm camadas está na definição adequada das várias camadas. Como uma camada só pode usar as camadas que estão em um nível inferior, é preciso haver um planejamento cuidadoso. Por exemplo, o driver de dispositivo para o armazenamento auxiliar (espaço em disco usado por algoritmos de memória virtual) deve estar em um nível abaixo das rotinas de gerência de memória, porque a gerência de memória requer a capacidade de usar o armazenamento auxiliar.

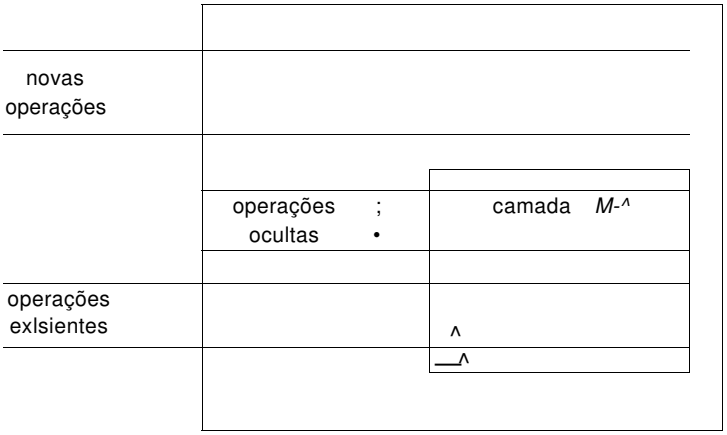


Figura 3.8 Camada de um sistema operacional.

Outros requisitos podem não ser tão óbvios. O driver do armazenamento auxiliar normalmente estaria acima do escalonador de CPU, porque o driver pode precisar esperar por l/O e a CPU pode ser reescalonada durante esse período. No entanto, em um sistema grande, o escalonador de CPU pode ter mais informações sobre todos os processos ativos que a memória comporta. Portanto, essas informações talvez precisem ser transferidas da memória para o disco e vice-versa, exigindo que a rotina de driver do armazenamento auxiliar esteja abaixo do escalonador de CPU.

Um último problema com implementações em camadas é que elas tendem a ser menos eficientes que outros tipos. Por exemplo, quando um programa de usuário executa uma operação de l/O, ele executa uma chamada ao sistema que é desviada para a camada de l/O, que chama a camada de gerência de memória, que, por sua vez, chama a camada de escalonamento de CPU, que é então passada para o hardware. Em cada camada, os parâmetros podem ser modificados, os dados podem precisar ser transferidos e assim por diante. Cada camada acrescenta custo à chamada ao sistema; o resultado final é uma chamada ao sistema que demora mais que em um sistema sem camadas.

Essas limitações causaram alguma reação contra a estrutura em camadas nos últimos anos. Menos camadas com mais funcionalidades estão sendo projetadas, fornecendo a maior parte das vantagens do código modularizado e evitando os difíceis problemas da definição e interação em camadas. Por exemplo, o OS/2 é descendente do MS-DOS que acrescenta operações multitarefas e em modo dual, assim como outros novos recursos. Devido a essa maior complexidade e ao hardware mais poderoso para o qual o OS/2 foi projetado, o sistema foi implementado de uma forma com mais camadas. Compare a estrutura do MS-DOS com aquela apresentada na Figura 3.9; do ponto de vista do projetn e implementação do sistema, o OS/2 tem vantagens. Por exemplo, não é permitido acesso direto do usuário a recursos de baixo nível, fornecendo ao sistema operacional maior controle sobre o hardware e maior conhecimento dos recursos que cada programa está utilizando.

Como outro exemplo, considere a história do Windows NT. A primeira versão tinha uma organização fortemente orientada a camadas. No entanto, essa versão tinha um desempenho menor quando comparada com o Windows 95. O Windows NT 4.0 parcialmente resolveu o problema de desempenho movendo camadas do espaço de usuário para o espaço do kernel e promovendo uma maior integração entre elas.

3.5.3 Microkemels

A medida que o UNIX se expandiu, o kernel tornou-se grande e difícil de gerenciar. Em meados dos anos 80, pesquisadores da Carnegie Mellon University desenvolveram um sistema operacional chamado Mach que modularizou o kernel, usando a abordagem de microkemels. Este método estrutura o sistema operacional removendo todos os componentes não-essenciais do kernel c implementando-os como programas de sistema c de nível de usuário. O resultado é um kernel menor. Existe pouco consenso com relação a quais serviços devem permanecer no kernel e quais devem ser implementados no espaço de usuário. Em geral, no entanto, os microkemels geralmente fornecem gerência mínima de memória e processos, além de um recurso de comunicação.

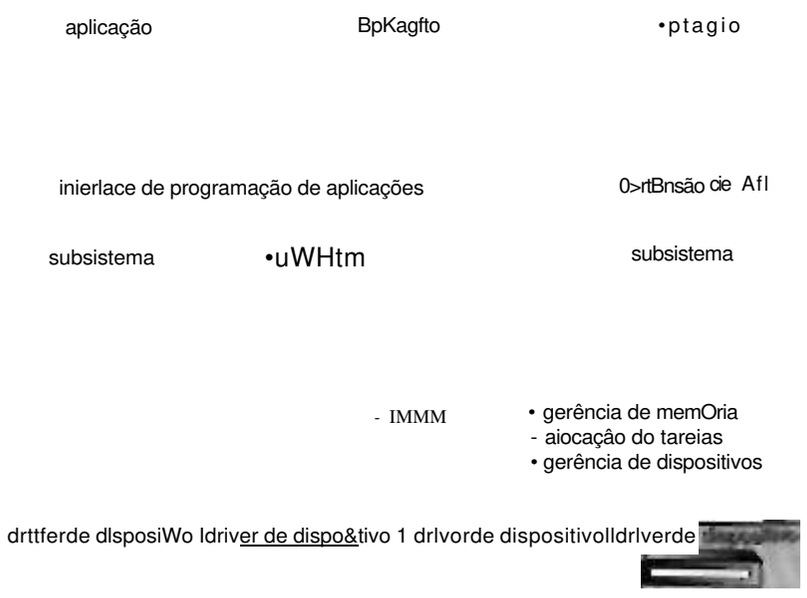


Figura 3.9 Estrutura em camadas do OS/2.

A principal função do microkernel é fornecer um recurso de comunicação entre o programa cliente e os vários serviços que também estão em execução no espaço de usuário. A comunicação é fornecida por meio de troca de mensagens, como descrito na Seção 3.3.5. Por exemplo, se o programa cliente desejar acessar um arquivo, ele deverá interagir com o servidor de arquivos. O programa cliente e o serviço nunca vão interagir diretamente. Km vez disso, eles se comunicam indiretamente trocando mensagens com o microkernel.

Os benefícios da abordagem do microkernel incluem a facilidade de expandir o sistema operacional. Todos os novos serviços são adicionados ao espaço de usuário c, consequentemente, não exigem modificação do kernel. Quando o kernel precisa ser modificado, as alterações tendem a ser menores, porque o microkernel í um kernel menor. K mais fácil transporrar o sistema operacional resultante de um projeto de hardware para outro. O microkernel também fornece mais segurança c confiabilidade, pois a maior parte dos serviços estão sendo executados como processos de usuário, em vez de kernel. Sc um serviço falhar, o resto do sistema operacional permanece inalterado.

Vários sistemas operacionais contemporâneos usaram a abordagem de microkernels. O UNIX Digital fornece uma interface UNIX com o usuário, mas é implementado com um kernel Mach. O Mach mapcia as chamadas ao sistema UNIX em mensagens para os serviços apropriados de nível de usuário. O sistema operacional Apple MacOS X Server bascia-se no kernel Mach. O Windows NT utiliza uma estrutura híbrida. Já mencionamos que parte da arquitetura do Windows NT utiliza camadas. O Windows NT é projetado para executar várias aplicações, incluindo Win32 (aplicações nativas do Windows), OS/2 e POSIX. Fornece um servidor que executa no espaço de usuário para cada tipo de aplicação. Os programas cliente para cada tipo de aplicação também executam no espaço de usuário. O kernel coordena a troca de mensagens entre as aplicações cliente e servidores de aplicação. A estrutura cliente-servidor do Windows NT está representada na Figura 3.10.

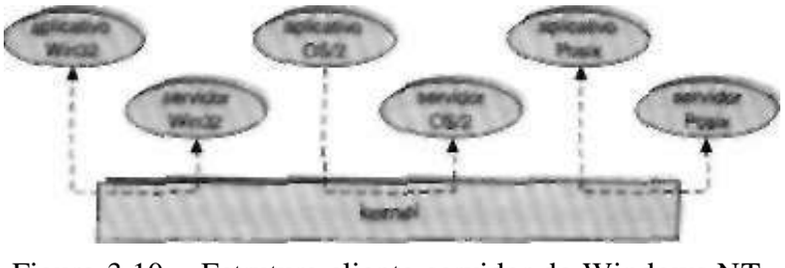


Figura 3.10 Estrutura cliente-servidor do Windows NT.

3.6 • Máquinas virtuais

Conceituai mentc,/um sistema de computação é formado por camadas/O hardware c" o nível mais baixo em todos esses sistemas, p kemel executando no próximo nível utiliza instruções de hardware para criar um conjunto de chamadas ao sistema para uso por camadas externas. Os programas de sistema acima do kernel, portanto, são capazes de usar as chamadas ao sistema ou as instruções de hardware e, em certos aspectos, esses programas não fazem a distinção entre ambos/Assim, embora sejam acessados de forma diferente, ambos fornecem funcionalidade que *o* programa pode usar para criar funções até mais avançadas. Os programas de sistema, por sua vez, tratam o hardware e as chamadas ao sistema como se ambos estivessem no mesmo nível.

/Alguns sistemas levam esse esquema um passo adiante, permitindo que os programas de sistema sejam chamados facilmente pelos programas aplicativos. Como antes, embora os programas de sistema estejam em um nível mais alto que os de outras rotinas, os programas aplicativos podem visualizar tudo abaixo deles na hierarquia como se os últimos fossem parte da máquina propriamente dita. Essa abordagem em camadas é levada a sua conclusão lógica no conceito de uma máquina virtuaj/O sistema operacional VM para os sistemas IBM *é* o melhor exemplo do conceito de máquina virtual, porque a IBM *é* pioneira nessa área.

Usando as técnicas de escalonamento de CPU (Capítulo 6) *c* de memória virtual (Capítulo 10), um sistema operacional pode criar a ilusão de que um processo tem seu próprio processador com sua própria memória (virtual). Ê claro que, normalmente, o processo tem recursos adicionais, tais como chamadas ao sistema e um sistema de arquivos, que não são fornecidos unicamente pelo hardware. A abordagem de máquina virtual, por outro lado, não fornece funcionalidade adicional, em vez disso, fornece uma interface que é idêntica ao hardware subjacente. Cada processo recebe uma cópia (virtual) do computador subjacente (Figura 3.11).

O computador físico compartilha recursos para criar máquinas virtuais. O escalonamento de CPU pode compartilhar;»CPU para criar a ilusão de que os usuários têm seus próprios processadores. O spooling e um sistema de arquivos podem fornecer leitoras de cartões virtuais e impressoras de linhas virtuais. Um terminal de usuário normal de tempo compartilhado fornece a função de console do operador da máquina virtual.

Uma grande dificuldade da abordagem de máquina virtual envolve os sistemas de disco. Vamos supor que a máquina física tenha três unidades de disco, mas deseja suportar sete máquinas virtuais. Evidentemente, não é possível alocar uma unidade de disco para cada máquina virtual. Lembre-se de que o software de máquina virtual por si só precisará de muito espaço em disco para fornecer memória virtual e spooling. A solução é fornecer discos virtuais, que são idênticos em todos os aspectos, exceto tamanho - chamados minidiscos no sistema operacional VM da IBM. O sistema implementa cada minidisco alocando tantas trilhas nos discos físicos quantas o minidisco precisar. Obviamente a soma dos tamanhos de todos os minidiscos deve ser menor do que o espaço em disco físico disponível.

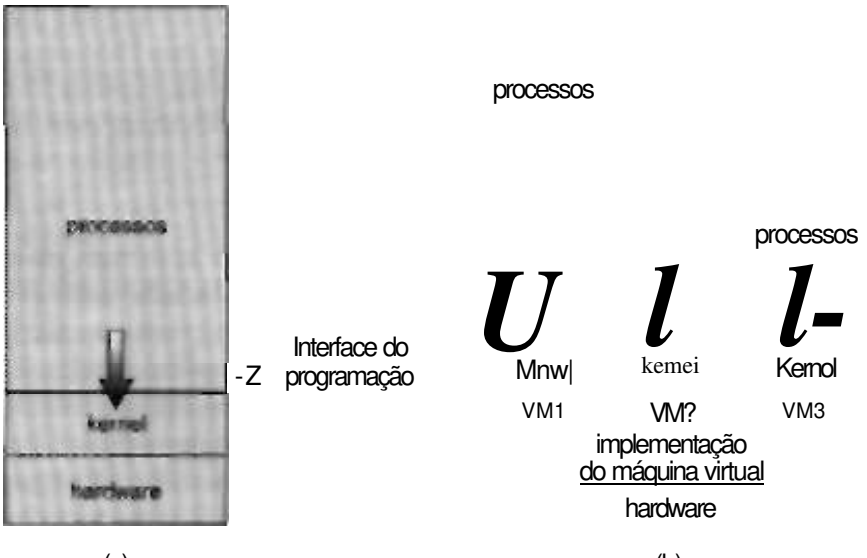


Figura 3.11 Modelos de sistema, (a) Máquina não-virtual. (b) Máquina virtual

Portanto, os usuários recebem as próprias máquinas virtuais. Podem executar qualquer um dos sistemas operacionais ou pacotes de software disponíveis na máquina subjacente. Para o sistema IBM VM, um usuário normalmente executa CMS - um sistema operacional monousuário interativo. O software de máquina virtual preocupa-se com a multiprogramação de várias máquinas virtuais cm uma máquina física, mas não precisa levar em consideração software de suporte a usuário. Dessa forma, o problema de projetar um sistema multiusuário interativo pode ser particionado de forma útil em partes menores.

3.6.1 Implementação

Embora o conceito de máquina virtual seja útil, cie é difícil de implementar. Muito trabalho é necessário para fornecer uma duplicata exata da máquina subjacente. Lembre-se de que a máquina subjacente tem dois modos: o modo usuário e o modo monitor. O software da máquina virtual pode executar no modo monitor, pois é o sistema operacional. A máquina virtual propriamente dita só pode executar no modo usuário. Assim como a máquina física tem dois modos, no entanto, a máquina virtual também precisa tê-los. Consequentemente, é preciso ter um modo usuário virtual e um modo monitor virtual, amhos executando em um modo usuário físico. As ações que causam a transferência do modo usuário para o modo monitor em uma máquina real (como uma chamada ao sistema ou uma tentativa de executar uma instrução privilegiada) também devem causar uma transferencia do modo usuário virtual para o modo monitor virtual em uma máquina virtual.

Essa transferência em geral pode ser feita facilmente. Quando uma chamada ao sistema, por exemplo, é feita por um programa que executa cm uma máquina virtual, em modo usuário virtual, ela causará a transferência para o monitor de máquina virtual na máquina real. Quando o monitor de máquina virtual obtiver o controle, ele poderá mudar o conteúdo de registradores e do contador do programa para que a máquina virtual simule o efeito da chamada ao sistema. Em seguida, poderá reiniciar a máquina virtual, observando que agora está cm modo monitor virtual. Se a máquina virtual tentar, por exemplo, ler a partir da leitora de cartões virtual, ela executará uma instrução de l/O privilegiada. Como a máquina virtual está executando em modo usuário físico, essa instrução vai eferuar uma exceção ao monitor de máquina virtual. O monitor de máquina virtual deverá então simular o efeito da instrução de I/O. Em primeiro lugar, ele encontra o arquivo em spool que implementa a leitora de cartões virtual. Em seguida, traduz a leitura do cartão virtual em uma leitura no arquivo de disco em spool e transfere a próxima "imagem de cartão" virtual para a memória virtual da máquina virtual. Finalmente, cie pode reiniciar a máquina virtual. O estado da máquina virtual foi modificado exatamente como se a instrução de l/O tivesse sido executada com uma leitora de cartões real para uma máquina real executando em modo monitor normal.

A principal diferença, é claro, é o tempo. Enquanto a l/O real pode levar 100 milissegundos, a I/O virtual pode levar menos tempo (porque está em spool) ou mais tempo (porque é interpretada). Além disso, a CPU está sendo multiprogramada entre várias máquinas virtuais, diminuindo ainda mais a velocidade das máquinas virtuais, de forma imprevisível. Nos casos extremos, pode ser necessário simular todas as instruções para fornecer uma verdadeira máquina virtual. O VM funciona para as máquinas IBM porque as instruções normais para as máquinas virtuais podem executar diretamente no hardware. Apenas instruções privilegiadas (necessárias basicamente para I/O) devem ser simuladas e, portanto, executam mais lentamente.

3.6.2 Benefícios

O conceito de máquina virtual tem várias vantagens. Observe que, neste ambiente, existe proteçâo total dos vários recursos do sistema. Cada máquina virtual é completamente isolada de todas as outras máquinas virtuais, por isso não existem problemas de segurança. Por outro lado, não existe compartilhamento direto dos recursos. Duas abordagens para fornecer compartilhamento foram implementadas. Em primeiro lugar, é possível compartilhar um minidisco. Esse esquema é modelado com base em um disco físico compartilhado, mas é implementado por software. Com essa técnica, os arquivos podem ser compartilhados. Em segundo lugar, é possível definir uma rede de máquinas virtuais, cada qual podendo enviar informações cm uma rede de comunicação virtual. Novamente, a rede é modelada com base em redes de comunicação físicas, mas é implementada por software.

Tal sistema de máquina virtual é um veículo perfeito para a pesquisa e desenvolvimento em sistemas operacionais. Normalmente, alterar um sistema operacional éuma tarefa difícil. Como os sistemas operacionais

são programas grandes e complexos, é difícil ter certeza de que uma alteração em uma parte não causará bugs obscuros em outra parte. Essa situação pode ser particularmente perigosa devido ao poder do sistema operacional. Como o sistema operacional executa em modo monitor, uma alteração errada em um ponteiro poderia causar um erro que poderia destruir todo o sistema de arquivos. Assim, é necessário testar todas as alterações do sistema operacional cuidadosamente.

O sistema operacional, no entanto, executa e controla toda a máquina. Portanto, o sistema atual deve ser interrompido c retirado de uso enquanto as mudanças estão sendo feitas e testadas. Esse período é normalmente chamado de tempo de desenvolvimento do sistema. Como ele torna o sistema indisponível aos usuários, o tempo de desenvolvimento do sistema é geralmente programado para tarde da noite ou nos fins de semana, quando a carga do sistema é baixa.

Um sistema de máquina virtual pode eliminar boa parte do problema. Os programadores de sistema recebem sua própria máquina virtual, e o desenvolvimento do sistema é feito na máquina virtual, em vez de na máquina física. A operação normal do sistema raramente precisa ser perturbada para desenvolvimento do sistema.

Apesar dessas vantagens, até recentemente poucas melhorias na técnica foram feitas. As máquinas virtuais estão voltando à moda como meio de resolver problemas de compatibilidade de sistema. Por exemplo, existem milhares de programas disponíveis para MS-DOS em sistemas baseados em CPU Intel. Eabricantes de computadores, como a Sun Microsystems e a Digital Equipmcnt Corporation (DEC) utilizam outros processadores mais rápidos, mas gostariam que seus clientes pudessem executar esses programas MS-DOS. A solução é criar uma máquina virtual Intel sobre o processador nativo. Um programa MS-DOS é executado neste ambiente, e suas instruções Intel são traduzidas no conjunto de instruções nativo. O MS-DOS também é executado nesta máquina virtual, por isso o programa pode fazer suas chamadas ao sistema como sempre. O resultado final é um programa que aparenta estar executando em um sistema baseado em Intel, mas está realmente executando em um processador diferente. Se o processador for suficientemente rápido, o programa MS-DOS executará rapidamente, embora toda instrução esteja sendo traduzida em várias instruções nativas para execução. Da mesma forma, o Apple Macintosh baseado em PowerPC inclui uma máquina virtual Motorola 68000 para permitir a execução de códigos binários que foram escritos para as máquinas Macintosh baseados cm 6S000 mais antigas. Infelizmente, quanto mais complexa a máquina sendo emulada, mais difícil é construir uma máquina virtual precisa e mais lenta é a execução dessa máquina virtual. Um dos principais recursos da plataforma Java é que ela executa em uma máquina virtual. A seção a seguir discute Java e a máquina virtual Java.

3.7 • Java

Java é uma tecnologia introduzida pela Sun Microsystems no final de 1995. Estamos nos referindo a ela como tecnologia em vez de uma linguagem de programação, porque ela fornece mais do que uma linguagem de programação convencional. A tecnologia Java consiste em três componentes essenciais:

1. Especificação da linguagem de programação
2. Interface de programação de aplicações (Al'l)
3. Especificação de máquina virtual

Vamos apresentar uma visão geral desses três componentes.

3.7.1 Linguagem de programação

A linguagem de programação Java é mais bem caracterizada como uma linguagem de programação orientada a objetos, independentes de arquítetura, distribuída e com múltiplos threads. Os objeros Java são especificados com a estrutura class; um programa Java consiste em uma ou mais classes. Para cada classe Java, o compilador Java produz um arquivo de saída *bytecode* independente de arquitetura (.class) que executará em qualquer implementação da máquina virtual Java (JVM). A comunidade de programação para Internet deu preferência inicial Java, por causa do seu suporte a applets, que são programas com acesso limitado a recursos que executam em um navegador Web. Java também fornece suporte de alto nível para redes e objetos distribuídos. Também é uma linguagem de múltiplos threads, o que significa que um programa Java pode ter vários threads, ou fluxos, de controle distintos. Os objetos distribuídos que utilizam invocação remota de

métodos (Rcmotc Mcthod Invocation - RMI) de Java são discutidos no Capítulo 15, c os programas Java com múltiplos rhreads são abordados no Capítulo S.

Java é considerada uma linguagem segura. Esse recurso *é* especialmente importante considerando que um programa Java pode estar sendo executado em um ambiente distribuído. Analisamos a segurança Java no Capítulo 19. Alem disso, Java gerência a memória de forma automática, realizando a coleta de lixo: a prática de recuperar memória de objetos que já não estão mais em uso e retorná-la ao sistema.

3.7.2 API

A APl para Java consiste em uma API base e unia API de extensão padrão. A API base fornece suporte básico da linguagem a gráficos, l/O, utilitários e redes. Os exemplos incluem: Java.lang, java.awt, java.io e java.net. A API de extensão padrão inclui suporte para empresa, comércio, segurança e mídia. À medida que a linguagem evolui, muitos pacotes que antes eram parte da API de extensão padrão estão se tornando parte da API base.

3.7.3 **Máquina virtual**

A JVM é uma especificação para um computador abstraio e consiste em um carregador de classes (class loader) e em um interpretador Java que executa os bytecodes independentes de arquitetura. O carregador de classes carrega arquivos .class do programa Java c da API Java para execução pelo interpretador Java. Esse, por sua vez, pode ser um interpretador de software que interpreta um bytecode de cada vez, ou pode ser um compilador jusl'in-rime (JIT) que transforma os bytecodes independentes de arquitetura em linguagem de máquina nativa para o computador host. Em outros casos, o interpretador pode ser implementado em um chip de hardware que executa bytecodes Java de forma nativa. A JVM é apresentada na Figura 3.12.

A plataforma Java consiste em JVM eem API Java; está representada na Figura 3.13. A plataforma Java pode ser implementada sobre um sistema operacional host, como UNIX ou Windows; como parte de um navegador Web ou em hardware.

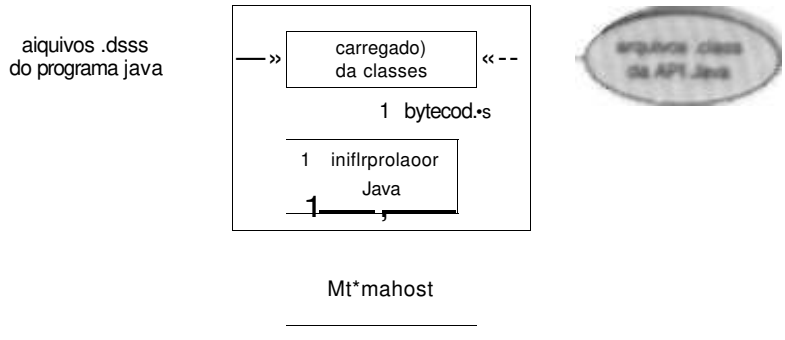


Figura 3.12 A máquina virtual Java.

Uma instância da JVM é criada sempre que uma aplicação ou apptct Java é executada. Essa instância da JVM inicia a execução quando o mc"rodo main( ) de um programa é chamado. Isso também se aplica a applcts, embora o programador não defina um main( ). Nesse caso, o navegador executa o método main( ) antes de criar o applet. Se executarmos simultaneamente dois programas Java e um applet Java no mesmo computador, teremos três instâncias da JVM.

E a plataforma Java que torna possível desenvolver programas independentes de arquitetura e portáveis. A implementação da plataforma é específica ao sistema e abstrai o sistema de modo padrão ao programa Java, fornecendo uma interface limpa e independente de arquitetura. Essa interface permite que um arquivo -class execute cm qualquer sistema que tenha implementado a JVM e a API; a Figura 3.14 representa isso. A implementação da plataforma Java consiste em desenvolver a JVM e a API Java para um sistema específico (como Windows ou UNIX), de acordo com a especificação da JVM.

Usamos a JVM em todo o texto para ilustrar os conceitos de sistema operacional. Fazemos referencia à especificação da JVM, em vez de a qualquer implementação específica.

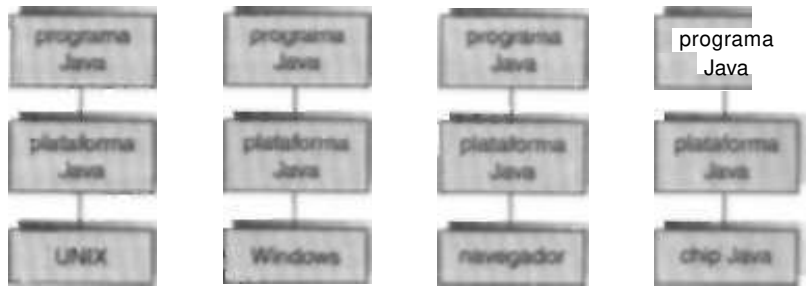


Figura 3.13 A plataforma Java.

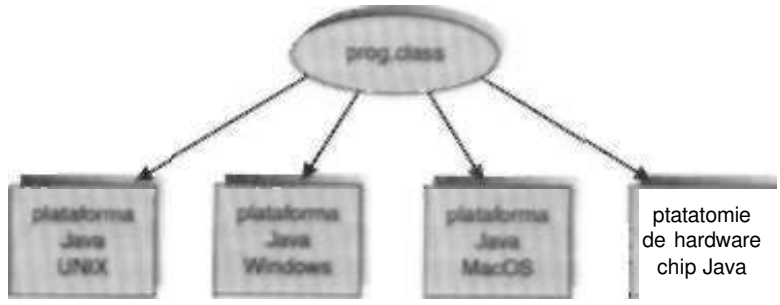


Figura 3.14 Arquivo Java .class cm múltiplas plataformas.

3.7.4 Ambiente de desenvolvimento Java

O ambiente de desenvolvimento Java consiste em um ambiente de compilação e em um ambiente de execução. O ambiente de compilação transforma um arquivo fonte Java em um bytecode (arquivo . cl ass). O arquivo fonte Java pode ser um programa ou applet Java. O ambiente de execução é a plataforma Java para o sistema host. O ambiente de desenvolvimento está representado na Figura 3.15.

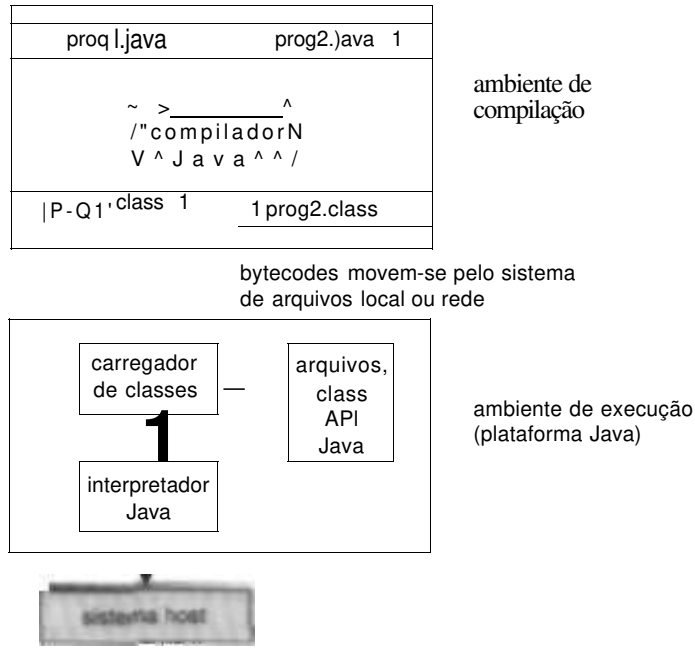


Figura 3-15 Ambiente de desenvolvimento Java.

3.8 • Projeto e implementação de sistemas

Nesta seção, vamos discutir os problemas enfrentados durante o projeto c implementação de um sistema. Evidentemente, não existem soluções completas para os problemas de projeto, mas algumas abordagens têm sido bem-sucedidas.

3.8.1 Objetivos do projeto

O primeiro problema no projeto de um sistema é definir os seus objetivos e especificações. No nível mais alto, 0 projeto do sistema será afetado pela escolha de hardware e o tipo de sistema: batch, tempo compartilhado, monousuário, multiusuário, distribuído, tempo real ou de uso geral.

Além desse nível de projeto mais alto, os requisitos podem ser muito mais difíceis de especificar. Os requisitos podem ser divididos em dois grupos básicos: objetivos de *usuário* e objetivos do *sistema.*

Os usuários desejam certas propriedades óbvias em um sistema: ele deve ser conveniente e fácil de usar, fácil de aprender, confiável, seguro e rápido. E claro que essas especificações não são particularmente úteis no projeto do sistema, pois não há concordância geral quanto a como alcançar esses objetivos.

Um conjunto semelhante de requisitos pode ser definido pelas pessoas que devem projetar, criar, manter e operar o sistema: o sistema operacional deve ser fácil de projetar, implementar e manter; deve ser flexível, confiável, livre de erros e eficiente. Mais uma vez, os requisitos são vagos e nào têm uma solução geral.

Não existe uma solução única para o problema de definir os requisitos de um sistema operacional. A grande gama de sistemas mostra que diferentes requisitos podem resultar em uma grande variedade de soluções para ambientes diferentes. Por exemplo, os requisitos para MS-DOS, um sistema monousuário para microcomputadores, devem ter sido substancialmente diferentes dos requisitos para o MVS, o grande sistema operacional multiusuário com acesso múltiplo para mainframes IBM.

A especificação e o projeto de um sistema operacional são tarefas altamente criativas. Embora nenhum livro didático ensine como fazê-lo, existem princípios gerais, desenvolvidos pela área de engenharia de software, que se aplicam especificamente a sistemas operacionais.

3.8.2 Mecanismos e políticas

Um princípio importante é a separação entre política c mecanismo. Os mecanismos determinam *como* faier alguma coisa-, as políticas determinam o *que* será feito. Por exemplo, o timer (consulte a Seção 2.5) é um mecanismo para garantir a proteção da CPU, mas decidir durante quanto tempo o timer deverá ser configurado para determinado usuário é uma decisão que envolve políticas.

A separação entre política e mecanismo é importante para fins de flexibilidade. As políticas tendem a mudar em diferentes locais ou com o tempo. Na pior das hipóteses, cada mudança na política exigiria uma mudança no mecanismo subjacente. Um mecanismo geral seria mais desejável. Com isso a mudança na política exigiria a redefinição de apenas certos parâmetros do sistema. Por exemplo, se, em um sistema de computação, uma política determinar que programas com uso intensivo de I/O devem ter prioridade sobre aqueles com uso intensivo de CPU, a política oposta poderia ser facilmente instituída em algum outro sistema de computação se o mecanismo fosse adequadamente separado e independente da política.

Os sistemas operacionais baseados em microkernel levam a separação entre mecanismo e política ao extremo, implementando um conjunto básico de blocos de construção primitivos. Esses blocos são praticamente independentes de política, permitindo o acréscimo de mecanismos c políticas mais avançados através de módulos do kernel criados pelo usuário ou através de programas de usuário. No outro extremo está um sistema como o sistema operacional Apple Macintosh, no qual mecanismos e políticas são codificados no sistema para dar a ele aparência e comportamento globais. Todas as aplicações têm interfaces semelhantes, porque a própria interface é incorporada ao kernel.

As decisões baseadas em políticas são importantes para todos os problemas de alocação de recursos e de escalonamento. Sempre que for necessário decidir se determinado recurso deve ou não ser alocado, uma decisão baseada em política deve ser tomada. Sempre que a questão envolver *como* em vez de o *quê,* um mecanismo deve ser determinado.

3.8.3 Implementação

Depois que o sistema operacional é projetado, ele deve ser implementado. Tradicionalmente, os sistemas operacionais têm sido escritos em linguagem assembiy. Agora, no entanto, geralmente são escritos em linguagens de nível mais alto. Por exemplo, uma implementação da JVM foi escrita cm Java na Sun Microsystems.

O primeiro sistema que não foi escrito em linguagem assembly provavelmente foi o Master Control Program (MCP) para computadores Burroughs. MCP foi escrito em uma variante de ALGOL. O MULTICS, desenvolvido no MIT, foi escrito basicamente cm PL/1. O sistema operacional Primos para computadores Prime foi escrito cm um dialcto de Fortran. Os sistemas operacionais UNIX, OS/2 e Windows NT são basicamente escritos em linguagem C. Apenas 900 linhas do código do UNIX original foram escritas em linguagem assembly, boa parte sendo do escalonador e dos drivers de dispositivo.

As vantagens de usar uma linguagem de alto nível, ou pelo menos uma linguagem de implementação de sistemas, para implementar sistemas operacionais são as mesmas obtidas quando a linguagem é usada para programas aplicativos: o código pode ser escrito de forma mais rápida, é mais compacto c mais fácil de entender e depurar. Alem disso, melhorias na tecnologia de compiladores aperfeiçoarão o código gerado para todo o sfstema operacional por simples recompilação. Finalmente, é muito mais fácil *portar* o sistema operacional -movê-lo para algum outro hardware-se ele estiver escrito em uma linguagem de alto nível. Por exemplo, o MS-DOS foi escrito na linguagem assembly Intel 8088. Consequentemente, está disponível apenas na família Intel de CPUs. O sistema operacional UNIX, por outro lado, que foi escrito basicamente em C, está disponível em várias CPUs diferentes, incluindo Intel 80X86, Motorola 680X0, SPARC c MIPS RX000.

As principais desvantagens da implementação de um sistema operacional em uma linguagem de nível mais alto são velocidade reduzida e maiores requisitos de armazenamento. Embora um programador especialista em linguagem assembly possa gerar pequenas rotinas eficientes, para programas grandes os compiladores modernos podem efetuar análises complexas e aplicar otimizações sofisticadas que geram código excelente. Os processadores modernos têm múltiplas unidades funcionais e pipelining massivo, podendo lidar com dependências complexas que podem superar a limitada capacidade humana de controlar detalhes.

Como acontece em outros sistemas, importantes melhorias de desempenho nos sistemas operacionais tendem a ser resultado de melhores estruturas de dados e algoritmos do que de excelente código cm linguagem assembly. Além disso, embora os sistemas operacionais sejam grandes, apenas uma pequena quantidade de código é crítica ao alto desempenho; o gerenciador de memória e o escalonador de CPU são provavelmente as rotinas mais críticas. Depois que o sistema tiver sido escrito e estiver funcionando corretamente, rotinas que representem gargalos podem ser identificadas c substituídas por equivalentes em linguagem assembly.

Para identificar gargalos, é preciso monitorar o desempenho do sistema. Código deve ser acrescentado para calcular e exibir medidas do comportamento do sistema. Em vários sistemas, o sistema operacional realiza essa tarefa produzindo listagens do comportamento do sistema. Todos os eventos interessantes são registrados com a respectiva hora e parâmetros importantes, sendo gravados cm um arquivo. Mais tarde, um programa de análise pode processar o arquivo de log para determinar o desempenho do sistema e identificar gargalos c ineficiências. Esses mesmos registros de log podem ser fornecidos como entrada para uma simulação de um sistema melhorado sugerido. Os registros de log também podem ajudar as pessoas a encontrar erros no comportamento de um sistema operacional.

Uma alternativa é calcular e exibir medidas de desempenho em tempo real Por exemplo, um timer pode acionar uma rotina para armazenar o valor corrente do ponteiro de instruções. O resultado é um quadro estatístico das posições mais frequentemente usadas no programa. Essa abordagem pode permitir que os operadores do sistema fiquem familiarizados com o comportamento do sistema e modifiquem a operação do sistema cm tempo real.

3.9 • Geração do sistema

E possível projetar, codificar e implementar um sistema operacional especificamente para uma máquina em determinado local. Mais comumente, no entanto, os sistemas operacionais são projetados para executar em qualquer máquina de uma determinada classe, em uma variedade de locais de instalação, com uma série de configurações periféricas. O sistema deve, então, ser configurado ou gerado para cada local de instalação específico, um processo às vezes denominado geração do sistema (SYSGEN).

O sistema operacional geralmente é distribuído em disco ou fita. Para gerar um sistema, usamos um programa especial. O programa SYSGEN lê um arquivo específico ou solicita que o operador do sistema forneça in-

formações relativas à configuração específica do sistema de hardware, ou examina diretamente o hardware para determinar quais os componentes presentes. Os seguintes tipos de informações devem ser determinados.

- Que CPU deve ser usada? Que opções (conjuntos de instruções estendidos, aritmética de ponto flutuante etc.) estão instaladas? Para sistemas com várias CPUs, cada CPU deve ser descrita.
- Quanta memória está disponível? Alguns sistemas determinarão esse valor por conta própria fazendo referencia a posições na memória de forma sucessiva ate" que uma falha do tipo "endereço ilegal" seja gerada. Esse procedimento define o endereço válido final c, portanto, a quantidade de memória disponível.
- Que dispositivos estão disponíveis? O sistema precisará saber como endereçar cada dispositivo (o número do dispositivo), o número de interrupção do dispositivo, o tipo e modelo do dispositivo e qualquer característica especial de dispositivo.
- Que opções de sistema operacional são desejadas, ou que valores de parâmetros deverão ser usados? Essas opções ou valores podem incluir quantos buffers de tal tamanho devem ser usados, que tipo de algoritmo de escalonamento de CPU é desejado, qual o número máximo de processo a ser suportado etc

Uma vez determinadas essas informações, elas podem ser usadas de várias fornias. Em um extremo, um administrador de sistema pode usá-las para modificar uma cópia do código-fonte do sistema operacional. O sistema, então, é compilado por completo. Declarações de dados, inicializações e constantes, juntamente com a compilação condicional, produzem uma versão objeto de saída do sistema operacional que é adaptada ao sistema descrito.

Em um nível ligeiramente menos personalizado, a descrição do sistema pode causar a criação de tabelas e a Seleção de módulos de uma biblioteca pré -com pilada. Esses módulos são ligados para formar o sistema operacional gerado. A Seleção permite que a biblioteca contenha os drivers de dispositivo para todos os dispositivos de l/O suportados, mas apenas aqueles necessários são incorporados ao sistema operacional. Como o sistema não c recompilado, a geração do sistema é mais rápida, mas o sistema resultante pode ser demasiadamente geral.

No outro extremo, é possível construir um sistema que seja totalmente baseado em tabelas. O código todo faz parte do sistema, e a Seleção ocorre no momento da execução, em vez de no momento de compilação ou ligação. A geração do sistema envolve simplesmente a criação das tabelas adequadas para descrever o sistema.

As principais diferenças entre essas abordagens são o tamanho e a generalidade do sistema gerado e a facilidade de modificação à medida que a configuração de hardware muda. Considere o custo de modificar o sistema para dar suporte a um terminal gráfico recém-adquirido ou outra unidade de disco. Comparado com esse custo está, *é* claro, a frequência (ou pouca frequência) em que essas mudanças são realizadas.

Depois que um sistema operacional *é* gerado, ele deve estar disponível para uso pelo hardware. Como o hardware sabe onde está o kernel ou como carregar esse kernet? O procedimento para iniciar um computador carregando o kernel *ê* chamado de *iniciaUzação* do sistema. Na maioria dos sistemas de computação, existe uma pequena parte do código, armazenado na ROM, conhecida como *rotina de partida* ou *programa de partida-* Esse código é capaz de localizar o kernel, carregá-lo na memória principal e iniciar sua execução. Alguns sistemas de computação, taiscomo PCs, utilizam um processo em duas etapas no qual uma única rotina de partida busca um programa de inicialização mais complexo do disco, que, por sua vez, carrega o kernel. A Seção 13.3.2 e o Capítulo 20 discutem a inicialização do sistema.

3.10 • Resumo

Os sistemas operacionais fornecem muitos serviços. No nível mais baixo, as chamadas ao sistema permitem que um programa em execução faça pedidos do sistema operacional diretamente. Em um nível mais alto, o interpretador de comandos ou shell fornece um mecanismo para o usuário efetuar uni pedido sem escrever um programa. Os comandos podem vir dos arquivos durante a execução cm modo batch, ou diretamente de um terminal quando em modo interativo ou de tempo compartilhado. Programas de sistema são fornecidos para satisfazer muitos pedidos comuns de usuários.

Os tipos de pedido variam de acordo com o nível do pedido. O nível da chamada ao sistema deve fornecer as funções básicas, tais como controle de processo e manipulação de arquivos e dispositivos. Pedidos de nível mais alto, satisfeitos pelo interpretador de comandos ou programas de sistema, são traduzidos em uma sequência de chamadas ao sistema. Os serviços de sistema podem ser classificados em várias categorias: controle de programa, pedidos de status e pedidos de I/O. Os erros de programa podem ser considerados pedidos implícitos de serviço.

Depois que os serviços do sistema estiverem definidos, a estrutura do sistema operacional pode ser desenvolvida. Várias tabelas são necessárias para registrar as informações que definem o estado do sistema de computação e o status dos jobs do sistema.

O projeto de um novo sistema operacional é uma tarefa importante. E essencial que os objetivos do sistema sejam bem definidos antes que o projeto comece. O tipo de sistema desejado é a base para a escolha dentre vários algoritmos e estratégias que serão necessários.

Como um sistema operacional é grande, a modularidade é importante. Projetar um sistema como uma sequência de camadas ou usando um microkernel é considerado uma boa técnica. O conceito de máquina virtual utiliza a abordagem em camadas e trata o kernel do sistema operacional e o hardware como se fossem apenas hardware. Mesmo outros sistemas operacionais podem ser carregados sobre essa máquina virtual.

Qualquer sistema operacional que tenha implementado a JVM pode executar todos os programas Java, já que a JVM abstrai o sistema subjacente para o programa Java, fornecendo uma interface independente de arquitetura.

Km todo o ciclo de projeto do sistema operacional, é preciso ter cuidado para separar as decisões baseadas em políticas dos detalhes de implementação (mecanismos). Essa separação permite flexibilidade máxima caso as decisões baseadas em políticas sejam alteradas posteriormente.

Os sistemas operacionais agora são quase sempre escritos em uma linguagem de implementação de sistema ou em uma linguagem de nível mais alto. Esse recurso melhora a sua implementação, manutenção e portabilidade. Para criar um sistema operacional para determinada configuração de máquina, devemos realizar a geração de sistema.

• Exercícios

3.1 Quais são as cinco principais atividades de um sistema operacional com relação à gerência de processos?

3.2 Quais são as três principais atividades de um sistema operacional com relação à gerência de memória?

3.3 Quais são as três principais atividades de um sistema operacional com relação â gerência de armazenamento secundário?

3.4 Quais são as cinco principais atividades de um sistema operacional com relação à gerência de arquivos?

3.5 Qual o objetivo do interpretador de comandos? Por que geralmente ele é separado do kernel?

3.6 Liste cinco serviços fornecidos por um sistema operacional. Explique como cada um fornece conveniência aos usuários. Explique em que casos seria impossível para os programas de nível de usuário fornecerem esses serviços.

3.7 Qual o propósito das chamadas ao sistema?

3.8 Usando as chamadas ao sistema, escreva um programa em C ou C+ + que leia dados de um arquivo e copie-os para outro. Tal programa foi descrito na Scção *Í3.*

3.9 Por que Java fornece a capacidade de chamar de um programa Java métodos nativos escritos em C ou C+ + ? Forneça um exemplo em que um método nativo é útil.

3.10 Qual o propósito dos programas de sistema?

3.11 Qual a principal vantagem da abordagem em camadas ao projeto do sistema?

3.12 Qual a principal vantagem da abordagem de microkernel ao projeto do sistema?

3.13 Qual a principal vantagem de um projetista de sistemas operacionais utilizar uma arquitetura de máquina virtual? Qual a principal vantagem para o usuário?

3.14 Por um compilador just-in-time é útil para executar programas Java?

3.15 Por que a separação entre mecanismo e política é uma propriedade desejável?

3.16 O sistema operacional experimental Synthcsis tem um montador incorporado no kernel. Para Otimizar o desempenho de chamadas ao sistema, o kernel monta rotinas dentro do espaço do kernel para minimizar o caminho que a chamada ao sistema deve utilizar através do kernel. Essa abordagem *é* a antítese da abordagem em camadas, na qual o caminho através do kernel é estendido, de modo que a construção do sistema operacional fique mais fácil. Discuta os prós e contras da abordagem Synthcsis ao projeto de kernel e à otimização do desempenho do sistema.

Notas bibliográficas

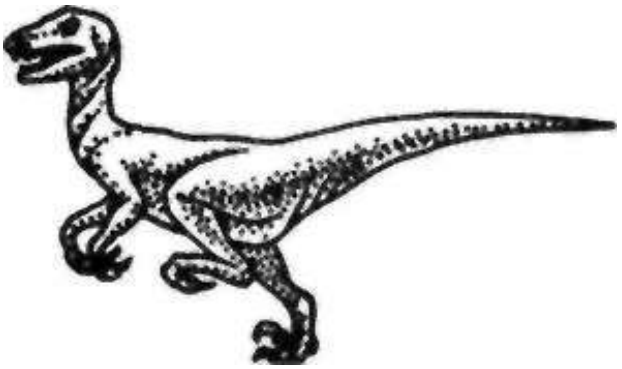
Dijkstra [1968] defendeu a abordagem em camadas ao projeto de sistemas operacionais. Brinch Hansen 11970] foi um dos primeiros a propor a construção de um sistema operacional como um kernel (ou núcleo) sobre o qual sistemas mais completos podem ser construídos.

O primeiro sistema operacional a fornecer uma máquina virtual foi o CP/67 em um IBM 360/67. O sistema operacional comercialmente disponível IBM VM/370 derivou do CP/67. Discussões gerais relativas a máquinas virtuais foram apresentadas por Hendricks e Hartmann (1979J, por MacKinnon |1979], e por Schultz [1988]. Cheung e Loong [1995| exploraram questões relativas à estruturação de sistemas operacionais desde microkerne! a sistemas extensíveis.

O MS-DOS, Versão 3.1, está descrito em [Microsoft 1986|. O Windows NT é descrito por Solomon | 1998|. O sistema operacional Apple Macintosh está descrito em [Apple 1987]. O UNIX de Berkeley está descrito em (CSRG 1986). O sistema padrão UNIX System V da AT&T está descrito em [AT&T 1986|. Uma boa descrição do OS/2 foi feita por lacobueci (1988].OMach foi introduzido por Accettae colegas [1986], e o AIX foi apresentado por Loucks e Sauer [1987]. O sistema operacional experimental Synthesisé discutido porMassalinePu|1989|.

A especificação para a linguagem Java c a máquina virtual Java é apresentada por Gosling e colegas [1996] e por Lindholm e Yellin [1998], respectivamente. O funcionamento interno da máquina virtual Java foi descrito por Venners [1998] e por Meyer e Downing [1997]. Existem vários bons livros de propósito geral sobre programação em Java (Flanagan 1997, Horstmanne Cornell 1998a, Nicmeyere Peck 1997]. Mais informações sobre Java estão disponíveis na Web no endereço http://www.javasoft.com.

Parte Dois



GERÊNCIA DE PROCESSOS

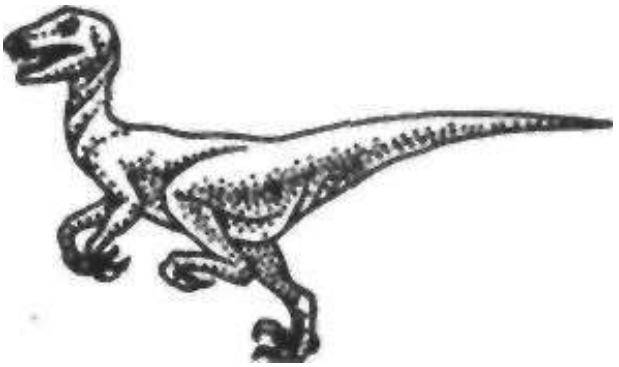
Um *processo* pode ser considerado um programa cm execução. Ele precisa de recursos - tais como tempo de CPU, memória, arquivos e dispositivos de I/O - para realizar uma tarefa. Esses recursos são alocados quando 0 processo é criado ou durante sua execução.

Um processo é a unidade de trabalho na maioria dos sistemas. Um sistema consiste em uma coleção de processos: os processos do sistema operacional executam código do sistema e os processos de usuário executam código de usuário. Todos esses processos podem ser executados de forma simultânea.

O sistema operacional é responsável pelas seguintes attvidades em relação à gerência de processos: a criação e exclusão de processos de sistema c de usuário; o escalonamento de processos e o fornecimento de mecanismos para sincronização, comunicação e tratamento de deadlocks para processos.

Embora tradicionalmente um processo contivesse um único *thrcad* de controle durante sua execução, a maioria dos sistemas operacionais modernos agora fornecem suporte a processos que possuem múltiplos threads.

Capítulo 4



PROCESSOS

Os primeiros sistemas de computação só permitiam que um programa fosse executado de cada vez. Esse programa tinha controle completo do sistema e acesso a todos os recursos do sistema. Os sistemas de computação modernos permitem que múltiplos programas sejam carregados na memória e executados de forma concorrente. Essa evolução exigiu maior controle e mais comparti mental ização dos vários programas. Essas necessidades resultaram na noção de um processo, que é um programa em execução. Um processo é a unidade de trabalho cm um sistema moderno de tempo compartilhado.

Quanto mais complexo for um sistema operacional, mais benefícios aos usuários são esperados. Embora sua principal preocupação seja a execução de programas de usuário, ele também precisa cuidar das várias tarefas de sistema que são mais bem executadas fora do kernel propriamente dito. Um sistema, portanto, consiste em uma coleção de processos: processos de sistema operacional que executam código do sistema e processos de usuário que executam código de usuário. Todos esses processos podem executar ao mesmo tempo, sendo a CPU (ou CPUs) multiplexada(s) entre eles. Ao alternar a CPU entre os processos, o sistema operacional pode tornar o computador mais produtivo.

4.1 • Conceito de processo

Um obstáculo à discussão de sistemas operacionais é que existe dificuldade em denominar todas as atividades da CPU. Um sistema em batch executa *jobs,* enquanto um sistema de tempo compartilhado executa *programas de usuário* ou *tarefas.* Mesmo em um sistema monousuário, como o Microsoft Windows ou Macintosh OS, um usuário pode executar vários programas de uma vez: um processador de textos, um navegador Web e um pacote de e-mail. Mesmo se o usuário só puder executar um programa de cada vez, o sistema operacional poderá precisar dar suporte a suas próprias atividades internas programadas, como gerência de memória. Em muitos aspectos, todas essas atividades são semelhantes, por isso chamamos todas de *processos.*

Os termos *job* e *processo* são usados quase que indistintamente neste livro. Apesar de pessoalmente preferirmos usar o termo *processo** boa parte da teoria e terminologia sobre sistemas operacionais foi desenvolvida em uma época na qual a principal atividade dos sistemas operacionais era o processamento de jobs. Seria erróneo evitar o uso de termos comumente aceitos que incluem a palavra *job* (como *escalonamento de jobs)* simplesmente porque *processo* suplantou *job.*

4.1.1 O processo

Informalmente, um processo é um programa em execução. Um processo é mais do que o código do programa, que às vezes *è* chamado seção de texto. Também inclui a atividade corrente, conforme representado pelo valor do contador do programa e 0 conteúdo dos registradores do processador. Um processo geralmente inclui a pilha de processo, que contém dados temporários (como parâmetros de métodos, endereços de retorno e variáveis locais) c uma seção de dados, que contém variáveis globais.

Enfatizamos que uni programa por si só não é um processo; um programa é uma entidade *passivo,* como o conteúdo de um arquivo armazenado em disco, enquanto um processo é uma entidade *atwd,* com um contador de programa especificando a próxima instrução a ser executada e um conjunto de recursos associados.

Embora dois processos possam ser associados com o mesmo programa, são considerados duas sequências separadas de execução. Por exemplo, vários usuários podem estar executando cópias diferentes do programa de correio ou o mesmo usuário pode chamar muitas cópias do programa editor. Cada uma dessas atividades é um processo separado e, embora as seções de texto sejam equivalentes, as seçóes de dados variam. Também *é* comum ter um processo que produza muitos processos durante sua execução. Essas questões são tratadas na Seção 4.4.

4.1.2 Estado do processo

A medida que o processo executa, ele muda de estado. O estado de uni processo é definido em parte pela atividade atual desse processo. Cada processo pode estar cm um dos seguintes estados:

- Novo: o processo está sendo criado.
- Em execução: as instruções estão sendo executadas.
- Em espera: o processo está esperando a ocorrência de algum evento {como conclusão de operação de I/O ou recepção de um sinal).
- Pronto: o processo está esperando para ser atribuído a um processador.
- Encerrado: o processo terminou sua execução.

Esses nomes são arbitrários, e variam dependendo do sistema operacional. Os estados que representam, no entanto, são encontrados em todos os sistemas. Certos sistemas operacionais também delineiam estados de processo de forma mais detalhada. Ê importante observar que apenas um processo pode estar *em execução* em qualquer processador a qualquer instante. No entanto, muitos processos podem estar *prontos* e *em espera** O diagrama de estado correspondente a esses estados está representado na Figura 4.1.

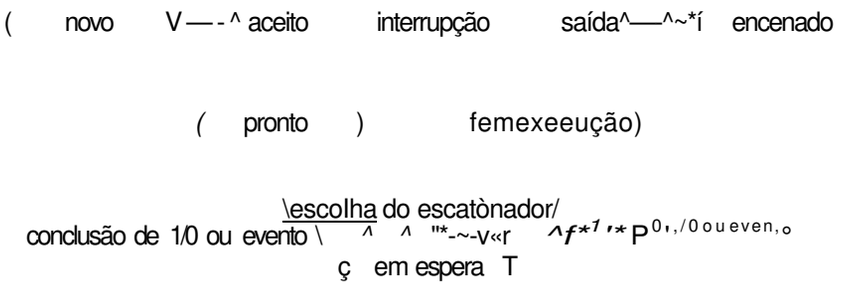


Figura 4.1 Diagrama de estado de um processo.

4.1.3 Bloco de controle de processo

Cada processo é representado no sistema operacional por um bloco de controle de processo (PCB - process control block), também chamado de bloco de controle de tarefa. A Figura 4.2 mostra um PCB. Ele contém muitas informações associadas a um processo específico, incluindo:

- *listado do processo:* O estado pode ser novo, pronto, em execução, em espera, suspenso, e assim por diante.
- *Contador do programai* O contador indica o endereço da próxima instrução a ser executada para esse processo.
- *Registradores de CPU:* Os registradores variam em número e tipo, dependendo da arquitetura do computador. Incluem acumuladores, registradores de índice, ponteiros de pilha e registradores de uso geral, além de informações de código de condição. Juntamente com o contador do programa, essas informações de estado devem ser salvas quando ocorre uma interrupção, para permitir que O processo continue corretamente depois disso (Figura 4..Í).

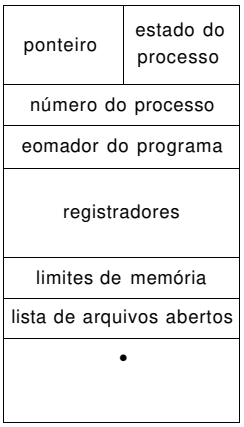


Figura 4.2 Bloco de controle de processo (PCB).

- *In for/nações de escalonamento de CPU:* Essas informações incluem prioridade de processo, ponteiros para filas de escalonamento e quaisquer outros parâmetros de escalonamento. (O Capítulo 6 descreve o escalonamento de processos.)
- *Informações de gerência de memória:* Essas informações podem incluir dados como o valor dos registradores de base e limite, as tabelas de páginas ou as tabelas de segmentos, dependendo do sistema de memória usado pelo sistema operacional (Capítulo 9).
- *Informações de contabilização:* Essas informações incluem a quantidade de CPU e o tempo real usados, limites de tempo, números de contas, números de jobs ou processos etc.
- *Informações de status de HO:* As informações incluem a lista de dispositivos de l/O alocados para este processo, uma lista de arquivos abertos etc.

O PCB serve simplesmente como repositório de informações que podem variar de processo a processo.

4.1.4 Threads

O modelo de processo discutido até agora considerava implicitamente que um processo é um programa que realiza um único fluxo de execução. Por exemplo, se um processo está executando um programa processador de textos, existe um único fluxo de instruções sendo executado. Esse fluxo único de controle só permite que o processo execute uma tarefa de cada vez. O usuário não pode digitar caracteres e passar o corretor ortográ-

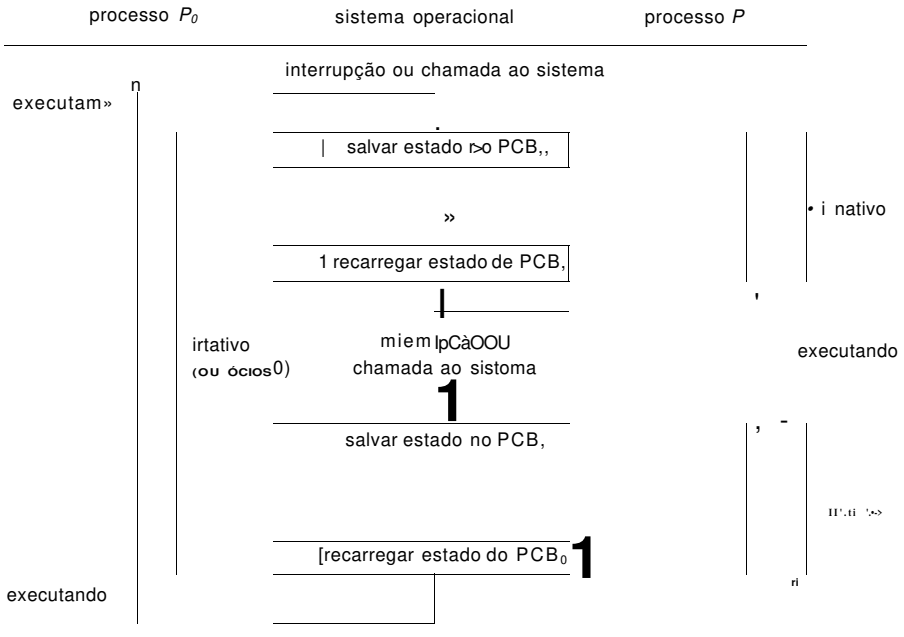


Figura 4.3 Diagrama mostrando a CPU alternando entre processos.

fico ao mesmo tempo no mesmo processo. Muitos sistemas operacionais modernos estenderam o conceito de processo para permitir que um processo tenha múltiplos fluxos de execução, ou threads. Assim, o processo pode executar mais de uma tarefa de cada vez. O Capítulo 5 explora processos com múltiplos threads.

4.2 • Escalonamento de processos

O objetivo da multiprogramação é ter processos cm execução o tempo todo, para maximizar a utilização de CPU. O objetivo do tempo compartilhado é alternar a CPU entre processos de forma tão frequente que os usuários possam interagir com cada programa durante sua execução. Para um sistema uniprocessador, nunca haverá mais de um processo cm execução. Sc houver mais processos, os demais terão de esperar até que a CPU esteja liberada e possa ser reescalonada.

4.2.1 Filas de escalonamento

À medida que os processos entram no sistema, são colocados em uma fila de jobs. Essa fila consiste em todos os processos do sistema. Os processos que estão residindo na memória principal e estão prontos e esperando para executar são mantidos em uma lista chamada fila de processos prontos *(ready queué).* Essa fila geralmente é armazenada como uma lista encadeada. Um cabeçalho de fila de processos prontos contém ponteiros ao primeiro e Último PCBs na lista. Estendemos cada PCB para incluir um campo de ponteiro apontando para o próximo PCB na fila de processos prontos.

Existem também outras filas no sistema. QuandoaCPUéalocadaa um processo, ele executa durante um tempo e *é* encerrado, interrompido ou espera pela ocorrência de determinado evento, como a conclusão de um pedido de l/O, por exemplo. No caso de um pedido de I/O, esse pedido pode ser para uma unidade de fita dedicada ou para um dispositivo compartilhado, como um disco. Como existem muitos processos no sistema, o disco pode estar ocupado com o pedido de l/O de algum outro processo. O processo, portanto, pode ter de esperar pelo disco. A lista de processos esperando por determinado dispositivo de l/O é chamada fila de dispositivo. Cada dispositivo tem sua própria fila de dispositivo (Figura 4.4).

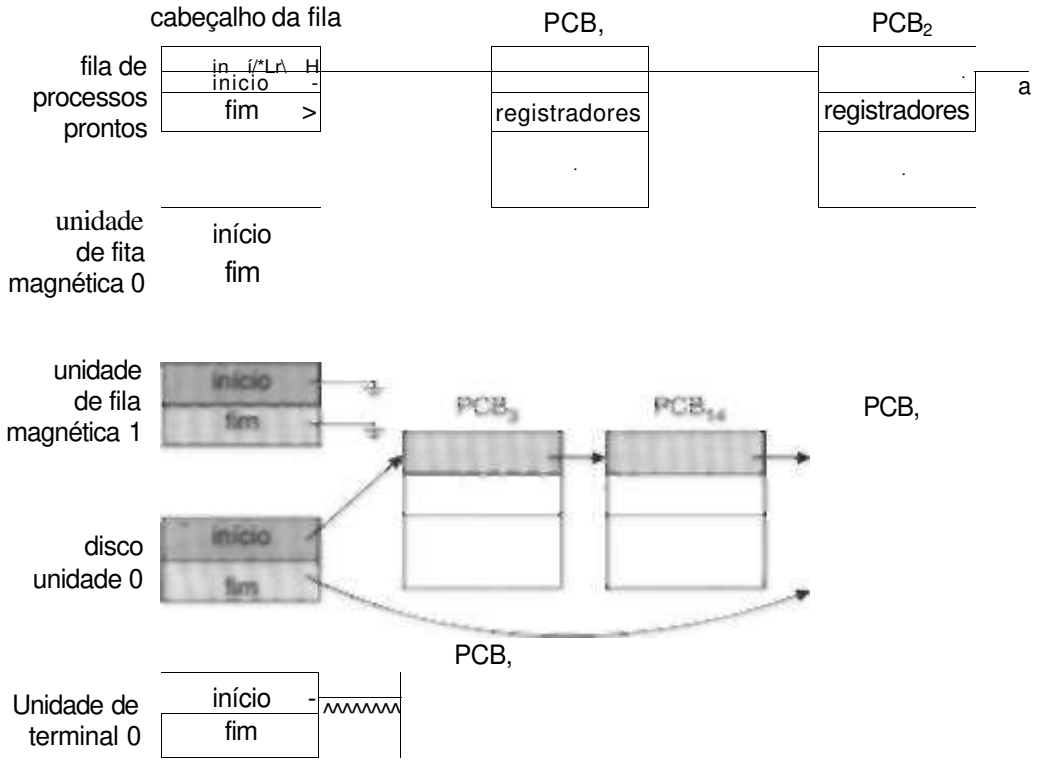


Figura 4.4 A fila de processos prontos e várias filas de dispositivos de l/O.

Uma representação comum pari uma discussão sobre escalonamento de processos é um diagrama de filas, como o da Figura 4.5. Cada caixa retangular representa uma fila. Dois tipos de fila estão presentes: a fila de processos prontos e um conjunto de filas de dispositivo. Os círculos representam os recursos que servem as filas e as setas indicam o fluxo de processos no sistema.

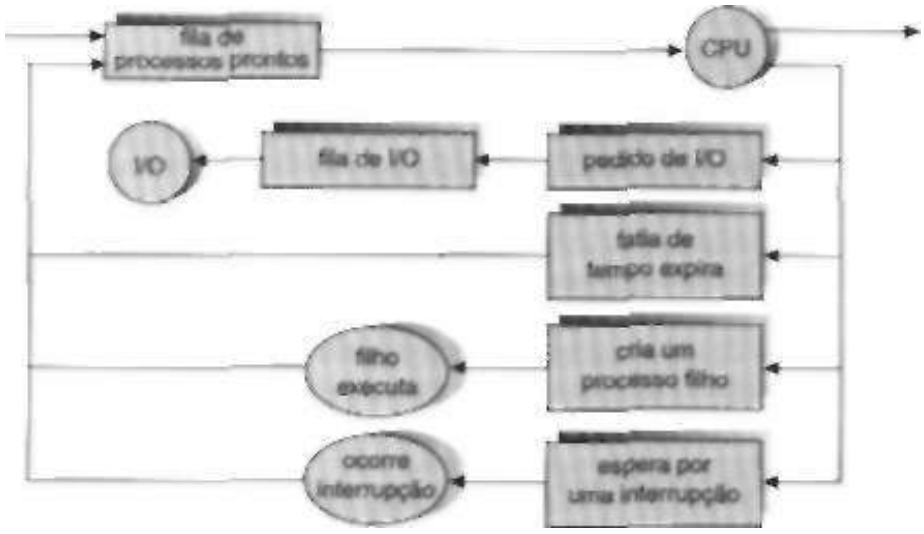


Figura 4.5 Representação do diagrama de filas do escalonamento de processos.

Um processo novo é colocado inicialmente na fila de processos prontos. Ele espera na fila até ser selecionado para execução ou ser submetido (dispatched). Depois que o processo recebe a CPU e está cm execução, um dos vários eventos a seguir poderá ocorrer:

- O processo pode emitir um pedido de I/O e ser colocado em uma fila de l/O.
- O processo pode criar um novo subprocesso c esperar seu término.
- O processo pode ser removido à força da CPU, como resultado de uma interrupção e ser colocado de volta na fila de processos prontos.

Nos dois primeiros casos, o processo acaba alternando do estado de espera para o estado de pronto e, em seguida, é colocado de volta na fila de processos prontos. Um processo continua o seu ciclo até terminar e, nesse ponto, é removido de todas as filas, com seu PCB e recursos sendo desalocados.

4.2.2 Escalonadores

Um processo migra entre as várias filas de escalonamento ao longo de toda sua vida. O sistema operacional deve selecionar, para fins de escalonamento, os processos dessas filas de alguma forma. O processo de Seleção *ê* executado pelo escalonador *(scheàuler)* adequado.

Em um sistema cm batch, existem geralmente mais processos submetidos do que processos que podem ser executados imediatamente. Esses processos são colocados em uni spool em um dispositivo de armazenamento de massa (geralmente um disco), onde são mantidos para execução posterior. O escalonador de longo prazo, ou o escalonador de jobs, seleciona processos desse conjunto c os carrega na memória para execução. O escalonador de curto prazo, ou o escalonador de CPU, seleciona dentre os processos que estão prontos para execução e aloca a CPU a um deles.

A principal distinção entre esses dois escalonadores é a frequência da sua execução. O escalonador de curto prazo deve selecionar um novo processo para a CPU com frequência. Um processo pode executar por apenas alguns milissegundos antes de esperar por um pedido de I/O. Em geral, o escalonador de curto prazo executa pelo menos uma vez a cada 100 milissegundos. Devido à breve duração de tempo entre as execuções, o escalonador de curto prazo deve ser rápido. Se levar 10 milissegundos para decidir executar um processo por 100 milissegundos, então 10/(100 + 10) = 9% da CPU está sendo usado (desperdiçado) simplesmente para escalonar o trabalho.

O cscalonador de longo prazo, por outro lado, executa com muito menos frequência. Pode haver um intervalo de minutos entre a criação de novos processos no sistema. O escalonador de longo prazo controla o grau de multiprogramação (o número de processos na memória). Sc o grau de mui ti programação for estável, a taxa media de criação de processos deve ser igual a taxa média de partida de processos que saem do sistema. Assim, o escalonador de longo prazo pode precisar ser chamado apenas quando um processo sair do sistema. Devido ao intervalo maior entre as execuções, o escalonador de longo prazo pode levar mais tempo para decidir que processos devem ser selecionados para execução.

E importante que o escalonador de longo prazo faça uma Seleção cuidadosa. Em geral, a maioria dos processos podem ser descritos como limitados por I/O ou limitados pela CPU. Um processo limitado por I/O passa mais tempo realizando operações de I/O do que efetuando cálculos. Um processo limitado pela CPU, por outro lado. gera pedidos de I/O com pouca frequência, usando mais o seu tempo na computação. E importante que o escalonador de longo prazo selecione uma boa combinação de processos incluindo processos limitados por I/O e pela CPU. Sc todos os processos forem limitados por I/O, a fila de processos prontos quase sempre estará vazia e o escalonador de curto prazo terá pouco trabalho. Se todos os processos forem limitados pela CPU, a fila de espera de I/O quase sempre estará vazia, os dispositivos ficarão sem uso e mais uma vez o sistema ficará desequilibrado. O sistema com o melhor desempenho terá uma combinação de processos limitados pela CPU e por I/O.

Em alguns sistemas, o cscalonador de longo prazo pode estar ausente ou ser mínimo. Por exemplo, sistemas de tempo compartilhado, como o UNIX, geralmente não têm um escalonador de longo prazo, mas simplesmente colocam todo novo processo na memória para o escalonador de curto prazo. A estabilidade desses sistemas depende de uma limitação física (como o número de terminais disponíveis) ou da natureza de auto-ajuste dos usuários humanos. Se o desempenho cair para níveis inaceitáveis, alguns usuários simplesmente vão desistir.

Alguns sistemas operacionais, como os de tempo compartilhado, podem introduzir um nível intermediário adicional de escalonamento. O escalonador de médio prazo está representado na Figura 4.6. A principal ideia por trás de um cscalonador de médio prazo *é* que às vezes pode ser vantajoso remover processos da memória (e da disputa ativa por CPU) e, assim, reduzir o grau de multiprogramação. Em algum momento posterior, o processo pode ser introduzido novamente na memória e sua execução pode ser retomada do ponto onde parou. Esse esquema é chamado de *swapping* (troca). O escalonador de médio prazo realiza as operações de swapping. O swapping pode ser necessário para melhorar a combinação de processos ou porque uma mudança nos requisitos de memória comprometeu a memória disponível, exigindo a liberação de memória. O Capítulo 9 discute o conceito de swapping.

Figura 4.6 Acréscimo do escalonamento de médio prazo ao diagrama de filas.

4.2.3 Troca de contexto

Alternar a CPU para outro processo requer salvar o estado do processo antigo e carregar o estado salvo do novo processo. Essa tarefa é chamada de troca de contexto. O contexto de um processo é representado no PCB de um processo; inclui o valor dos registradores de CPU, o estado do processo (consulte a Figura 4.1) e as informações de gerência de memória. Quando ocorre uma troca de contexto, qjcerncl salva o contexto do processo antigo em seu PCB e carrega o çontextosàlvo do novo pTocesso escolhido para execução. O tempo de troca de contexto é puro *overbead,* já que o sistema não efetua trabalho útil durante o processo de troca.

Sua velocidade varia de máquina a máquina dependendo da velocidade da memória, do número de registradores que devem ser copiados c da existência de instruções especiais (como uma única instrução para carregar ou armazenar todos os registradores). Velocidades típicas estão entre 1 a 1000 microssegundos.

Os tempos de troca de contexto são altamente dependentes do suporte de hardware. Por exemplo, alguns processadores (como o Sun UltraSPARC) fornecem vários conjuntos de registradores. Uma troca de contexto simplesmente inclui mudar o ponteiro para o conjunto de registradores atual. E claro que se houver mais processos ativos do que conjuntos de registradores, o sistema vai copiar os dados do registrador de e para a memória como antes. Além disso, quanto mais complexo o sistema operacional, mais trabalho deve ser realizado durante uma troca de contexto. Como será discutido no Capítulo 9, técnicas avançadas de gerência de memória podem exigir que dados extras sejam trocados com cada contexto. Por exemplo, o espaço de endereçamento do processo atual deve ser preservado à medida que o espaço da próxima tarefa é preparado para uso. Como o espaço de endereçamento é preservado e quanto trabalho será necessário para preservá-lo dependerão do método de gerência de memória do sistema operacional. Como será discutido no Capítulo 5, a troca de contexto tomou-se um gargalo de desempenho de tal ordem que os programadores estão utilizando novas estruturas (threads) para evitá-la sempre que possível.

4.3 • Operações nos processos

Os processos no sistema podem executar de forma concorrente e devem ser criados e excluídos de forma dinâmica. Assim, o sistema operacional deve fornecer um mecanismo para a criação e término de processos.

4.3.1(Criação de processo^)

Um processo pode criar vários novos processos através de uma chamada ao sistema para a criação de processo, durante sua execução. O processo criador é chamado **de processo pai, enquanto** os **novos** processos^são chamados de filhos desse processo. Cada um dos novos processos, por sua vez, pode criar outros processos, formando uma árvore de processos (Figura 4.7).

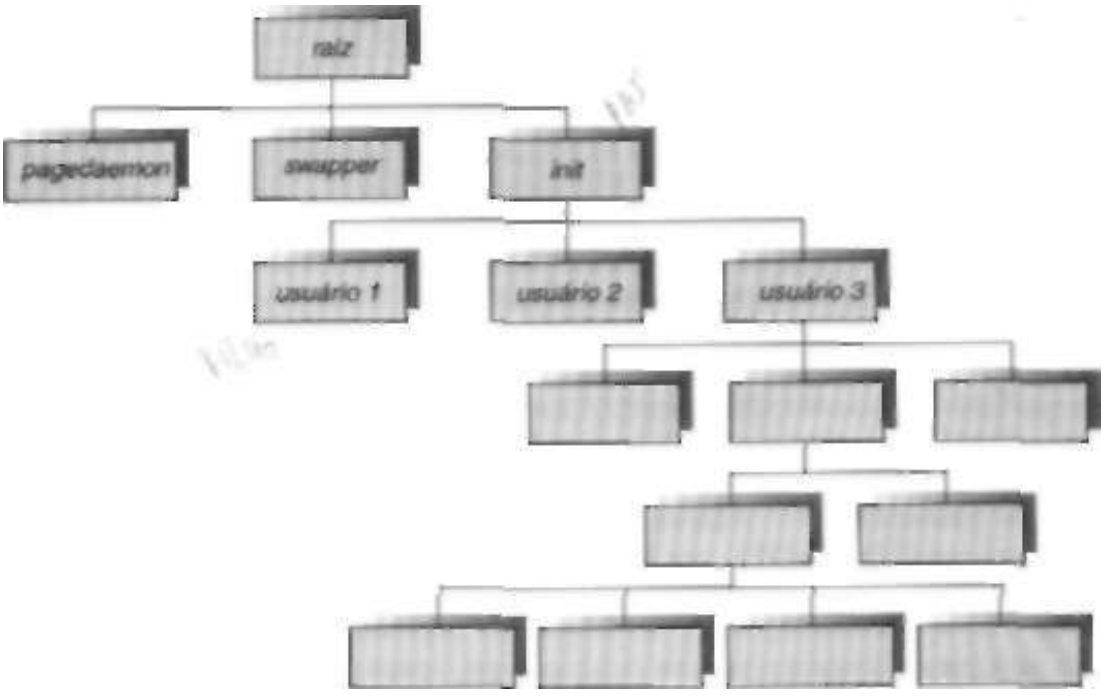


Figura 4.7 Uma árvore de processos cm um sistema UNIX típico.

Em geral, um processo precisará de determinados recursos (tempo de CPU, memória, arquivos, dispositivos de I/O) para realizar sua tarefa. Quando um processo cria um subprocesso, esse subprocesso pode ser capaz de obter seus recursos diretamente do sistema operacional ou pode ser limitado a um subconjunto dos recursos do processo pai. O pai pode ter de dividir seus recursos entre os filhos, ou talvez possa compartilhar

alguns recursos (como memória ou arquivos) entre vários filhos. Restringir um processo filho a um subconjunto dos recursos do pai evita que algum processo sobrecarregue o sistema criando subprocessos demais.

Além dos vários recursos físicos e lógicos que um processo obtém quando é criado, os dados de inicialização (entrada) podem ser passados pelo processo pai para o processo filho. Por exemplo, considere um processo cuja função seja exibir o status de um arquivo, digamos Fí, na tela de um terminal. Quando ele for criado, receberá como entrada do seu processo pai o nome de arquivo *F1* e executará usando esse dado para obter as informações desejadas. Também pode receber o nome do dispositivo de saída. Alguns sistemas operacionais passam recursos para os processos filhos. Km um sistema como esses, o novo processo pode obter dois arquivos abertos, Fí e o dispositivo do terminal, e pode apenas precisar transferir o dado entre os dois.

Quando um processo cria um novo processo, existem duas possibilidades em termos de execução:

1, O pai continua a executar de forma concorrente com os filhos.

2. O pai espera até que alguns ou todos os filhos tenham terminado.

Existem também duas possibilidades em termos de espaço de endereçamento do novo processo:

1. O processo filho é uma duplicata do processo pai.
2. O processo filho tem um programa carregado nele.

Para ilustrar essas implementações diferentes, vamos considerar o sistema operacional UNIX. No UNIX, cada processo é identificado por seu identificador de processo, que é um inteiro único. Um novo processo é criado pela chamada ao sistema f ork. O novo processo consiste em uma cópia do espaço de endereçamento do processo original. Esse mecanismo permite que o processo pai se comunique facilmente com o processo filho. Ambos os processos (o pai e o filho) continuam a execução da instrução após o f ork com uma diferença: o código de retorno para f ork é zero para o novo processo (filho), enquanto o identificador de processo filho (diferente de zero) é retornado ao pai.

Geralmente, a chamada ao sistema execlp é usada após uma chamada fork por um dos dois processos, para substituir o espaço de memória do processo por um novo programa. A chamada execl p carrega um arquivo binário na memória (destruindo a imagem na memória do programa que contém a chamada ao sistema execl p) e inicia sua execução. Dessa maneira, os dois processos podem se comunicar e, em seguida, continuar cada um o seu caminho. O pai pode criar mais filhos ou, se não tiver nada a fazer enquanto o filho executa, ele poderá emitir uma chamada ao sistema wai t para sair da fila de processos prontos até o término do filho. O programa C mostrado na Figura 4.8 ilustra as chamadas ao sistema UNIX descritas anteriormente. O pai cria um processo filho usando a chamada fork( ). Agora temos dois processos diferentes executando uma cópia do mesmo programa. O valor de pid para o processo filho é zero; o valor para o processo pai é um valor inteiro maior do que zero. O processo filho sobrepõe seu espaço de endereçamento com o comando UNIX /bi n/l s (usado para obter uma listagem de diretórios) utilizando a chamada ao sistema execl p ( ). O pai espera que o processo filho termine, usando a chamada wai t( ). Quando o processo filho termina, o processo pai retoma a partir da chamada wait( ), onde conclui usando a chamada ao sistema exit( ).

O sistema operacional VMS da DEC, por outro lado, cria um novo processo, carrega um programa especificado nesse processo c começa a sua execução. O sistema operacional Microsoft Windows NT suporta ambos os modelos: o espaço de endereçamento do pai pode ser duplicado, ou o pai pode especificar o nome de um programa para que o sistema operacional o carregue no espaço de endereçamento do novo processo.

4.3.2 Término do processo

Um processo termina quando acaba de executar sua instrução final e pede que o sistema operacional o exclua usando a chamada ao sistema exi t. Nesse ponto, o processo pode retomar dados (saída) ao seu processo pai (via chamada wai t). Todos os recursos do processo - incluindo memória física e virtual, arquivos abertos e buffers de I/O - são desalocados pelo sistema operacional.

```
íinclude <stdio.h>

void main(int arge, char *argv[ ])
(
int pi d;

    /•bifurcação em outro processo */
    pid •fagj J; *          | ,

    if {pid < 0) { /• ocorreu um erro */
      fprintffstderr, "Fork Falhou');
      exit(-l);
    I
   ^else if (pid •• 0) { /" processo filho •/
 (4);execlp(7b1n/1s\"ls\NULL);

    else ( /* processo pai •/
      /* pai espçrarS a conclusão do filho */
      Mlt(NULLh
      prtntfCFilho concluiu");
    vexit(0>;
      )
```

Figura 4.8 Programa C criando uma nova imagem de processo.

Existem circunstâncias adicionais nas quais ocorre o término. Um processo pode causar o término de outro processo via uma chamada ao sistema adequada (por exemplo, abort). Geralmente, essa chamada pode ser feita apenas pelo pai do processo que deverá ser terminado. Caso contrário, os usuários podem interromper arbitrariamente os jobs uns dos outros. Observe que um pai precisa conhecer as identidades de seus filhos. Assim, quando um processo cria um novo processo, a identidade do processo recém-criado é passada para o pai.

Um pai pode terminar a execução de um de seus filhos por vários motivos, a saber:

- O filho excedeu seu uso de alguns dos recursos que foram alocados.
- A tarefa atribuída ao filho não é mais exigida.
- O pai está saindo, c o sistema operacional não permite que um filho continue se seu pai terminar.

Para determinar o primeiro caso, o pai deve ter um mecanismo para inspecionar o estado de seus filhos.

Muitos sistemas, incluindo o VMS, não permitem a existência de um filho se seu pai tiver terminado. Em tais sistemas, se um processo terminar (de forma normal ou anormal), todos os filhos também devem ser terminados. Esse fenómeno, chamado de término cm cascata, é normalmente iniciado pelo sistema operacional.

Para ilustrar a execução <* o término de processos, consideramos que, no UNIX, podemos terminar um processo usando a chamada ao sistema exi t; seu processo pai pode esperar pelo término de um processo filho usando a chamada wai t. A chamada wai t retorna o identificador de processo de um filho terminado, de modo que o pai possa dizer qual dos seus possíveis muitos filhos terminou. Se o pai terminar, no entanto, todos os seus filhos recebem como novo pai o processo *init.* Assim, os filhos ainda têm um pai para colctar seu status e estatísticas de execução.

4.4 • Processos cooperativos

Os processos concorrentes executando no sistema operacional podem ser processos independentes ou processos cooperativos. Um processo é independente se não puder afetar ou ser afetado pelos outros processos executando no sistema. Claramente, qualquer processo que não compartilhe dados (temporários ou persistentes) com outro processo é independente. Por outro lado, um processo é cooperativo se puder afetar ou ser afetado por outro processo executando no sistema. Claramente, qualquer processo que compartilhe dados com outros processos é um processo cooperativo.

Existem vários motivos para fornecer um ambiente que permita a cooperação entre processos:

- *Compartilhamento de informações:* Como vários usuários podem estar interessados na mesma informação (por exemplo, um arquivo compartilhado), é preciso fornecer um ambiente para permitir o acesso concorrente a esses tipos de recursos.
- *Velocidade de computação:* Se queremos que determinada tarefa execute mais rápido, é preciso quebrá-la em subtarefas, cada qual sendo executada cm paralelo com as demais. Observe que o aumento na velocidade só pode ser alcançado se o computador tiver múltiplos elementos de processamento (tais como CPUs ou canais de I/O).
- *Modularidade:* Talvez queiramos construir o sistema de forma modular, dividindo as funções do sistema em processos ou ihreads separados, como discutido no Capítulo 3.
- *Conveniência:* Mesmo um usuário individual pode ter muitas tarefas nas quais trabalhar cm determinado momento. Por exemplo, um usuário pode estar editando, imprimindo e compilando cm paralelo.

A execução concorrente que requer a cooperação entre os processos exige mecanismos para permitir que os processos se comuniquem entre si (Seção 4.5) c sincronizem suas ações (Capítulo 7).

Para ilustrar o conceito de processos cooperativos, vamos considerar o problema do produtor-consumidor, que é um paradigma comum para os processos cooperativos. Um processo produtor produz informações que são consumidas por um processo consumidor. Por exemplo, um programa de impressão produz caracteres que são consumidos pelo driver de impressão. Um compilador pode produzir código assembly, que é consumido por um montador. O montador, por sua vez, pode produzir módulos objeto, que são consumidos pelo utilitário de carga.

Para permitir que os processos de produtor e consumidor executem de forma concorrente, *é* preciso ter disponível um buffer de itens que pode ser preenchido pelo produtor e esvaziado pelo consumidor. Um produtor pode produzir um item enquanto um consumidor está consumindo outro item. O produtor e o consumidor devem ser sincronizados, de modo que o consumidor não tente consumir um item que ainda não tenha sido produzido. Nessa situação, o consumidor deve esperar até que um item seja produzido.

O problema do produtor-consumidor de buffer náo-limitado não impõe limite prático no tamanho do buffer. O consumidor talvez tenha de esperar por novos itens, mas o produtor sempre poderá produzir novos itens. O problema do produtor-consumidor de buffer limitado assume que existe um tamanho fixo de buffer. Nesse caso, o consumidor deve esperar se o buffer estiver vazio e o produtor deve esperar se o buffer estiver cheio.

O buffer pode ser fornecido pelo sistema operacional através do uso de um recurso de comunicação entre processos (IPC, *interprocess communication)* (Seção 4.5), ou explicitamente codificado pelo programador da aplicação com o uso de memória compartilhada. Vamos ilustrar uma solução de memória compartilhada para o problema de buffer limitado. Os processos produtor e consumidor compartilham uma instância da classe BoundedBuffer (Figura 4.9).

O buffer compartilhado é implementado como um vetor circular com dois ponteiros lógicos: in e out. A variável i n aponta para a próxima posição livre no buffer; out aponta para a primeira posição cheia no buffer. counté o número de itens presentes no buffer no momento. O buffer está vazio quando count == 0 e está cheio quando count •- BUFFERSIZE. O processo produtor chama o método enter( ) (Figura 4.10) quando deseja inserir um item no buffer e o consumidor chama o método removei) (Figura 4.11) quando deseja consumir um item do buffer. Observe que o produtor e o consumidor ficarão bloqueados no laço whi 1 e se o buffer estiver, respectivamente, cheio ou vazio.

No Capítulo 7, discutimos como a sincronização entre processos cooperativos pode ser implementada de forma eficiente em um ambiente de memória compartilhada.

4.5 • Comunicação entre processos

Na Seção 4.4, mostramos como os processos cooperativos podem se comunicar em um ambiente de memória compartilhada. O esquema requer que esses processos compartilhem um poot de buffers comum e que o código para implementar o buffer seja explicitamente escrito pelo programador da aplicação. Outra forma de obter o mesmo efeito é fornecer o meio para os processos cooperativos se comunicarem entre si através de um recurso de comunicação entre processos (IPC).

```
import java.util.";

public class BoundedBuffer
|
    public BoundedBuffer( ) {
       // buffer esta* inicialmente vazio
       count • 0;
       In * 0;
       out • 0;

       buffer • new Object[BUFFER SIZE];
    )

    // produtor chana este método
    public void enter(Object item) {
    // Figura 4.10
    )
    // Consumidor chama este método
    public Object removei ) I
    // Figura 4.11
    I

    public static final int NAP_TIHE = 5;
    private static final int BUFFER_S1ZE • 5;
    private volatile int count;
    private int in; // aponta para a próxima posição livre
    private int out; // aponta para a próxima posição cheia
    private ObjectE ] buffer;
1
```

Figura 4.9 Solução de memória compartilhada para o problema do produtor-consumidor.

```
public void enter(0bject item) {
    while (count •- BUFFER_SIZE)
          ; // náo faz nada

    // adiciona um item ao buffer
    ••count;
    buffer[in] = item;
    in - (in + 1) % BUFFER_SIZE;

    if (count == BUFFERJIZE)
          System.out.println ("Producer Entered " •
              item * "Buffer FULL');
    else
          System.out.println ("Producer Entered " •
              item • " Buffer Size = " • count);
)
```

Figura 4.10 O método enter( ).

O IPC fornece um mecanismo para permitir que os processos comuniquem e sincronizem suas ações sem compartilhar o mesmo espaço de endereçamento. O IPC é particularmente útil em um ambiente distribuído no qual os processos em comunicação podem residir em diferentes computadores conectados via rede. Um exemplo é um programa de batc-papo (chat) usado na World Wide Web. No Capítulo 15, analisamos a computação distribuída usando Java.

A melhor forma de fornecer IPC é através do sistema de troca de mensagens, e os sistemas de mensagem podem ser definidos de várias formas diferentes. Vamos analisar diferentes aspectos de projeto e apresentamos uma solução Java para o problema do produtor-consumidor que utiliza troca de mensagens.

4.5.1 Sistema de troca de mensagens

A função de um sistema de mensagens é permitir que os processos se comuniquem entre si sem necessidade de recorrer a dados compartilhados. Um recurso de IPC fornece pelo menos duas operações: *senâ(message)* e *recei ve(message).*

```
public Object removei ) {
    Object item;

    while (count •• 0)
          ; // nào faz nada
    // remove um item do buffer
    —count;
    item = buffer[outJ;
    out - (out + 1) % BUFFER_SIZE;

    if (count « 0)
          System.out.println("Consumer Consumed " +
              item * " Buffer EMPTY-);
    else
          System.out.printlnfConsumer Consumed " *
              item • " Buffer Size • " + count);

    return item;
Í
```

Figura 4. II O método removei).

As mensagens enviadas por um processo podem ser de tamanho fixo ou variável. Se apenas mensagens de tamanho fixo puderem ser enviadas, a implementação no nível do sistema será simples. Essa restrição, no entanto, torna a tarefa de programação mais difícil. Por outro lado, as mensagens de tamanho variável exigem uma implementação mais complexa no nível do sistema, mas a tarefa de programação se torna mais simples.

Se os processos Pe *Q* quiserem se comunicar, deverão enviar c receber mensagens um do outro; um canal de comunicação deve existir entre eles. Esse canal pode ser implementado de várias formas. Estamos preocupados não com a implementação física do canal (como memória compartilhada, barramento de hardware ou rede, que são tratados no Capitulo 14), mas sim com sua implementação lógica. Existem vários métodos para implementar logicamente um canal e as operações de send/receive:

- Comunicação direta ou indireta
- Comunicação simétrica ou assimétrica
- lluffering automático ou explícito
- Enviar por cópia ou enviar por referência
- Mensagens de tamanho fixo ou variável

A seguir analisamos cada um desses tipos de sistemas de mensagens.

4.5.2 Nomeação

Os processos que desejam se comunicar precisam ter uma forma de fazer referencia mútua. Eles podem usar comunicação direta ou indireta.

4.5.2.1 Comunicação direta

Na comunicação direta, cada processo que deseja se comunicar precisa identificar explicitamente o destinatário ou remetente da comunicação. Neste esquema, as primitivas send e receive são definidas como:

- send(P, message) - Envia uma mensagem para o processo P.
- receive(Q, message) - Recebe uma mensagem do processo Q.

Um canal de comunicação neste esquema possui as seguintes propriedades:

- Um canal é estabelecido automaticamente entre cada par de processos que deseja se comunicar. Os processos precisam conhecer as respectivas identidades para se comunicar.
- Um canal é associado a exatamente dois processos.
- Entre cada par de processos, existe exatamente um canal.

Esse esquema apresenta simetria no endereçamento; ou seja, os processos de envio e recepção precisam se identificar mutuamente para haver comunicação. Uma variante do esquema utiliza assimetria no endereçamento. Apenas o remetente identifica o destinatário; o destinatário não precisa identificar o remetente. Neste esquema, as primitivas send e receive são definidas assim:

- send(P, message) - Envia uma mensagem para o processo P.
- receive(id, message) - Recebe uma mensagem de qualquer processo; a variável *id* retorna o nome do processo com o qual foi estabelecida a comunicação.

A desvantagem desses esquemas (simétrico e assimétrico) c a modularidade limitada das definições de processos resultantes. Mudar o nome de um processo pode exigir a verificação de todas as definições de processos. Todas as referencias ao nome anterior devem ser encontrados para que possam ser alteradas para o novo nome. Essa situação não c desejável do ponto de vista de compilação separada.

4.5.2.2 Comunicação indircta

Com a comunicação indireta, as mensagens são enviadas e recebidas através de caixas de correio ou portas. Uma caixa de correio pode ser vista de forma abstrata como um objeto no qual as mensagens podem ser colocadas por processos e do qual as mensagens podem ser retiradas. Cada caixa de correio tem uma identificação exclusiva. Nesse esquema, um processo pode se comunicar com outro processo através de várias caixas de correio diferentes. Dois processos só podem se comunicar se tiverem uma caixa de correio comum. As primitivas send e receive são definidas da seguinte forma:

- send (A, message) - Envia uma mensagem para a caixa de correio A.
- receive(A, message) - Recebe uma mensagem da caixa de correio A.

Neste esquema, um canal de comunicação possui as seguintes propriedades:

- Um canal c estabelecido entre um par de processos somente se os dois membros do par tiverem uma caixa de correio compartilhada.
- Um canal pode estar associado a mais de dois processos.
- Entre cada par de processos comunicantes, pode haver vários canais diferentes, com cada canal correspondendo .. uma caixa de correio.

Vamos supor que os processos P, P_t e P, compartilhem a mesma caixa de correio A. O processo P, envia uma mensagem para/\, enquanto P, e P_t executam uma operação de receive em A Que processo receberá a mensagem enviada por P,} A resposta depende do esquema escolhido:

- Permitir que um canal seja associado com no máximo dois processos.
- Permitir que no máximo um processo de cada vez execute uma operação de receive.
- Permitir que o sistema selecione arbitrariamente que processo receberá a mensagem (ou seja, $P<$ ou P_t, mas não ambos, receberá a mensagem). O sistema poderá identificar o destinatário para o remetente.

Uma caixa de correio pode ser de propriedade de um processo ou do sistema operacional. Se a caixa de correio pertencer a um processo (ou seja, a caixa de correio é parte do espaço de endereçamento do processo), fazemos a distinção entre o proprietário (que só pode receber mensagens através desta caixa de correio) e o usuário da caixa de correio (que só pode enviar mensagens para a caixa de correio). Como cada caixa tem um proprietário exclusivo, não pode haver confusão sobre quem deve receber uma mensagem enviada para esta caixa de correio. Quando um processo que possui uma caixa de correio termina, a caixa desaparece.

Qualquer processo subsequente que envie uma mensagem para essa caixa de correio deverá ser avisado de que a caixa não existe mais.

Por outro lado, uma caixa de correio pertencente ao sistema operacional tem existência própria. É independente e não está ligada a nenhum processo específico. O sistema operacional deverá fornecer um mecanismo para permitir que um processo faça o seguinte:

- Crie uma nova caixa de correio
- Envie e receba mensagens através da caixa de correio
- Exclua uma caixa de correio

O processo que cria uma nova caixa de correio é por default o proprietário da caixa. Inicialmente, o proprietário é o único processo que pode receber mensagens através dessa caixa de correio. No entanto, a propriedade e o privilégio de recebimento podem ser passados para outros processos através de chamadas ao sistema adequadas. E claro que esse procedimento pode resultar em vários destinatários para cada caixa de correio.

4.5.3 **Sincronização**

A comunicação entre processos ocorre por meio de chamadas às primitivas send e recei ve. Existem diferentes opções de projeto para implementar cada primitiva. A troca de mensagens pode ser do tipo bloqueante ou não-bloqucante - também chamado de síncrono c assíncrono.

- Envio bloqueante: O processo de envio é bloqueado até que a mensagem seja recebida pelo processo receptor ou pela caixa de correio.
- Envio não-bloqucante: O processo de envio envia a mensagem e retoma a operação.
- Recepção bloqueante: O receptor é bloqueado até que uma mensagem esteja disponível.
- Recepção não-bloqueante: O receptor recebe uma mensagem válida ou então nada.

Diferentes combinações de send c recei ve são possíveis. Quando ambos forem bloqueantes, temos um *rendezvous* - é um termo usual (encontro, em francês) entre o remetente e o destinatário.

4.5.4 Buffering

Quer a comunicação seja direta ou indireta, as mensagens trocadas pelos processos em comunicação residem em uma fila temporária. Basicamente, existem três formas de implementar uma fila desse tipo:

- Capacidade zero: A fila tem tamanho máximo 0; assim, o canal não pode ter mensagens em espera. Nesse caso, o remetente deve bloquear até que o destinatário receba a mensagem.
- Capacidade limitada: A fila tem tamanho finito n; assim, no máximo;; mensagens podem residir nela. Se a fila não estiver cheia quando uma nova mensagem for enviada, esta é colocada na fila (a mensagem é copiada ou é mantido um ponteiro para a mensagem) e o remetente pode continuar a execução sem esperar. No entanto, o canal tem capacidade finita. Se o canal estiver cheio, o remetente deverá bloquear (no sentido de "ficar bloqueado") até que haja espaço disponível na fila.
- Capacidade náo-limitada ou ilimitada: Potencialmente, a fila tem tamanho infinito; assim, qualquer número de mensagens pode ficar esperando na fila. O remetente nunca bloqueia.

O caso de capacidade zero às vezes é chamado de sistema de mensagem sem buffering; os outros casos são chamados de buffering automático.

4.5.5 Exemplo do produtor-consumidor

Agora podemos apresentar uma solução ao problema do produtor-consumidor que utiliza troca de mensagem. O produtor e o consumidor se comunicam de forma indireta utilizando a caixa de correio compartilhada mostrada na Figura 4.12.

O buffer é implementado usando a classe java.uti 1 .Vector, o que significa que será um buffer de capacidade ilimitada. Além disso, os métodos send( ) e receive( ) são não-bloqueantes.

Quando o produtor gera um item, ele o coloca na caixa de correio através do método $end ( ). O código para o produtor aparece na Figura 4.13.

O consumidor obtém um item da caixa de correio usando o método receive( ). Como receive( ) é não-bloqueante, o consumidor deve avaliar o valor doObject retornado de recei ve( ). Se for nul1,a caixa de correio estará vazia. O código para o consumidor é apresentado na Figura *4.*14.

O Capítulo 5 mostra como implementar o produtor e o consumidor como threads separados e como permitir que a caixa de correio seja compartilhada entre os threads.

```
import java.util.*;

public class HessageQueue
i

    public HessageQueue( ) {
      queue • new Vector( );
    Í

    // implementa um envio náo-bloqueaite
    public void send(Object item) {
      queue.addElement{item);
    |

    // implementa uma recepção não-bloqueante
    public Object receive( ) {
      Object item;

      ff (queue.size{ ) " 0)
            return null;
      else {
            item • queue.firstElement( );
            queue.removeElementAt(0);

            return item;
      }
    )

    private Vector queue;
Í
```

Figura 4.12 Caixa de correio para troca de mensagens.

```
HessageQueue mailBox;

while (true) {
    Oate message • new 0ate( );
    ma)1 Box.send(message);
1
```

Figura 4.13 O processo Produtor.

```
HessageQueue mailBox;

while (true) (
    Date message = (Date) mailBox.receitei );
    if (message ! • null)
        // consome a mensagem
Í
```

Figura 4.14 O processo Consumidor.

4.5.6 Um exemplo: Mach

Como exemplo de um sistema operacional baseado em mensagens, considere o sistema Mach, desenvolvido na Carnegie Mellon University. O kernel do Mach oferece suporte à criação e destruição de várias tarefas, que são semelhantes aos processos mas possuem múltiplos fluxos de controle. A maior parte da comunicação no Mach - incluindo boa parte das chamadas ao sistema e todas as informações entre tarefas- é realizada por *mensagens.* As mensagens são enviadas e recebidas através das caixas de correio, chamadas *portas* no Mach.

Mesmo as chamadas ao sistema são feitas por mensagens. Quando cada tarefa é criada, duas caixas de correio especiais - a caixa Kernel e a caixa Notify - também são criadas. A caixa de correio Kernel é usada pelo kernel para se comunicar com a tarefa. O kernel envia notificações de ocorrência de eventos para a porta Notify. Apenas três chamadas ao sistema são necessárias para a transferência de mensagens. A chamada msgsend envia uma mensagem a uma caixa de correio. Uma mensagem é recebida via msg recei ve. As chamadas de procedimento remoto (RPCs, *Remote Procedure Calls)* são executadas via msgrpc, que envia uma mensagem e espera por exatamente uma mensagem de retorno do remetente. Dessa forma, RPC modela uma chamada de procedimento típica, mas pode funcionar entre sistemas.

A chamada ao sistema porta 11 ocate cria uma nova caixa de correio e aloca espaço para sua fila de mensagens. O tamanho máximo da fila de mensagens pressupõe como padrão oito mensagens. A tarefa que cria a caixa de correio é proprietária da caixa. A proprietária também recebe acesso para receber mensagens na caixa de correio. Apenas uma tarefa de cada vez pode ser proprietária ou receber mensagens de uma caixa de correio, mas esses direitos podem ser enviados para outras tarefas, se desejado.

Inicialmente, a caixa de correio tem uma fila vazia de mensagens. A medida que as mensagens são enviadas para a caixa de correio, elas são copiadas para a caixa. Todas as mensagens têm a mesma prioridade. O Mach garante que múltiplas mensagens do mesmo remetente sejam colocadas na fila em ordem FIFO (primeiro a entrar, primeiro a sair), mas não garante um ordenamento absoluto. Por exemplo, mensagens enviadas de dois remetentes distintos podem ser colocadas na fila em qualquer ordem.

As mensagens em si consistem em um cabeçalho de tamanho fixo, seguido por uma porção de dados de tamanho variável. O cabeçalho inclui o tamanho da mensagem e dois nomes de caixas de correio. Quando uma mensagem é enviada, um dos nomes de caixa de correio é a caixa para a qual a mensagem está sendo enviada. Muitas vezes, o thread de envio espera uma resposta; o nome da caixa de correio do remetente é passado para a tarefa receptora, que pode usá-la como um "endereço de retorno" para enviar mensagens de volta.

Aparte variável de uma mensagem é uma lista dos itens de dados digitados. Cada entrada na lista tem um tipo, tamanho e valor. O tipo dos objetos especificado na mensagem é importante, já que os objetos definidos pelo sistema operacional - tais como propriedade ou direitos de acesso em recepção, estados de tarefa c segmentos de memória - podem ser enviados nas mensagens.

As operações de envio e recepção em si são flexíveis. Por exemplo, quando uma mensagem é enviada para uma caixa de correio, esta pode estar cheia. Sc a caixa não estiver cheia, a mensagem c copiada para a caixa de correio e o thread de envio continua. Se a caixa estiver cheia, o thread de envio tem quatro opções:

1. Esperar indefinidamente até haver espaço na caixa de correio.
2. Esperar no máximo n milissegundos.
3. Não esperar, mas retornar imediatamente.
4. Temporariamente armazenar a mensagem em cache. Uma mensagem pode ser entregue ao sistema operacional para ser mantida por ele, mesmo que a caixa de correio para a qual está sendo enviada esteja cheia. Quando a mensagem puder ser colocada na caixa de correio, uma mensagem é enviada de volta ao remetente; apenas uma mensagem desse tipo para uma caixa de correio cheia pode estar pendente em qualquer momento, para determinado thread de envio.

A última opção é dedicada a tarefas de servidor, como um driver de impressora de linhas. Depois de terminar um pedido, essas tarefas podem precisar enviar uma resposta única à tarefa que solicitou o serviço, mas também devem continuar com outros pedidos de serviço, mesmo se a caixa de correio de resposta para um cliente estiver cheia.

A operação de recepção deve especificar a partir de que caixa de correio ou conjunto de caixas de correio determinada mensagem será recebida. Um conjunto de caixas de correio é uma coleção de caixas de correio, conforme declarado pela tarefa, que podem ser agrupadas e tratadas como uma caixa única para os objetivos da tarefa. Threads em uma tarefa só podem receber de uma caixa de correio ou conjunto de caixas de correio para a qual a tarefa tenha acesso de recepção. Uma chamada ao sistema port^status retoma o número de mensagens em determinada caixa de correio. A operação de recepção tenta receber de (1) qualquer caixa de correio cm um conjunto de caixas de correio ou (2) uma caixa de correio especifica (identificada). Se nenhuma mensagem estiver esperando para ser recebida, o thread receptor poderá esperar, esperar no máximo n milissegundos, ou não esperar.

O sistema Mach foi especialmente projetado para sistemas distribuídos, que são discutidos nos Capítulos 14 a 16, mas o Mach também é adequado para sistemas monoprocessador. O principal problema com sistemas de mensagens geralmente tem sido o baixo desempenho causado pela cópia da mensagem primeiro do remetente para a caixa de correio e depois da caixa de correio para o receptor. O sistema de mensagens do Mach tenta evitar as operações de cópia dupla usando técnicas de gerência de memória virtual (Capítulo 10). Basicamente, o Mach mapeia o espaço de endereçamento que contém a mensagem do remetente no espaço de endereçamento do receptor. A mensagem em si nunca é copiada. Essa técnica de gerência de mensagens melhora em muito o desempenho, mas só funciona para mensagens intra-sistema. O sistema operacional Mach é discutido no capítulo extra encontrado no nosso site na Web: (http://www.bell-labs.com/topic/books/os-book).

^f 4.5.7 Um exemplo: Windows NT

O sistema operacional Windows NT é um exemplo de projeto moderno que utiliza a modularidade para aumentar a funcionalidade e diminuir o tempo necessário para implementar novos recursos. O NT fornece suporte a vários ambientes operacionais ou *subsistemas,* com os quais os programas aplicativos se comunicam via um mecanismo de troca de mensagens. Os programas aplicativos podem ser considerados clientes do servidor de subsistema NT.

O recurso de troca de mensagens no NT é chamado de recurso de chamada de procedimento local (LPC - local procedure call). O LPC no Windows NT permite a comunicação entre dois processos que estão na mesma máquina. È semelhante ao mecanismo RPC padrão é que amplamente usado, mas é otimizado e específico ao NT. O Windows NT, como o Mach, utiliza um objeto porta para estabelecer e manter uma conexão entre dois processos. TCKJO cliente que chama um subsistema precisa de um canal de comunicação, que é fornecido por um objeto porta e nunca é herdado. O NT utiliza dois tipos de portas: portas de conexão e portas de comunicação. Na verdade, são os mesmos, mas recebem nomes diferentes de acordo com a forma em que são usados. As portas de conexão são objetos com nome (consulte o Capítulo 22 para obter mais informações sobre objetos do NT), visíveis a todos os processos; eles dão às aplicações uma forma de estabelecer um canal de comunicação. Essa comunicação funciona da seguinte maneira:

- O cliente abre um descritor para o objeto de porta de conexão do subsistema.
- O cliente envia um pedido de conexão.
- O servidor cria duas portas de comunicação privadas e retorna o descritor de um deles para o cliente.
- O cliente e o servidor usam o descritor da porta correspondente para enviar mensagens ou *callbacks* e ouvir respostas.

O NT utiliza três tipos de técnicas de troca de mensagens em uma porta que o cliente especifica quando estabelece o canal. O mais simples, que é usado para mensagens pequenas, utiliza a fila de mensagensda porta como um meio de armazenamento intermediário e copia a mensagem de um processo para outro. Com esse método, mensagens de até 256 bytes podem ser enviadas.

Se um cliente precisar enviar uma mensagem maior, ele passa a mensagem através de um objeto de seção (memória compartilhada). O cliente precisa decidir, quando estabelecer o canal, se precisará ou não enviar uma mensagem grande. Se o cliente determinar que deseja enviar mensagens grandes, ele solicitará que um objeto de seção seja criado. Da mesma forma, se o servidor decidir que as respostas serão grandes, ele criará um objeto de seção. Para que o objeto possa ser usado, uma mensagem pequena será enviada contendo um

ponteiro e informações de tamanho sobre o objeto de seção. Esse método é mais complicado do que o primeiro método, mas evita a cópia de dados. Em ambos os casos, um mecanismo de *callback* pode ser usado quando o cliente ou o servidor não puderem responder imediatamente a um pedido. O mecanismo de *callback* permite que eles efetuem o tratamento de mensagens de forma assíncrona.

Uma das desvantagens de usar a troca de mensagens para executar funções rudimentares, tais como funções gráficas, é que o desempenho talvez não seja tão bom quanto em um sistema sem troca de mensagens (ou seja, de memória compartilhada). Para aumentar o desempenho, o ambiente NT nativo (Win32) utiliza um terceiro método de troca de mensagens, chamado LPC rápido. Um cliente envia para a porta de conexão do servidor um pedido de conexão indicando que ele estará usando LPC rápido. O servidor estabelece um thread servidor dedicado para manipular os pedidos, um objeto de seção de 64 kilobytes e um objeto *event-pair*. A partir daí, as mensagens são passadas no objeto de seção, e a sincronização é realizada pelo objeto *event-pair*. Esse esquema elimina a cópia de mensagem, o custo de usar o objeto porta e o custo de determinar que thread de cliente está chamando, já que existe um thread para cada servidor thread de cliente. O kernel também dá ao thread dedicado preferência de escalonamento. A desvantagem é que esse método utiliza mais recursos do que os outros dois métodos. Como existe custo envolvido na passagem de mensagens, o NT poderá reunir várias mensagens em uma única mensagem para reduzir esse custo. Para obter maiores informações sobre o Windows NT, consulte o Capítulo 22.

4.6 • Resumo

Um processo é um programa em execução. A medida que um processo executa, ele muda de estado. O estado de um processo é definido pela atividade atual daquele processo. Cada processo pode estar em um dos seguintes estados: novo, pronto, cm execução, em espera ou encerrado. Cada processo é representado no sistema operacional pelo seu próprio bloco de controle de processo (PCB).

Um processo, quando não estiver executando, é colocado em alguma fila de espera. Existem duas classes principais de filas em um sistema operacional: filas de pedido de l/O e a fila de processos prontos. A fila de processos prontos contém todos os processos que estão prontos para executar e estão esperando pela CPU. Cada processo é representado por um PCB, e os PCBs podem ser unidos para formar uma fila de processos prontos. O escalonamento de longo prazo (de jobs) é a Seleção de processos que receberão permissão para disputar CPU. Normalmente, o escalonamento de longo prazo é altamente influenciado por considerações de alocação de recursos, especialmente a gerência de memória. O escalonamento de curto prazo (de CPU) é a Seleção de um processo da fila de processos prontos.

Os processos no sistema podem executar concorrentemente. Existem vários motivos para permitir a execução concorrente: compartilhamento de informações, velocidade de computação, modularidade e conveniência. A execução concorrente requer um mecanismo para a criação e exclusão de processos.

Os processos executando no sistema operacional podem ser independentes ou cooperativos. Os processos cooperativos devem ter meios de se comunicar entre si. Em princípio, existem dois esquemas de comunicação complementares: sistemas de memória compartilhada c de mensagens. O método de memória compartilhada requer que os processos em comunicação compartilhem algumas variáveis. Os processos devem trocar informações usando essas variáveis compartilhadas. Em um sistema de memória compartilhada, a responsabilidade de fornecer a comunicação está com os programadores da aplicação; o sistema operacional precisa fornecer apenas a memória compartilhada. O método do sistema de mensagens permite que os processos troquem mensagens entre si. A responsabilidade por fornecer a comunicação poderá estar com o próprio sistema operacional. Esses dois esquemas não são mutuamente exclusivos, e podem ser usados ao mesmo tempo em um único sistema operacional.

Podemos implementar a memória compartilhada ou troca de mensagens usando Java. Threads separados de controle podem ser criados c podem se comunicar usando qualquer um dos métodos.

• Exercícios

4.1 O MS-DOS não permitia o processamento concorrente. Discuta as três principais complicações que o processamento concorrente acrescenta a um sistema operacional.

4.2 Descreva as diferenças entre o escalonamento de curto, médio e longo prazo.

4.3 Num computador DECSYSTEM-20 possui vários conjuntos de registradores. Descreva as ações de uma troca de contexto, caso O novo contexto já esteja carregado em um dos conjuntos de registradores. O que mais precisa acontecer se o novo contexto estiver na memória em vez de em um conjunto de registradores, e se todos os conjuntos de registradores estiverem em uso?

4.4 Descreva as ações tomadas por um kernel para fazer a troca de contexto entre processos.

4.5 Quais são as vantagens e desvantagens de cada um dos itens a seguir? Considere os níveis de sistema e dos programadores.

a. Comunicação simétrica e assimétrica
b. Buffering automático e explícito
c. Enviar por copia e enviar por referência
d. Mensagens de tamanho fixo e variável

4.6 Considere o esquema de comunicação entre processos no qual são usadas caixas de correio.

a. Vamos supor que um processo *P* deseje esperar duas mensagens, uma da caixa A e outra da caixa *B.* Que sequência de send e receive deve ser executada?
b. Que sequência de sende receive/* deve executar se quiser esperar por uma mensagem da caixa *A* ou da caixa B (ou de ambas)?
c. Uma operação de recei ve faz o processo esperar até que a caixa de correio não esteja vazia. Crie um esquema que permita que um processo espere até que a caixa de correio fique vazia ou explique por que esse esquema não pode existir.

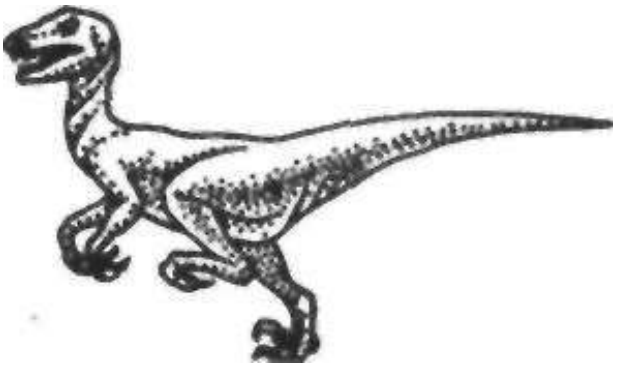
Notas bibliográficas

O tópico "comunicação entre processos" foi discutido por Brinch Hansen [1970} com relação ao sistema RC 4000. Schlichting e Schneidcr 11982) discutiram primitivas de troca de mensagens assíncrona. O recurso IPC implementado no nível de usuário foi descrito por liershad e colegas [1990].

Detalhes da comunicação entre processos nos sistemas UNIX foram apresentados por Gray (*1997],* Barrera |1991| e Vahalia 11996] apresentaram a comunicação entre processos no sistema Mach. Solomon [1998] delineou a comunicação entre processos no Windows NT.

As discussões relativas à implementação de RPCs foram apresentadas por Birrell e Nelson [1984]. O projetode um mecanismo de RPCconfiável foi apresentado por Shrivastava e Panzieri [1982]. Um estudo sobre RPCs foi apresentada por Tay e Ananda [19901- Stankovic (1982] eStaunstrup (1982] discutiram as chamadas de procedimento *versus* a comunicação por troca de mensagens.

Capítulo 4



PROCESSOS

Os primeiros sistemas de computação só permitiam que um programa fosse executado de cada vez. Esse programa tinha controle completo do sistema e acesso a todos os recursos do sistema. Os sistemas de computação modernos permitem que múltiplos programas sejam carregados na memória e executados de forma concorrente. Essa evolução exigiu maior controle e mais comparti mental ização dos vários programas. Essas necessidades resultaram na noção de um processo, que é um programa em execução. Um processo é a unidade de trabalho cm um sistema moderno de tempo compartilhado.

Quanto mais complexo for um sistema operacional, mais benefícios aos usuários são esperados. Embora sua principal preocupação seja a execução de programas de usuário, ele também precisa cuidar das várias tarefas de sistema que são mais bem executadas fora do kernel propriamente dito. Um sistema, portanto, consiste em uma coleção de processos: processos de sistema operacional que executam código do sistema e processos de usuário que executam código de usuário. Todos esses processos podem executar ao mesmo tempo, sendo a CPU (ou CPUs) multiplexada(s) entre eles. Ao alternar a CPU entre os processos, o sistema operacional pode tornar o computador mais produtivo.

4.1 • Conceito de processo

Um obstáculo à discussão de sistemas operacionais é que existe dificuldade em denominar todas as atividades da CPU. Um sistema em batch executa *jobs,* enquanto um sistema de tempo compartilhado executa *programas de usuário* ou *tarefas.* Mesmo em um sistema monousuário, como o Microsoft Windows ou Macintosh OS, um usuário pode executar vários programas de uma vez: um processador de textos, um navegador Web e um pacote de e-mail. Mesmo se o usuário só puder executar um programa de cada vez, o sistema operacional poderá precisar dar suporte a suas próprias atividades internas programadas, como gerência de memória. Em muitos aspectos, todas essas atividades são semelhantes, por isso chamamos todas de *processos.*

Os termos *job* e *processo* são usados quase que indistintamente neste livro. Apesar de pessoalmente preferirmos usar o termo *processo** boa parte da teoria e terminologia sobre sistemas operacionais foi desenvolvida em uma época na qual a principal atividade dos sistemas operacionais era o processamento de jobs. Seria erróneo evitar o uso de termos comumente aceitos que incluem a palavra *job* (como *escalonamento de jobs)* simplesmente porque *processo* suplantou *job.*

4.1.1 O processo

Informalmente, um processo é um programa em execução. Um processo é mais do que o código do programa, que às vezes *è* chamado seção de texto. Também inclui a atividade corrente, conforme representado pelo valor do contador do programa e 0 conteúdo dos registradores do processador. Um processo geralmente inclui a pilha de processo, que contém dados temporários (como parâmetros de métodos, endereços de retorno e variáveis locais) c uma seção de dados, que contém variáveis globais.

Enfatizamos que uni programa por si só não é um processo; um programa é uma entidade *passivo,* como o conteúdo de um arquivo armazenado em disco, enquanto um processo é uma entidade *atwd,* com um contador de programa especificando a próxima instrução a ser executada e um conjunto de recursos associados.

Embora dois processos possam ser associados com o mesmo programa, são considerados duas sequências separadas de execução. Por exemplo, vários usuários podem estar executando cópias diferentes do programa de correio ou o mesmo usuário pode chamar muitas cópias do programa editor. Cada uma dessas atividades é um processo separado e, embora as seções de texto sejam equivalentes, as seçóes de dados variam. Também *é* comum ter um processo que produza muitos processos durante sua execução. Essas questões são tratadas na Seção 4.4.

4.1.2 Estado do processo

A medida que o processo executa, ele muda de estado. O estado de uni processo é definido em parte pela atividade atual desse processo. Cada processo pode estar cm um dos seguintes estados:

- Novo: o processo está sendo criado.
- Em execução: as instruções estão sendo executadas.
- Em espera: o processo está esperando a ocorrência de algum evento {como conclusão de operação de I/O ou recepção de um sinal).
- Pronto: o processo está esperando para ser atribuído a um processador.
- Encerrado: o processo terminou sua execução.

Esses nomes são arbitrários, e variam dependendo do sistema operacional. Os estados que representam, no entanto, são encontrados em todos os sistemas. Certos sistemas operacionais também delineiam estados de processo de forma mais detalhada. Ê importante observar que apenas um processo pode estar *em execução* em qualquer processador a qualquer instante. No entanto, muitos processos podem estar *prontos* e *em espera** O diagrama de estado correspondente a esses estados está representado na Figura 4.1.

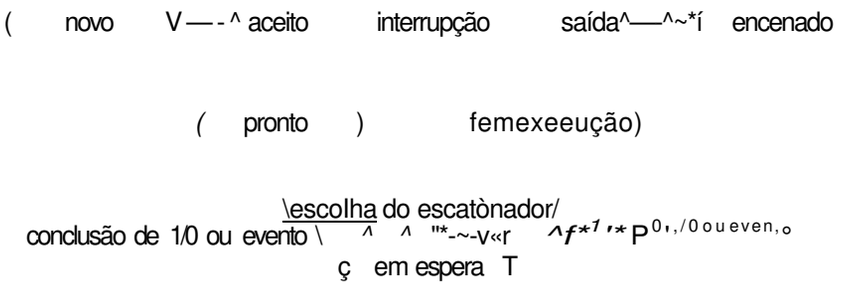


Figura 4.1 Diagrama de estado de um processo.

4.1.3 Bloco de controle de processo

Cada processo é representado no sistema operacional por um bloco de controle de processo (PCB - process control block), também chamado de bloco de controle de tarefa. A Figura 4.2 mostra um PCB. Ele contém muitas informações associadas a um processo específico, incluindo:

- *listado do processo:* O estado pode ser novo, pronto, em execução, em espera, suspenso, e assim por diante.
- *Contador do programai* O contador indica o endereço da próxima instrução a ser executada para esse processo.
- *Registradores de CPU:* Os registradores variam em número e tipo, dependendo da arquitetura do computador. Incluem acumuladores, registradores de índice, ponteiros de pilha e registradores de uso geral, além de informações de código de condição. Juntamente com o contador do programa, essas informações de estado devem ser salvas quando ocorre uma interrupção, para permitir que O processo continue corretamente depois disso (Figura 4..Í).

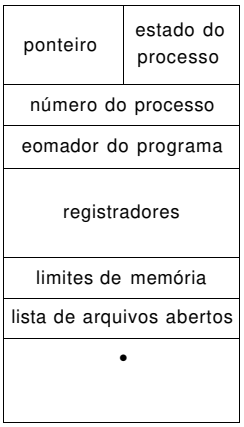


Figura 4.2 Bloco de controle de processo (PCB).

- *In for/nações de escalonamento de CPU:* Essas informações incluem prioridade de processo, ponteiros para filas de escalonamento e quaisquer outros parâmetros de escalonamento. (O Capítulo 6 descreve o escalonamento de processos.)
- *Informações de gerência de memória:* Essas informações podem incluir dados como o valor dos registradores de base e limite, as tabelas de páginas ou as tabelas de segmentos, dependendo do sistema de memória usado pelo sistema operacional (Capítulo 9).
- *Informações de contabilização:* Essas informações incluem a quantidade de CPU e o tempo real usados, limites de tempo, números de contas, números de jobs ou processos etc.
- *Informações de status de HO:* As informações incluem a lista de dispositivos de l/O alocados para este processo, uma lista de arquivos abertos etc.

O PCB serve simplesmente como repositório de informações que podem variar de processo a processo.

4.1.4 Threads

O modelo de processo discutido até agora considerava implicitamente que um processo é um programa que realiza um único fluxo de execução. Por exemplo, se um processo está executando um programa processador de textos, existe um único fluxo de instruções sendo executado. Esse fluxo único de controle só permite que o processo execute uma tarefa de cada vez. O usuário não pode digitar caracteres e passar o corretor ortográ-

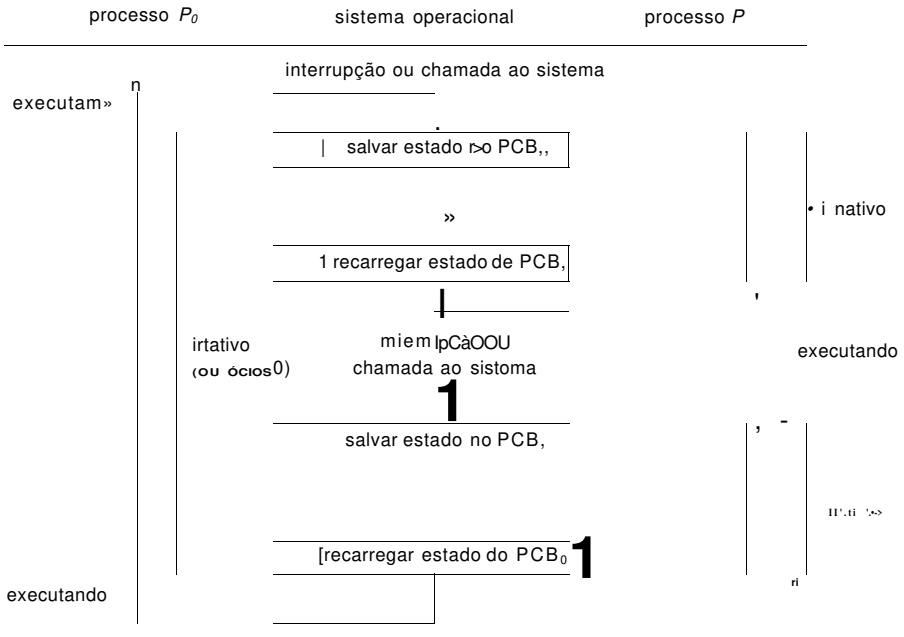


Figura 4.3 Diagrama mostrando a CPU alternando entre processos.

fico ao mesmo tempo no mesmo processo. Muitos sistemas operacionais modernos estenderam o conceito de processo para permitir que um processo tenha múltiplos fluxos de execução, ou threads. Assim, o processo pode executar mais de uma tarefa de cada vez. O Capítulo 5 explora processos com múltiplos threads.

4.2 • Escalonamento de processos

O objetivo da multiprogramação é ter processos cm execução o tempo todo, para maximizar a utilização de CPU. O objetivo do tempo compartilhado é alternar a CPU entre processos de forma tão frequente que os usuários possam interagir com cada programa durante sua execução. Para um sistema uniprocessador, nunca haverá mais de um processo cm execução. Sc houver mais processos, os demais terão de esperar até que a CPU esteja liberada e possa ser reescalonada.

4.2.1 Filas de escalonamento

À medida que os processos entram no sistema, são colocados em uma fila de jobs. Essa fila consiste em todos os processos do sistema. Os processos que estão residindo na memória principal e estão prontos e esperando para executar são mantidos em uma lista chamada fila de processos prontos *(ready queué).* Essa fila geralmente é armazenada como uma lista encadeada. Um cabeçalho de fila de processos prontos contém ponteiros ao primeiro e Último PCBs na lista. Estendemos cada PCB para incluir um campo de ponteiro apontando para o próximo PCB na fila de processos prontos.

Existem também outras filas no sistema. QuandoaCPUéalocadaa um processo, ele executa durante um tempo e *é* encerrado, interrompido ou espera pela ocorrência de determinado evento, como a conclusão de um pedido de l/O, por exemplo. No caso de um pedido de I/O, esse pedido pode ser para uma unidade de fita dedicada ou para um dispositivo compartilhado, como um disco. Como existem muitos processos no sistema, o disco pode estar ocupado com o pedido de l/O de algum outro processo. O processo, portanto, pode ter de esperar pelo disco. A lista de processos esperando por determinado dispositivo de l/O é chamada fila de dispositivo. Cada dispositivo tem sua própria fila de dispositivo (Figura 4.4).

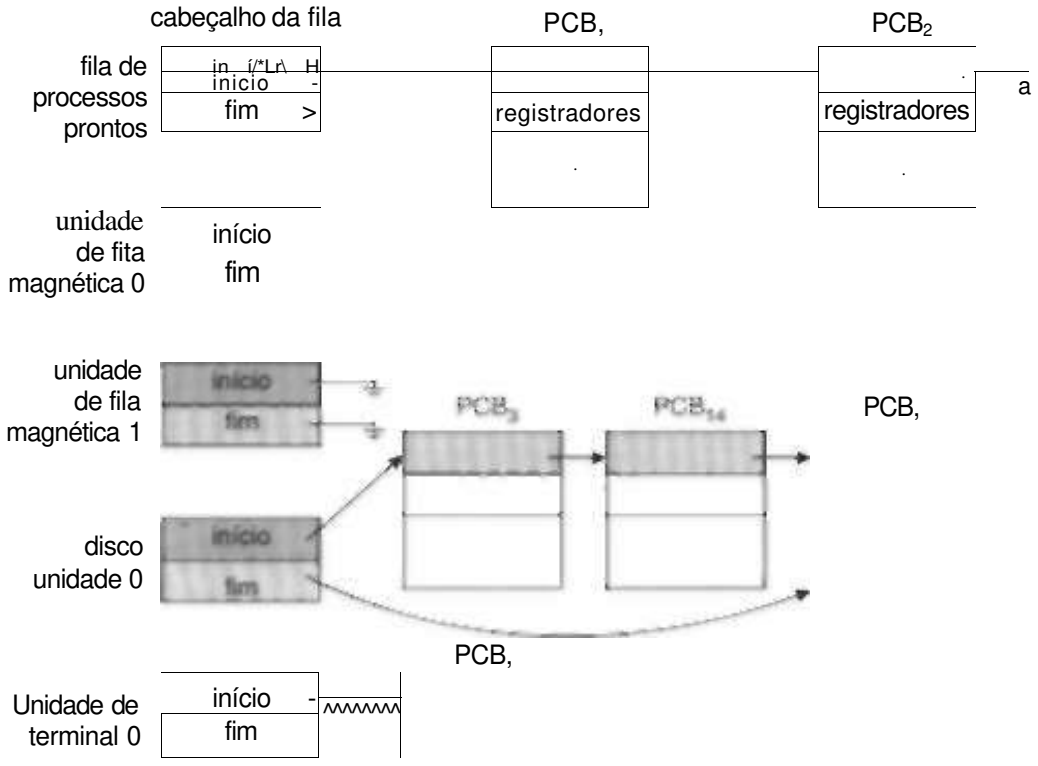


Figura 4.4 A fila de processos prontos e várias filas de dispositivos de l/O.

Uma representação comum pari uma discussão sobre escalonamento de processos é um diagrama de filas, como o da Figura 4.5. Cada caixa retangular representa uma fila. Dois tipos de fila estão presentes: a fila de processos prontos e um conjunto de filas de dispositivo. Os círculos representam os recursos que servem as filas e as setas indicam o fluxo de processos no sistema.

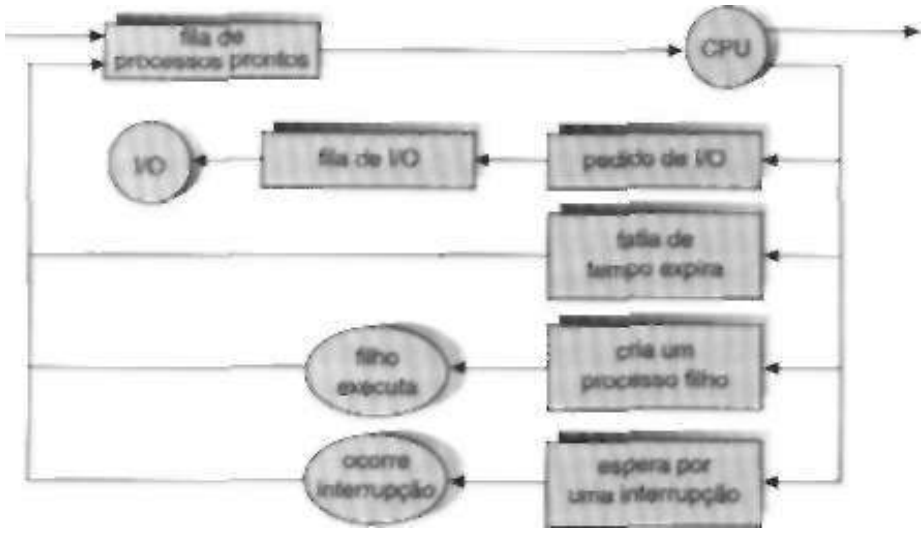


Figura 4.5 Representação do diagrama de filas do escalonamento de processos.

Um processo novo é colocado inicialmente na fila de processos prontos. Ele espera na fila até ser selecionado para execução ou ser submetido (dispatched). Depois que o processo recebe a CPU e está cm execução, um dos vários eventos a seguir poderá ocorrer:

- O processo pode emitir um pedido de I/O e ser colocado em uma fila de l/O.
- O processo pode criar um novo subprocesso c esperar seu término.
- O processo pode ser removido à força da CPU, como resultado de uma interrupção e ser colocado de volta na fila de processos prontos.

Nos dois primeiros casos, o processo acaba alternando do estado de espera para o estado de pronto e, em seguida, é colocado de volta na fila de processos prontos. Um processo continua o seu ciclo até terminar e, nesse ponto, é removido de todas as filas, com seu PCB e recursos sendo desalocados.

4.2.2 Escalonadores

Um processo migra entre as várias filas de escalonamento ao longo de toda sua vida. O sistema operacional deve selecionar, para fins de escalonamento, os processos dessas filas de alguma forma. O processo de Seleção *ê* executado pelo escalonador *(scheàuler)* adequado.

Em um sistema cm batch, existem geralmente mais processos submetidos do que processos que podem ser executados imediatamente. Esses processos são colocados em uni spool em um dispositivo de armazenamento de massa (geralmente um disco), onde são mantidos para execução posterior. O escalonador de longo prazo, ou o escalonador de jobs, seleciona processos desse conjunto c os carrega na memória para execução. O escalonador de curto prazo, ou o escalonador de CPU, seleciona dentre os processos que estão prontos para execução e aloca a CPU a um deles.

A principal distinção entre esses dois escalonadores é a frequência da sua execução. O escalonador de curto prazo deve selecionar um novo processo para a CPU com frequência. Um processo pode executar por apenas alguns milissegundos antes de esperar por um pedido de I/O. Em geral, o escalonador de curto prazo executa pelo menos uma vez a cada 100 milissegundos. Devido à breve duração de tempo entre as execuções, o escalonador de curto prazo deve ser rápido. Se levar 10 milissegundos para decidir executar um processo por 100 milissegundos, então 10/(100 + 10) = 9% da CPU está sendo usado (desperdiçado) simplesmente para escalonar o trabalho.

O cscalonador de longo prazo, por outro lado, executa com muito menos frequência. Pode haver um intervalo de minutos entre a criação de novos processos no sistema. O escalonador de longo prazo controla o grau de multiprogramação (o número de processos na memória). Sc o grau de mui ti programação for estável, a taxa media de criação de processos deve ser igual a taxa média de partida de processos que saem do sistema. Assim, o escalonador de longo prazo pode precisar ser chamado apenas quando um processo sair do sistema. Devido ao intervalo maior entre as execuções, o escalonador de longo prazo pode levar mais tempo para decidir que processos devem ser selecionados para execução.

E importante que o escalonador de longo prazo faça uma Seleção cuidadosa. Em geral, a maioria dos processos podem ser descritos como limitados por I/O ou limitados pela CPU. Um processo limitado por I/O passa mais tempo realizando operações de I/O do que efetuando cálculos. Um processo limitado pela CPU, por outro lado. gera pedidos de I/O com pouca frequência, usando mais o seu tempo na computação. E importante que o escalonador de longo prazo selecione uma boa combinação de processos incluindo processos limitados por I/O e pela CPU. Sc todos os processos forem limitados por I/O, a fila de processos prontos quase sempre estará vazia e o escalonador de curto prazo terá pouco trabalho. Se todos os processos forem limitados pela CPU, a fila de espera de I/O quase sempre estará vazia, os dispositivos ficarão sem uso e mais uma vez o sistema ficará desequilibrado. O sistema com o melhor desempenho terá uma combinação de processos limitados pela CPU e por I/O.

Em alguns sistemas, o cscalonador de longo prazo pode estar ausente ou ser mínimo. Por exemplo, sistemas de tempo compartilhado, como o UNIX, geralmente não têm um escalonador de longo prazo, mas simplesmente colocam todo novo processo na memória para o escalonador de curto prazo. A estabilidade desses sistemas depende de uma limitação física (como o número de terminais disponíveis) ou da natureza de auto-ajuste dos usuários humanos. Se o desempenho cair para níveis inaceitáveis, alguns usuários simplesmente vão desistir.

Alguns sistemas operacionais, como os de tempo compartilhado, podem introduzir um nível intermediário adicional de escalonamento. O escalonador de médio prazo está representado na Figura 4.6. A principal ideia por trás de um cscalonador de médio prazo *é* que às vezes pode ser vantajoso remover processos da memória (e da disputa ativa por CPU) e, assim, reduzir o grau de multiprogramação. Em algum momento posterior, o processo pode ser introduzido novamente na memória e sua execução pode ser retomada do ponto onde parou. Esse esquema é chamado de *swapping* (troca). O escalonador de médio prazo realiza as operações de swapping. O swapping pode ser necessário para melhorar a combinação de processos ou porque uma mudança nos requisitos de memória comprometeu a memória disponível, exigindo a liberação de memória. O Capítulo 9 discute o conceito de swapping.

Figura 4.6 Acréscimo do escalonamento de médio prazo ao diagrama de filas.

4.2.3 Troca de contexto

Alternar a CPU para outro processo requer salvar o estado do processo antigo e carregar o estado salvo do novo processo. Essa tarefa é chamada de troca de contexto. O contexto de um processo é representado no PCB de um processo; inclui o valor dos registradores de CPU, o estado do processo (consulte a Figura 4.1) e as informações de gerência de memória. Quando ocorre uma troca de contexto, qjcerncl salva o contexto do processo antigo em seu PCB e carrega o çontextosàlvo do novo pTocesso escolhido para execução. O tempo de troca de contexto é puro *overbead,* já que o sistema não efetua trabalho útil durante o processo de troca.

Sua velocidade varia de máquina a máquina dependendo da velocidade da memória, do número de registradores que devem ser copiados c da existência de instruções especiais (como uma única instrução para carregar ou armazenar todos os registradores). Velocidades típicas estão entre 1 a 1000 microssegundos.

Os tempos de troca de contexto são altamente dependentes do suporte de hardware. Por exemplo, alguns processadores (como o Sun UltraSPARC) fornecem vários conjuntos de registradores. Uma troca de contexto simplesmente inclui mudar o ponteiro para o conjunto de registradores atual. E claro que se houver mais processos ativos do que conjuntos de registradores, o sistema vai copiar os dados do registrador de e para a memória como antes. Além disso, quanto mais complexo o sistema operacional, mais trabalho deve ser realizado durante uma troca de contexto. Como será discutido no Capítulo 9, técnicas avançadas de gerência de memória podem exigir que dados extras sejam trocados com cada contexto. Por exemplo, o espaço de endereçamento do processo atual deve ser preservado à medida que o espaço da próxima tarefa é preparado para uso. Como o espaço de endereçamento é preservado e quanto trabalho será necessário para preservá-lo dependerão do método de gerência de memória do sistema operacional. Como será discutido no Capítulo 5, a troca de contexto tomou-se um gargalo de desempenho de tal ordem que os programadores estão utilizando novas estruturas (threads) para evitá-la sempre que possível.

4.3 • Operações nos processos

Os processos no sistema podem executar de forma concorrente e devem ser criados e excluídos de forma dinâmica. Assim, o sistema operacional deve fornecer um mecanismo para a criação e término de processos.

4.3.1(Criação de processo^)

Um processo pode criar vários novos processos através de uma chamada ao sistema para a criação de processo, durante sua execução. O processo criador é chamado **de processo pai, enquanto** os **novos** processos^são chamados de filhos desse processo. Cada um dos novos processos, por sua vez, pode criar outros processos, formando uma árvore de processos (Figura 4.7).

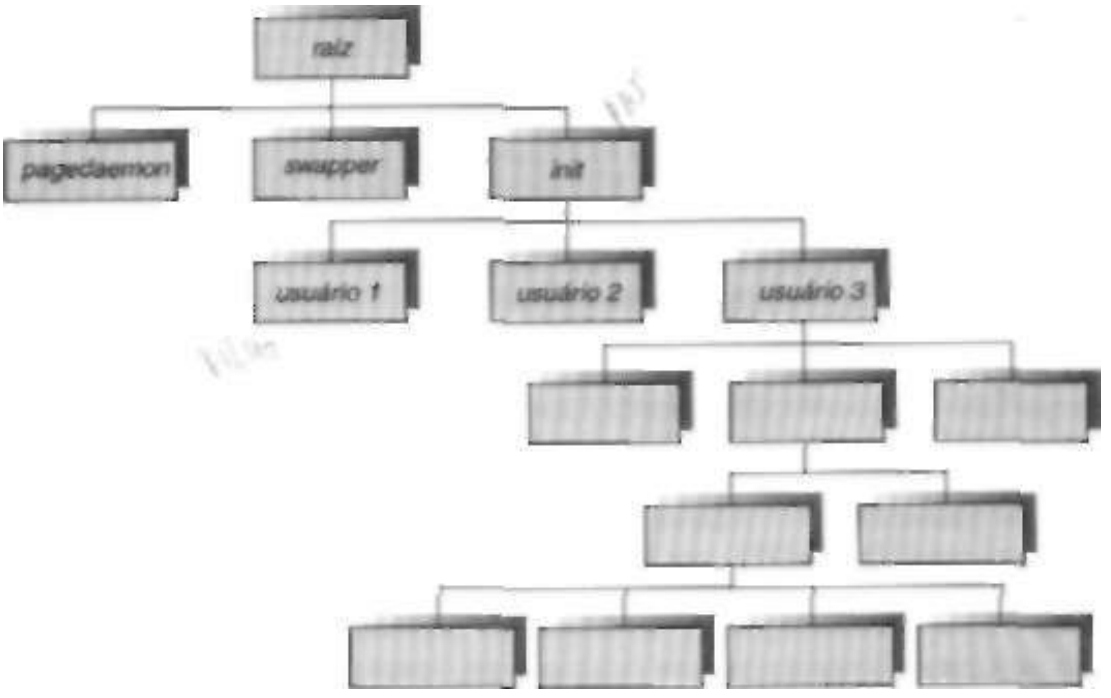


Figura 4.7 Uma árvore de processos cm um sistema UNIX típico.

Em geral, um processo precisará de determinados recursos (tempo de CPU, memória, arquivos, dispositivos de I/O) para realizar sua tarefa. Quando um processo cria um subprocesso, esse subprocesso pode ser capaz de obter seus recursos diretamente do sistema operacional ou pode ser limitado a um subconjunto dos recursos do processo pai. O pai pode ter de dividir seus recursos entre os filhos, ou talvez possa compartilhar

alguns recursos (como memória ou arquivos) entre vários filhos. Restringir um processo filho a um subconjunto dos recursos do pai evita que algum processo sobrecarregue o sistema criando subprocessos demais.

Além dos vários recursos físicos e lógicos que um processo obtém quando é criado, os dados de inicialização (entrada) podem ser passados pelo processo pai para o processo filho. Por exemplo, considere um processo cuja função seja exibir o status de um arquivo, digamos Fí, na tela de um terminal. Quando ele for criado, receberá como entrada do seu processo pai o nome de arquivo *F1* e executará usando esse dado para obter as informações desejadas. Também pode receber o nome do dispositivo de saída. Alguns sistemas operacionais passam recursos para os processos filhos. Km um sistema como esses, o novo processo pode obter dois arquivos abertos, Fí e o dispositivo do terminal, e pode apenas precisar transferir o dado entre os dois.

Quando um processo cria um novo processo, existem duas possibilidades em termos de execução:

1, O pai continua a executar de forma concorrente com os filhos.

2. O pai espera até que alguns ou todos os filhos tenham terminado.

Existem também duas possibilidades em termos de espaço de endereçamento do novo processo:

1. O processo filho é uma duplicata do processo pai.
2. O processo filho tem um programa carregado nele.

Para ilustrar essas implementações diferentes, vamos considerar o sistema operacional UNIX. No UNIX, cada processo é identificado por seu identificador de processo, que é um inteiro único. Um novo processo é criado pela chamada ao sistema f ork. O novo processo consiste em uma cópia do espaço de endereçamento do processo original. Esse mecanismo permite que o processo pai se comunique facilmente com o processo filho. Ambos os processos (o pai e o filho) continuam a execução da instrução após o f ork com uma diferença: o código de retorno para f ork é zero para o novo processo (filho), enquanto o identificador de processo filho (diferente de zero) é retornado ao pai.

Geralmente, a chamada ao sistema execlp é usada após uma chamada fork por um dos dois processos, para substituir o espaço de memória do processo por um novo programa. A chamada execl p carrega um arquivo binário na memória (destruindo a imagem na memória do programa que contém a chamada ao sistema execl p) e inicia sua execução. Dessa maneira, os dois processos podem se comunicar e, em seguida, continuar cada um o seu caminho. O pai pode criar mais filhos ou, se não tiver nada a fazer enquanto o filho executa, ele poderá emitir uma chamada ao sistema wai t para sair da fila de processos prontos até o término do filho. O programa C mostrado na Figura 4.8 ilustra as chamadas ao sistema UNIX descritas anteriormente. O pai cria um processo filho usando a chamada fork( ). Agora temos dois processos diferentes executando uma cópia do mesmo programa. O valor de pid para o processo filho é zero; o valor para o processo pai é um valor inteiro maior do que zero. O processo filho sobrepõe seu espaço de endereçamento com o comando UNIX /bi n/l s (usado para obter uma listagem de diretórios) utilizando a chamada ao sistema execl p ( ). O pai espera que o processo filho termine, usando a chamada wai t( ). Quando o processo filho termina, o processo pai retoma a partir da chamada wait( ), onde conclui usando a chamada ao sistema exit( ).

O sistema operacional VMS da DEC, por outro lado, cria um novo processo, carrega um programa especificado nesse processo c começa a sua execução. O sistema operacional Microsoft Windows NT suporta ambos os modelos: o espaço de endereçamento do pai pode ser duplicado, ou o pai pode especificar o nome de um programa para que o sistema operacional o carregue no espaço de endereçamento do novo processo.

4.3.2 Término do processo

Um processo termina quando acaba de executar sua instrução final e pede que o sistema operacional o exclua usando a chamada ao sistema exi t. Nesse ponto, o processo pode retomar dados (saída) ao seu processo pai (via chamada wai t). Todos os recursos do processo - incluindo memória física e virtual, arquivos abertos e buffers de I/O - são desalocados pelo sistema operacional.

```
íinclude <stdio.h>

void main(int arge, char *argv[ ])
(
int pi d;

    /•bifurcação em outro processo */
    pid •fagj J; *          | ,

    if {pid < 0) { /• ocorreu um erro */
      fprintffstderr, "Fork Falhou');
      exit(-l);
    I
   ^else if (pid •• 0) { /" processo filho •/
 (4);execlp(7b1n/1s\"ls\NULL);

    else ( /* processo pai •/
      /* pai espçrarS a conclusão do filho */
      Mlt(NULLh
      prtntfCFilho concluiu");
    vexit(0>;
      )
```

Figura 4.8 Programa C criando uma nova imagem de processo.

Existem circunstâncias adicionais nas quais ocorre o término. Um processo pode causar o término de outro processo via uma chamada ao sistema adequada (por exemplo, abort). Geralmente, essa chamada pode ser feita apenas pelo pai do processo que deverá ser terminado. Caso contrário, os usuários podem interromper arbitrariamente os jobs uns dos outros. Observe que um pai precisa conhecer as identidades de seus filhos. Assim, quando um processo cria um novo processo, a identidade do processo recém-criado é passada para o pai.

Um pai pode terminar a execução de um de seus filhos por vários motivos, a saber:

- O filho excedeu seu uso de alguns dos recursos que foram alocados.
- A tarefa atribuída ao filho não é mais exigida.
- O pai está saindo, c o sistema operacional não permite que um filho continue se seu pai terminar.

Para determinar o primeiro caso, o pai deve ter um mecanismo para inspecionar o estado de seus filhos.

Muitos sistemas, incluindo o VMS, não permitem a existência de um filho se seu pai tiver terminado. Em tais sistemas, se um processo terminar (de forma normal ou anormal), todos os filhos também devem ser terminados. Esse fenómeno, chamado de término cm cascata, é normalmente iniciado pelo sistema operacional.

Para ilustrar a execução <* o término de processos, consideramos que, no UNIX, podemos terminar um processo usando a chamada ao sistema exi t; seu processo pai pode esperar pelo término de um processo filho usando a chamada wai t. A chamada wai t retorna o identificador de processo de um filho terminado, de modo que o pai possa dizer qual dos seus possíveis muitos filhos terminou. Se o pai terminar, no entanto, todos os seus filhos recebem como novo pai o processo *init.* Assim, os filhos ainda têm um pai para colctar seu status e estatísticas de execução.

4.4 • Processos cooperativos

Os processos concorrentes executando no sistema operacional podem ser processos independentes ou processos cooperativos. Um processo é independente se não puder afetar ou ser afetado pelos outros processos executando no sistema. Claramente, qualquer processo que não compartilhe dados (temporários ou persistentes) com outro processo é independente. Por outro lado, um processo é cooperativo se puder afetar ou ser afetado por outro processo executando no sistema. Claramente, qualquer processo que compartilhe dados com outros processos é um processo cooperativo.

Existem vários motivos para fornecer um ambiente que permita a cooperação entre processos:

- *Compartilhamento de informações:* Como vários usuários podem estar interessados na mesma informação (por exemplo, um arquivo compartilhado), é preciso fornecer um ambiente para permitir o acesso concorrente a esses tipos de recursos.
- *Velocidade de computação:* Se queremos que determinada tarefa execute mais rápido, é preciso quebrá-la em subtarefas, cada qual sendo executada cm paralelo com as demais. Observe que o aumento na velocidade só pode ser alcançado se o computador tiver múltiplos elementos de processamento (tais como CPUs ou canais de I/O).
- *Modularidade:* Talvez queiramos construir o sistema de forma modular, dividindo as funções do sistema em processos ou ihreads separados, como discutido no Capítulo 3.
- *Conveniência:* Mesmo um usuário individual pode ter muitas tarefas nas quais trabalhar cm determinado momento. Por exemplo, um usuário pode estar editando, imprimindo e compilando cm paralelo.

A execução concorrente que requer a cooperação entre os processos exige mecanismos para permitir que os processos se comuniquem entre si (Seção 4.5) c sincronizem suas ações (Capítulo 7).

Para ilustrar o conceito de processos cooperativos, vamos considerar o problema do produtor-consumidor, que é um paradigma comum para os processos cooperativos. Um processo produtor produz informações que são consumidas por um processo consumidor. Por exemplo, um programa de impressão produz caracteres que são consumidos pelo driver de impressão. Um compilador pode produzir código assembly, que é consumido por um montador. O montador, por sua vez, pode produzir módulos objeto, que são consumidos pelo utilitário de carga.

Para permitir que os processos de produtor e consumidor executem de forma concorrente, *é* preciso ter disponível um buffer de itens que pode ser preenchido pelo produtor e esvaziado pelo consumidor. Um produtor pode produzir um item enquanto um consumidor está consumindo outro item. O produtor e o consumidor devem ser sincronizados, de modo que o consumidor não tente consumir um item que ainda não tenha sido produzido. Nessa situação, o consumidor deve esperar até que um item seja produzido.

O problema do produtor-consumidor de buffer náo-limitado não impõe limite prático no tamanho do buffer. O consumidor talvez tenha de esperar por novos itens, mas o produtor sempre poderá produzir novos itens. O problema do produtor-consumidor de buffer limitado assume que existe um tamanho fixo de buffer. Nesse caso, o consumidor deve esperar se o buffer estiver vazio e o produtor deve esperar se o buffer estiver cheio.

O buffer pode ser fornecido pelo sistema operacional através do uso de um recurso de comunicação entre processos (IPC, *interprocess communication)* (Seção 4.5), ou explicitamente codificado pelo programador da aplicação com o uso de memória compartilhada. Vamos ilustrar uma solução de memória compartilhada para o problema de buffer limitado. Os processos produtor e consumidor compartilham uma instância da classe BoundedBuffer (Figura 4.9).

O buffer compartilhado é implementado como um vetor circular com dois ponteiros lógicos: in e out. A variável i n aponta para a próxima posição livre no buffer; out aponta para a primeira posição cheia no buffer. counté o número de itens presentes no buffer no momento. O buffer está vazio quando count == 0 e está cheio quando count •- BUFFERSIZE. O processo produtor chama o método enter( ) (Figura 4.10) quando deseja inserir um item no buffer e o consumidor chama o método removei) (Figura 4.11) quando deseja consumir um item do buffer. Observe que o produtor e o consumidor ficarão bloqueados no laço whi 1 e se o buffer estiver, respectivamente, cheio ou vazio.

No Capítulo 7, discutimos como a sincronização entre processos cooperativos pode ser implementada de forma eficiente em um ambiente de memória compartilhada.

4.5 • Comunicação entre processos

Na Seção 4.4, mostramos como os processos cooperativos podem se comunicar em um ambiente de memória compartilhada. O esquema requer que esses processos compartilhem um poot de buffers comum e que o código para implementar o buffer seja explicitamente escrito pelo programador da aplicação. Outra forma de obter o mesmo efeito é fornecer o meio para os processos cooperativos se comunicarem entre si através de um recurso de comunicação entre processos (IPC).

```
import java.util.";

public class BoundedBuffer
|
    public BoundedBuffer( ) {
       // buffer esta* inicialmente vazio
       count • 0;
       In * 0;
       out • 0;

       buffer • new Object[BUFFER SIZE];
    )

    // produtor chana este método
    public void enter(Object item) {
    // Figura 4.10
    )
    // Consumidor chama este método
    public Object removei ) I
    // Figura 4.11
    I

    public static final int NAP_TIHE = 5;
    private static final int BUFFER_S1ZE • 5;
    private volatile int count;
    private int in; // aponta para a próxima posição livre
    private int out; // aponta para a próxima posição cheia
    private ObjectE ] buffer;
1
```

Figura 4.9 Solução de memória compartilhada para o problema do produtor-consumidor.

```
public void enter(0bject item) {
    while (count •- BUFFER_SIZE)
          ; // náo faz nada

    // adiciona um item ao buffer
    ••count;
    buffer[in] = item;
    in - (in + 1) % BUFFER_SIZE;

    if (count == BUFFERJIZE)
          System.out.println ("Producer Entered " •
              item * "Buffer FULL');
    else
          System.out.println ("Producer Entered " •
              item • " Buffer Size = " • count);
)
```

Figura 4.10 O método enter( ).

O IPC fornece um mecanismo para permitir que os processos comuniquem e sincronizem suas ações sem compartilhar o mesmo espaço de endereçamento. O IPC é particularmente útil em um ambiente distribuído no qual os processos em comunicação podem residir em diferentes computadores conectados via rede. Um exemplo é um programa de batc-papo (chat) usado na World Wide Web. No Capítulo 15, analisamos a computação distribuída usando Java.

A melhor forma de fornecer IPC é através do sistema de troca de mensagens, e os sistemas de mensagem podem ser definidos de várias formas diferentes. Vamos analisar diferentes aspectos de projeto e apresentamos uma solução Java para o problema do produtor-consumidor que utiliza troca de mensagens.

4.5.1 Sistema de troca de mensagens

A função de um sistema de mensagens é permitir que os processos se comuniquem entre si sem necessidade de recorrer a dados compartilhados. Um recurso de IPC fornece pelo menos duas operações: *senâ(message)* e *recei ve(message).*

```
public Object removei ) {
    Object item;

    while (count •• 0)
          ; // nào faz nada
    // remove um item do buffer
    —count;
    item = buffer[outJ;
    out - (out + 1) % BUFFER_SIZE;

    if (count « 0)
          System.out.println("Consumer Consumed " +
              item * " Buffer EMPTY-);
    else
          System.out.printlnfConsumer Consumed " *
              item • " Buffer Size • " + count);

    return item;
Í
```

Figura 4. II O método removei).

As mensagens enviadas por um processo podem ser de tamanho fixo ou variável. Se apenas mensagens de tamanho fixo puderem ser enviadas, a implementação no nível do sistema será simples. Essa restrição, no entanto, torna a tarefa de programação mais difícil. Por outro lado, as mensagens de tamanho variável exigem uma implementação mais complexa no nível do sistema, mas a tarefa de programação se torna mais simples.

Se os processos Pe *Q* quiserem se comunicar, deverão enviar c receber mensagens um do outro; um canal de comunicação deve existir entre eles. Esse canal pode ser implementado de várias formas. Estamos preocupados não com a implementação física do canal (como memória compartilhada, barramento de hardware ou rede, que são tratados no Capitulo 14), mas sim com sua implementação lógica. Existem vários métodos para implementar logicamente um canal e as operações de send/receive:

- Comunicação direta ou indireta
- Comunicação simétrica ou assimétrica
- lluffering automático ou explícito
- Enviar por cópia ou enviar por referência
- Mensagens de tamanho fixo ou variável

A seguir analisamos cada um desses tipos de sistemas de mensagens.

4.5.2 Nomeação

Os processos que desejam se comunicar precisam ter uma forma de fazer referencia mútua. Eles podem usar comunicação direta ou indireta.

4.5.2.1 Comunicação direta

Na comunicação direta, cada processo que deseja se comunicar precisa identificar explicitamente o destinatário ou remetente da comunicação. Neste esquema, as primitivas send e receive são definidas como:

- send(P, message) - Envia uma mensagem para o processo P.
- receive(Q, message) - Recebe uma mensagem do processo Q.

Um canal de comunicação neste esquema possui as seguintes propriedades:

- Um canal é estabelecido automaticamente entre cada par de processos que deseja se comunicar. Os processos precisam conhecer as respectivas identidades para se comunicar.
- Um canal é associado a exatamente dois processos.
- Entre cada par de processos, existe exatamente um canal.

Esse esquema apresenta simetria no endereçamento; ou seja, os processos de envio e recepção precisam se identificar mutuamente para haver comunicação. Uma variante do esquema utiliza assimetria no endereçamento. Apenas o remetente identifica o destinatário; o destinatário não precisa identificar o remetente. Neste esquema, as primitivas send e receive são definidas assim:

- send(P, message) - Envia uma mensagem para o processo P.
- receive(id, message) - Recebe uma mensagem de qualquer processo; a variável *id* retorna o nome do processo com o qual foi estabelecida a comunicação.

A desvantagem desses esquemas (simétrico e assimétrico) c a modularidade limitada das definições de processos resultantes. Mudar o nome de um processo pode exigir a verificação de todas as definições de processos. Todas as referencias ao nome anterior devem ser encontrados para que possam ser alteradas para o novo nome. Essa situação não c desejável do ponto de vista de compilação separada.

4.5.2.2 Comunicação indircta

Com a comunicação indireta, as mensagens são enviadas e recebidas através de caixas de correio ou portas. Uma caixa de correio pode ser vista de forma abstrata como um objeto no qual as mensagens podem ser colocadas por processos e do qual as mensagens podem ser retiradas. Cada caixa de correio tem uma identificação exclusiva. Nesse esquema, um processo pode se comunicar com outro processo através de várias caixas de correio diferentes. Dois processos só podem se comunicar se tiverem uma caixa de correio comum. As primitivas send e receive são definidas da seguinte forma:

- send (A, message) - Envia uma mensagem para a caixa de correio A.
- receive(A, message) - Recebe uma mensagem da caixa de correio A.

Neste esquema, um canal de comunicação possui as seguintes propriedades:

- Um canal c estabelecido entre um par de processos somente se os dois membros do par tiverem uma caixa de correio compartilhada.
- Um canal pode estar associado a mais de dois processos.
- Entre cada par de processos comunicantes, pode haver vários canais diferentes, com cada canal correspondendo .. uma caixa de correio.

Vamos supor que os processos P, P_t e P, compartilhem a mesma caixa de correio A. O processo P, envia uma mensagem para/\, enquanto P, e P_t executam uma operação de receive em A Que processo receberá a mensagem enviada por P,} A resposta depende do esquema escolhido:

- Permitir que um canal seja associado com no máximo dois processos.
- Permitir que no máximo um processo de cada vez execute uma operação de receive.
- Permitir que o sistema selecione arbitrariamente que processo receberá a mensagem (ou seja, $P<$ ou P_t, mas não ambos, receberá a mensagem). O sistema poderá identificar o destinatário para o remetente.

Uma caixa de correio pode ser de propriedade de um processo ou do sistema operacional. Se a caixa de correio pertencer a um processo (ou seja, a caixa de correio é parte do espaço de endereçamento do processo), fazemos a distinção entre o proprietário (que só pode receber mensagens através desta caixa de correio) e o usuário da caixa de correio (que só pode enviar mensagens para a caixa de correio). Como cada caixa tem um proprietário exclusivo, não pode haver confusão sobre quem deve receber uma mensagem enviada para esta caixa de correio. Quando um processo que possui uma caixa de correio termina, a caixa desaparece.

Qualquer processo subsequente que envie uma mensagem para essa caixa de correio deverá ser avisado de que a caixa não existe mais.

Por outro lado, uma caixa de correio pertencente ao sistema operacional tem existência própria. É independente e não está ligada a nenhum processo específico. O sistema operacional deverá fornecer um mecanismo para permitir que um processo faça o seguinte:

- Crie uma nova caixa de correio
- Envie e receba mensagens através da caixa de correio
- Exclua uma caixa de correio

O processo que cria uma nova caixa de correio é por default o proprietário da caixa. Inicialmente, o proprietário é o único processo que pode receber mensagens através dessa caixa de correio. No entanto, a propriedade e o privilégio de recebimento podem ser passados para outros processos através de chamadas ao sistema adequadas. E claro que esse procedimento pode resultar em vários destinatários para cada caixa de correio.

4.5.3 **Sincronização**

A comunicação entre processos ocorre por meio de chamadas às primitivas send e recei ve. Existem diferentes opções de projeto para implementar cada primitiva. A troca de mensagens pode ser do tipo bloqueante ou não-bloqucante - também chamado de síncrono c assíncrono.

- Envio bloqueante: O processo de envio é bloqueado até que a mensagem seja recebida pelo processo receptor ou pela caixa de correio.
- Envio não-bloqucante: O processo de envio envia a mensagem e retoma a operação.
- Recepção bloqueante: O receptor é bloqueado até que uma mensagem esteja disponível.
- Recepção não-bloqueante: O receptor recebe uma mensagem válida ou então nada.

Diferentes combinações de send c recei ve são possíveis. Quando ambos forem bloqueantes, temos um *rendezvous* - é um termo usual (encontro, em francês) entre o remetente e o destinatário.

4.5.4 Buffering

Quer a comunicação seja direta ou indireta, as mensagens trocadas pelos processos em comunicação residem em uma fila temporária. Basicamente, existem três formas de implementar uma fila desse tipo:

- Capacidade zero: A fila tem tamanho máximo 0; assim, o canal não pode ter mensagens em espera. Nesse caso, o remetente deve bloquear até que o destinatário receba a mensagem.
- Capacidade limitada: A fila tem tamanho finito n; assim, no máximo;; mensagens podem residir nela. Se a fila não estiver cheia quando uma nova mensagem for enviada, esta é colocada na fila (a mensagem é copiada ou é mantido um ponteiro para a mensagem) e o remetente pode continuar a execução sem esperar. No entanto, o canal tem capacidade finita. Se o canal estiver cheio, o remetente deverá bloquear (no sentido de "ficar bloqueado") até que haja espaço disponível na fila.
- Capacidade náo-limitada ou ilimitada: Potencialmente, a fila tem tamanho infinito; assim, qualquer número de mensagens pode ficar esperando na fila. O remetente nunca bloqueia.

O caso de capacidade zero às vezes é chamado de sistema de mensagem sem buffering; os outros casos são chamados de buffering automático.

4.5.5 Exemplo do produtor-consumidor

Agora podemos apresentar uma solução ao problema do produtor-consumidor que utiliza troca de mensagem. O produtor e o consumidor se comunicam de forma indireta utilizando a caixa de correio compartilhada mostrada na Figura 4.12.

O buffer é implementado usando a classe java.uti 1 .Vector, o que significa que será um buffer de capacidade ilimitada. Além disso, os métodos send( ) e receive( ) são não-bloqueantes.

Quando o produtor gera um item, ele o coloca na caixa de correio através do método $end ( ). O código para o produtor aparece na Figura 4.13.

O consumidor obtém um item da caixa de correio usando o método receive( ). Como receive( ) é não-bloqueante, o consumidor deve avaliar o valor doObject retornado de recei ve( ). Se for nul1,a caixa de correio estará vazia. O código para o consumidor é apresentado na Figura *4.*14.

O Capítulo 5 mostra como implementar o produtor e o consumidor como threads separados e como permitir que a caixa de correio seja compartilhada entre os threads.

```
import java.util.*;

public class HessageQueue
i

    public HessageQueue( ) {
      queue • new Vector( );
    Í

    // implementa um envio náo-bloqueaite
    public void send(Object item) {
      queue.addElement{item);
    |

    // implementa uma recepção não-bloqueante
    public Object receive( ) {
      Object item;

      ff (queue.size{ ) " 0)
            return null;
      else {
            item • queue.firstElement( );
            queue.removeElementAt(0);

            return item;
      }
    )

    private Vector queue;
Í
```

Figura 4.12 Caixa de correio para troca de mensagens.

```
HessageQueue mailBox;

while (true) {
    Oate message • new 0ate( );
    ma)1 Box.send(message);
1
```

Figura 4.13 O processo Produtor.

```
HessageQueue mailBox;

while (true) (
    Date message = (Date) mailBox.receitei );
    if (message ! • null)
        // consome a mensagem
Í
```

Figura 4.14 O processo Consumidor.

4.5.6 Um exemplo: Mach

Como exemplo de um sistema operacional baseado em mensagens, considere o sistema Mach, desenvolvido na Carnegie Mellon University. O kernel do Mach oferece suporte à criação e destruição de várias tarefas, que são semelhantes aos processos mas possuem múltiplos fluxos de controle. A maior parte da comunicação no Mach - incluindo boa parte das chamadas ao sistema e todas as informações entre tarefas- é realizada por *mensagens.* As mensagens são enviadas e recebidas através das caixas de correio, chamadas *portas* no Mach.

Mesmo as chamadas ao sistema são feitas por mensagens. Quando cada tarefa é criada, duas caixas de correio especiais - a caixa Kernel e a caixa Notify - também são criadas. A caixa de correio Kernel é usada pelo kernel para se comunicar com a tarefa. O kernel envia notificações de ocorrência de eventos para a porta Notify. Apenas três chamadas ao sistema são necessárias para a transferência de mensagens. A chamada msgsend envia uma mensagem a uma caixa de correio. Uma mensagem é recebida via msg recei ve. As chamadas de procedimento remoto (RPCs, *Remote Procedure Calls)* são executadas via msgrpc, que envia uma mensagem e espera por exatamente uma mensagem de retorno do remetente. Dessa forma, RPC modela uma chamada de procedimento típica, mas pode funcionar entre sistemas.

A chamada ao sistema porta 11 ocate cria uma nova caixa de correio e aloca espaço para sua fila de mensagens. O tamanho máximo da fila de mensagens pressupõe como padrão oito mensagens. A tarefa que cria a caixa de correio é proprietária da caixa. A proprietária também recebe acesso para receber mensagens na caixa de correio. Apenas uma tarefa de cada vez pode ser proprietária ou receber mensagens de uma caixa de correio, mas esses direitos podem ser enviados para outras tarefas, se desejado.

Inicialmente, a caixa de correio tem uma fila vazia de mensagens. A medida que as mensagens são enviadas para a caixa de correio, elas são copiadas para a caixa. Todas as mensagens têm a mesma prioridade. O Mach garante que múltiplas mensagens do mesmo remetente sejam colocadas na fila em ordem FIFO (primeiro a entrar, primeiro a sair), mas não garante um ordenamento absoluto. Por exemplo, mensagens enviadas de dois remetentes distintos podem ser colocadas na fila em qualquer ordem.

As mensagens em si consistem em um cabeçalho de tamanho fixo, seguido por uma porção de dados de tamanho variável. O cabeçalho inclui o tamanho da mensagem e dois nomes de caixas de correio. Quando uma mensagem é enviada, um dos nomes de caixa de correio é a caixa para a qual a mensagem está sendo enviada. Muitas vezes, o thread de envio espera uma resposta; o nome da caixa de correio do remetente é passado para a tarefa receptora, que pode usá-la como um "endereço de retorno" para enviar mensagens de volta.

Aparte variável de uma mensagem é uma lista dos itens de dados digitados. Cada entrada na lista tem um tipo, tamanho e valor. O tipo dos objetos especificado na mensagem é importante, já que os objetos definidos pelo sistema operacional - tais como propriedade ou direitos de acesso em recepção, estados de tarefa c segmentos de memória - podem ser enviados nas mensagens.

As operações de envio e recepção em si são flexíveis. Por exemplo, quando uma mensagem é enviada para uma caixa de correio, esta pode estar cheia. Sc a caixa não estiver cheia, a mensagem c copiada para a caixa de correio e o thread de envio continua. Se a caixa estiver cheia, o thread de envio tem quatro opções:

1. Esperar indefinidamente até haver espaço na caixa de correio.
2. Esperar no máximo n milissegundos.
3. Não esperar, mas retornar imediatamente.
4. Temporariamente armazenar a mensagem em cache. Uma mensagem pode ser entregue ao sistema operacional para ser mantida por ele, mesmo que a caixa de correio para a qual está sendo enviada esteja cheia. Quando a mensagem puder ser colocada na caixa de correio, uma mensagem é enviada de volta ao remetente; apenas uma mensagem desse tipo para uma caixa de correio cheia pode estar pendente em qualquer momento, para determinado thread de envio.

A última opção é dedicada a tarefas de servidor, como um driver de impressora de linhas. Depois de terminar um pedido, essas tarefas podem precisar enviar uma resposta única à tarefa que solicitou o serviço, mas também devem continuar com outros pedidos de serviço, mesmo se a caixa de correio de resposta para um cliente estiver cheia.

A operação de recepção deve especificar a partir de que caixa de correio ou conjunto de caixas de correio determinada mensagem será recebida. Um conjunto de caixas de correio é uma coleção de caixas de correio, conforme declarado pela tarefa, que podem ser agrupadas e tratadas como uma caixa única para os objetivos da tarefa. Threads em uma tarefa só podem receber de uma caixa de correio ou conjunto de caixas de correio para a qual a tarefa tenha acesso de recepção. Uma chamada ao sistema port^status retoma o número de mensagens em determinada caixa de correio. A operação de recepção tenta receber de (1) qualquer caixa de correio cm um conjunto de caixas de correio ou (2) uma caixa de correio especifica (identificada). Se nenhuma mensagem estiver esperando para ser recebida, o thread receptor poderá esperar, esperar no máximo n milissegundos, ou não esperar.

O sistema Mach foi especialmente projetado para sistemas distribuídos, que são discutidos nos Capítulos 14 a 16, mas o Mach também é adequado para sistemas monoprocessador. O principal problema com sistemas de mensagens geralmente tem sido o baixo desempenho causado pela cópia da mensagem primeiro do remetente para a caixa de correio e depois da caixa de correio para o receptor. O sistema de mensagens do Mach tenta evitar as operações de cópia dupla usando técnicas de gerência de memória virtual (Capítulo 10). Basicamente, o Mach mapeia o espaço de endereçamento que contém a mensagem do remetente no espaço de endereçamento do receptor. A mensagem em si nunca é copiada. Essa técnica de gerência de mensagens melhora em muito o desempenho, mas só funciona para mensagens intra-sistema. O sistema operacional Mach é discutido no capítulo extra encontrado no nosso site na Web: (http://www.bell-labs.com/topic/books/os-book).

^f 4.5.7 Um exemplo: Windows NT

O sistema operacional Windows NT é um exemplo de projeto moderno que utiliza a modularidade para aumentar a funcionalidade e diminuir o tempo necessário para implementar novos recursos. O NT fornece suporte a vários ambientes operacionais ou *subsistemas,* com os quais os programas aplicativos se comunicam via um mecanismo de troca de mensagens. Os programas aplicativos podem ser considerados clientes do servidor de subsistema NT.

O recurso de troca de mensagens no NT é chamado de recurso de chamada de procedimento local (LPC - local procedure call). O LPC no Windows NT permite a comunicação entre dois processos que estão na mesma máquina. È semelhante ao mecanismo RPC padrão é que amplamente usado, mas é otimizado e específico ao NT. O Windows NT, como o Mach, utiliza um objeto porta para estabelecer e manter uma conexão entre dois processos. TCKJO cliente que chama um subsistema precisa de um canal de comunicação, que é fornecido por um objeto porta e nunca é herdado. O NT utiliza dois tipos de portas: portas de conexão e portas de comunicação. Na verdade, são os mesmos, mas recebem nomes diferentes de acordo com a forma em que são usados. As portas de conexão são objetos com nome (consulte o Capítulo 22 para obter mais informações sobre objetos do NT), visíveis a todos os processos; eles dão às aplicações uma forma de estabelecer um canal de comunicação. Essa comunicação funciona da seguinte maneira:

- O cliente abre um descritor para o objeto de porta de conexão do subsistema.
- O cliente envia um pedido de conexão.
- O servidor cria duas portas de comunicação privadas e retorna o descritor de um deles para o cliente.
- O cliente e o servidor usam o descritor da porta correspondente para enviar mensagens ou *callbacks* e ouvir respostas.

O NT utiliza três tipos de técnicas de troca de mensagens em uma porta que o cliente especifica quando estabelece o canal. O mais simples, que é usado para mensagens pequenas, utiliza a fila de mensagensda porta como um meio de armazenamento intermediário e copia a mensagem de um processo para outro. Com esse método, mensagens de até 256 bytes podem ser enviadas.

Se um cliente precisar enviar uma mensagem maior, ele passa a mensagem através de um objeto de seção (memória compartilhada). O cliente precisa decidir, quando estabelecer o canal, se precisará ou não enviar uma mensagem grande. Se o cliente determinar que deseja enviar mensagens grandes, ele solicitará que um objeto de seção seja criado. Da mesma forma, se o servidor decidir que as respostas serão grandes, ele criará um objeto de seção. Para que o objeto possa ser usado, uma mensagem pequena será enviada contendo um

ponteiro e informações de tamanho sobre o objeto de seção. Esse método é mais complicado do que o primeiro método, mas evita a cópia de dados. Em ambos os casos, um mecanismo de *callback* pode ser usado quando o cliente ou o servidor não puderem responder imediatamente a um pedido. O mecanismo de *callback* permite que eles efetuem o tratamento de mensagens de forma assíncrona.

Uma das desvantagens de usar a troca de mensagens para executar funções rudimentares, tais como funções gráficas, é que o desempenho talvez não seja tão bom quanto em um sistema sem troca de mensagens (ou seja, de memória compartilhada). Para aumentar o desempenho, o ambiente NT nativo (Win32) utiliza um terceiro método de troca de mensagens, chamado LPC rápido. Um cliente envia para a porta de conexão do servidor um pedido de conexão indicando que ele estará usando LPC rápido. O servidor estabelece um thread servidor dedicado para manipular os pedidos, um objeto de seção de 64 kilobytes e um objeto *event-pair*. A partir daí, as mensagens são passadas no objeto de seção, e a sincronização é realizada pelo objeto *event-pair*. Esse esquema elimina a cópia de mensagem, o custo de usar o objeto porta e o custo de determinar que thread de cliente está chamando, já que existe um thread para cada servidor thread de cliente. O kernel também dá ao thread dedicado preferência de escalonamento. A desvantagem é que esse método utiliza mais recursos do que os outros dois métodos. Como existe custo envolvido na passagem de mensagens, o NT poderá reunir várias mensagens em uma única mensagem para reduzir esse custo. Para obter maiores informações sobre o Windows NT, consulte o Capítulo 22.

4.6 • Resumo

Um processo é um programa em execução. A medida que um processo executa, ele muda de estado. O estado de um processo é definido pela atividade atual daquele processo. Cada processo pode estar em um dos seguintes estados: novo, pronto, cm execução, em espera ou encerrado. Cada processo é representado no sistema operacional pelo seu próprio bloco de controle de processo (PCB).

Um processo, quando não estiver executando, é colocado em alguma fila de espera. Existem duas classes principais de filas em um sistema operacional: filas de pedido de l/O e a fila de processos prontos. A fila de processos prontos contém todos os processos que estão prontos para executar e estão esperando pela CPU. Cada processo é representado por um PCB, e os PCBs podem ser unidos para formar uma fila de processos prontos. O escalonamento de longo prazo (de jobs) é a Seleção de processos que receberão permissão para disputar CPU. Normalmente, o escalonamento de longo prazo é altamente influenciado por considerações de alocação de recursos, especialmente a gerência de memória. O escalonamento de curto prazo (de CPU) é a Seleção de um processo da fila de processos prontos.

Os processos no sistema podem executar concorrentemente. Existem vários motivos para permitir a execução concorrente: compartilhamento de informações, velocidade de computação, modularidade e conveniência. A execução concorrente requer um mecanismo para a criação e exclusão de processos.

Os processos executando no sistema operacional podem ser independentes ou cooperativos. Os processos cooperativos devem ter meios de se comunicar entre si. Em princípio, existem dois esquemas de comunicação complementares: sistemas de memória compartilhada c de mensagens. O método de memória compartilhada requer que os processos em comunicação compartilhem algumas variáveis. Os processos devem trocar informações usando essas variáveis compartilhadas. Em um sistema de memória compartilhada, a responsabilidade de fornecer a comunicação está com os programadores da aplicação; o sistema operacional precisa fornecer apenas a memória compartilhada. O método do sistema de mensagens permite que os processos troquem mensagens entre si. A responsabilidade por fornecer a comunicação poderá estar com o próprio sistema operacional. Esses dois esquemas não são mutuamente exclusivos, e podem ser usados ao mesmo tempo em um único sistema operacional.

Podemos implementar a memória compartilhada ou troca de mensagens usando Java. Threads separados de controle podem ser criados c podem se comunicar usando qualquer um dos métodos.

• Exercícios

4.1 O MS-DOS não permitia o processamento concorrente. Discuta as três principais complicações que o processamento concorrente acrescenta a um sistema operacional.

4.2 Descreva as diferenças entre o escalonamento de curto, médio e longo prazo.

4.3 Num computador DECSYSTEM-20 possui vários conjuntos de registradores. Descreva as ações de uma troca de contexto, caso O novo contexto já esteja carregado em um dos conjuntos de registradores. O que mais precisa acontecer se o novo contexto estiver na memória em vez de em um conjunto de registradores, e se todos os conjuntos de registradores estiverem em uso?

4.4 Descreva as ações tomadas por um kernel para fazer a troca de contexto entre processos.

4.5 Quais são as vantagens e desvantagens de cada um dos itens a seguir? Considere os níveis de sistema e dos programadores.

a. Comunicação simétrica e assimétrica
b. Buffering automático e explícito
c. Enviar por copia e enviar por referência
d. Mensagens de tamanho fixo e variável

4.6 Considere o esquema de comunicação entre processos no qual são usadas caixas de correio.

a. Vamos supor que um processo *P* deseje esperar duas mensagens, uma da caixa A e outra da caixa *B.* Que sequência de send e receive deve ser executada?
b. Que sequência de sende receive/* deve executar se quiser esperar por uma mensagem da caixa *A* ou da caixa B (ou de ambas)?
c. Uma operação de recei ve faz o processo esperar até que a caixa de correio não esteja vazia. Crie um esquema que permita que um processo espere até que a caixa de correio fique vazia ou explique por que esse esquema não pode existir.

Notas bibliográficas

O tópico "comunicação entre processos" foi discutido por Brinch Hansen [1970} com relação ao sistema RC 4000. Schlichting e Schneidcr 11982) discutiram primitivas de troca de mensagens assíncrona. O recurso IPC implementado no nível de usuário foi descrito por liershad e colegas [1990].

Detalhes da comunicação entre processos nos sistemas UNIX foram apresentados por Gray (*1997],* Barrera |1991| e Vahalia 11996] apresentaram a comunicação entre processos no sistema Mach. Solomon [1998] delineou a comunicação entre processos no Windows NT.

As discussões relativas à implementação de RPCs foram apresentadas por Birrell e Nelson [1984]. O projetode um mecanismo de RPCconfiável foi apresentado por Shrivastava e Panzieri [1982]. Um estudo sobre RPCs foi apresentada por Tay e Ananda [19901- Stankovic (1982] eStaunstrup (1982] discutiram as chamadas de procedimento *versus* a comunicação por troca de mensagens.

• Exercícios

4.1 O MS-DOS não permitia o processamento concorrente. Discuta as três principais complicações que o processamento concorrente acrescenta a um sistema operacional.

4.2 Descreva as diferenças entre o escalonamento de curto, médio e longo prazo.

4.3 Num computador DECSYSTEM-20 possui vários conjuntos de registradores. Descreva as ações de uma troca de contexto, caso O novo contexto já esteja carregado em um dos conjuntos de registradores. O que mais precisa acontecer se o novo contexto estiver na memória em vez de em um conjunto de registradores, e se todos os conjuntos de registradores estiverem em uso?

4.4 Descreva as ações tomadas por um kernel para fazer a troca de contexto entre processos.

4.5 Quais são as vantagens e desvantagens de cada um dos itens a seguir? Considere os níveis de sistema e dos programadores.

a. Comunicação simétrica e assimétrica
b. Buffering automático e explícito
c. Enviar por copia e enviar por referência
d. Mensagens de tamanho fixo e variável

4.6 Considere o esquema de comunicação entre processos no qual são usadas caixas de correio.

a. Vamos supor que um processo *P* deseje esperar duas mensagens, uma da caixa A e outra da caixa *B.* Que sequência de send e receive deve ser executada?
b. Que sequência de sende receive/* deve executar se quiser esperar por uma mensagem da caixa *A* ou da caixa B (ou de ambas)?
c. Uma operação de recei ve faz o processo esperar até que a caixa de correio não esteja vazia. Crie um esquema que permita que um processo espere até que a caixa de correio fique vazia ou explique por que esse esquema não pode existir.

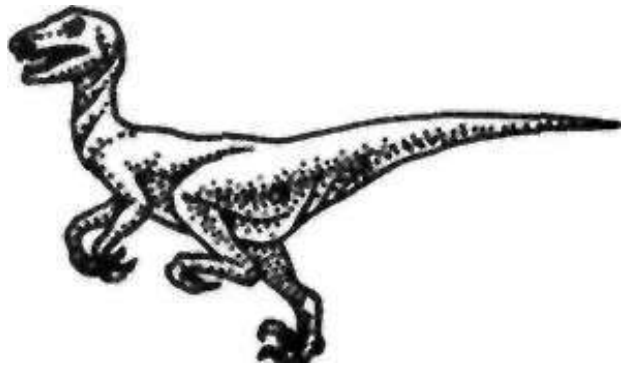
Notas bibliográficas

O tópico "comunicação entre processos" foi discutido por Brinch Hansen [1970} com relação ao sistema RC 4000. Schlichting e Schneidcr 11982) discutiram primitivas de troca de mensagens assíncrona. O recurso IPC implementado no nível de usuário foi descrito por liershad e colegas [1990].

Detalhes da comunicação entre processos nos sistemas UNIX foram apresentados por Gray (*1997],* Barrera |1991| e Vahalia 11996] apresentaram a comunicação entre processos no sistema Mach. Solomon [1998] delineou a comunicação entre processos no Windows NT.

As discussões relativas à implementação de RPCs foram apresentadas por Birrell e Nelson [1984]. O projetode um mecanismo de RPCconfiável foi apresentado por Shrivastava e Panzieri [1982]. Um estudo sobre RPCs foi apresentada por Tay e Ananda [19901- Stankovic (1982] eStaunstrup (1982] discutiram as chamadas de procedimento *versus* a comunicação por troca de mensagens.

Capítulo 5



THREADS

O modelo de processo apresentado no Capítulo 4 supôs que um processo era um programa em execução com um único fluxo de controle. Muitos sistemas operacionais modernos agora oferecem recursos para que um processo contenha múltiplos fluxos de controle, ou threads. Este capitulo apresenta muitos conceitos associados com os sistemas de computação com threads múltiplos e aborda o uso de Java para criar e manipular threads.

5.1 • Visão geral

Um thread, às vezes chamado de processo leve *(lightweighl ftrocess), é* uma unidade básica de utilização de CPU; compreende um ID de thread, um contador de programa, um conjunto de registradores e uma pilha. Compartilha com outros threads pertencentes ao mesmo processo sua seção de código, seção de dados e outros recursos do sistema operacional, tais como arquivos abertos e sinais. Um processo tradicional, ou pesado *(hemyu/eight),* tem um único fluxo de controle. Processos com múltiplos threads podem realizar mais de uma tarefa de cada vez. A Higura 5.1 ilustra a diferença entre um processo de thread único tradicional e um processo com múltiplos threads.

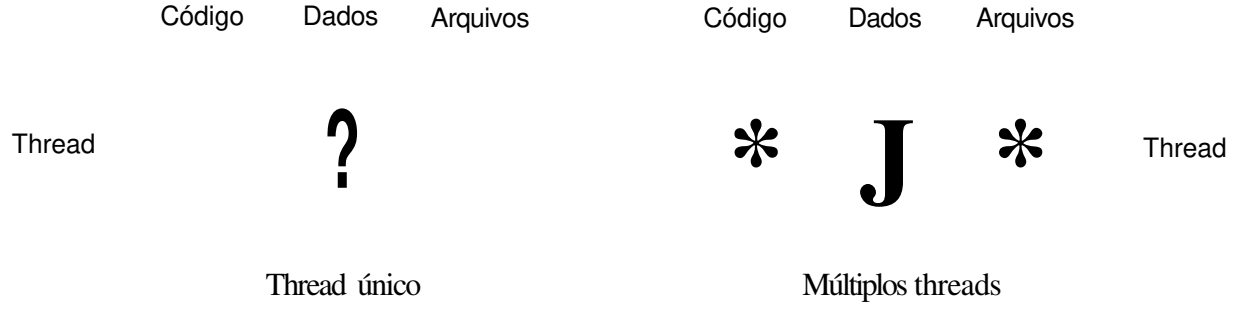


Figura 5.1 Processos com um e com múltiplos threads.

Muitos pacotes de software que executam em PCs desktop modernos têm múltiplos threads. Uma aplicação geralmente é implementada como um processo separado com vários threads de controle. Um navegador Web pode fazer um thread exibir imagens ou texto enquanto outro recupera dados da rede. Um processador de textos pode ter um thread para exibir gráficos, outro thread para ler sequências de teclas do usuário e um terceiro thread para efetuar a verificação ortográfica e gramatical em segundo plano.

Existem também situações em que uma única aplicação pode ser solicitada a realizar várias tarefas semelhantes. Por exemplo, um servidor Web aceita pedidos de cliente para páginas Web, imagens, som e assim por diante. Um servidor Web ocupado pode ter vários (talvez ate centenas) de clientes acessando-o de forma concorrente. Se o servidor Web executasse como um processo tradicional de único thread, poderia servir

apenas um cliente de cada vez com seu processo único. O tempo que um cliente teria de esperar para que seu pedido seja atendido poderia ser enorme. Uma solução é fazer o servidor executar como um único processo que aceita pedidos. Quando o servidor recebe um pedido, ele cria um processo separado para atender a esse pedido. No entanto, em vez de incorrer no custo de criar um novo processo para atender a cada pedido, pode ser mais eficiente ter um processo que contenha vários threads para atender ao mesmo propósito. Essa abordagem utilizaria múltiplos threads em um processo de servidor Web. O servidor criaria um thread separado que ouviria os pedidos dos clientes; quando um pedido fosse feito, em vez de criar outro processo, ele criaria outro thread para atender ao pedido.

Java tem outros usos para os threads. De um lado, Java não tem conceito de comportamento assíncrono. Por exemplo, quando ela tiver de efetuar uma conexão telnet cm um servidor, o cliente c bloqueado até que a conexão seja feita ou ocorra um *timeout.* Um *timeout* é um exemplo de evento assíncrono e Java não tem suporte direto para eventos assíncronos. Se um programa Java tentar se conectar a um servidor, ele bloqueará até a conexão ser efetuada. (Considere o que acontecerá se o servidor estiver fora do ar!) A solução Java é configurar um thread que tentará fazer a conexão ao servidor e outro thread que inicialmente ficará suspenso durante um prazo (por exemplo, 60 segundos) e depois será ativado. Quando esse thread de temporização for ativado, ele verificará se o thread de conexão ainda está tentando se conectar com o servidor. Sc esse thread ainda estiver tentando, ele irá interrompê-lo e impedirá que continue a tentar.

5.2 • Benefícios

Os benefícios da programação com múltiplos threads podem ser divididos em quatro categorias básicas:

1. *Capacidade de resposta;* O multithreading de uma aplicação interativa pode permitir que um programa continue executando mesmo se parte dele estiver bloqueada ou executando uma operação demorada, aumentando, assim, a capacidade de resposta para o usuário. Por exemplo, um navegador Vfeb com múltiplos threads ainda poderia permitir a interação do usuário em um thread enquanto uma imagem está sendo carregada em outro thread. . ,
2. *Compartilhamento de recursos:* Por defatilt, os threads compartilham a memória e os recursos do processo aos quais pertencem. O benefício do compartilhamento do código permite que uma aplicação tenha vários threads diferentes de atividade todos dentro do mesmo espaço de endereçamento.
3. *Economia*: Alocar memória e recursos para a criação de processos é caro. Como alternativa, como os threads compartilham recursos do processo aos quais pertencem, é mais económico criar e realizar a troca de contexto de threads. Pode ser difícil avaliar empiricamente a diferença em custo de criar e manter um processo em vez de um thread, mas em geral é muito mais demorado criar e gerenciar processos do que threads. No Solaris, criar um processo é aproximadamente 30 vezes mais lento do que criar um thread, e a troca de contexto é cinco vezes mais lenta.
4. *Utilização de arquiteturas multiprocessador.* Os benefícios do multithreading podem ser muito aumentados em uma arquitetura multiprocessador, na qual cada thread pode estar executando em paralelo em um processador diferente. Em uma arquitetura no processador, a CPU geralmente move-se entre cada thread de forma tão rápida que existe a ilusão de paralelismo, mas na verdade apenas um thread está sendo executado de cada vez.

5.3 • Threads de usuário c de kernel

Nossa discussão até aqui tratou threads de forma genérica. No entanto, pode ser fornecido suporte a threads no nível do usuário, para threads de usuário, ou pelo kernel, para threads de kernel. Essa distinção será discutida a seguir.

5.3.1 Threads de usuário

Os threads de usuário são suportados acima do kernel e são implementados por uma biblioteca de threads no nível do usuário. A biblioteca fornece suporte à criação, escalonamento e gerência de threads, sem suporte do

kcrnel. Como o kcrncl não está a par dos threads de usuário, todas as atividades de criação e escalonamento de threads são feitas no espaço de usuário, sem necessidade da intervenção do kernel. Portanto, os threads de usuário geralmente são rápidos de criar e gerenciar; no entanto, também apresentam desvantagens. Por exemplo, se o kernel tiver um único thread, qualquer thread de usuário realizando uma chamada bloqueante ao sistema causará o bloqueio de todo o processo, mesmo se houver outros threads disponíveis para execução na aplicação. As bibliotecas de threads de usuário incluem *Píhreads* do POSIX, *C-threads* do Mach e *threads* do Solaris.

5.3.2 **Threads de kcrnel**

Os threads de kernel são suportados diretamente pelo sistema operacional: a criação, o escalonamento e a gerência de threads são feitos pelo kcrnel no espaço do kcrnel. Como a gerência de threads é feita pelo sistema operacional, os threads de kernel são geralmente mais lentos para criar c gerenciar do que os threads de usuário. No entanto, como o kernel está gerenciando os threads, se um thread realizar uma chamada bloqueante ao sistema, o kernel poderá escalonar outro thread na aplicação para execução. Além disso, em um ambiente multiprocessador, o kernel pode escalonar threads em diferentes processadores. Windows NT, Solaris e Digital UNIX são sistemas operacionais que suportam threads de kernel.

5.4 • Modelos de multithreading

Muitos sistemas fornecem suporte a threads de usuário e de kernel, resultando em diferentes modelos de multithreading. Vamos analisar três tipos comuns de implementação de threading.

5.4.1 Modelo muitos-para-um

O modelo muitos-para-um *(ttiany-to-one model)* (Figura 5.2) mapeia muitos threads de usuário em um thread de kernel. A gerência de threads é feita no espaço de usuário, sendo assim eficiente, mas o processo inteiro será bloqueado se um thread efetuar uma chamada bloqueante ao sistema. Além disso, como apenas um thread pode acessar o kernel de cada vez, não é possível executar múltiplos threads em multiprocessadores. As bibliotecas de threads de usuário implementadas nos sistemas operacionais que não suportam os threads de kernel utilizam o modelo muitos-para-um.

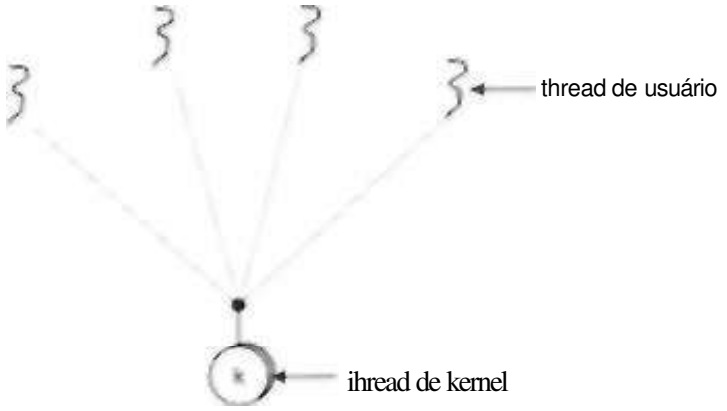


Figura 5.2 Modelo miútos-para-um.

Figura 5.3 Modelo um-para-um.

5.4.2 Modelo um-para-um

O modelo um-para-um *(ofte-to-one ntodel)* (Figura 5.3) mapcia cada thrcad de usuário em um thread de kernel. Fornece mais concorrência do que <> modelo muitos-para-um, permitindo que outro thread execute quando um thread efetuar uma chamada bloqueante ao sistema; também permite de múltiplos threads^xecutem em paralelo em multiprocessadores. A única desvantagem desse modelo é que criar um thread de usuário requer criar o thread de kernel correspondente. Como o custo de criar threads de kernel pode prejudicar o desempenho de uma aplicação, a maior parte das implementações desse modelo restringem o número de threads suportados pelo sistema. O Windows NT e o OS/2 implementam o modelo um-para-um.

5.4.3 Modelo muitos-para-muitos

O modelo muitos-para-muitos *(many-to-many model)* (Figura 5.4) multiplexa muitos threads de usuário em um número menor ou igual de threads de kernel. O número de threads de kernel pode ser específico para determinada aplicação ou determinada máquina (uma aplicação pode receber mais threads de kernel em um multiprocessador do que em um uniprocessador). Enquanto o modelo muitos-para-um permite ao desenvolvedor criar tantos threads de usuário quantos desejar, a verdadeira concorrência não é obtida porque apenas um thread pode ser escalonado pelo kernel de cada vez. O modelo um-para-um permite maior concorrência, mas o desenvolvedor precisa ter cuidado para não criar um número excessivo de threads em uma aplicação (c, em algumas instâncias, pode ser limitado no número de threads que podem ser criados). O modelo muitos-para-muitos não apresenta essas desvantagens: os descnvolvedores podem criar tantos threads quantos forem necessários, e os threads de kernel correspondentes podem executar em paralelo em um multiprocessador. Além disso, quando um thread realiza uma chamada bloqueante ao sistema, o kernel pode escalonar outro thread para execução. Solaris, IRIX e Digital UNIX suportam esse modelo.

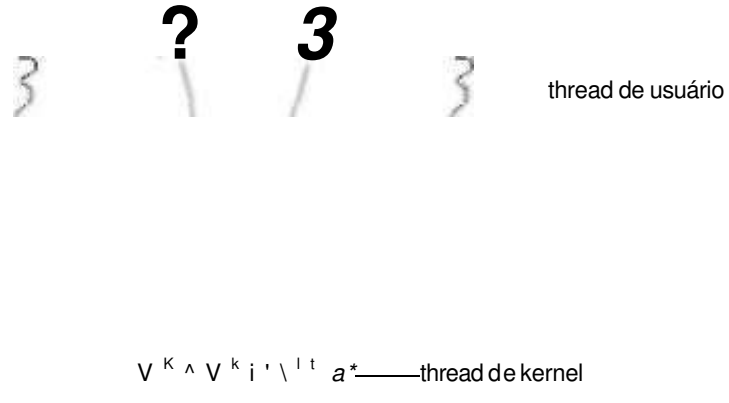


Figura 5.4 Modelo muitos-para-muitos.

5.5 • Threads do Solaris 2

O Solaris 2 é uma versão do UNI X que até 1992 só oferecia suporte a processos tradicionais pesados com um único thread de controle. Foi transformado em um sistema operacional moderno com suporte a threads nos níveis de kernel c usuário, multiprocessamento simétrico (SMP) e escalonamento de tempo real.

O Solaris 2 suporta threads de usuário com uma biblioteca contendo APls para a criação e a gerência de threads. O Solaris 2 define também um nível intermediário de threads. Entre os threads de usuário e de kernel estão processos leves (I.WP). Cada processo contém pelo menos um LWP. A biblioteca de threads multiplexa os threads de usuário no pool de i.WPs para o processo, e apenas os threads de usuário conectados no momento a um LWP realizam o trabalho. O restante é bloqueado ou fica esperando por um l.WP no qual possa rodar.

Todas as operações no kernel são executadas por threads standard de kernel. Existe um thread de kernel para cada LWP, e há alguns threads de kernel que executam tarefas do kernel e não têm LWP associado (por exemplo, um thread para atender aos pedidos de disco). Os threads de kernel são os únicos objetos escalona-

dos no sistema (consulte o Capítulo 6 para uma discussão sobre escalonamento). O Solaris implementa o modelo muitos-para-muitos; todo seu sistema de threads é representado na Figura 5.5.

Os threads de usuário podem ser limitados ou ilimitados. Um thread de usuário limitado é permanentemente associado a um LWP. Apenas esse thread executa no l.WP c mediante solicitação o LWP pode ficar dedicado a um único processador (ver o thread à extrema direita na Figura 5.5). Restringir um thread é útil em situações que exigem um rápido tempo de resposta, tais como uma aplicação de tempo real. Um thread ilimitado não é ligado permanentemente a LWP algum. Todos os threads desse tipo em uma aplicação são multiplexados no pool de LWPs disponíveis para a aplicação. Os threads são ilimitados por dcfault.

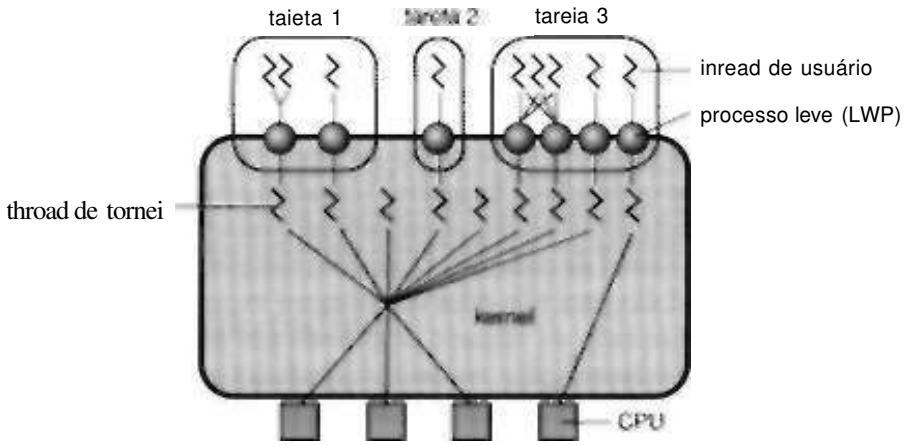


Figura 5.5 Threads do Solaris 2.

Considere o sistema em operação. Qualquer processo pode ter muitos threads de usuário. Esses threads de usuário podem ser escalonados e alternados entre os LWPs pela biblioteca de threads sem intervenção do kcrncl. Os threads de usuário são extremamente eficientes porque não há necessidade de suporte do kerncl para a biblioteca de threads realizar a troca de contexto de um thread de usuário para outro.

Cada LWP é conectado exatamente a um thread de kerncl, enquanto cada thread de usuário independe do kernel. Pode haver muitos 1. WPs em um processo, mas eles só são necessários quando o thread precisa se comunicar com o kernel. Por exemplo, um LWP c necessário para cada thread que pode bloquear de forma concorrente cm chamadas ao sistema. Considere cinco pedidos distintos de leitura de arquivo que poderiam estar ocorrendo simultaneamente. Cinco LWPs seriam então necessários, porque todos poderiam estar esperando pela conclusão de l/O no kernel. Se a tarefa tivesse apenas quatro LWPs, o quinto pedido teria de esperar que um dos LWPs retornasse do kernel. Adicionar um sexto LWP não seria vantajoso se só houvesse trabalho para cinco.

Os threads de kcrncl são escalonados pelo escalonador do kernel e executam na CPU ou CPUs do sistema. Se um thread de kernel bloquear (como quando espera pela conclusão de uma operação de l/O), o processador fica livre para executar outro thread de kernel. Se o thread que bloqueou estava executando em nome de um LWP, o LWP também bloqueia. Em consequência, o thread de usuário ligado ao LWP naquele momento também bloqueia. Se um processo tiver mais de um LWP, outro poderá ser escalonado pelo kerncl.

A biblioteca de threads ajusta dinamicamente o número de LWPs no pool para garantir o melhor desempenho para a aplicação. Por exemplo, se todos os LWPs em um processo estiverem bloqueados e houver outros threads que podem ser executados, a biblioteca de threads automaticamente cria outro LWP para atribuir a um thread em espera. Assim, evita-se o bloqueio de um programa pela falta de LWPs não-bloqueados. Além disso, os LWPs são recursos de kernel de manutenção cara se não estiverem sendo usados. A biblioteca de threads gerência os LWPs e os exclui quando ficarem sem uso por muito tempo, geralmente em torno de 5 minutos.

Os desenvolvedores usaram as seguintes estruturas de dados para implementar threads no Solaris 2:

- Um thread de usuário contém um ID de thread, um conjunto de registradores (incluindo um contador de programa e um ponteiro de pilha), pilha e prioridade (usada pela biblioteca de threads para fins de escalonamento). Nenhuma dessas estruturas de dados são recursos do kernel; todas existem no espaço de usuário.

- Um LWP tem um conjunto de registradores para o thread de usuário que ele está executando, assim como informações de memória e contabilidade. Um LWP é uma estrutura de dados do kernel e reside no espaço do kernel.
- Um thread de kernel só tem uma pequena estrutura de dados e uma pilha. A estrutura de dados inclui uma cópia dos registradores de kernel, um ponteiro para o LWP ao qual está ligado e informações de prioridade e escalonamento.

Cada processo no Solaris 2 contém boa parte das informações descritas no bloco de controle de processo (PCB), que foi discutido na Seção 4.1.3. Especificamente, um processo do Solaris contém um identificador de processo (PlD), mapa de memória, lista de arquivos abertos, prioridade e um ponteiro para uma lista de LWPs associados ao processo (consulte a Figura 5.6).

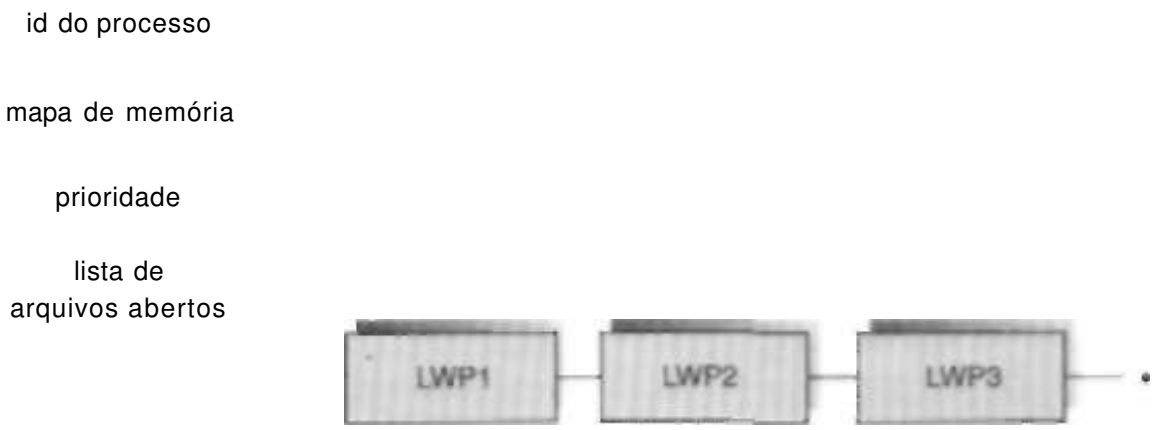


Figura 5.6 Processo do Solaris.

5.6 • Threads de Java

Como já observado, o suporte para threads é fornecido pelo sistema operacional ou por uma biblioteca de threads. Por exemplo, a biblioteca WÍn32 fornece um conjunto de APIs para efetuar multithreading em aplicações nativas do Windows e Pihreads fornece uma biblioteca de funções de gerência de threads para sistemas compatíveis com POSIX. Java é notável, pois fornece suporte no nível da linguagem para a criação e gerência de threads.

Todos os programas Java consistem cm pelo menos um thread de controle. Mesmo um programa Java simples com apenas um método mai rt( ) executa como um único thread na JVM. Além disso, Java fornece comandos que permitem ao desenvolvedor criar c manipular threads de controle adicionais no programa.

5.6.1 Criação de threads

Uma forma de criar um thread explicitamente é criar uma nova classe derivada da classe Thread e redefinir o método run( ) da classe Thread. Essa abordagem é apresentada na Figura 5.7.

Um objeto dessa classe derivada executará como um thread de controle separado na JVM. No entanto, criar um objeto derivado da classe Thread não cria especificamente o novo thread; em vez disso, é o método start ( ) que realmente cria o novo thread. Chamar o método start ( ) para o novo objeto (1) aloca memória e inicial iza um novo thread na JVM e (2) chama o método run ( ) tornando o thread passível de execução pela JVM. (Observação: Nunca chame o método run( ) diretamente. Chame o método start( ),e ele chamará o método run {) por você.)

Quando esse programa executa, dois threads são criados pela JVM. O primeiro é o thread associado com a aplicação - o thread que começa a execução no método mai n ( ). O segundo thread é o thread runner que é explicitamente criado com o método start ( ). O thread runner começa a execução no seu método run( ).

```
class Workerl extends Thread
f
    public void run( ) {
      System.out.println("I Am a Worker Thread");
    I
|

public class First
í
    public static void main(String args[ ]) {
      Workerl runner ' new Workerl( );

      runner.start( );

      System.out.println("I Am The Hain Thread");
    )
í
```

Figura 5.7 Criação de thread pela extensão da classe Thread.

Outra opção para criar um thread separado é definir uma classe que implementa a interface Runnable. Essa interface é definida desta forma:

```
public interface Runnable
(
    public abstract void run( );
)
```

Portanto, quando uma classe implementa Runnable, deverá definir um método run( ). (A classe Thread, além de definir métodos estáticos e de instância, também implementa a interface Runnable. Isso explica por que uma classe derivada de Thread deve definir um método run( ).)

Implementar a interface Runnable é similar a estender a classe Thread, a única mudança sendo substituir "extends Thread" por "implements Runnable":

```
Class Worker2 implements Runnable
í
    public void run{ ) (
        System.out.println('I am a worker thread. ");
    }
|
```

Criar um novo thread a partir de uma classe que implemente Runnable é ligeiramente diferente de criar um thread de uma classe que estende Thread. Como a nova classe não estende Thread, ela não tem acesso aos métodos estáticos ou de instância da classe Thread como o método start( ). No entanto, um objeto da classe Thread ainda é necessário porque é o método start( ) que cria um novo thread de controle. A Figura 5.8 mostra como a criação de thrcads usando a interface Runnable pode ser obtida.

```
public classe Second
(
    public static void main(String args[ ]) {
        Runnable runner - new Worker2( );

        Thread thrd • new Thread(runner);

        thrd.startf );
        System.out.println("I Am The Main Thread");
    I
J
```

Figura 5.8 Criação de thrcads implementando a interface Runnable.

Na classe Second, um novo objeto Thread é criado, sendo passado um objcto Runnable no seu construtor. Quando o thread é criado com o método stai-t( ), 0 novo thraed começa a execução no método run( ) do objeto Runnable.

Ambos os métodos de criação de threads são igualmente populares. Por que Java suporta duas abordagens para criar threads? Que abordagem é mais apropriada para uso em que situações?

A primeira pergunta é fácil de responder. Como Java não suporta herança múltipla, se uma classe já tiver derivado de outra, não poderá estender também a classe Thread. Um bom exemplo é que um applet já estende a classe Applet. Paracfetuar multithreadem um applet, estenda a classe Applet e implemente a interface Runnable:

```
public class ThreadedApplet extends Applet
        implements Runnabl*
|
    . . .
1
```

A resposta para a segunda pergunta é menos óbvia. Os puristas da orientação a objetos podem dizer que, a menos que você esteja refinando a classe Thread, a classe não deve ser estendida. (Embora esse ponto possa ser discutível, muitos dos métodos na classe Thread são definidos como f i nal.) No entanto, se a nova classe implementar a interface Runnabl e e não estender Thread, nenhum dos métodos de instância da classe Thread poderão ser chamados de dentro da nova classe. Não tentaremos determinar a resposta correra nesse debate. Para clareza de código, a menos que uma classe requeira multithreading c já seja estendida, adotaremos a abordagem de estender a classe Thread.

5.6.2 Gerência de threads

Java fornece várias APIs para gerenciar threads, incluindo:

- suspend( ): Suspende a execução de um thread que estiver processando no momento
- sleep( ): Suspende o thread em execução no momento durante determinado período
- resume( ): Retoma a execução de um thread que tinha sido suspenso
- stop( ): Interrompe a execução de um thread; depois que um thread tiver sido interrompido, ele não poderá ser retomado ou iniciado

Cada um dos diferentes métodos para controlar o estado de um thread pode ser útil em determinadas situações. Por exemplo, os applets são exemplos naturais de multithreading porque normalmente têm gráficos, animação e áudio - todos bons candidatos à gerência como threads separados. No entanto, pode não fazer sentido para um applet executar enquanto ele não estiver sendo exibido, particularmente se o applet estiver executando uma tarefa com uso intensivo de CPU. Uma forma de lidar com essa situação é fazer o applet executar como um thread de controle separado, suspendendo o thread quando o applet não estiver sendo exibido e retomando-o quando o applet for exibido novamente.

Você pode fazer isso observando que o método start( ) de um applet é chamado quando um applet é exibido pela primeira vez. Se o usuário deixar a página Web ou o applet sair da área visível da tela, o método stop( ) do applet é chamado. (Infelizmente, métodos start( ) e stop( ) diferentes são associados com threads c applets.) Se o usuário retornar à página Web do applet, o método start ( ) é chamado novamente. O método destroyt) de um applet é chamado quando o applet é removido do cache do navegador. E possível evitar que um applet execute enquanto não estiver sendo exibido em um navegador da Web fazendo que o método stop( ) do applet suspenda o thread e retomando a execução de um thread no método stort() do applet.

O applet ClockAppl et (mostrado na Figura 5.9) exibe a data e a hora em um navegador. Se o applet estiver sendo exibido pela primeira vez, o método start ( ) cria um novo thread; caso contrário, ele retoma a execução de um thread. O método stop( ) do applet suspende a execução de um thread. O método destroy( ) do applet interrompe a execução de um thread. O método run( ) do thread tem um laço infinito que alterna entre a suspen-

sãoporumsegundoeachamadaaométodorepaint( ). O método repaint( ) chamaométodopatnt( )queexibe a data na janela do navegador.

Com a versão Java 2, os métodos suspend( ),resume( )estop( ) tornaram-se obsoletos. Os métodos obsoletos ainda são implementados na API atua); no entanto, seu uso é desencorajado. Os motivos para tornar esses métodos obsoletos estão apresentados no Capítulo 8. Observe que, apesar de serem obsoletos, esses métodos ainda fornecem uma alternativa razoável (consulte a Figura 5.9, por exemplo).

```
import java.applet.";
irrport java.awt.*;
publlc class ClockApplet extends Applet
      implements Runnable
{
    public void run( ) {
        while (true) {
            try (
                Thread.sleep(lOOO);
            )
            catch (InterruptedException e) í }
            repaint( );
        I
    1
    public void start( ) (
        if ( clockThread •• null ) (
          clockThread • ne* Thread(this);
          clockThread.start( );
        I
        else
          clockThread.resume( );
    I
    public void stop( ) (
        if (clockThread I-null)
          clockThread.suspend( );
    )
    public void destroy( ) (
        if (clockThread !-nu11) (
          clockThread.stop{ );
          clockThread = null;
        )
    1
    public void paint(Graphics g) {
      g.drawString(new java.util.Oate( ).toString( ),10,30);
    I
    private Thread clockThread;
Í
```

Figura 5.9 Applet que exibe a data e hora do dia.

5.6.3 Estados de um thread

Um thread Java pode estar em um dos quatro estados possíveis a seguir:

1. *Novo:* Um thread está neste estado quando um objeto para o thread é criado (ou seja, a instrução new( )).
2. *Executável:* Chamar o método start( ) aloca memória para o novo thread na JVM e chama o método run( ) para o objeto thread. Quando o método run( ) de um thread é chamado, o thread passa do estado Novo para o estado Executável. Um thread no estado Executável pode ser executado pela JVM. Observe que Java não faz distinção entre um thread que é passível de execução e um thread que está sendo executado no momento pela JVM. Um thread em execução ainda está no estado Executável.

3. *Bloqueado:* Um thrcad torna-sc Bloqueado se executar uma instrução bloqueante, como ao realizar uma operação de I/O ou se chamar certos métodos Thread Java, como sleepí) ou suspend( ).
4. *Terminado:* Um thread passa para o estado terminado quando seu método run( ) termina ou quando o método stop( ) é chamado.

Não é possível definir o estado exato de um thread, embora o método isAlive( ) retorne um valor hooleano que um programa pode usar para determinar se um thread está ou não no estado terminado. A Figura 5.10 ilustra os diferentes estados de um thread c identifica várias transições possíveis.

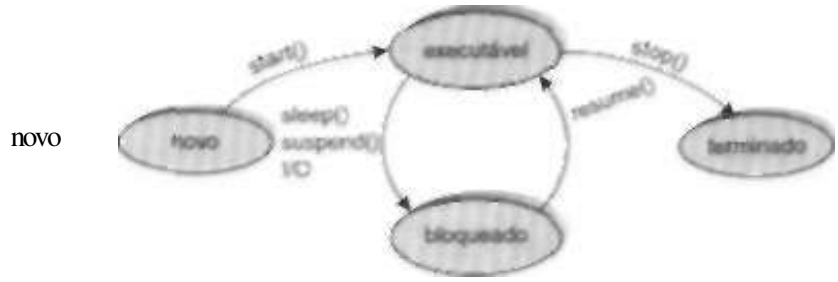


Figura 5.10 Estados de um thread Java.

5.6.4ThreadseaJVM

Além de um programa Java que contém vários thrcads de controle distintos, existem muitos threads executando assincronamente para a JVM que tratam de tarefas ao nível de sistema, tais como gerência de memória e controles gráficos. Esses threads incluem um thread de coleta de lixo, que avalia os objetos na JVM para verificar se ainda estão em uso. Se não estiverem, ele devolve a memória para o sistema. Dentre os outros threads está um que trata eventos de temporização, tais como chamadas ao método sleepí)> um que trata eventos dos controles gráficos, tais como pressionar um botão, e um que atualiza a tela. Assim, uma aplicação Java típica contém vários threads de controle diferentes na JVM. Certos threads são criados explicitamente pelo programa; outros threads são tarefas de nível de sistema executando em nome da JVM.

5.6.5 **A JVM** c **o** sistema **operacional host**

A implementação típica da JVM normalmente é feita sobre um sistema operacional host. Essa configuração permite que a JVM oculte os detalhes da implementação do sistema operacional subjacente e forneça um ambiente consistente e abstraio que permite aos programas Java operar em qualquer plataforma que suporte uma JVM. A especificação da JVM não indica como os threads Java serão mapeados no sistema operacional subjacente, deixando essa decisão para cada implementação particular da JVM. Em geral, um thread Java é considerado um thread de usuário, e a JVM é responsável pela gerência de threads. O Windows NT utiliza o modelo um-para-unv, portanto, cada thread Java para uma JVM executando no NT émapeado em um thread de kcrnel. O Solaris 2 inicialmente implementou a JVM usando o modelo muitos-para-um (chamado *threads verdes* pela Sun). No entanto, já na Versão 1.1 da JVM com o Solaris 2.6, a JVM foi implementada usando o modelo muitos-para-muitos.

5.6.6 **Exemplo de** solução com **multithread**

Nesta seção, apresentamos uma solução completa com multithread para o problema do produtor^consumidor que utiliza troca de mensagens. A classe Server na Figura 5.11 primeiro cria uma caixa de torreio para servir de buffer de mensagens, usando a classe MessageQueue desenvolvida no Capítulo 4. Em seguida, ela cria threads separados de produtor e consumidor (Figuras 5.12 e 5.13, respectivamente) e passa a cada thread uma referência à caixa de correio compartilhada. O thread produtor alterna entre o estado suspenso, a produção de um item e a inserção desse item na caixa de correio. O consumidor alterna vntre o estado suspenso c a busca de um item da caixa de correio, consumindo-o. Como o método receive( ) da classe MessageQueueé não-bloqueante, o consumidor deve verificar se a mensagem recuperada é nula.

```
public class Server
|
    public Server( ) {
        //primeiro cria o buffer de mensagens
        HessageQueue mailBox • new MessageQueue{ );

        //agora cria os threads de produtor e consumidor
        Producer producerThread • new Producer(mailBox);
        Consumer consumerlhread • new Consumer(mailBox);

        producerThread.start( );
        consumerThread.starl( );
    »

    public static void main(StrÍng args[ J) (
      Server server - new Server( J;
    I

    public static final int NAP_T1HE • 5;
)
```

Figura 5.11 A classe Server.

```
imoort java.util.";

clíss Producer extends Thread
<
    public Producer(HessageQueue m) (
      mbox - m;
    I

    public void run( ) {
      Date message;

      while (true) {
        int sleeptime » (int) (Server.NAP_TIME *
          Hath.random( ) );
        System.out.printlnfProducer sleeping for * +
          sleeptime + • seconds");
        try {
          Thread.sleep(sleeptime-lOOO);
        )
        catch{InterruptedException e) { )

        //produz um item e o insere
        //no buffer
        message • new Date( );
        System.out.println{"Producer produced • +
          message);
        mbox.send(message);
      |
    Í
    private MessageQueue mbox;
)
```

Figura 5.12 Thread do produtor.

5.7 • Resumo

Um thread é um fluxo de controle cm um processo. Um processo multilliread contém vários fluxos de controle distintos no mesmo espaço de endereçamento. Os benefícios do multithreading incluem maior capacidade de resposta ao usuário, compartilhamento de recursos no processo, economia c capacidade de aprovei-

lar as arquiteturas com múltiplos processadores. Os rhrcads de usuário são threads visíveis ao programador e desconhecidos do kernel. Alem disso, geralmente são gerenciados por uma biblioteca de threads no espaço de usuário. Os threads de kernel são suportados e gerenciados por uma kernel do sistema operacional. Em geral, os threads de usuário são mais rápidos de criar e gerenciar do que os threads de kernel. Existem três tipos diferentes de modelos relacionados os threads de usuário aos de kernel: o modelo muitos-para-um mapeia muitos threads de usuário em uni único thread de kernel. O modelo um-para-um mapeia cada thread de usuário em um thread de kernel correspondente. O modelo muitos-para-muitos multiplexa muitos threads de usuário em um número menor ou igual de threads de kernel.

Java é notável por fornecer suporte para threads DO nível da linguagem. Todos os programas Java consistem em pelo menos um thread de controle, sendo fácil criar vários threads de controle no mesmo programa. Java tamhcm fornece um conjunto de APIs para gerenciar threads, incluindo métodos para suspender e retomar threads, suspender um thread por determinado período de tempo e interromper um thread cm execução. Um thread Java também pode estar em um de quatro estados possíveis: Novo, Executável, Bloqueado c Terminado. As diferentes APIs para gerenciar threads geralmente mudam o estado do thread. Apresentamos como exemplo uma solução Java com multithread ao problema do produtor-consumidor.

```
import java.util.-;

class Consumer extends Thread
(
    public Consumer(MessageQueue m) |
      mbox = m;
    )

    public void run( ) (
      Date message;

      whilç (true) (
        int sleeptime • (int) (Server.NAPJIHE * Math.random( ));
        System.out.println("Consumer sleeping for • +
            sleeptime * " seconds");
        try (
            Thread.sleep(sleeptime*1000);
        í
        catch(InterruptetlException e) { )

        //consome um item do buffer
        message • (Date)mbox.receive( );

        if (message !• null)
          System.out.println("Consumer consumed " * message);
      í
  |

  private MessageQueue mbox;
í
```

Figura 5.13 Thread do consumidor.

Exercícios

5.1 Forneça dois exemplos de programação de multithreading com desempenho melhorado em relação a uma solução de thread único.

5.2 Forneça dois exemplos de programação de multithreading que *não* melhoram o desempenho em relação a uma solução de thread único.

5.3 Quais são as duas diferenças entre threads de usuário e de kernel? Em que circunstâncias um tipo é melhor do que o outro?

5.4 Descreva as ações tomadas por um kerncl para trocar contexto entre threads de kerncl.

5.5 Descreva as açôes tomadas por uma biblioteca de rhreads para trocar o contexto entre threads de usuário.

5.6 Que recursos sáo usados quando um thread é criado? Como eles diferem daqueles usados quando um processo é criado?

5.7 Modifique a classe MessageQueue de modo que o método receive( ) bloqueie até que haja uma mensagem disponível na fila.

5.8 Implemente os métodos send( ) ereceive( ) para um sistema de troca de mensagens que tenha capacidade zero. Implemente o método send ( ) de modo que o remetente bloqueie até que o receptor tenha aceito a mensagem. Implemente o método receive( ) de modo que o receptor bloqueie até que haja uma mensagem disponível na fila.

5.9 Implemente os métodos send ( ) ereceive( ) para um sistema de troca de mensagens que tenha capacidade limitada. Implemente o método send( ) de modo que o remetente bloqueie até que haja espaço disponível na fila. Implemente o método receive( ) de modo que o receptor bloqueie até que haja uma mensagem disponível na fila.

5.10 Escreva um programa Java com multithread que gere a série de Fibonacci. A operação desse programa deve ser a seguinte: o usuário executará o programa e entrará na linha de comandos quantos números de Fibonacci o programa deverá gerar. O programa criará então um thread separado que gerará os números de Fibonacci.

5.11 Escreva um programa Java com multithread que gere como saída números primos. Esse programa deve operar assim: o usuário executará o programa e entrará um numero na linha de comandos. O programa criará então um thread separado que gerará como saída todos os números primos menores ou iguais ao número que o usuário digitou.

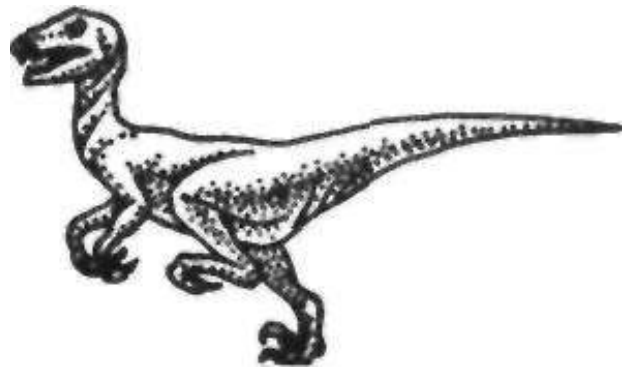
Notas bibliográficas

As questões relativas ao desempenho de threads foram discutidas por Anderson e colegas [1989), que continuaram seu trabalho 11991| avaliando o desempenho dos threads de usuário com suporte de kernel. Marsh e associados |1991] discutiram threads de usuário de primeira classe. Bcrshad c colegas [1990] descreveram a combinação de threads com RPC. Dravese colegas] 1991| discutiram o uso de continuações para implementar a gerência e a comunicação de threads nos sistemas operacionais.

O sistema operacional IBM OS/2 é um sistema com multithread que executa em computadores pessoais [Kogan e Rawson 1988]. A estrutura de threads do Solaris 2 da Sun Microsystems foi descrita por Eykhott e colegas [1992]. Os threads de usuário foram detalhados por Stcin e Shaw [1992]. Peacock 11992] discutiu o multithreading do sistema de arquivos no Solaris 2.

Informações sobre a programação com multithread são fornecidas em Lewis e Berg (1998], em Sunsoft (1995], e em Kleiman e colegas [1996], embora essas referencias tendam a favorecer Pthreads. Oakse Wong *[1999],* Lea'[1997] c Hartley (1998] discutem multithreading em Java. Solomon [1998| descreve como os threads são implementados no Windows NT; Beveridge e Wiener [1997) discutem multithreading usando Win32, e Pham e Garg |1996| descrevem a programação multithreading usando Windows NT. Vahalia [1996| e Graham (19951 descrevem como o Solaris implementa o multithreading.

Capítulo 6



ESCALONAMENTO DE CPU

O escalonamento de CPU é a base dos sistemas operacionais multiprogramados. Ao alternar a CPU entre os processos, o sistema operacional pode tornar o computador produtivo. Neste capítulo, apresentamos os conceitos básicos de escalonamento *e* vários algoritmos distintos de escalonamento de CPU. Além disso, consideramos o problema de selecionar um algoritmo para um determinado sistema. No Capítulo 5, apresentamos threads no modelo de processo. Nos sistemas operacionais que os suportam, são os threads do nível do kernel - e náo processos-que estão sendo escalonados pelo sistema operacional. No entanto, os termos escalonamento de processos e escalonamento de threads são muitas vezes usados indistintamente. Neste capítulo, usaremos *escalonamento de processos* quando estivermos discutindo conceitos gerais de escalonamento e *escalonamento de threads* para fazer referência a ideias específicas sobre threads.

6.1 • Conceitos básicos

O objetivo da multiprogramação é ter sempre algum processo cm execução para maximizar a utilização de CPU. Para um sistema uni processador, nunca haverá mais de um processo em execução. Se houver mais processos, o restante terá de esperar até que a CPU esteja livre e possa ser reescalonada.

A ideia da multipropramação *é* relativamente simples. Um processo é executado até ter de esperar, geralmeute-^e4axojiclujão^ejirr^pedido de lCX Em um sistema de computação simpJeí^a-CPU_fícarÍa_pciosa. t^doxssejempode esperaé pgnjido; nenhunurabalhsútil é realizado. Com a multiprogramação, essc_t£m-pají-usado de forma produtiva. Vários processos são mantidos na memória ao mesmo tempo. Quando um processo precisa esperar, o sistema operacional tira a CPU do processo e a passa para outro processo. Essepadrão continua: toda vez que um processo precisa esperar, outro pode presumir o uso da CPU.

O escalonamento é uma função fundamental do sistema operacional. Quase todos os recursos do computador são escalonados antes do uso. A CPU, é claro, é um dos principais recursos do computador. Assim, o escalonamento é central ao projero de sistemas operacionais.

6.1.1 Ciclo de surtos de CPU e I/O

0 sucesso do escalonamento da CPU depende da seguinte propriedade observada dos processos: a execução de um processo consiste em um ciclo de execução de CPU e espera de I/O. Os processos alternam entre esses dois estados. A execução de um processo começa com um surto de CPU. A isso se segue um surto de I/O que, por sua vez, é seguido por outro surto de CPU, depois outro surto de l/O e assim por diante. Por fim, o último surto de CPU termina com um pedido do sistema para encerrar a execução, em vez de com outro surro de l/O (Figura 6.1).

As durações desses surtos de CPU foram medidas exaustivamente. Embora possam variar muito de processo a processo e de computador a computador, tendem a ter uma curva de frequência semelhante àquela indicada na Figura 6.2. A curva geralmente é caracterizada como exponencial ou hiperexponencial. Existe um grande número de surtos de CPU curtos e um número pequeno de surtos de CPU longos. Um programa limi-

fado pela entrada/saída geralmente terá muitos surtos de CPU curtos. Um programa limitado pela CPU poderá ter alguns surtos de CPU longos. Essa distribuição pode ser importante na Seleção de um algoritmo adequado de escalonamento de CPU.

carregar valor da memória adicionar valor ler do arquivo	surto de CPU
espera por l/O	surto de 1/0
incrementar valor do índice gravar no arquivo	surto de CPU
espera por l/O	surto de l/O
carregar valor da memória adicionar valor ler do arquivo	surto de CPU
espera por l/O	surto de l/O

Figura 6.1 Sequência de troca de surtos de CPU c I/O.

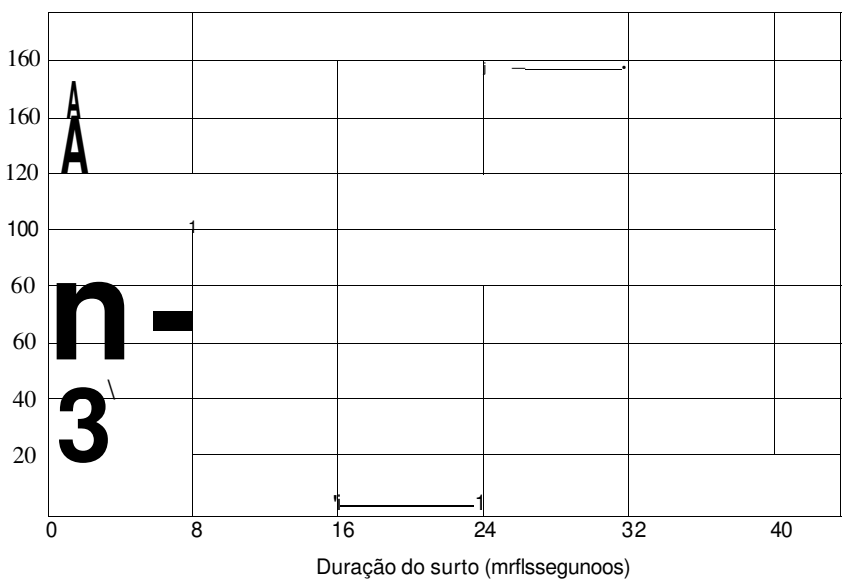


Figura 6.2 Histograma das durações de surto de CPU.

6.1.2 Escalonador de CPU

Sempre que a CPU ficar ociosa, o sistema operacional deverá escolher um dos processos para r\<vu$_v$u > da iil.i—— de processos prontos. O processo de Seleção é executado pelo escalonador de curto prazo (ou escalonador de CPU). O escalonador seleciona dentre os processos na memória aqueles que estão prontos para executar e aloca CPU a um deles.

Observe que a fila de processos prontos não é necessariamente uma fila FIFO (primeiro a entrar, primeiro a sair). Como veremos mais adiante ao tratarmos dos vários algoritmos de escalonamento, uma fila de processos prontos pode ser implementada como uma fila FIFO, uma fila de prioridades, uma árvore ou simplesmente uma lista ligada desordenada. Conceitualmente, no entanto, rodos os processos na fila de processos prontos estão à espera de uma chance na CPU. Os registros nas filas geralmente são os PCBs dos processos.

6.1.3 Escalonamento preemptivo

As decisões de escalonamento de CPU podem ocorrer em quatro circunstâncias:

1. Quando um processo passa do estado em execução para o estado de espera (por exemplo, um pedido de l/O ou chamada de espera para término de um dos processos filhos)
2. Quando um processo passa do estado em execução para o estado de pronto (por exemplo, quando ocorre uma interrupção)
3. Quando um processo passa do estado de espera para o estado de pronto (por exemplo, conclusão de l/O)
4. Quando um processo termina

Para os casos 1 e 4, não há opção em termos de escalonamento. Um novo processo (se houver na fila de processos prontos) deve ser selecionado para execução. Existe uma opção, no entanto, para os casos 2 e 3.

Quando o escalonamento ocorre apenas nos casos 1 e 4, dizemos que o esquema de escalonamento é não-preemptivo ou cooperativo; caso contrário, c preemptivo. No escalonamento não-preemptivo, depois que a CPU foi alocada a um processo, o processo mantém a CPU até liberá-la terminando ou passando para o estado de espera. Esse método de escalonamento foi usado pelo Microsoft Windows 3.x; o Windows 95 introduziu o escalonamento preemptivo. O sistema operacional Apple Macintosh introduziu o escalonamento preemptivo no MacOS 8 para a plataforma PowerPC. As versões anteriores do MacOS usavam escalonamento cooperativo. O escalonamento cooperativo é o único método que pode ser usado em determinadas plataformas de hardware, porque não requer hardware especial (por exemplo, um timer) necessário para o escalonamento preemptivo.

Infelizmente, o escalonamento preemptivo incorre em um custo associado com a coordenação de acesso aos dados compartilhados. Considere o caso de dois processos que compartilham dados. Enquanto um está atualizando os dados, ele é interrompido para que o segundo processo possa executar. O segundo processo tenta ler os dados, que estão em estado inconsistente. Assim, precisamos de novos mecanismos para coordenar o acesso a dados compartilhados; esse tema é tratado no Capítulo 7.

A preempção também tem um efeito no projeto do kernel do sistema operacional. Durante o processamento de uma chamada ao sistema, o kernel pode estar ocupado com uma atividade solicitada por um processo. Tais atívidades podem implicar na alteração dados importantes do kernel (por exemplo, filas de l/O). O que acontece se o processo for interrompido no meio dessas mudanças, c o kernel (ou o driver de dispositivo) precisar ler ou modificar a mesma estrutura? Seria o caos. Certos sistemas operacionais, incluindo a maior parte das versões do UNIX, lidam com esse problema esperando a conclusão de uma chamada ao sistema ou de um bloco de operações de I/O, antes de efetuar uma troca de contexto. Esse esquema garante que a estrutura do kernel seja simples, já que o kernel não interromperá um processo enquanto estruturas de dados do kernel estiverem em estado inconsistente. Infelizmente, esse modelo de execução do kernel não é sólido o bastante para dar suporte ao multiprocessamento e à computação de tempo real. Esses problemas, e suas soluções, estão descritos nas Seções 6.4 e 6.5.

No caso do UNIX, existem ainda seções do código em risco. Como as interrupções, por definição, podem ocorrer a qualquer momento, e como as interrupções nem sempre podem ser ignoradas pelo kernel, as seções de código afetadas pelas interrupções devem ser protegidas do uso simultâneo. O sistema operacional precisa aceitar interrupções praticamente o tempo todo; caso contrário, a entrada pode se perder ou a saída pode ser sobreposta. Para que essas seções de código não sejam acessadas de forma concorrente por vários processos, elas desabilitam as interrupções ao iniciar c as reabilitam ao terminar.

6.1.4 Dispatcher

Outro componente envolvido na função de escalonamento de CPU é o dispatcher (executor). O dispatcher c um módulo que dá controle da CPU ao processo selecionado pelo escalonador de curto prazo. Essa função envolve o seguinte:

- Troca de contexto
- Passar para o modo usuário
- Pular para a posição adequada no programa de usuário para reiniciar esse programa

O dispatcher deve ser o mais rápido possível, considerando que ele é chamado durante cada troca de processo. O tempo necessário para o dispatcher interromper um processo e iniciar a execução de outro é chamado de latência de dispatch.

6.2 • Critérios de escalonamento

Diferentes algoritmos de escalonamento têm diferentes propriedades e podem favorecer uma classe de processos mais que outra. Ao escolher que algoritmo usar em determinada situação, devemos considerar as diferentes propriedades dos vários algoritmos.

Muitos critérios foram sugeridos para comparar os algoritmos de escalonamento de CPU. As características usadas para comparação podem fazer diferença substancial na determinação do melhor algoritmo. Os critérios usados incluem:

- *Utilização de CPU:* A CPU deverá ficar o mais ocupada possível. A utilização da CPU pode variar de 0 a 100%. Em um sistema real, deverá variar de 40% (para um sistema não muito carregado) a 90% (para sistemas muito utilizados).
- *Throughput:* Sc a CPU estiver ocupada executando processos, o trabalho estará sendo feito. Uma medida de trabalho é o número de processos completados por unidade de tempo, denominado throughput. Para processos longos, essa taxa pode ser de um processo por hora; para transações curtas, o throughput pode ser de 10 processos por segundo.
- *Tempo de retorno:* Do ponto de vista de determinado processo, o critério importante é quanto tempo leva para executar esse processo. O intervalo entre a submissão de um processo até o seu tempo de conclusão é o tempo de retorno. Esse tempo é a soma dos períodos gastos esperando para acessar a memória, aguardando na fila de processos prontos, executando na CPU e realizando operações de entrada/saída.
- *Tempo de espera:* O algoritmo de escalonamento de CPU não afeta a quantidade de tempo durante a qual determinado processo executa ou efetua I/O; afeta apenas o tempo que um processo gasta esperando na fila de processos prontos. O tempo de espera é a soma dos períodos gastos esperando na fila de processos prontos.
- *Tempo de resposta:* Em um sistema interativo, o tempo de retorno pode não ser o melhor critério. Em geral, um processo pode produzir algum resultado logo de início c pode continuar computando novos resultados enquanto os anteriores estiverem sendo repassadas para o usuário. Assim, outra medida é o tempo entre a submissão de um pedido até a primeira resposta ser produzida. Essa medida, chamada de tempo de resposta, é 0 tempo que O processo leva para começar a responder, mas não é o tempo que leva para gerar a resposta. O tempo de resposta geralmente é limitado pela velocidade do dispositivo de saída.

E desejável maximizar a utilização de CPU e o throughput, e minimizar o tempo de retorno, o tempo de espera e o tempo de resposta. Em muitos casos, otimizamos a medida média. No entanto, existem casos em que é desejável otimizar os valores mínimos ou máximos, cm vez do valor médio. Por exemplo, para garantir que todos os usuários obtenham um bom serviço, talvez seja necessário minimizar o tempo de resposta máximo.

Pesquisadores sugeriram que, para sistemas interativos (como os de tempo compartilhado), é mais importante minimizar a variância no tempo de resposta do que minimizar o tempo de resposta médio. Um sistema com um tempo de resposta razoável e previsível pode ser considerado mais desejável do que um sistema

que é mais rápido em média, mas é altamente variável. No entanto, pouco trabalho foi realizado em algoritmos de escalonamento de CPU que minimizem a variância.

Ao discutirmos vários algoritmos de escalonamento de CPU, vamos ilustrar sua operação. Um quadro preciso deve envolver muitos processos, cada qual sendo uma sequencias de várias centenas de surtos de CPU e surtos de l/O. Para fins de simplicidade de ilustração, consideramos apenas um surto de CPU (em milissegundos) por processo nos nossos exemplos. Nossa medida de comparação é* o tempo de espera médio. Mecanismos de avaliação mais elaborados são discutidos na Scção 6.8.

6.3 • Algoritmos de escalonamento

O escalonamento de CPU lida com o problema de decidir a quais processos na fila de processos prontos a CPU deverá ser alocada. Existem muitos algoritmos diferentes de escalonamento de CPU. Nesta seçáo, vamos descrever vários desses algoritmos.

6.3.1 Escalonamento first-come, first-served

O algoritmo de escalonamento mais simples é de longe o algoritmo de escalonamento first-come, first-served (FCFS). Nesse esquema, o processo que solicita a CPU primeiro, a recebe primeiro. A implementação da política FCFS é facilmente gerenciada com uma fila **FIFO.** Quando um processo entra na fila de processos prontos, seu PCB é ligado ao final da fila. Quando a CPU é liberada, ela é alocada ao processo no início da fila. O processo em execução é então removido da fila. O código para escalonamento FCFS é simples de escrever e entender.

O tempo de espera médio com a política FCFS, no entanto, muitas vezes é bem longo. Considere o seguinte conjunto de processos que chegam no instante 0, com a duração de surto de CPU expressa em milissegundos:

Processo	Duração de surto
P,	24
P_2	3
P_s	3

Se os processos chegarem na ordem P,, P, e $_{P,,}$ e forem atendidos na ordem FCFS, teremos o resultado apresentado no seguinte diagrama de Gantt:

O tempo de espirra é 0 milissegundo para o processo P, 24 milissegundos para o processo P,e 27 milissegundos para o processo P^ Assim, o tempo de espera médio é (0 + 24 + 27)/3= 17 milissegundos. Entretanto, se os processos chegarem na ordem P_2, Pj e P],os resultados serão apresentados no seguinte diagrama de Gantt:

O tempo de espera médio é agora (6 + 0 + 3)/3 = 3 milissegundos. Essa redução é substancial. Assim, o tempo de espera médio em uma política FCFS geralmente não é mínimo, e pode variar substancialmente se os tempos de surto de CPU do processo variarem muito.

Além disso, considere o desempenho do escalonamento FCFS em uma situação dinâmica. Vamos supor que tenhamos um processo limitado pela CPU e muitos processos limitados por l/O. A medida que os processos fluem pelo sistema, a seguinte situação poderá ocorrer. O processo limitado pela CPU obterá e manterá a CPU. Durante esse tempo, todos os demais processos terminarão sua operação de entrada/saída e passarão para a fila de processos prontos, esperando pela CPU. Enquanto os processos esperam na fila de processos prontos, os dispositivos de l/O estão ociosos. Por fim, o processo limitado pela CPU termina seu surto de CPU e passa para um dispositivo de l/O. Todos os processos limitados por I/O, que têm surtos de CPU curtos, executam rapidamente e voltam para as filas de l/O. Nesse ponto, a CPU fica ociosa. O processo limitado pela CPU passará então para a fila de processos prontos e receberá a CPU. Mais uma vez, todos os processos de I/O acabam esperando na fila de processos prontos até que o processo limitado pela CPU seja realizado. Existe um efeito comboio, enquanto todos os outros processos esperam que um grande processo saia da CPU. Esse efeito resulta em menor utilização de CPU e dispositivos que o possível, caso os processos mais curtos pudessem ser atendidos primeiro.

O algoritmo de escalonamento FCFS é não-preemptivo. Assim que a CPU tiver sido alocada a um processo, esse processo mantém a CPU até liberá-la, terminando ou realizando um pedido de l/O. O algoritmo FCFS é particularmente problemático para sistemas de tempo compartilhado, onde é importante que cada usuário tenha uma parte da CPU em intervalos regulares. Seria desastroso permitir que um processo ficasse com a CPU por um período prolongado.

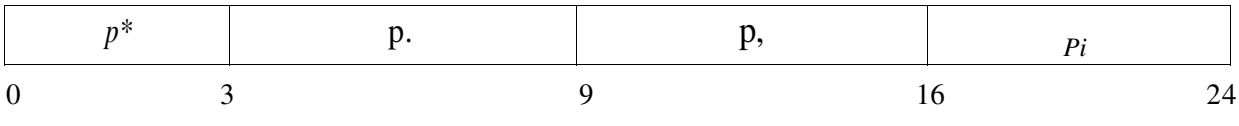
6.3.2 Escalonamento job mais curto primeiro

Uma abordagem diferente ao escalonamento de CPU é o algoritmo job mais curto primeiro (SJF). Esse algoritmo associa a cada processo a duração de seu próximo surto de CPU. Quando a CPU está disponível, ela é atribuída ao processo que tem o próximo surto de CPU menor. Se dois processos tiverem seus próximos surtos de CPU de mesma duração, o escalonamento FCFS será usado para desempate. Observe que um termo mais apropriado seria próximo surto de CPU mais curto, porque o escalonamento é feito examinando-se a duração do próximo surto de CPU de um processo, em vez de sua duração total. Usamos o termo SJF porque muitas pessoas e livros didáticos referem-se a esse tipo de disciplina de escalonamento como SJF.

Como exemplo, considere o seguinte conjunto de processos, com a duração do surto de CPU expressa em milissegundos:

Processo	Duração de surto
P,	6
Pz	8
P,	*7*
P>	3

Usando o escalonamento SJF, poderíamos escalonar esses processos de acordo com o seguinte diagrama de Gantt:



O tempo de espera é 3 milissegundos para o processo /*„ 16 milissegundos para o processo *P^ 9* milissegundos para o processo F, eO mi lissegundo para o processo P_4. Assim, o tempo de espera médio é (3 + 16 +9 + 0)/4 = 7 milissegundos. Se estivéssemos usando o esquema de escalonamento FCFS, o tempo de espera médio seria 10,25 milissegundos.

O algoritmo de escalonamento SJF é comprovadamente *átimo,* pois fornece o tempo médio de espera mínimo para um determinado conjunto de processos. Quando um processo curto é colocado antes de um

processo longo, o tempo de espera do processo curro diminui mais do que aumenta o tempo de espera do processo longo. Consequentemente, o tempo de espera *médio* diminui.

A verdadeira dificuldade com o algoritmo SJF *6* conhecer a duração do próximo pedido de CPU. Para o escalonamento de longo prazo de jobs em um sistema em batcli, podemos usar como duração o limite de tempo do processo que um determinado usuário especifica quando submete o job. Assim, os usuários ficam motivados a estimar o limite de tempo do processo com precisão, já que um valor menor pode significar resposta mais rápida. (Um valor baixo demais poderá causar um erro de estouro do limite de tempo e exigir nova submissão.) O escalonamento SJF é usado frequentemente no escalonamento de longo prazo.

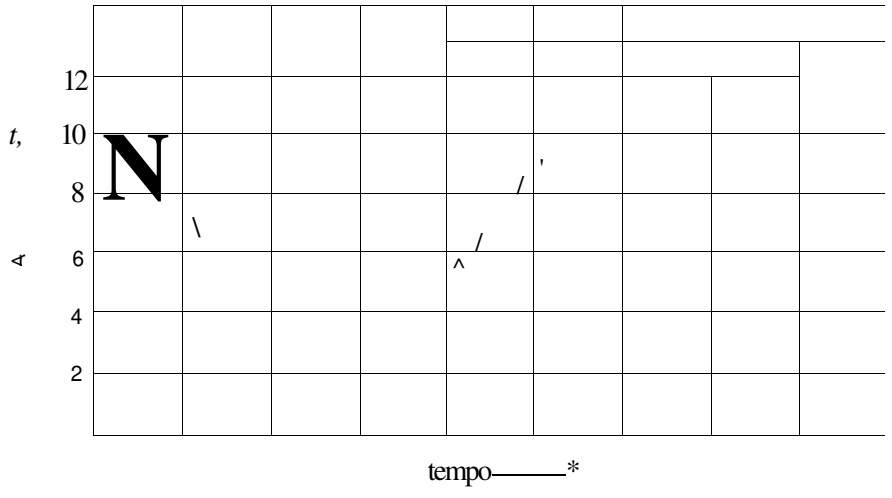
Embora o algoritmo SJF seja ótimo, ele não pode ser implementado no nível do escalonamento de CPU de curto prazo. Não existe como saber a duração do próximo surto de CPU. Uma abordagem é tentar se aproximar do escalonamento SJF. Talvez não seja possível *saber* a duração do próximo surto de CPU, mas podemos tentar *prever* o seu valor. Esperamos que o próximo surto de CPU seja semelhante em duração aos anteriores. Assim, ao calcular uma aproximação da duração do próximo surto de CPU, podemos selecionar o processo com o menor surto de CPU previsto.

O próximo surto de CPU geralmente é previsto como uma média exponencial das durações medidas dos surtos de CPU anteriores. Seja t„ a duração do enésimo surto de CPU e r„+, nosso valor previsto para o próximo surto. Em seguida, para a , 0 S a S 1 , define-se

Essa fórmula define uma média exponencial. O valor de t„contém nossa informação mais recente; ^armazena o histórico. O parâmetro *a* controla o peso relativo do histórico recente e passado na nossa previsão. Sc a = 0 , r„, , = *T„e* c> histórico recente não tem efeito (as condições atuais são consideradas transitórias); se *a* — 1, então r„. , — *t„e* apenas o surto de CPU mais recente importa (o histórico é considerado velho e irrelevante). Mais comumenre, *a* = *Vi,* de modo que os históricos recente e passado têm peso igual. O r„ inicial pode ser definido como uma constante ou como uma média geral do sistema. A Figura 6.3 mostra uma média exponencial com *a* = *Vz c* x_0 = 10.

Para entender o comportamento da média exponencial, expandimos a fórmula para r„ , , substituindo *x„* para encontrar

$$r_{n+1} = at_n + (1-a)at_{n-1} + \dots + (1-a)^j at_{n-j} + \dots + (1-a)^{n+1} x_0$$



surto *de* CPUty	6	4	6	4	13	13	13
"previsão"*!,) 10	8	6	6	5	9	11	12

Figura 6.3 Previsão da duração do próximo surto de CPU.

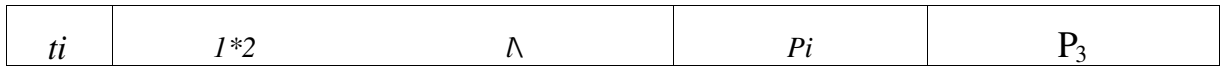
Como *a* e (1 *-a)* são menores ou iguais a 1, cada termo sucessivo tem menos peso do que seu precedente.

O algoritmo SJF pode ser preemptivo ou não-preetnptivo. A opção surge quando um novo processo chega na fila de processos prontos enquanto um processo anterior está em execução. O novo processo pode ter seu próximo surto de CPU mais curto do que o que restou do processo em execução no momento. Um algoritmo SJF preemptivo interromperá o processo em execução no momento, enquanto um algoritmo não-preemptivo permitirá que o processo em execução no momento termine seu surto de CPU. O escalonamento SJF preemptivo às vezes e* chamado escalonamento menor tempo restante primeiro.

Como exemplo, considere os quatro processos seguintes, com a duração do surto de CPU expressa em milissegundos:

Processo	Instante de chegada	Duração de surto
P,	0	8
P,	1	4
P_i	*2*	9
P_4,	3	5

Se os processos chegarem na fila de processos prontos nos tempos indicados e precisarem dos tempos de surto a seguir, o escalonamento SJF preemptivo resultante será o apresentado no seguinte diagrama de Gantt:



O processo /*, ê iniciado no instante 0, já que é o único processo da fila. O processo *P,* chega no instante 1. O tempo restante para o processo /*, (7 milissegundos) c maior do que o tempo exigido pelo processo P_2 (4 milissegundos), de modo que o processo /*, é interrompido e o processo P_1 é escalonado. O tempo de espera médio para esse exemplo é **((10-1)** + **(1- l)+(17-2)** + (5-3))/4 = **26/4** = 6,5 milissegundos. Um escalonamento SJF não-preemptivo resultaria em um rempo de espera médio de 7,75 milissegundos.

6.3.3 Escalonamento **por** prioridade

O algoritmo SJF é um caso especial do algoritmo geral de escalonamento por prioridade. Uma prioridade está associada a cada processo, c a CPU é alocada ao processo com prioridade mais alta. Processos de prioridade igual são escalonados na ordem FCFS.

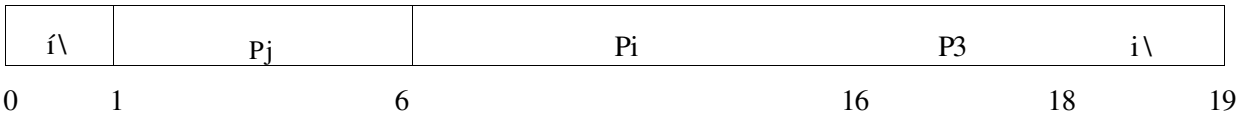
Um algoritmo SJF nada mais é do que um algoritmo por prioridade no qual a prioridade (/>) é 0 inverso do próximo surto de CPU (previsto). Quanto maior o surto, menor a prioridade, e vice-versa.

Observe que discutimos o escalonamento em termos de *alta* e *baixa* prioridade. As prioridades em geral estão em uma faixa fixa de números, tais como 0 a 7, ou 0 a 4095. No entanto, não existe consenso geral sobre se 0 é a prioridade mais alta ou mais baixa. Alguns sistemas usam números baixos para representar baixa prioridade; outros usam números baixos para alta prioridade. Essa diferença pode levar à confusão. Neste texto, consideramos que os números baixos representam alta prioridade.

Como um exemplo, considere o seguinte conjunto de processos, que têm seu instante de chegada definido em 0, na ordem /', *P& ...*, P_5> com a duração do surto de CPU expressa cm milissegundos:

Processo	Duração do surto	Prioridade
P,	10	3
P,	1	1
P_t	2	4
P_4	1	5
P_s	5	2

Usando o escalonamento por prioridade, esses processos seriam escalonados de acordo com o seguinte diagrama de Gantt:



O tempo de espera médio é de 8,2 milissegundos.

As prioridades podem ser definidas interna ou externamente. As prioridades definidas internamente utilizam alguma quantidade ou quantidades mensuráveis a fim de calcular a prioridade de um processo. Por exemplo, os limites de tempo, requisitos de memória, o número de arquivos abertos e a razão entre o surto de I/O médio c o surto de CPU médio tem sido usados no cálculo das prioridades. As prioridades externas são definidas por critérios que são externos ao sistema operacional, tais como a importância do processo, o tipo e a quantidade de fundos pagos para uso do computador, o departamento patrocinando o trabalho c outros fatores, geralmente políticos.

O escalonamento por prioridade pode ser preemptivo ou não-preemptivo. Quando um processo chega na fila de processos prontos, sua prioridade é comparada com a prioridade do processo em execução no momento. Um algoritmo de escalonamento por prioridade preemptivo interromperá a CPU se a prioridade do processo recem-chegado for maior do que a prioridade do processo em execução no momento. Um algoritmo de escalonamento por prioridade não-preemptivo simplesmente colocará o novo processo no topo da fila de processos prontos.

J* Um problema crítico com os algoritmos de escalonamento por prioridade é o bloqueio por tempo indefinido ou starvation (estagnação). Um processo que está pronto para executar mas que não tem a CPU pode ser considerado bloqueado, esperando pela CPU. Um algoritmo de escalonamento por prioridade pode deixar que alguns processos de baixa prioridade fiquem esperando indefinidamente pela CPU. Em um sistema de computação altamente carregado, um fluxo constante de processos de maior prioridade pode impedir que um processo de baixa prioridade obtenha a CPU. Geralmente, ocorre uma de duas opções. O processo acabará sendo executado (às *Z* horas da manhã de domingo, quando o sistema finalmente ficar com menos carga) 0l1 o sistema de computador acabará falhando e perderá todos os processos de baixa prioridade não terminados. (Dizem, inclusive, que quando o IBM 7094 do M1T foi desligado em 1973, encontraram um processo de baixa prioridade que tinha sido submetido em 1967 e que ainda não tinha sido executado.)

Uma solução para o problema do bloqueio indefinido de processos de baixa prioridade é o envelhecimento (aging). O envelhecimento é uma técnica para aumentar gradualmente a prioridade dos processos que ficam esperando no sistema durante muito tempo. Por exemplo, se as prioridades tiverem uma faixa de 127 (baixa) a 0 (alta), poderíamos aumentar a prioridade de um processo em espera em 1 ponto a cada 15 minutos. Por fim, mesmo um processo com prioridade inicial de 127 teria a maior prioridade no sistema e poderia ser executado. Na verdade, bastariam 32 horas para um processo de prioridade 127 passar para um processo de prioridade 0.

6.3.4 Escalonamento Round-Robin

O algoritmo de escalonamento Round-Robin (RR) (revezamento circular) foi projetado especialmente para sistemas de tempo compartilhado. E semelhante ao escalonamento FCFS, mas a preempção é acrescentada para alternar entre processos. Uma pequena unidade de tempo, chamada quantum de tempo, ou fatia de tempo, é definida. Um quantum geralmente é de 10 a 100 milissegundos. A fila de processos prontos é tratada como uma fila circular. O escalonador de CPU percorre a fila de processos prontos, alocando a CPU a cada processo por um intervalo de tempo de até 1 quantum de tempo.

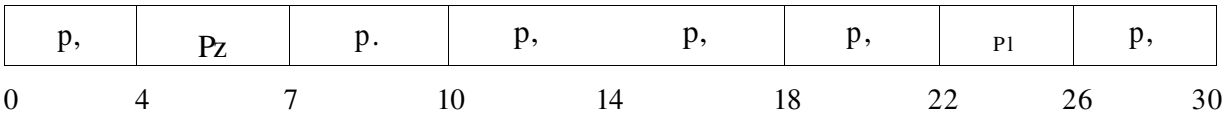
Para implementar o escalonamento Round-Robin, a fila de processos prontos é mantida como uma fila FIFO dos processos. Novos processos são adicionados ao final da fila. O escalonador de CPU seleciona o primeiro processo da fila, define um temporizador para interromper depois de 1 quantum e submete o processo.

Neste momento, duas opções podem ocorrer. O processo poderá ter um surto de CPU de menos de 1 quantum. Nesse caso, o próprio processo liberará a CPU voluntariamente. O escalonador procederá então para o próximo processo na fila de processos prontos. Caso contrário, se o surto de CPU do processo em execução no momento for maior do que 1 quantum de tempo, o temporizador se esgotará e causará uma interrupção para o sistema operacional. Uma troca de contexto será executada, e o processo será colocado no final da fila de processos prontos. O escalonador de CPU selecionará então o próximo processo na fila.

O tempo de espera médio na política RR, no entanto, é geralmente longo. Considere o seguinte conjunto de processos que chegam no instante 0, com duração de surto de CPU expressa em milissegundos:

Processo	Duração de surto
P,	24
P_2	3
f_3	3

Se usarmos um quantum de 4 milissegundos, o processo P_l obtém os 4 primeiros milissegundos. Como mais 20 milissegundos são necessários, ele será interrompido depois do primeiro quantum de tempo, c a CPU será passada ao próximo processo na fila, o processo P>. Como o processo *P,* não precisa de 4 milissegundos, ele encerra antes que o seu quantum expire. A CPU é então passada para o próximo processo, o processo P_y Assim que cada processo tiver recebido 1 quantum, a CPU é retornada ao processo P_1 para um quantum adicional. O escalonamento Round-Robin resultante é:



O tempo de espera médio é 17/3 = 5,66 milissegundos.

No algoritmo de escalonamento Round-Robin (RR), nenhum processo recebe CPU por mais do que 1 quantum de tempo consecutivo (a menos que seja o único processo pronto). Se o surto de CPU de um processo exceder 1 quantum, esse processo será interrompido e colocado de volta na fila de processos prontos. O algoritmo de Round-Robin (RR) é preemptivo.

\ Se houver *n* processos na fila de processos prontos e o quantum for *q,* então cada processo obterá *Mn* do tempo de CPU cm lotes de no máximo 4 unidades de tempo. Cada processo deve esperar no máximo (« - 1) x *q* unidades de tempo até seu próximo quanrum. Por exemplo, se houver cinco processos com um quantum de tempo de 20 milissegundos, cada processo obterá até 20 milissegundos a cada 100 milissegundos.

O desempenho do algoritmo RR depende muito do tamanho do quantum de tempo. Por um lado, se o quantum for extremamente grande, a regra de RR será a mesma que a regra FCFS. Sc o quantum de tempo for extremamente pequeno (por exemplo, 1 microssegundo), a abordagem RR será chamada de compartilhamento de processador c, para os usuários, é como se em teoria cada um dos *n* processos tivesse seu próprio processador executando a l/« da velocidade do processador real. A abordagem foi usada no hardware da Control Data Corporation (CDC) para implementar 10 processadores periféricos com apenas um conjunto de hardware e 10 conjuntos de registradores. O hardware executa uma instrução para um conjunto de registradores c passa para o próximo. Esse ciclo continua, resultando em 10 processadores lentos cm vez de um rápido. (Na verdade, como o processador era muito mais rápido do que a memória c cada instrução referenciava a memória, os processadores não eram muito mais lentos do que 10 processadores reais teriam sido.)

No software, no entanto, também precisamos considerar o efeito da troca de contexto no desempenho do escalonamento RR. Vamos supor que temos apenas um processo com 10 unidades de tempo. Se o quantum for 12 unidades de tempo, o processo terminará em menos de 1 quanrum, sem custo adicional. Sc o quantum for 6 unidades de tempo, no entanto, o processo exigirá 2 quanta, resultando em uma troca de contexto. Se o quantum for 1 unidade de tempo, nove trocas de contexto ocorrerão, tornando a execução do processo mais lenta (Figura 6.4).

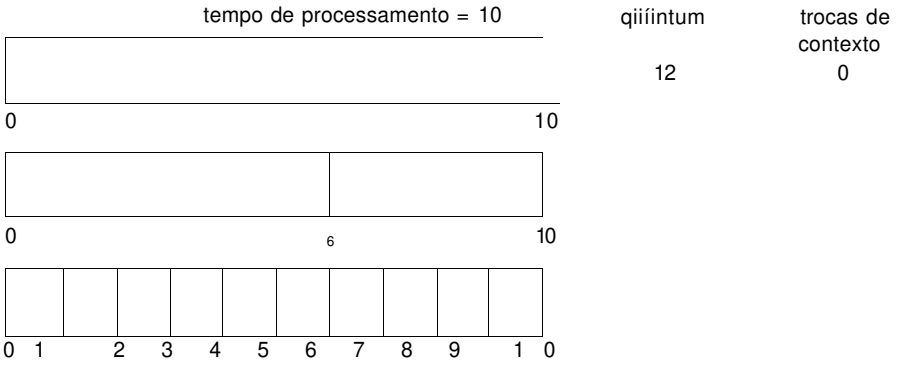
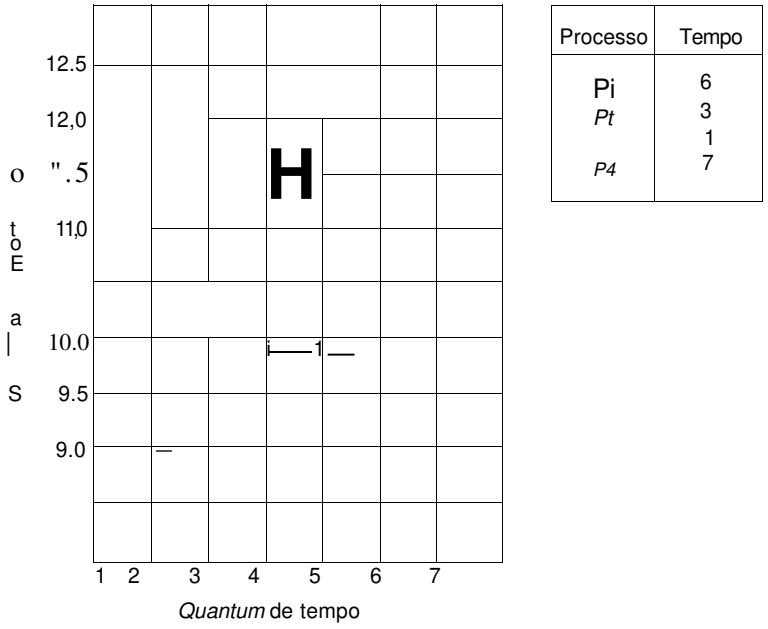


Figura 6.4 A forma como um quantum de tempo menor aumenta as trocas de contexto.

Assim, o quantum deve ser grande com relação ao tempo de troca de contexto. Se o tempo de troca for de cerca de 10% do quantum, então cerca de 10% do tempo da CPU será gasto em troca de contexto.

O tempo de retorno também depende do tamanho do quantum. Como podemos ver na Figura 6.5, o tempo de retorno médio de um conjunto de processos não melhora necessariamente à medida que o quantum aumenta. Em gerai, o tempo de retorno médio pode ser melhorado se a maioria dos processos terminar seu próximo surto de CPU em um único quantum. Por exemplo, dados três processos com 10 unidades de tempo cada e um quantum de 1 unidade de tempo, o tempo de retorno médio é 29. Se o quantum for 10, no entanto, o tempo de retorno cairá para 20. Se o tempo de troca de contexto for acrescentado, o tempo médio aumentará para um quantum menor, já que um número maior de trocas de contexto será necessário.



Processo	Tempo
Pi	6
Pt	3
	1
P4	7

Figura 6.5 A forma como o tempo de retorno varia com a duração do quantum.

Por outro lado, se o quantum for muito grande, o Round-Robin degenera para a política FCFS. Uma regra geral é que 80% dos surtos de CPU devem ser menores do que o quantum de tempo.

6.3.5 Escalonamento por múltiplas filas

Outra classe de algoritmos de escalonamento foi criada para situações em que os processos são facilmente classificados em diferentes grupos. Por exemplo, uma divisão comum é feita entre processos de primeiro plano (interativos) e processos de segundo plano (batch). Esses dois tipos de processos têm diferentes requisitos de tempo de resposta c, por isso, podem ter diferentes necessidades de escalonamento. Além disso, os processos de primeiro plano podem ter prioridade (externamente definida) em relação aos processos de segundo plano.

Um algoritmo de escalonamento por múltiplas filas divide a fila de processos prontos em várias filas separadas (Figura 6.6). Os processos são permanentemente atribuídos a uma fila, geralmente com base em alguma propriedade do processo, tais como tamanbo da memória, prioridade do processo ou tipo de processo. Cada fila tem seu próprio algoritmo de escalonamento. Por exemplo, filas separadas podem ser usadas para processos de primeiro e segundo planos. A fila de primeiro plano pode ser escalonada por um algoritmo RR, enquanto a fila de segundo piano é escalonada por um algoritmo FCFS.

Além disso, deve haver escalonamento entre as filas, que é normalmente implementado como escalonamento preemptivo de prioridade fixa. Por exemplo, a fila de primeiro plano pode ter prioridade absoluta sobre a fila de segundo plano.

Vamos analisar um exemplo de um algoritmo de escalonamento por múltiplas filas com cinco filas:

1. Processos do sistema
2. Processos interativos
3. Processos de edição interativa
4. Processos cm batch
5. Processos secundários

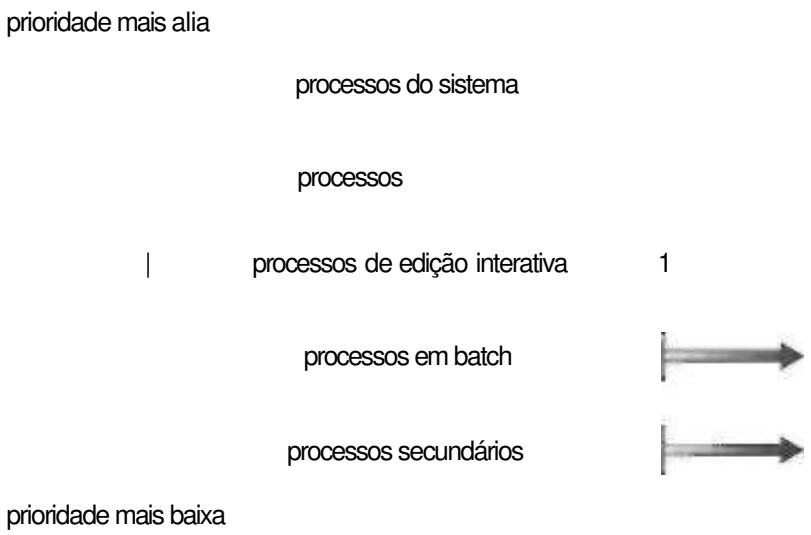


Figura 6.6 Escalonamento por múltiplas filas.

Cada fila tem prioridade absoluta sobre as filas de menor prioridade. Nenhum processo na fila batch, por exemplo, poderia executar a menos que as filas para os processos do sistema, processos interativos e processos de edição interativa estivessem todas vazias. Sc um processo de edição interativa entrasse na fila de processos prontos enquanto um processo em batch estivesse executando, o processo em batch seria interrompido.

Outra possibilidade é fracionar o tempo entre as filas. Cada fila obtém uma certa parte do tempo de CPU, que pode então ser escalonado entre os vários processos na fila. Por exemplo, no exemplo de fila de primeiro e segundo planos, a fila de primeiro plano pode receber 80% do tempo de CPU para o escalonamento Round-Robin (RR) entre seus processos, enquanto a fila de segundo plano recebe 20% da CPU para dar aos seus processos de modo FCFS.

6.3.6 Escalonamento por múltiplas filas com realimentação

Normalmente, em um algoritmo de escalonamento por múltiplas filas, os processos são permanentemente atribuídos a uma fila ao entrar no sistema. Os processos não se movem entre as filas. Se houver filas separadas para processos de primeiro e segundo planos, por exemplo, os processos não passam de uma fila para outra, já que não mudam sua natureza de primeiro ou segundo plano. Essa configuração tem a vantagem de apresentar baixo custo de escalonamento, mas é inflexível.

O escalonamento por múltiplas filas com realimentação, no entanto, permite que um processo se mova entre as filas. A ideia é separar os processos com diferentes características de surto de CPU. Se um processo

usar tempo de CPU excessivo, será movido para uma fila de menor prioridade. Esse esquema deixa os processo limitados por entrada/saída e interaf ivos nas filas de prioridade mais alta. Da mesma forma, um processo que espera demais em uma fila de baixa prioridade pode ser passado para uma fila de maior prioridade. Essa forma de envelhecimento evita a estagnação.

Por exemplo, considere um escalonador por múltiplas filas com realimentação que tenha três filas, numeradas de 0 a 2 (Figura 6.7). O escalonador primeiro executa todos os processos na fila 0. Somente quando a fila 0 estiver vazia, ela executará os processos na fila 1. Da mesma forma, os processos na fila 2 só serão executados se as filas 0 e 1 estiverem vazias. Um processo que chega na fila 1 interromperá um processo na fila *2.* Um processo na fila 1, por sua vez, será interrompido por um processo que chega na fila 0.

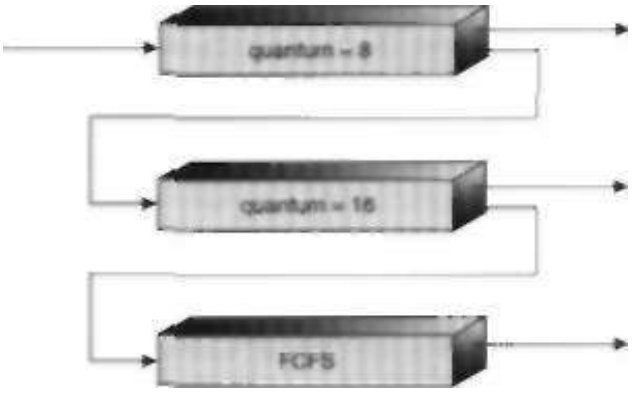


Figura 6.7 Escalonamento por múltiplas filas com realimentação.

Um processo que entra na fila de processos prontos é colocado na fila 0. Um processo na fila 0 recebe um quantum de 8 milissegundos. Se não terminar dentro desse prazo, ele é passado para o final da fila 1. Se a fila 0 estiver vazia, o processo no início da fila 1 recebe um quantum de 16 milissegundos. Se não completar, cie será interrompido e colocado na fila 2. Os processos na fila 2 são executados em modo FCFS, apenas quando as filas 0 e 1 ficam vazias.

Esse algoritmo de escalonamento fornece prioridade mais alta a qualquer processo com um surto de CPU de 8 milissegundos ou menos. Tal processo obterá CPU rapidamente, terminará seu surto de CPU e passará para o próximo surto de I/O. Os processos que precisam de mais de 8, mas menos de 24 milissegundos, também são atendidos rapidamente, embora com menor prioridade do que os processos mais curros. Os processos mais longos automaticamente caem para fila 2 e são servidos na ordem FCFS com quaisquer ciclos de CPU deixados das filas 0 e 1.

Em geral, um escalonador por múltiplas filas com realimentação é definido pelos seguintes parâmetros:

- O número de filas
- O algoritmo Je escalonamento para cada fila
- O método usado para determinar quando promover um processo para uma fila de maior prioridade
- O método usado para determinar quando rebaixar um processo para uma fila de menor prioridade
- O método usado para determinar em que fila um processo entrará quando esse processo precisar de serviço

A definição de um escalonador por múltiplas filas com realimentação torna o algoritmo de escalonamento de CPU mais geral. Pode ser configurado para corresponder a um sistema específico cm desenvolvimento. Infelizmente, requer algum meio segundo o qual seja possível selecionar valores para todos os parâmetros para definir o melhor escalonador. Embora uma fila multinível com realimentação seja o esquema mais geral, também c o mais complexo.

6.4 • Escalonamento com múltiplos processadores

Nossa discussão até agora enfocou os problemas do escalonamento da CPU em um sistema com um único processador. Se várias CPUs estiverem disponíveis, o problema do escalonamento será proporcionalmente

mais complexo. Muitas possibilidades foram testadas e, como vimos com 0 escalonamento de CPU de um único processador, não existe uma solução melhor. Discutimos rapidamente as preocupações com o escalonamento com múltiplos processadores. Nosso foco é em sistemas nos quais os processadores são idênticos - homogéneos - em termos de funcionalidade; podemos usar qualquer processador disponível para executar quaisquer processos na fila.

Mesmo com multiprocessadores homogéneos, às vezes existem limitações no escalonamento. Considere um sistema com um dispositivo de I/O conectado a um barramento privado de um processador. Os processos que desejarem utilizar esse dispositivo devem ser escalonados para executar naquele processador.

Se vários processadores idênticos estiverem disponíveis, então poderá haver compartilhamento de carga. Poderíamos definir uma fila separada para cada processador. Nesse caso, no entanto, um processador poderia ficar ocioso, com uma fila vazia, enquanto outro processador estaria extremamente ocupado. Para evitar essa situação, usamos uma fila de processos prontos comum. Todos os processos vão para uma fila e são escalonados para qualquer processador disponível.

Em um esquema como esse, uma de duas abordagens pode ser usada. Uma abordagem utiliza o multiprocessamento simétrico (SMP), no qual cada processador faz seu próprio escalonamento. Cada processador examina a fila de processos prontos comum e selcciona um processo para executar. Como veremos no Capítulo 7, se tivermos vários processadores tentando acessar e atualizar uma estrutura de dados comum, cada processador deverá ser programado com cuidado: devemos garantir que dois processadores não escolham o mesmo processo, c que processos não sejam perdidos da fila.

Alguns sistemas levam essa estrutura um passo adiante, fazendo com que todas as decisões de escalonamento, processamento de l/O e outras atividades do sistema sejam tratadas por um único processador - o processador mestre. Os outros processadores executam apenas código de usuário. Esse multiprocessamento assimétrico é muito mais simples do que o multiprocessamento simétrico, porque apenas um processador acessa as estruturas de dados do sistema, aliviando a necessidade de compartilhamento de dados.

6.5 • Escalonamento de tempo real

No Capítulo 1, discutimos a crescente importância dos sistemas operacionais de tempo real. Aqui, vamos descrever o recurso de escalonamento necessário para dar suporte à computação de tempo real em um sistema de computador de uso geral.

Existem dois tipos de computação de tempo real: sistemas de tempo real crítico são necessários para completar uma tarefa crítica dentro de um período garantido. Em geral, um processo é submetido juntamente com uma instrução sobre a quantidade de tempo necessária para concluir ou efetuar uma operação de entrada e saída. O escalonador admite o processo, garantindo que ele concluirá no prazo, ou rejeita o pedido como sendo impossível. Essa garantia, feita mediante reserva de recurso, exige que o escalonador saiba exatamente quanto tempo leva para realizar cada tipo de função do sistema operacional; portanto, deve-se garantir a quantidade máxima de tempo que cada operação utilizará. Essa garantia é impossível em um sistema com armazenamento secundário ou memória virtual, como veremos nos Capítulos 9 a 13, porque esses subsistemas causam variação inevitável e imprevisível na quantidade de tempo utilizada para executar determinado processo. Portanto, os sistemas de tempo real crítico são compostos por software de propósito especial executando cm hardware dedicado aos seus processos críticos e não tem a plena funcionalidade dos computadores C sistemas operacionais modernos.

A computação de tempo real não-crítico é menos restritiva. Requer que os processos críticos recebam prioridade cm relação a outros menos favorecidos. Embora acrescentar funcionalidade de tempo real não-crítico a um sistema de tempo compartilhado possa causar alocação injusta de recursos e resultar em atrasos maiores, ou mesmo *starvation* (paralisação), para alguns processos, é pelo menos possível de alcançar. O resultado é um sistema de uso geral que também pode suportar mulfimídia, gráficos interativos de alta velocidade e uma variedade de tarefas que não funcionariam aceitavelmente em um ambiente que não suporta computação de tempo real não-crítico.

Implementar a funcionalidade de tempo real não-crítico requer o projeto cuidadoso do escalonador e aspectos relacionados do sistema operacional. Em primeiro lugar, o sistema deve ter escalonamento por prioridade, e os processos de tempo real devem ter a prioridade mais alta. A prioridade dos processos de tempo real

não deve se degradar com o tempo, embora a prioridade de outros processos possa. Em segundo lugar, a latência de dispatch deve ser pequena. Quanto menor a latência, mais rápido 0 processo de tempo real poderá começar a execução assim que estiver no estado executável ou pronto.

È relativamente simples garantir a validade da primeira propriedade. Por exemplo, podemos dcsabilitar o envelhecimento em processos de tempo real, garantindo assim que a prioridade dos vários processos não mude. No entanto, garantir a segunda propriedade envolve muitos outros aspectos. O problema é que muitos sistemas operacionais, incluindo a maioria das versões do UNIX, são forçados a esperar a conclusão de uma chamada ao sistema ou a finalização de um bloco de operações de I/O antes que possam realizar uma troca de contexto. A latência de dispatch em tais sistemas pode ser demorada, já que algumas chamadas ao sistema são complexas e alguns dispositivos de l/O são lentos.

Para manter a latência de dispatch baixa, precisamos permitir que as chamadas ao sistema possam ser interrompidas. Existem várias maneiras de atingir essa meta. Uma delas é inserir pontos de preempção em chamadas ao sistema de longa duração, que verificam se um processo de alta prioridade precisa ser executado. Sc precisar, ocorre uma troca de contexto; quando o processo de alta prioridade termina, o processo interrompido continua sua chamada ao sistema. Os pontos de preempção só podem ser colocados em posições *seguras* no kerncl - somente onde as estruturas de dados do kcrnel não estiverem sendo modificadas. Mesmo com pontos de preempção, a latência de dispatch pode ser grande, porque é praticável acrescentar apenas alguns pontos de preempção ao kerncl.

Outro método para lidar com a preempção é tornar todo o kcrnel preemptível. Para garantir a operação correta, todas as estruturas de dados do kernel devem ser protegidas com o uso de vários mecanismos de sincronização que serão discutidos no Capítulo 7. Com esse método, o kernel sempre pode ser preemptível, porque quaisquer dados sendo atualizados são protegidos contra modificação pelo processo de alta prioridade. Esse método *é* usado no Solaris 2.

O que acontece se o processo de prioridade mais alta precisar ler ou modificar os dados do kernel que estão sendo acessados no momento por outro processo de menor prioridade ? O processo de alta prioridade estaria esperando o término de um processo de menor prioridade. Essa situação é chamada de inversão de prioridade. Na verdade, pode haver uma cadeia de processos, todos acessando recursos que o processo de alta prioridade precisa. Esse problema pode ser resolvido via protocolo de herança de prioridade, no qual todos esses processos (os processos que estão acessando recursos que o processo de alta prioridade precisa) herdam a alta prioridade até terminarem com o recurso em questão. Quando tiverem concluído, a sua prioridade volta ao valor original.

Na Figura 6.8, os componentes da latência de dispatch estão representados. A fase de conflito da latência de dispatch tem dois componentes:

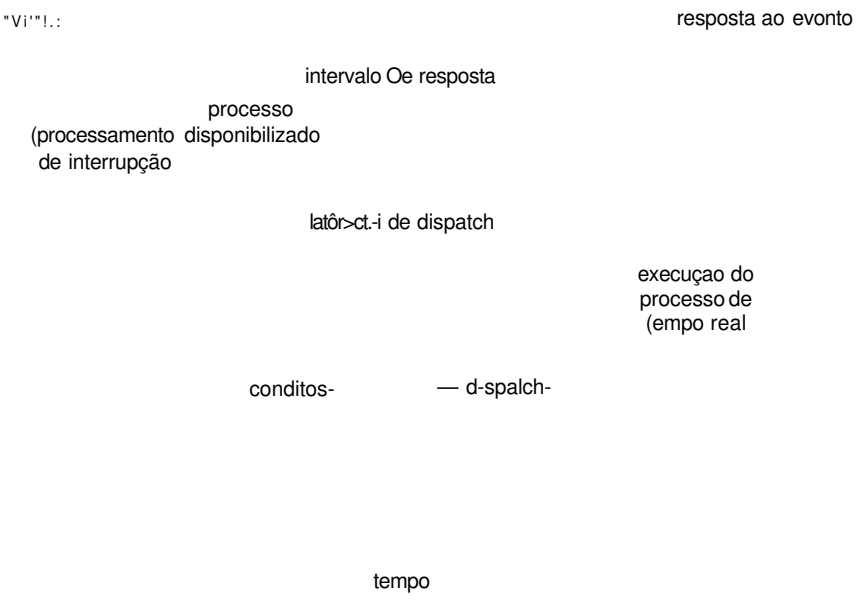


Figura 6.8 Latência de dispatch.

1. Preempção de qualquer processo em execução no kernel
2. Liberação por parte de processos de baixa prioridade dos recursos que o processo de alta prioridade necessita

Como exemplo, no Solaris 2, com a preempção desabilitada, a latência de dispatch fica acima de 100 milissegundos; com a preempção habilitada, geralmente cai para 2 milissegundos.

6.6 • Escalonamento de threads

No Capítulo 5, apresentamos os threads no modelo de processo, permitindo que um único processo tivesse múltiplos fluxos de controle. Além disso, fizemos a distinção entre threads de *usuário* e de *kernel.* Os threads de usuário são gerenciados por uma biblioteca de threads, e o kernel não está ciente deles. Para executar em uma CPU, os threads de usuário são mapcados em um thread de kernel associado, embora esse mapeamento possa ser indireto e usar um processo leve (Lightweight Process - I.WP). Uma distinção entre threads de usuário e de kernel está na forma como são escalonados. A biblioteca de threads escalona threads de usuário para execução em um LWP disponível, um esquema chamado escalonamento local ao processo (o escalonamento de threads c feito localmente em relação à aplicação). Por outro lado, o kernel utiliza o escalonamento global do sistema para decidir que thread de kernel escalonar. Não abrangemos em detalhes como as diferentes bibliotecas de threads escalonam threads localmente - isso é mais do escopo de uma biblioteca de software do que uma preocupação de sistemas operacionais. O escalonamento global é tratado, porque é feito pelo sistema operacional. A seguir, vamos analisar como o Solaris 2 escalona threads.

6.6.1 Escalonamento do Solaris 2

O Solaris 2 utiliza escalonamento de processos baseado em prioridades. São definidas quatro classes de escalonamento, que são, cm ordem de prioridade: de tempo real, de sistema, de tempo compartilhado e interativa. Em cada classe, existem diferentes prioridades e diferentes algoritmos de escalonamento (as classes de tempo compartilhado e interativa utilizam as mesmas políticas de escalonamento). O escalonamento do Solaris 2 está ilustrado na Figura 6.9.

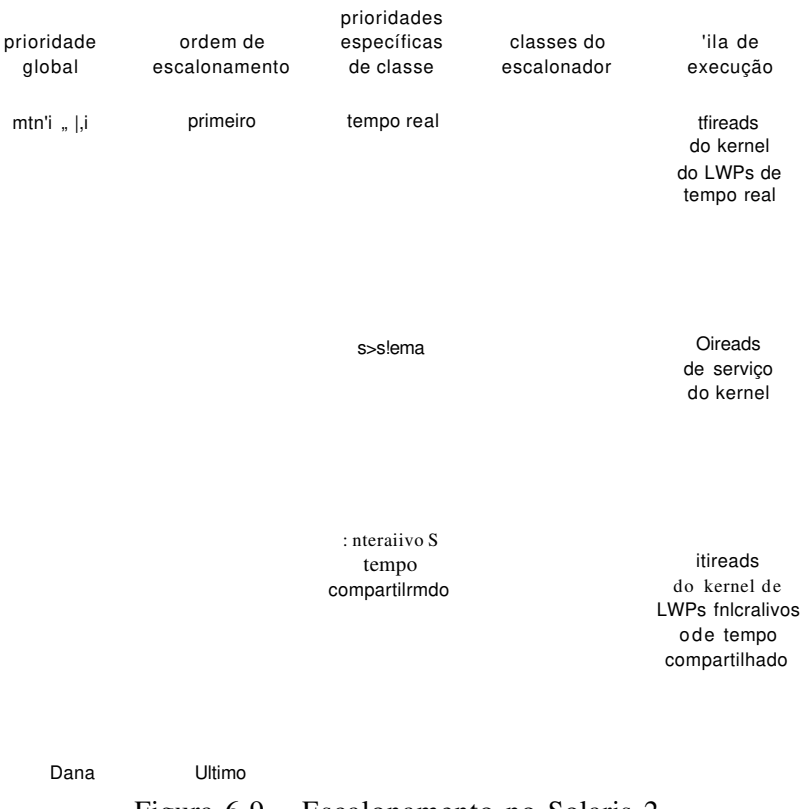


Figura 6.9 Escalonamento no Solaris 2.

Um processo começa com um 1 AVI* c pode criar novos I.WPs conforme necessário. Cada l.WP herda a classe de escalonamento e a prioridade do processo pai. A classe de escalonamento padrão para um processo c a de tempo compartilhado. A política de escalonamento para 0 tempo compartilhado altera prioridades de forma dinâmica e atribui fatias de tempo de diferentes tamanhos usando filas múltiplas com realimentação. Por default, existe um relacionamento inverso entre prioridades e fatias de tempo: quanto maior a prioridade, menor a fatia de tempo e quanto menor a prioridade, maior a fatia de tempo. Os processos interativos normalmente tem prioridade mais alta; os processos limitados por CPU têm prioridade mais baixa. Essa política de escalonamento fornece bom tempo de resposta para os processos interativos e bom throughput para os processos limitados por CPU. O Solaris 2.4 introduziu a classe interativa no escalonamento de processos. A classe interativa usa a mesma política de escalonamento que a classe de- tempo compartilhado, mas fornece às aplicações com janelas gráficas uma prioridade mais alta para melhor desempenho.

0 Solaris utiliza a classe de sistema para executar processos do kerncl, tais como o escalonador e o daemon de paginação. Uma vez estabelecida, a prioridade de um processo de sistema não muda. A classe de sistema é reservada para uso do kernel (os processos de usuário em execução em modo kernel não estão na classe de sistema).

Os threads na classe de tempo real recebem a mais alta prioridade para executar entre todas as classes. Essa atribuição permite que um processo de tempo real tenha uma resposta garantida do sistema em um período limitado de tempo. Um processo de tempo real executará antes que um processo em qualquer outra classe. Em geral, poucos processos pertencem à classe de tempo real.

Existe um conjunto de prioridades em cada classe de escalonamento. No entanto, o escalonador converte as prioridades específicas de classe em prioridades globais e selcciona executar o thread com a prioridade global mais alta. O thread selecionado executa na CPU até (1) bloquear, (2) utilizar sua fatia de tempo, ou (3) ser interrompido por um thread de prioridade mais alta. Se houver múltiplos threads com a mesma prioridade, o escalonador utiliza uma fila circular.

6.7 • Escalonamento de threads Java

A JVM escalona threads usando um algoritmo de escalonamento preemptivo baseado em prioridades. Todos os threads Java recebem uma prioridade e a JVM escalona o thread executável com a prioridade mais alta para execução. Se dois ou mais threads executáveis tiverem a prioridade mais alta, a JVM escalonará os threads usando uma fila FIFO.

A JVM escalona um thread para execução quando um dos seguintes eventos ocorre:

1. O thread em execução no momento sai do estado Executável. Um thread pode deixar o estado Executável de várias formas, tais como bloqueando operações de l/O, saindo do seu método run( ) ou chamando o método suspendi) ou stop{).
2. Um thread com prioridade mais alta do que o thread em execução no momento entra no estado Executável. Nesse caso, a JVM interrompe o thread em execução no momento e escalona o thread com prioridade mais alta para execução.

6.7.1 Fatia de tempo

A especificação da JVM não indica se os threads têm ou não fatias de tempo - isso cabe à implementação específica da JVM. Se os threads tiverem fatias de tempo, um thread Executável será executado durante seu quantum de tempo ou até sair do estado Executável ou ser interrompido por um thread de prioridade mais alta. Se os threads não tiverem fatia de tempo, um thread será executado até que um de dois eventos listados ocorra.

De modo que todos os threads tenham uma quantidade igual de tempo de CPU em um sistema que não realiza fracionamento de tempo, um thread poderá passar o controle da CPU com o método yield( ). Ao chamar o método yi el d ( ), um thread abandona o controle da CPU permitindo que outro thread de prioridade igual seja executado. Quando um thread voluntariamente cede o controle da CPU temos o que se chama de multitarefa cooperativo. O uso do método yield( ) aparece como

```
public void run( ) {
   while (true) {
```

```
        // realiza uma tarefa de uso intensivo da CPU
           .
        // passa o controle da CPU
        Thread.yield( )
    )
|
```

6.7.2 Prioridades de thread

Conforme mencionado, a JVM escalona threads de acordo com a prioridade. A JVM seleciona executar um thread Executável com a prioridade mais alta. Todos os threads Java recebem como prioridade um inteiro positivo dentro de um determinado intervalo. Os threads recebem uma prioridade default quando são criados e - a menos que sejam alterados explicitamente pelo programa - eles mantêm a mesma prioridade em toda sua vida; a JVM não altera prioridades de forma dinâmica. A classe Thread de Java identifica as seguintes prioridades de thread:

Prioridade	Comentário
Thread.NORM_PRIORITY	A prioridade default de thread.
Thread.MIN_PRIORITY	A prioridade mínima de thread.
Thread.MAX_PRIORITY	A prioridade máxima de thread.

O valor de MINPRIORITY é 1, o de MAX_PRIORITY é 10 e o de NORM_PRIORITY é 5. Todo thread Java tem uma prioridade que se encaixa nesse intervalo. Quando um thread é criado, recebe a mesma prioridade que o thread que o criou. A menos que especificado de outro modo, a prioridade default para todos os threads é NORM_PRIORITY. A prioridade de um thread também pode ser definida explicitamente com o método setPriority( ). A prioridade pode ser definida antes de um thread ser iniciado ou enquanto um thread está ativo. A classe HighThread (Figura 6.10) aumenta a prioridade em 1 a mais do que a prioridade default antes de executar o restante do seu método run( ).

```
public class
\
    public void run( ) {
      this.setPrioHty(Thread.NORM_PRIORITY  t  1);
      // restante do método run( )
```

Figura 6.10 Definindo uma prioridade usando setPriority( ).

6.7.3 Escalonador Round-Robin com base em Java

A classe Schedul er (Figura 6.11) implementa um escalonador de threads circular com fatias de tempo. O escalonador contém uma única fila de threads da qual seleciona um thread para execução. O escalonador executa como um thread separado com prioridade 6. Os threads que ele escalona têm prioridade 2 ou 4, dependendo se o thread foi ou não escalonado para executar.

A operação do escalonador é a seguinte: quando um thread é adicionado à fila do escalonador (com o método addThreadf)), recebe a prioridade default 2. Quando o escalonador seleciona um thread para executar, ele define a prioridade do thread para 4. O escalonador então *6* suspenso por um quantum de tempo. Como a JVM usa o algoritmo de escalonamento baseado em prioridade, o thread com a prioridade 4 é o thread Executável de prioridade mais alta, por isso recebe a CPU pelo quantum de tempo. Quando o quantum expira, o escalonador é reativado. Como o escalonador está executando na prioridade 6, ele interrompe o thread em execução no momento. O escalonador define, então, a prioridade do thread interrompido novamente para 2 e seleciona um novo thread para execução, repetindo o mesmo processo.

Se não houver threads na fila, o escalonador entra em um laço esperando para recuperar o próximo thread. Em geral, laços de espera ocupada representam um desperdício dos recursos da CPU. No entanto, provavelmente não importa nesse caso, já que não existem threads disponíveis para execução além do escalonador. Assim que outro thrcad esiivcr pronto para executar, o escalonador o escalonará e será suspenso. Uma modificação bem simples na classe Ci reul arU st é acrescentar o método 1 sEmpty ( ) que indica ao escalonador se a fila está vazia ou não. Se a fila estiver vazia, o escalonador poderá ser suspenso durante algum tempo e, cm seguida, reativado para verificar a fila novamente. Tal modificação evita a atividade de espera ocupada do escalonador.

A classe TestSchedul er (Figura 6,12) ilustra como o escalonador pode ser usado criando uma instância do mesmo e adicionando três threads à sua fila. Como o fracionamento do tempo não pode ser presumido, a classe TestSchedul er está executando com a prioridade mais alta para que continue executando a fim de iniciar todos os threads.

```
public class Scheduler extends Thread
{
    public Scheduler( ) {
      tlmeSHce « DEFAUtJ_TIHE_SLICE;
      queue = new C1rcularlist( );
    »

    public Scheduler(int quantura) (
      timeSlice • quantum;
      queue • new CircularList( );
    1

    public void addThread(Thread t) {
      t.setPriority(2);
      queue.additem(t);
    I

    private void $chedulerSleep( ) {
      try {
        Thread.sieepítimeSlice);
      } catch (InterruptException e) ( };
    )

    public void run( ) {
      Thread current;

      this.setPriority(6);

      while (true) {
        // obtém o próximo thread
        current • (Thr&ad)queue.getNetx{ );
        if ( ( current !« null) && (current.isAlive( )) ) {
          current.setPriority(4);
          schedulerSleçp( );
          current.setPrlorlty(2);
        1
      )
  )
  private CircularList queue;
  private int timeSlice;
  private static final int DEFAUUTIME SLICE » 1000;
|
```

Figura 6.11 Escalonador Round-Robin.

6.8 • Avaliação de algoritmos

Como selecionamos um algoritmo de escalonamento de CPU para determinado sistema? Como vimos na Scção 6.3, existem muitos algoritmos de escalonamento, cada qual com seus próprios parâmetros. Como resultado, selccionar um algoritmo pode ser difícil.

O primeiro problema é definir os critérios usados na Seleção de um algoritmo. Como vimos na Seção 6.2, os critérios são geralmente definidos cm termos de utilização da CPU, tempo de resposta ou throughpuf. Para selccionar um algoritmo, devemos primeiro definir a importância relativa dessas medidas. Nossos critérios podem incluir várias medidas, como:

- Maximizar a utilização de CPU sob a limitação de que o tempo de resposta máximo é 1 segundo
- Maximizar o throughput de modo que o tempo de retorno seja (em média) linearmente proporcional ao tempo de execução total

Assim que os critérios de Seleção tiverem sido definidos, devemos avaliar os vários algoritmos em consideração. Existem vários métodos de avaliação diferentes, que estão descritos nas Scções 6.8.1 a 6.8.4.

```
public class TestScheduler
{
    public static void matn(String args[ ]) {
      Thread.currentThread( ).setPriority(Thread.MAX_PRIORITY);
      Scheduler CPUScheduter • new Scheduler( );
      CPUScheduler.start( );

      //usa Testlhread. embora possa
      //ser qualquer objeto Thread.
      Testlhread ti - new TestThreadfThread 1');
      tl.startí );
      CPUScheduler.addThread(tl);
      TestThread t2 - new TestThread("Thread 2");
      t2.start( );
      CPUScheduler.addThread(t2);

      TestThread t3 " new TestThreadfThread 3");
      t3.start( );
      CPUScheduler.addThread(t3);
    )
```

Figura 6.12 Programa Java que ilustra o uso do escalonador.

6.8.1 Modelagem determinista

Um classe importante de métodos de avaliação é a avaliação analítica. A avaliação analítica utiliza o algorit' mo dado e o volume de trabalho do sistema para gerar uma fórmula ou número que avalia o desempenho do algoritmo para aquele volume de trabalho.

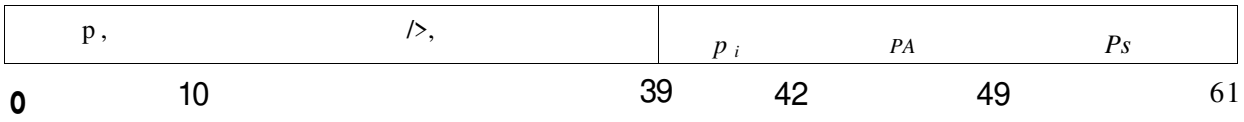
Um tipo de avaliação analítica é a modelagem determinista. Esse método pega um determinado volume de trabalho predeterminado e define o desempenho de cada algoritmo para esse volume de trabalho.

Por exemplo, considere que temos o volume de trabalho indicado. Todos os cinco processos chegam no instante 0, na ordem dada, com a duração do surto de CPU expressa em milissegundos:

Processo	Duração de surto
P,	10
P,	29
P>	3
P<	*?*
P_5	12

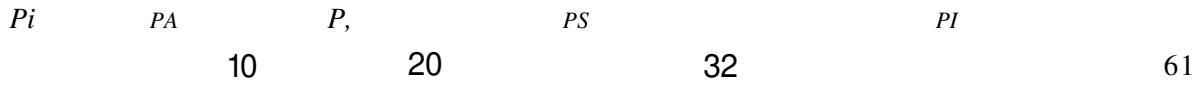
Considere os algoritmos de escalonamento FCFS, SJF e RR (quantum = 10 milissegundos) para esse conjunto de processos. Qual dos algoritmos deve dar O mínimo tempo de espera médio?

Para o algoritmo FCFS, os processos seriam executados assim:



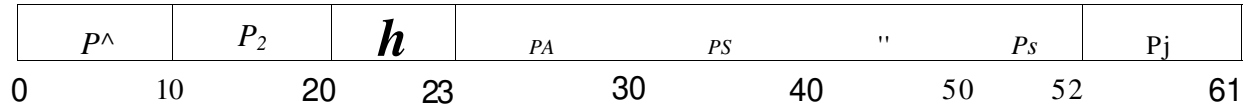
O tempo de espera é 0 milissegundo para o processo P, 10 milissegundos para o processo P>,39 milissegundos para o processo P(, 42 milissegundos para o processo P_4 e 49 milissegundos para o processo P_5. Assim, o tempo de espera médio é (0 + 10 + 39 + 42 + 49)/5 = 28 milissegundos.

Com o escalonamento SJF não-preemptivo, os processos seriam executados assim:



O tempo de espera é 10 milissegundos para o processo P, 32 milissegundos para o processo P:,0 milissegundo para o processo P,, 3 milissegundos para o processo P_4 e 20 milissegundos para o processo P_s. Assim, o tempo de espera médio é (10 + 32 + 0 + 3 + 20)/5 = 13 milissegundos.

Com o algoritmo RR, os processos seriam executados assim:



O tempo de espera é 0 milissegundo para o processo P_L 32 milissegundos para o processo P_2» 20 milissegundos para o processo P,, 23 milissegundos para o processo P_4 e 40 milissegundos para o processo P(. Assim, o tempo de espera médio é (0 + 32 + 20 + 23 + 40)/5 = 23 milissegundos.

Vemos que, *nesse caso,* a regra SJF resulta em menos do que metade do tempo de espera médio obtido com o escalonamento FCFS; o algoritmo RR nos dá um valor intermediário.

A modelagem determinista é simples e rápida. Fornece números exatos, permitindo que os algoritmos sejam comparados. No entanto, requer números exatos como entrada, e suas respostas só se aplicam nesses casos. Os principais usos da modelagem determinista são em descrever algoritmos de escalonamento e em fornecer exemplos. Nos casos em que os mesmos programas estejam sendo executados seguidamente e nos quais seja possível medir exatamente os requisitos de processamento do programa, a modelagem determinista talvez possa ser usada para selccionar um algoritmo de escalonamento. Em um conjunto de exemplos, a modelagem determinista poderá indicar tendências que podem ser analisadas e comprovadas separadamente. Por exemplo, pode-se demonstrar que, para o ambiente descrito (todos os processos e seus tempos disponíveis no instante 0), a regra SJF sempre resultará no tempo de espera mínimo.

Em geral, no entanto, a modelagem determinista é específica demais e requer muito conhecimento exato para ser útil.

6.8.2 Modelos de filas

Os processos que são executados cm muitos sistemas variam todos os dias, por isso não existe um conjunto fixo de processos (e tempos) para uso da modelagem determinista. O que pode ser determinado, no entanto, é a distribuição dos surtos de CPU e de l/O. Essas distribuições podem ser medidas e aproximadas, ou simplesmente estimadas. O resultado é uma fórmula matemática par descrever a probabilidade de determinado surto de CPU. Em geral, essa distribuição é exponencial e é descrita pela sua média. Da mesma forma, a distribuição dos tempos de chegada dos processos no sistema (a distribuição do tempo de chegada) deve ser dada.

A partir dessas duas distribuições, é possível calcular o valor médio de throughpur, utilização, tempo de espera ctc. pari a maioria dos algoritmos.

O sistema de computação é descrito como uma rede de servidores. Cada servidor tem uma fila de processos em espera. A CPU é um servidor com sua tila de processos prontos, assim como o sistema de entrada e saída com suas filas de dispositivos. Conhecendo as taxas de chegada e as taxas de serviço, podemos calcular a utilização, o tamanho médio da fila, O tempo de espera médio ctc. Essa área de estudo *é* chamada de análise de redes de filas.

Como exemplo, seja *n o* tamanho médio da fila (excluindo o processo sendo atendido), seja W o tempo de espera médio na fila c*X* a taxa de chegada média para novos processos na fila (como três processos por segundo). Assim, esperamos que durante o tempo *W*que determinado processo espera, *X* X *W* novos processos chegarão na fila. Se O sistema estiver em uma situação estável, o número de processos que saem da fila deverá sei" igual ao número de processos que chegam. Assim,

$$n = X \times W.$$

Essa equação, conhecida como a fórmula de Littlc, é particularmente útil porque é válida para qualquer algoritmo de escalonamento e distribuição de chegada.

Podemos usar a fórmula de Little para calcular uma das três variáveis, se soubermos as outras duas. Por exemplo, se sabemos que sete processos chegam a cada segundo (em média), e que existem normalmente 14 processos na fila, podemos calcular 0 tempo de espera médio por processo como sendo de 2 segundos.

A análise de filas pode ser útil na comparação dos algoritmos de escalonamento, mas também tem limitações. No momento, as classes de algoritmos e distribuições que podem ser tratadas são bem limitadas. A matemática dos algoritmos ou distribuições complicadas pode ser difícil de dominar. Assim, as distribuições de chegada c serviço geralmente são definidas de forma irrealista, mas matematicamente tratáveis. Geralmente também é necessário fazer uma série de hipóteses independentes, que talvez não sejam precisas. Assim, para que uma resposta possa ser calculada, os modelos de filas geralmente são apenas uma aproximação de um sistema real. Como resultado, a precisão dos resultados calculados pode ser questionável.

6.8.3 Simulações

Para obter uma avaliação mais precisa dos algoritmos de escalonamento, podemos usar simulações. Executar simulações envolve programar um modelo de sistema de computador. Estruturas de dados em software representam os principais componentes do sistema. O simulador tem uma variável que representa o relógio; à medida que o valor dessa variável aumenta, o simulador modifica o estado do sistema para refletir as atividades dos dispositivos, dos processos e do cscalonador. A medida que a simulação executa, estatísticas que indicam o desempenho do algoritmo são coletadas e impressas.

Os dados que conduzem a simulação podem ser gerados de várias formas. O método mais comum utiliza um gerador de números aleatórios, que é programado para gerar processos, tempos de surtos de CPU, chegadas, partidas e assim por diante, de acordo com distribuições de probabilidade. As distribuições podem ser definidas matematicamente (uniforme, exponencial, Poisson) ou empiricamente. Se a distribuição for definida empiricamente, são feitas medidas do sistema real sendo estudado. Os resultados definem a distribuição real de eventos no sistema real; essa distribuição pode então ser usada para conduzir a simulação.

Uma simulação baseada em distribuição, no entanto, pode ser imprecisa devido aos relacionamentos entre eventos sucessivos no sistema real. A distribuição de frequência indica apenas quantos eventos ocorrem; não revela nada sobre a ordem de sua ocorrência. Para corrigir esse problema, podemos usar registros de execução. Um registro de execução é criado por meio da monitoração do sistema real, registrando-se a sequencia de eventos reais (Figura 6.13). Em seguida, essa sequencia é usada para conduzir a simulação. Os registros de execução fornecem uma forma excelente de comparar dois algoritmos exatamente no mesmo conjunto de entradas reais. Esse método pode produzir resultados precisos para suas entradas.

As simulações podem ser caras, geralmente exigindo horas de tempo do computador. Uma simulação mais detalhada fornece resultados mais precisos, mas requer mais tempo de computação. Além disso, registros de execução podem exigir grandes quantidades de espaço de armazenamento. Finalmente, o projeto, codificação e depuração do simulador pode ser uma grande tarefa.

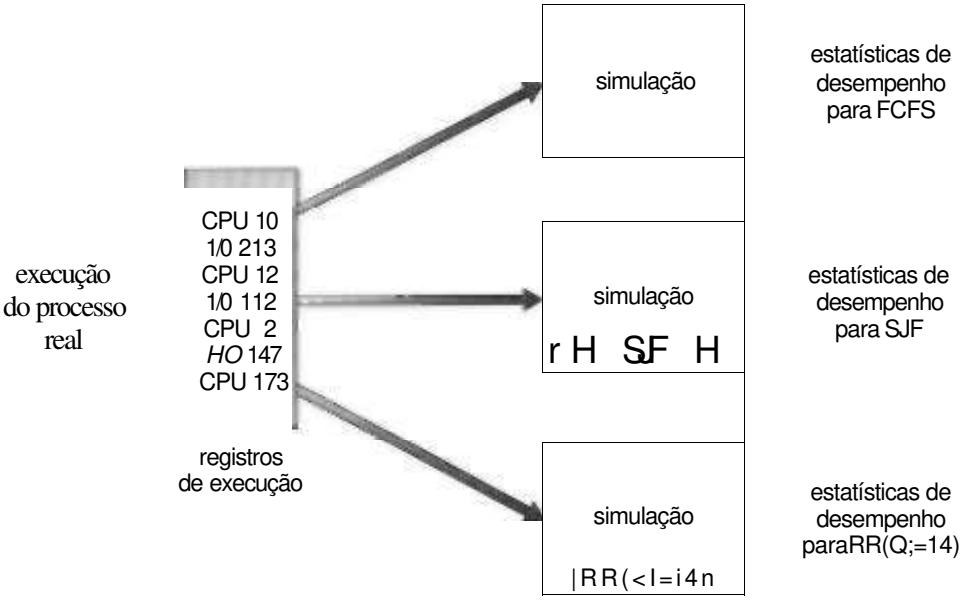


Figura 6.13 Avaliação dos escalonadores de CPU por simulação.

6.8.4 Implementação

Mesmo uma simulação *é* de precisão limitada. A única forma completamente precisa de avaliar um algoritmo de escalonamento c codificá-lo, colocá-lo no sistema operacional e verificar seu funcionamento. Essa abordagem coloca o algoritmo real no sistema real para avaliação em condições reais de operação.

A principal dificuldade dessa abordagem é o alto custo. Os custos envolvem não apenas a codificação do algoritmo e modificação do sistema operacional para suportá-lo e as estruturas de dados exigidas por ele, mas também a reação dos usuários a um sistema operacional em constante mudança. A maior parte dos usuários não está interessada em construir um sistema operacional melhor; simplesmente querem que seus processos sejam executados e desejam utilizar seus resultados. Um sistema operacional em constante mudança não ajuda os usuários a realizar seu trabalho.

A outra dificuldade com qualquer avaliação de algoritmo c que o ambiente no qual o algoritmo é usado será alterado. O ambiente mudará não só da forma normal, à medida que novos programas são escritos e os tipos de problemas mudam, mas também como resultado do desempenho do escalonador. Se processos curtos tiverem prioridade, os usuários poderão quebrar processos maiores em grupos de processos menores. Se os processos interativos tiverem prioridade sobre processos não-interativos, os usuários poderão mudar para uso interativo.

Por exemplo, pesquisadores projetaram um sistema que classificava os processos interativos e não-interativos automaticamente, analisando a quantidade de entrada e saída no terminal. Se um processo não tivesse entrada nem saída para o terminal em um intervalo de I segundo, ele era classificado como não-interativo e movido para uma fila de baixa prioridade. Essa regra resultou cm uma situação na qual um programador modificava seus programas para escrever um caractere arbitrário para o terminal em intervalos regulares de menos de 1 segundo. O sistema dava a seus programas uma alta prioridade, embora a saída 00 terminal fosse completamente sem significado.

Os algoritmos de escalonamento mais flexíveis podem ser alterados pelos gerentes do sistema ou pelos usuários de modo que possam ser ajustados para uma aplicação ou conjunto de aplicações específicas. Por exemplo, uma estação de trabalho que executa aplicações gráficas de alto nível pode ter necessidades de escalonamento diferentes daquelas de um servidor Web ou servidor de arquivos. Alguns sistemas operacionais - particularmente várias versões do UNIX - permitem ao administrador do sistema ajustar os parâmetros de escalonamento para uma configuração de sistema específica. Outra abordagem é usar APIscomoyieldí) e setPriori ty( ), permitindo assim que as aplicações tenham comportamento previsível, A desvantagem dessa abordagem é que o ajuste de desempenho de um sistema ou aplicação muitas vezes não resulta em desempenho melhorado em situações mais gerais.

6.9 • Resumo

O escalonamento de CPU consiste em selecionar um processo em espera na fila de processos prontos e alocar a CPU a esse processo. A CPU é alocada ao processo seleeionado pelo dispatcher.

O escalonamento first-come, first-served (FCFS) é o algoritmo de escalonamento mais simples, mas pode fazer com que processos curtos esperem por processos muito longos. O escalonamento do tipo job mais curto (SJF) é comprovadamente ideal, fornecendo o menor tempo de espera médio. A implementação do escalonamento SJF é difícil porque prever a duração do próximo surto de CPU é difícil. O algoritmo SJF é um caso especial de algoritmo geral de escalonamento por prioridade, que simplesmente aloca a CPU ao processo de prioridade mais alta. O escalonamento SJF e o escalonamento por prioridade podem sofrer de paralisação. Envelhecimento é a técnica usada para evitar a paralisação.

O Round-Robin (RR) é mais adequado para um sistema de tempo compartilhado (interativo). O escalonamento RR aloca a CPU ao primeiro processo na fila de processos prontos durante *q* unidades de tempo, em que *q* é o quantum de tempo. *Apósq* unidades de tempo, se o processo não tiver abandonado a CPU, ele é interrompido e colocado no final da fila de processos prontos. O principal problema é a Seleção do quantum de tempo. Se o quantum for grande demais, o escalonamento RR degenera para escalonamento FCFS; se *O* quantum for pequeno demais, o custo de escalonamento na forma de tempo de troca de contexto se torna excessivo.

O algoritmo FCFS é não-preemptivo; o algoritmo RR é preemptivo. Os algoritmos SJF e de prioridade podem ser préemptivos ou não-preemptivos.

Os algoritmos de múltiplas filas permitem que diferentes algoritmos sejam usados para várias classes de processos. O mais comum é ter uma fila interativa de primeiro plano, que utiliza o escalonamento RR, e uma fila cm batch de segundo plano, que utiliza o escalonamento FCFS. As múltiplas filas com rcalímcntação permitem que os processos passem de uma fila para outra.

Muitos sistemas de computação contemporâneos agora oferecem suporte a múltiplos processadores, cm que cada processador é escalonado de forma independente. Em geral, existe uma fila de processos (ou threads), todos os quais estão disponíveis para execução. Cada processador toma uma decisão de escalonamento e seleciona a partir dessa fila.

A JVM usa um algoritmo de escalonamento de threads preemptivo e baseado em prioridade que seleciona para execução o thread com a prioridade mais alta. A especificação não indica se a JVM deve fracionar o tempo dos threads; isso cabe à implementação específica da JVM.

A grande variedade de algoritmos de escalonamento exige que tenhamos métodos para selecionar dentre eles. Os métodos analíticos utilizam análise matemática para determinar o desempenho de um algoritmo. Os métodos de simulação determinam o desempenho imitando o algoritmo de escalonamento em uma amostra "representativa" de processos e calculando o desempenho resultante.

• Exercícios

6.1 Um algoritmo de escalonamento de CPU determina a ordem para execução dos seus processos escalonados. Considerando *n* processos a serem escalonados em um processador, quantos escalonamentos diferentes são possíveis? Apresente uma fórmula cm termos de *n.*

6.2 Defina a diferença entre escalonamento preemptivo e não-preemptivo. Explique por que o escalonamento não-preemptivo estrito não deve ser usado em um centro de computação.

6.3 Considere o seguinte conjunto de processos, com a duração de surto de CPU expressa em milissegundos:

Processo	Duração do surto	Prioridade
r>	10	3
P,	1	1
P,	2	3
P<	1	4
P_s	5	2

Os processos são considerados como tendo chegado na ordem Pj P, P_t P_A P$, todos no instante 0.

a. Desenhe quatro diagramas de Gantt que ilustrem a execução desses processos usando o escalonamento FCFS, SJF, por prioridade não-preemptiva (um número de prioridade mais baixo implica prioridade mais alta) e RR (quantum = 1).
b. Qual é o tempo de retorno de cada processo para cada um dos algoritmos de escalonamento do item a?
c. Qual é o tempo de espera de cada processo para cada um dos algoritmos de escalonamento do item a?
d. Qual dos escalonamentos do item a resulta no tempo de espera médio mínimo (em relação a todos os processos)?

6.4 Vamos supor que os seguintes processos cheguem para execução nos instantes indicados. Cada processo executará durante o período de tempo listado. Ao responder as perguntas, use o escalonamento não-preemptivo e baseie todas as decisões nas informações existentes no momento em que a decisão deverá ser tomada.

Processo	Instante de checada	Duração de surto
Pi	0,0	8
P,	0,4	4
P_3	1,0	1

a. Qual é o tempo de retorno médio para esses processos com o algoritmo de escalonamento FCFS?
b. Qual é o tempo de retorno médio para esses processos com o algoritmo de escalonamento SJF?
c. O algoritmo SJF deve melhorar o desempenho, mas observe que escolhemos executar o processo P, no tempo 0 porque não sabíamos que dois processos mais curtos chegariam logo a seguir. Calcule o tempo de retorno médio se a CPU ficar inativa pela primeira unidade de tempo (I) e o escalonamento SJF for utilizado. Lembre-sc de que os processos P] e *Pi* estão esperando durante esse tempo inativo, de modo que seu tempo de espera poderá aumentar. Esse algoritmo pode ser chamado de *escalonamento de conhecimento futuro.*

6.5 Considere uma variante do algoritmo de escalonamento RR no qual as entradas na fila de processos prontos são ponteiros para PCBs.

a. Qual seria o efeito de colocar dois ponteiros para o mesmo processo na fila de processos prontos?
b. Quais seriam as duas principais vantagens c duas desvantagens desse esquema?
c. Como você modificaria o algoritmo RR básico para obter o mesmo efeito sem os ponteiros duplicados?

6.6 Qual a vantagem cm ter diferentes tamanhos de quantum em diferentes níveis de um sistema de múltiplas filas?

6.7 Considere o seguinte algoritmo de escalonamento preemptivo por prioridade que se baseia em prioridades em constante mudança. Os números de prioridade maiores implicam prioridade mais alta. Quando um processo está esperando pela CPU (na fila de processos prontos, mas não em execução), sua prioridade muda a uma razão «; quando ele está executando, sua prioridade muda a uma razão /?. Todos os processos recebem prioridade 0 quando entram na fila de processos prontos. Os parâmetros *a* e/f podem ser definidos para ter muitos algoritmos de escalonamento diferentes.

a. Qual é o algoritmo resultante de/í > « > 0?
b. Qual é o algoritmo resultante de *a* < *fi* < 0?

6.8 Muitos algoritmos de escalonamento de CPU são parametrizados. Por exemplo, o algoritmo RR requer um parâmetro para indicar a fatia de tempo. As múltiplas filas com realimentação requerem parâmetros para definir o número de filas, os algoritmos de escalonamento para cada fila, os critérios usados para mover processos entre as filas c assim por diante.

Esses algoritmos são, na verdade, conjuntos de algoritmos (por exemplo, o conjunto de algoritmos RR para diversas fatias de tempo etc). Um conjunto de algoritmos pode incluir outro (por exemplo, o algoritmo FCFS <• o algoritmo RR com um quantum de tempo infinito). Que relação (se houver) existe entre os seguintes pares de conjuntos de algoritmos?

a. Prioridade e SJF
b. Múltiplas filas com realimentação c FCFS
c. Prioridade e FCFS
d. RR e SJF

6.9 Vamos supor que um algoritmo de escalonador (no nível do escalonamento de CPU de curto prazo) favoreça os processos que tenham usado o menor tempo de processador no passado recente. Por que esse algoritmo favorecerá os programas limitados por entrada/saída c ainda assim não causará paralisação permanente nos programas limitados por CPU?

6.10 Explique as diferenças no grau em que os seguintes algoritmos de escalonamento favorecem os processo curtos?

a. FCFS
b. RR
c. Múltiplas filas com realimentação

6.11 As perguntas seguintes referem-se ao escalonamento Round-Robin com base em Java descrito na Seção 6.7.3.

a. Se houver um thread 7", na fila para o Scheduler executando em prioridade 4 e outro thread T_2 com prioridade 2, é possível para o thread T_2 executar durante o quantum de Tj? Explique.
b. Explique o que acontece se o thread em execução no momento tiver seu método stop( ) chamado durante seu quantum de tempo.
c. Explique o que acontece se o escalonador selecionar um thread para execução que tenha tido seu método su$pend( ) chamado.

6.12 Modifique a classe Ci reul arLi st adicionando o método i sEmpty ( ) que retorna informação indicando se a fila está vazia ou não. Faça com o que o escalonador verifique se a fila está vazia. Se a fila estiver vazia, suspenda o escalonador durante algum tempo e ative-o para verificar a fila novamente.

6.13 Modifique o escalonador baseado em Java de modo que ele tenha múltiplas filas representando diferentes prioridades. Por exemplo, tenha três filas separadas, para as prioridades 2,3 e 4. Faça com que o escalonador selecione um thread da fila de prioridade mais alta, defina essa prioridade para 5 e deixe que o thread execute determinado quantum de tempo. Quando o quantum expirar, selecione o próximo thread da fila de prioridade mais alta e repita o processo. Além disso, você deve modificar a classe Schedul er de modo que, quando um thread é entregue ao escalonador, uma prioridade inicial seja especificada.

Notas bibliográficas

As filas com realimentação foram originalmente implementadas no sistema CTSS descrito em Corbato e colegas (1962]. Esse sistema de filas com realimentação foi analisado por Schrage [1967]. O algoritmo de escalonamento preemptivo por prioridade do Exercício 6.7 foi sugerido por Kleinrock (1975|.

Anderson e colegas (1989] e Lewis e Berg [1998] falaram sobre escalonamento de threads. Discussões relativas ao escalonamento com múltiplos processadores foram apresentadas por Tuckcr e Gupta [1989], Zahorjan c McCann [1990], Fcitclson e Rudolph [1990] e Leutenegger e Vernon (1990].

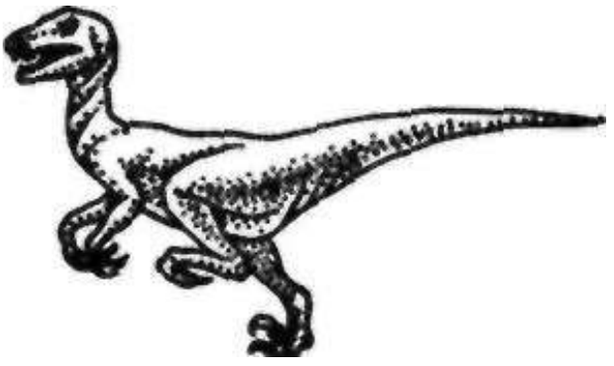
Discussões sobre o escalonamento cm sistemas de tempo real foram oferecidas por Abbot [1984], Jensen e colegas [1985], Hong e colegas 11989] e Khanna e colegas [1992]. Uma edição especial sobre sistemas operacionais de tempo real foi editada por Zhao [1989]. Eykholt e colegas [1992] descreveram o componente de tempo real do Solaris 2.

Escalonadores proporcionais foram descritos por Henry [1984], Woodside [1986], e Kay e Lauder [1988].

A discussão relativa às políticas de escalonamento usadas no sistema operacional OS/2 foi apresentada por Iacobucci [1988]; uma discussão relativa ao sistema operacional UNIX V foi apresentada por Bach [1987|; outra relativa ao UNIX BSD 4.4 foi apresentada por McKusick e colegas [1996], e outra para o sistema operacional Mach foi apresentada por Black [1990|. O escalonamento do Solaris 2 foi descrito por Granam [1995]. Solomon [1998] discutiu o escalonamento no Windows NT.

O escalonamento de threads Java de acordo com a especificação da JViM *é* descrito por Lindholm e Yellin [1998]. A ideia para o cscalonador Round-Robin com base em Java foi sugerida por Oaks e Wong 11999].

Capítulo 7



SINCRONIZAÇÃO DE PROCESSOS

Um processo cooperativo pode afetar ou ser afetado pelos outros processos que estão executando no sistema. Os processos cooperativos podem compartilhar diretamente um espaço de endereçamento lógico (ou seja, código e dados) ou ter permissão para compartilhar dados apenas através de arquivos. O primeiro caso *é* obtido com a utilização de threads, que são discutidos no Capítulo 5. O acesso concorrente a dados compartilhados pode resultar em inconsistência de dados. Neste capítulo, discutimos vários mecanismos para garantir a execução ordenada de processos ou threads cooperativos que compartilham um espaço de endereçamento lógico, permitindo que a consistência de dados seja mantida.

7.1 • Fundamentos

No Capítulo 4, desenvolvemos um modelo de sistema consistindo em processos ou threads sequenciais cooperativos, todos executando assincronamente e possivelmente compartilhando dados. Ilustramos esse modelo com o esquema de buffer limitado, que é representativo dos sistemas operacionais. O código para o thread produtor é o seguinte:

```
while (count " BUFFER_SIZE)
    ; // no-op
// adiciona um item ao buffer
••count;
buffer[in] • item;
In - (in • 1) * BUFFERJIZE;
```

O código para o consumidor é

```
while (count •» 0)
    ; // no-op
// remove um item do buffer
--count;
item - bufferfout];
out = (out + I) % BUFFER_SIZE;
```

Embora ambas as rotinas produtor e consumidor sejam corretas separadamente, podem não funcionar corretamente quando forem executadas ao mesmo tempo. O motivo pelo qual isso acontece é que os threads compartilham a variável count, que serve como contador do número de itens no buffer. Como exemplo, vamos supor que o valor atual da variável count seja 5, e que os threads produtor e consumidor executem as instruções **count e --count de forma concorrente. Após a execução dessas duas instruções, o valor da variável count pode ser 4, *5* ou 6! O único resultado correto para count é 5, que e* gerado corretamente apenas se o produtor e o consumidor executarem sequencialmente.

Podemos demonstrar como o valor resultante de count pode estar incorreto. Observe que a instrução ••count pode ser implementada em linguagem de máquina (em uma máquina típica) como:

registrador, = count;
registrador, = *registrador,* **+ 1;**
count = *registrador,*

onde *registrador, é* um registrador local da CPU. Da mesma forma, a instrução-count é implementada da seguinte maneira:

$registrador_z$ = count;
$registrador_z$ = $registrador_2$ - 1;
count = $registrador_2$;

onde $registrador_2$ *é* um registrador local da CPU. Lcmbre-se de que, embora *registrador,* e *registrador,* possam ser os mesmos registradores físicos (um acumulador, por exemplo), o conteúdo desse registrador será salvo e restaurado pela rotina de tratamento de interrupções (Seção 2.1).

A execução concorrente das instruções t+count e —count é equivalente a uma execução sequencial na qual as instruções de nível mais baixo apresentadas anteriormente são intercaladas em alguma ordem arbitrária (mas a ordem em cada instrução de alto nível é preservada). Uma intercalação desse tipo seria:

So	*produtor*	executa	*registrador,* = count	*{registrador,* = 5}
Si	*produtor*	executa	*registrador,* = *registrador,* + 1 count	*{registrador,* =6}
s_2	*consumidor*	executa	$registrador_2$ • count	*{registrador,* = 5}
Sj	*consumidor*	executa	*registrador,* = $registrador_2$ - 1	*{registrador,* = 4}
S_4	*produtor*	executa	count * *registrador,*	{count = 6}
Sj	*consumidor*	executa	count =* *registrador*^	{count = 4}

Observe que chegamos ao estado incorreto *"count* = 4", registrando que há quatro buffers cheios, quando, na verdade, existem cinco buffers cheios. Se invertermos a ordem das instruções em S_4 e Sj, chegaríamos ao estado incorreto *"count* = 6".

Chegaríamos a esse estado incorreto porque permitimos que ambos os threads manipulassem a variável count concorrentemente. Essa situação, na qual vários threads acessam e manipulam os mesmos dados concorrentemente e na qual o resultado da execução depende da ordem específica em que o acesso ocorre, é chamada de condição de corrida *(race conditioti).* Para evitar essa condição, precisamos garantir que apenas um thread de cada vez possa estar manipulando a variável count. Para ter essa garantia, precisamos de alguma forma de sincronização de threads. Tais situações ocorrem com frequência nos sistemas operacionais à medida que diferentes partes do sistema manipulam os recursos, e não queremos que as alterações interfiram umas mas outras. Uma boa parte deste capítulo será dedicada à sincronização c coordenação de threads.

7.2 • O problema da seção crítica

Como um primeiro método para controlar o acesso a um recurso compartilhado, declaramos uma seção de código como sendo *crítica;* em seguida, controlamos o acesso a essa seção. Considere um sistema que consista em» threads *{T_0)* T,, ..., T„..,h Cada thread tem um segmento de código, chamado seção crítica, no qual o thread pode estar alterando variáveis comuns, atualizando uma tabela, gravando um arquivo etc A característica importante do sistema é que, quando um thread estiver executando cm sua seção crítica, nenhum outro thread terá permissão de executar em sua seção crítica. Assim, a execução das seções críticas pelos threads é *mutuamente exclusiva* no tempo. O problema da seção crítica é como escolher um protocolo que os threads possam usar para cooperar.

Uma solução ao problema da seção crítica deve satisfazer os três requisitos seguintes:

1. *Exclusão mútua:* Se o thread T_i estiver executando em sua seção crítica, nenhum outro thread poderá estar executando em sua respectiva seção crítica.
2. *Progresso:* Se nenhum thread estiver executando em sua seção crítica e houver alguns threads que desejam entrar em suas seções críticas, apenas os threads que não estiverem executando em sua seção não-crítica poderão participar da decisão de qual deles entrará na sua seção crítica em seguida, e essa Seleção não poderá ser adiada indefinidamente.
3. *Espera limitada:* Existe um limite no número de vezes que outros threads podem entrar em suas seções críticas depois que um thread tiver solicitado para entrar em sua seção crítica e antes que o pedido seja concedido. Esse limite evita a *starvaliou* (paralisação) de qualquer thread único.

Consideramos que cada thread está executando a uma velocidade diferente de zero. No entanto, não podemos fazer nenhuma hipótese com relação à velocidade *relativa* dos;; threads. Na Seção 7.3, examinamos o problema de seção crítica e desenvolvemos uma solução que satisfaz esses três requisitos. As soluções não se baseiam em hipóteses relativas a instruções de hardware ou ao número de processadores que o hardware suporta. Suponhamos, no entanto, que as instruções básicas de linguagem de máquina (as instruções básicas como load, store e test) são executadas atomicamente. Ou seja, se duas instruções desse tipo forem executadas de forma concorrente, o resultado será equivalente à sua execução sequencial em alguma ordem desconhecida. Assim, se load e store forem executados concorrentemente, load obterá o valor antigo ou o novo, mas não uma combinação de ambos.

7.3 • Soluções para duas tarefas

Nesta seção, consideramos três implementações Java diferentes para coordenar as ações de dois threads diferentes. Os threads são numerados $T_{,}$ *e* $T_{,}$. Para fins de conveniência, quando estivermos representando T,, usamos T; para indicar o outro thread; ou seja,/' = 1 -/'. Antes de examinar os diferentes algoritmos, apresentamos os arquivos de classe Java necessários.

Cada thread é implementado utilizando a classe Worker apresentada na Figura 7.1. Uma chamada aos métodoscritical$ection( ) enonCriticatSection( ) representa os locais em que cada thread realiza suas seções críticas e náo-críticas. Antes de chamar sua seção crítica, cada thread chamará o método enteringCriticalSecti - on( ), passando o seu id de thread (que será 0 ou I). Um thread só retornará de enteringCriticalSection( ) quando lhe for possível entrar em sua seção crítica. Ao terminar sua seção crítica, o thread chamará o método leavingCHticalSectioní).

A classe abstrara Mutual Excl usl on (Figura 7.2) servirá como um modelo para os três algoritmos distintos.

```
public class Worker extends Thread
{
    public Horker(Stríng n. int i, MutualExclusto" s) {
      name • n;
      Id - i;
      shared = s;
    )
    public voul run( ) {
      while (true) {
        share.enteringCriticalSectIon(id);
        System.out.printIn(name • " is in criticai section");
        MutualExclusion.cri ti calSection( );
        shared.leavingCriticalSectionOd);
        System.out.printIn(name t • is out of criticai section");
        MutualExclusion.nonCriti calSection( );
      I
    J
    private String name;
    private int id;
    private MutualExclusion shared;
)
```

Figura 7.1 Thread Worker.

```
public abstract class HutualExclusion
{
    public static void critícalSection{ ) (
      try {
        Thread.sleepí (int) (Hath.random( ) • TIHE) );
      \
      catch (InterruptedException e) ) }
    \

    public static void nonCriticalSection( ) {
      try {
        Thread.sleep( (int) (Math.random( ) * TIHE) );
      }
      catch (InterruptedException e) ( )
      |

      public abstract void enteringCrit1cal$ection(int t);
      public abstract void leavingCriticalSection(int t);

      public static final int TURNJ) » 0;
      public static final int TURIM • 1;
      private static final int TIHE = 3000;
|
```

Figura 7.2 Classe abstrata MutualExclusion.

Os métodos estáticos criticalSection( ) e nonCriticalSection( ) são chamados por cada thread. Representamos os três algoritmos distintos estendendo a classe MutualExclusion e implementando os métodos abstratos enteringCriticalSection e leavingCritical$ecton( ).

Usamos a classe TestAlgorithm (Figura 7.3) para criar dois threads e testar cada algoritmo.

```
public class TestAlgorithm
i
    public static void main(String args[ ]) {
      MutualExclusion alg • new Algorithm_l( );

      Worker first • ne* Worker("Runner 0", 0, alg);
      Worker second • new Worker("Runner 1", 1, alg);

      first.start( );
      second.start( );
    )
|
```

Figura 7.3 A classe TestAlgorithm.

7.3.1 Algoritmo 1

Nossa primeira abordagem é deixar os threads compartilharem uma variável inteira comum turn, inicializadaemOou l.Se turn • i, então o thread *í* poderá executar sua scção crítica. Uma solução completa cm Java é apresentada na Figura 7.4.

Essa solução garante que apenas um thread de cada vez possa estar em sua seção crítica. No entanto, não satisfaz o requisito de progresso, já que requer a troca estrita dos threads na execução de sua seção crítica. Por exemplo, se turn = 0 e o thread *T,* estiver pronto para entrar em sua seção crítica, então T, não poderá fazê-lo, mesmo que T„ possa estar em sua seção não-crítica.

O algoritmo 1 introduz o método yield( ) apresentado na seção 6.7.1. Achamada ao mctodoyield( ) mantém o thread 00 estado Executável, mas também permite que a JVM selecione outro thread Executável de prioridade igual para execução.

```
public cUss Algorithm 1 extends MutualExclusion
(
    public Algorithm 1( ) {
      turn - TURN_0;~
    }
    public void enteringCriticalSection(int t) {
      while (turn != t)
        Thread.yield( );
    f

    public void leavingCriticalSection(int t) {
      turn • 1 - t;
    \

    prívate volatile int turn;
|
```

Figura 7.4 Algoritmo 1.

Essa solução também introduz uma nova palavra reservada Java: volatile. A especificação da linguagem Java permite que um compilador faça algumas otimizações, tais como fazer cache do valor de uma variável cm um registrador da máquina, em vez de continuamente atualizar o valor a partir da memória principal. Tais otimizações ocorrem quando um compilador reconhece que o valor da variável permanecerá inalterado, como na instrução

```
while (turn !• t)
    Thread.yield( );
```

No entanto, se outro thread puder alterar o valor de turn, como acontece com o algoritmo I, recomenda-se que o valor de turn seja relido da memória principal durante cada iteração. Declarar uma variável como volati le impede que o compilador faça tais otimizações.

7.3.2 Algoritmo 2

O problema do algoritmo 1 é que ele não retém informações suficientes sobre o estado de cada thread; ele se lembra apenas de qual thread tem permissão para entrar em sua seção crítica. Para resolver esse problema, podemos substituir a variável turn pelo seguinte vetor:

```
boolean[ ] flag • new boolean[2j;
```

Os elementos do vetor são inicializados como false. Sc flag[i] for true, esse valor indica que o thread"/', está pronto para entrar na sua seção crítica. A solução completa em Java é apresentada na Figura 7.5.

Nesse algoritmo, o thread '/'.primeiro define flagfi] como true, indicando que está pronto para entrar em sua seção crítica. Em seguida, T, verifica se o thread T, também não está pronto para entrar em sua seção crítica. Se T, estiver pronto, 7" espera até *T,* indicar que não precisa mais estar na seção crítica (ou seja, até que f 1 ag [j] seja f ai se). Nesse ponto, 7", entra em sua seção crítica. Ao sair da seção crítica, *T,* define seu f 1 ag para false, permitindo que outro thread (se ele estiver esperando) entre em sua seção crítica.

Nessa solução, o requisito de exclusão mútua é atendido. Infelizmente, o requisito de progresso ainda não foi atendido. Para ilustrar esse problema, considere a seguinte sequencia de execução: vamos supor que o thread '/',, defina f 1 ag[0] para true, indicando que ele deseja entrar em sua seção crítica. Antes que possa começar a execução do loop whi le, ocorre uma troca de contexto e o thread T, define f lag[1] também para true. Quando ambos os threads executarem a instrução whi le, eles entrarão em um laço infinito, já que o valor de flag para o outro thread é true.

```
public class Algorithm 2 extends MutualExclusion
1
    public Algorithm_2( ) {
      flag[0] = false;
      flagEl] - false;
    Í
    public void enteringCrtticalSection(int t) {
      int other;

      other • 1 - t;

      flag[t] " true;

      while (flag[other] " true)
        Thread.yield( );
    |

    public void 1eavingCritka1Section(int t) {
      flagft] - false;


    private volatile boolean[ ] flag = new boolean[2];
J
```

Figura 7.5 Algoritmo 2.

7.3.3 Algoritmo 3

Combinando as principais ideias dos algoritmos I e 2, obtemos uma solução correta para o problema de seção crítica, na qual todos os três requisitos são atendidos. Os threads compartilham duas variáveis:

```
booleanf ] flag • new boolean[2];
int turn;
```

Inicialmente, cada elemento do vetor é definido como f ai se, e o valor de turn é indefinido (pode ser 0 ou 1). A Figura 7.6 exibe a solução completa em Java.

```
public class Algorithm 3 extends MutualExclusion
1
    public Algorithm_3( ) {
      flag[0] = false;
      flag[l] • false;
      turn = TURN 0;
    Í
    public void enteringCriticalSection(int t) (
      int other;

      other = 1 - t;

      flag[t] = true;
      turn • other;

      while ( (flag[other] — true) && (turn == other) )
        Thread.yield( );
    )
    public void leavÍngCriticalSection(int t) {
      flagEtJ - false;
    )

    private volatile int turn;
    private volatile booleanf ] flag • new boolean[2];
1
```

Figura 7.6 Algoritmo 3.

Para entrarem sua seção crítica, o rhread 7* primeiro define flag[i] como true, e declara que *è* a vez do outro thread entrar também, se for o caso (turn == j). Se ambos os threads tentarem entrar ao mesmo tempo, turn é definido como i e como j praticamente ao mesmo tempo. Apenas uma dessas atribuições perdura; a outra ocorrerá, mas será sobrescrita imediatamente. O valor final de turn decide qual dos dois threads terá permissão para entrar em sua seção crítica primeiro.

7.4 • Hardware de sincronização

Como ocorre com outros aspectos do software, as características do hardware podem tornar a tarefa de programação mais fácil c melhorar a eficiência do sistema. Nesta seção, apresentamos instruções simples de hardware que estão disponíveis cm muitos sistemas c mostramos como elas podem ser usadas com eficácia para resolver o problema de seção crítica.

O problema de seção crítica poderia ser resolvido de forma simples cm um ambiente monoprocessador se pudéssemos desabilitar a ocorrência de interrupções enquanto uma variável compartilhada estiver sendo modificada. Dessa maneira, poderíamos ter certeza de que a sequência de instruções corrente teria permissão para executar de forma ordenada, sem preempção. Nenhuma outra instrução seria executada, por isso nenhuma modificação inesperada poderia ser feita à variável compartilhada. Infelizmente, essa solução não é viável em um ambiente com vários processadores. Desabilitar interrupções em um multiprocessador pode ser demorado, já que a mensagem deve ser passada a todos os processadores. Essa transmissão de mensagem atrasa a entrada em cada seção crítica, e a eficiência do sistema diminui. Além disso, considere o efeito no clock do sistema, se ele for atualizado através de interrupções.

Assim, muitas máquinas fornecem instruções de hardware especiais, que nos permitem testar e modificar o conteúdo de uma palavra ou trocar o conteúdo de duas palavras, de forma atómica. Podemos usar essas instruções especiais para resolver o problema de seção crítica de uma forma relativamente simples. Em vez de discutir uma instrução específica para uma máquina específica, vamos usar Java para abstrair os principais conceitos por trás desses tipos de instrução. A classe HardwareOata apresentada na Figura 7.7 mostra as instruções.

```
public class HardwareOata
Í
    public HardwareData(boo'ean v) {
      data • v;
    Í

    public boolean get( ) {
      return data;
    I

    public void set(boolean v) 1
      data = v;
    I

    private boolean data;
}
```

Figura 7.7 Estrutura de dados para as soluções de hardware.

A Figura 7.8 apresenta um método que implementa a instrução*Test-And-Set* nadasse HardwareSolution. A característica importante é que essa instrução é executada de forma atómica, ou seja, como uma unidade não-interrompível. Assim, se duas instruções *Test-And-Set* forem executadas simultaneamente (cada qual cm uma CPU diferente), elas serão executadas sequencialmente em alguma ordem arbitrária.

Se a máquina suportar a instrução *Test-And-Set,* poderemos implementar a exclusão mútua declarando lock como sendo um objeto da classe HardwareOata e inicializando-o como false. Todos os threads terão acesso compartilhado a lock. A Figura 7.9 ilustra a estrutura de um thread *Tf.*

```
public class HardwareSolution
{
    public static boolean testAndSet(HardwareData target) {
      HardwareData temp = new HardwareData(target.get( ));

      target.set(true);
      return temp.get( );
    )
```

Figura 7.8 Instrução *Tesí-atld-Set.*

```
HardwareData lock • new HardwareData(false);

wMle(true) t
    while (HardwareSolution.testAndSet (lock))
      Thread.yield( );

    criticalSection( );
    lock.set(false);
    nonCriticalSection( );
í
```

Figura 7.9 Thread utilizando o bloco de operações *Test-and'Set.*

z-n instrução *Swap,* definida no método swap( ) na Figura 7.10, opera com o conteúdo de duas palavras; como a instrução *Test-And-Set,* ela é executada de forma atómica.

Sc a máquina suportar a instrução *Swap,* a exclusão mútua poderá ser obtida da seguinte maneira: Todos os threads compartilham um objeto lock da classe HardwareData, que é inicializado como false. Além disso, cada thread também tem um objeto local key da classe HardwareData. A estrutura do thread T, aparece na Figura 7.11.

```
public static void swap(HardwareData s. Hardware Data b) {
    HardwareData temp = new HardwareData(a.get( ));
    a.set(b.get( ));
    b.set(temp.get( ));
I
```

Figura 7.10 Instrução *Swap.*

```
HardwareData lock • new HardwareData(false);
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
      HardwareSolution.swap(1ock,key);
    }
    while (key.get( ) " true);

    críticalSection( );
    lock.set(false);
    nonCriticalSection( );
```

Figura 7.11 Thread utilizando a instrução *Swap.*

- Semáforos

As soluções para o problema de seção crítica apresentadas na Seção 7.3 não são fáceis de generalizar para problemas mais complexos. Para superar essa dificuldade, podemos usar uma ferramenta de sincronização, denominada semáforo. Um semáforo S é uma variável inteira que, além da inicialização, só é acessada através de duas operações-padrão: P e V. Essas operações receberam seus nomes de termos holandeses: P de *proberen,* que significa testar, e Vde *verhogen,* que significa incrementar*. As definições de P c V são as seguintes:

```
P(S) {
    while S  0
      ;  // no-op

i

V(S) {

)
```

As modificações no valor inteiro do semáforo nas operações P e V devem ser executadas de forma indivisível. Ou seja, quando um thrcad modifica o valor do semáforo, nenhum outro rhread pode modificar simultaneamente o valor do mesmo semáforo. Além disso, no caso de P(S),o teste do valor inteiro de S(S S 0) e de sua possível modificação (S--) também deve ser executado sem interrupção. Na Seção 7.5.2, veremos como essas operações podem ser implementadas; primeiro vamos ver como os semáforos podem ser usados.

7.5.1 Uso

Os sistemas operacionais muitas vezes fazem a distinção entre semáforos de contagem e semáforos binários. O valor de um semáforo de contagem pode variar de modo irrestrito. O valor de um semáforo binário pode variar apenas entre 0 ç 1.

A estratégia geral para usar um semáforo binário no controle do acesso a uma seção crírica é a seguinte (vamos supor que o semáforo seja inicializado em 1):

```
Semaphore  S;

P(S);
criticalSectionf   );
V(S);
```

Assim, podemos usar o semáforo para controlar o acesso à seção crítica em um processo ou rhread. Uma solução generalizada para múltiplos threads aparece no programa Java da Figura 7.12. Cinco threads separados são criados, mas apenas um pode estar na sua seção crítica de cada vez. O semáforo sçm que é compartilhado por todo* os threads controla o acesso à seção crítica.

Os semáforos de contagem podem ser usados para controlar o acesso a determinado recurso em quantidade limitada (finita). O semáforo é inicializado com o número de recursos disponíveis. Cada thrcad que desejar usar um recurso executaria uma operação P no semáforo (decrementando, assim, a contagem). Quando um thrcad libera um recurso, ele realiza uma operação V (incrementando a contagem). Quando a contagem para o semáforo chegar a 0, todos os recursos estarão sendo utilizados. Os threads subsequentes que desejarem utilizar um recurso ficarão bloqueados até que a contagem fique maior do que 0.

' líssas instruções também são conhecidas, respectivamente, como /waitt c /signal/, ou /down/ c /up/. (N.R.T.)

```
public class Worfcer çxtends Thread
t
    public Worker(Semaphore s, String n) {
      name » n;
      sem = s;
    |
    publk void run{ ) {
      while (true) (
        &em.P( );
        System.out.println(name * " is in criticai sectiOn.");
        Runner.critkalSection( );
        iem.V( );
        System.out.println(name * " is out of criticai sextion.");
        Runner.nonCrlticalSection( );
      )
    )

    private Semaphore sem;
    private String name;
\
public class FirstSemaphore
|
    public static void main{String args[ ]) (
      Semaphore sem * new Semaphore(l);
      Workerf ] bees » new Worker[5];

      for (int i = 0; i < 5; i**)
        bees[i] • new Worker(sem, "Worker " +
          (new tnteger(i)).toString( ) );
      for (int i • 0; i < 5; i+*)
        bees[i].start( );
    1
)
```

Figura 7.12 Sincronização utilizando semáforos.

7.5.2 Implementação

A principal desvantagem das soluções de exclusão mútua da Scção 7.2, e da definição de semáforo apresentada aqui, é* que todas exigem espera ocupada. Enquanto um processo estiver em sua seção crítica, qualquer outro processo que tentar entrar em sua seção crítica deverá fazer um laço contínuo no código de entrada. Esse laço contínuo é claramente um problema em um sistema de multiprogramação, no qual uma única CPU é compartilhada entre muitos processos. A espera ocupada desperdiça ciclos de CPU que outros processos poderiam usar de fornia produtiva. Esse tipo de semáforo também é chamado de *spinlock* (porque o processo "gira" enquanto espera um bloco de operações). Os *sphúocks* são úteis nos sistemas com várias processadores. A vantagem de um *spinlock* é que não há necessidade de troca de contexto quando um processo precisa esperar em um bloco de operações, e uma troca de contexto pode demorar muito. Assim, quando os blocos de operações devem ser mais curtos, os *spinlocks* são úteis.

Para superar a necessidade da espera ocupada, podemos modificar a definição das operações de semáforo P e V. Quando um processo executar uma operação P e descobrir que o valor do semáforo não é positivo, ele deverá esperar. No entanto, em vez de utilizar a espera ocupada, o processo poderá se *bloquear.* A operação de bloco de operações coloca um processo cm uma fila de espera associada com o semáforo, e o estado do processo é passado para o estado de espera. Em seguida, O controle é transferido para o escalonador de CPU, que seleciona outro processo para executar.

Um processo bloqueado, esperando em um semáforo S, deve ser retomado quando algum outro processo executar uma operação V. O processo é reiniciado por uma operação *wakeup* (acordar), que muda o processo do estado de espera para o estado de pronto. O processo *é* então colocado na fila de processos prontos. (A CPU pode ou não ser passada do processo cm execução para o novo processo pronto, dependendo do algoritmo de escalonamento de CPU.)

Para implementar semáforos com essa definição, definimos um semáforo como sendo um valor inteiro e uiua lista de processos. Quando um processo precisa esperar em um semáforo, ele é acrescentado à lista de processos daquele semáforo. Uma operação V remove um processo da lista de processos em espera, e acorda esse processo.

As operações de semáforo agora podem ser definidas como:

```
P(S)Í
    value--;
    if(value < 0){
       add this process to list
      block;
    }
}

V(S){
    value + + ;
    if(value  0)í
     remove ã process P from list
     wakeup(P);
    )
}
```

A operação block suspende o processo que o chama. A operação wakeup (P) retoma a execução de um processo bloqueado P. Essas duas operações são fornecidas pelo sistema operacional como chamadas ao sistema básicas.

Observe que, embora em uma definição clássica de semáforos com espera ocupada o valor do semáforo nunca seja negativo, essa implementação pode ter valores de semáforo negativos. Se o valor do semáforo for negativo, seu valor absoluto é o número de processos esperando naquele semáforo. Esse fato é resultado da troca da ordem entre o decremento e o teste na implementação da operação P.

A lista de processos cm espera pode ser facilmente implementada por um ponteiro em cada bloco de controle de processo (PCB). Cada semáforo contém um valor inteiro e um ponteiro para uma lista de PCBs. Uma forma de adicionar e remover processos da lista, que garante uma espera limitada, seria utilizar uma fila FIFO, na qual o semáforo contenha ponteiros de início e fim para a fila. Em geral, no entanto, a lista poderá usar *qualquer* estratégia de enfileiramento. O uso correto de semáforos não depende de alguma estratégia específica para as listas de semáforo.

O aspecto crítico dos semáforos é que eles são executados de forma atómica. Devemos garantir que dois processos não podem executar as operações P e V no mesmo semáforo ao mesmo tempo. Essa situação cria uni problema de seção crítica, que pode ser resolvido de duas maneiras.

Em um ambiente monoprocessador (ou seja, onde só exista uma CPU), podemos simplesmente inibir interrupções durante a execução das operações P e V. Assim que as interrupções forem inibidas, as instruções dos diferentes processos não podem ser intercaladas. Apenas o processo em execução no momento executa, até que as interrupções sejam habilitadas novamente e o escalonador possa retomar o controle-

Em um ambiente multiprocessador, no entanto, a inibição de interrupções não funciona. As instruções de processos distintos (executando em diferentes processadores) podem ser intercaladas de forma arbitrária. Sc o hardware não fornecer instruções especiais, podemos empregar qualquer solução de software correia para o problema de seção crítica (Seção 7.2), na qual as seções críticas consistirão nas operações P e V.

Não eliminamos completamente a espera ocupada com essa definição das operações P c V. Em vez disso, passamos a espera ocupada para as seções críticas dos aplicativos. Além disso, limitamos espera ocupada apenas às seções críticas das operações P e V, c essas seções são curtas (se adequadamente codificadas, não devem ter mais do que 10 instruções). Assim, a seção crítica quase nunca é ocupada, e a espera ocupada ocorre raramente c sõ por um período limitado. Uma situação inteiramente diferente existe com aplicativos, cujas seções críticas talvez sejam longas (minutos ou mesmo horas) ou estejam quase sempre ocupadas. Nesse caso, a espera ocupada é extremamente ineficiente.

Na Seção 7.8.6, veremos como os semáforos podem ser implementados em Java.

7.5.3 Deadlocks e paralisação

A implementação de um semáforo com uma fila de espera poderá resultar cm uma situação na qual dois ou mais processos estão esperando indefinidamente por um evento que somente pode ser causado por um dos processos em espera. O evento em questão c* a execução de uma operação V. Quando tal estado é alcançado, esses processos são considerados em estado de impasse ou deadloek.

Como ilustração, consideramos um sistema que consiste cm dois processos, $P_{„}$ e $P_{„}$ cada qual acessando dois semáforos, *S* e *Q,* iniciadas em I:

Po	*Pi*
P(S);	P(0);
P(Q);	P(S);
V(S);	V(Q);
V(Q);	V(S);

```
public class BoundedBuffer
{
    public BoundedBufferf ) {
      // o buffer está inicialmente vazio
      count • 0;
      in • 0;
      out • 0;
      buffer • new Object[BUFFER_SIZE];
      mutex • new Semaphore(l);
      empty • new Semaphore(8UFFER_SIZE);
      full = new Semaphore(O);
    |
    // o produtor e o consumidor chamam este método para •cochilar"
    public static void napping( ) {
      int sleepTime - (int)(NAPTIME *
        Math.random( ));
      try <
        Thread.s1eep(sleepTime"1000);
      |
      catch(InterruptedExceptÍon e) 1 )
    1
    public void enter(0bject item) (
      //Figura 7.14
    \
    public Object removei ) |
    //Figura 7.15
    I
    private static final int NAP_TIME • 5;
    private static final int BUFFER_SIZE • 5;
    private Objectf ] buffer;
    private int count, in, out;
    // mutex controla o acesso a count, in, out
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;
```

Figura 7.13 Solução para o problema do buffer limitado utilizando semáforos.

```
public void enter(Object item) {
    empty.P( );
    mutex.P( );

    //adiciona um item ao buffer
    *+count;
    buffer[in] • item;
    in » (in + 1) % BUFFERSIZE;

    if (count « BUFFER_SIZE)
        System.out.println("Producer Entered " * item +
          • Buffer RIU.");
    else
        System.out.println{"Producer Entered " • item +
          • Buffer Size •" • count);

    mutex.Vt );
    full.VÍ >;
1
```

Figura 7.14 O método enter( ).

Vamos supor que $P_„$ executa P(S) e /', executa P(Q). Quando P_0 executar P(0), ele deve esperar até que $P,$ execute V (0). Da mesma forma, quando $P,$ executar P(S), ele deve esperar até que $P_„$ execute V (S). Como essas operações V( ) não podem ser executadas, $P_„$ e $P,$ estão em estado de deadlock.

Dizemos que um conjunto de processos está cm estado de deadlock quando todo processo no conjunto estiver esperando por um evento que só pode ser causado por outro processo no conjunto. Os eventos que enfocaremos aqui são a aquisição e a liberação de recursos; no entanto, outros tipos de eventos poderão resultar em deadlocks, conforme demonstrado no Capítulo 8. Nesse capítulo, descrevemos vários mecanismos para lidar com o problema de deadlocks.

Outro problema relacionado com deadlocks é o bloco de operações indefinido ou starvation (estagnação) - uma situação na qual os processos esperam indefinidamente no semáforo. O bloco de operações indefinido poderá ocorrer se adicionarmos e removermos processos da lista associada a um semáforo usando a ordem LIFO.

7.6 • Problemas clássicos de sincronização

Nesta scção, apresentamos uma série de problemas de sincronização diferentes que são importantes principalmente por serem exemplos de uma grande classe de problemas de controle de concorrência. Esses problemas são usados para testar praticamente todos os novos esquemas de sincronização propostos. Os semáforos são usados para sincronização nas nossas soluções.

7.6.1 O problema do buffer limitado

O problema do buffer limitado foi apresentado na Scção 7.1; ele é comumente usado para ilustrar o poder das primitivas de sincronização. A Figura 7.13 mostra uma solução. Um produtor coloca um item no buffer chamando o método enter( ); os consumidores removem os itens chamando o método remove( ).

O semáforo mutex fornece exclusão mútua para os acessos ao conjunto de buffers e é inicializado como 1. Os semáforos empty e fui 1 contam o número de buffers vazios e cheios, respectivamente. O semáforo empty é inicializado com a capacidade do buffer - BUFFER__SIZE; o semáforo fui 1 é inicializado como 0. Observe que a variável count só atende ao propósito de indicar se o buffer está vazio ou cheio para mensagens de saída.

O thread produtor é mostrado na Figura 7.16. O produtor alterna entre "cochilar" um pouco, a produção de uma mensagem e a tentativa de colocar essa mensagem no buffer via método enter{).

O thread consumidor é mostrado na Figura 7.17. O consumidor alterna entre "cochilar" um pouco e o consumo de um item utilizando o método remove( ).

A classe BoundedBufferServer (Figura 7.18) cria os thrcads produtor e consumidor, passando a cada um uma referencia ao objeto BoundedBuffer.

```
public Object remove( ) {
    Object item;

    full.PÍ );
    mutex.P( );

    //remove um item do buffer
    --count;
    item • buffer[out];
out • (out • 1) % BUFTER_SIZE;

    if (count -• 0)
      System.out.println("Consumer Consumed • + item +
        "Buffer EHPTY-);

    else
      System.out.printlnfConsumer Consumed " • item +
        "Buffer Slze •• + count);

    mutex.V( );
    empty.V( );

    return item;
1
```

Figura 7.15 O método removei).

```
import java.util.';

public class Producer extends Thread
|
    public Producer(8oundedBuffer b) (
      buffer • b;
    |
    public void run( ) {
      Oate message;

      while (true) {
        BoundedBuffer.napping( );
        //produz um item e insere o item no buffer
        message • new Date( );
        System.out.println("Producer produced " * message);
        buffer.enter(message);
    |
    |

    private BoundedBuffer buffer;
|
```

Figura 7.16 Thread produtor.

```
import java.util.*;

public class Consumer extends Thread
{
    public Consumer(BoundedBuffer b) {
      buffer • b;


    public void run( ) {
      Date message;

      while (true) (
        BoundedBuffer.nappingí );
        // consome um item do buffer
        System.out.printlnCConsumer wants to consume");
        message • (Date)buffer.remove( );
    }
    í

    private BoundeBuffer buffer;
»
```

Figura 7.17 Thread consumidor.

```
public class BoundedBufferServer
í
    public static void main(String args[ ]} {
        BoundedBuffer server • new Bounded8uffer( );

        // cria os threads consumidor e produtor
        Producer producerlhread = new Producer(server);
        Consumer consumerThread » new Con$umer(server);

        producerThread.start( );
        consumerThread.start( );
    }
}
```

Figura 7.18 A classe BoundedBufferServer.

7.6.2 O problema dos leitores-cscritores

Um banco de dados deve ser compartilhado entre vários threads concorrentes. Alguns desses threads talvez queiram apenas ler o banco de dados, enquanto outros talvez desejem atualizá-lo (ou seja, ler e gravar dados). Fazemos a distinção entre esses dois tipos de threads chamando os primeiros de leitores e os últimos de escritores. Obviamente se dois leitores acessarem os dados compartilhados ao mesmo tempo, não haverá efeitos adversos. No entanto, se um escritor e algum outro thread (leitor ou escritor) acessar o banco de dados ao mesmo tempo, poderá haver confusão.

Para evitar esse tipo de problema, os escritores deverão ter acesso exclusivo ao banco de dados compartilhado. Rsse requisito leva ao problema dos leitores-escritorcs. Como já mencionado, esse problema tem sido usado para testar praticamente todas as novas primitivas de sincronização. O problema dos leitores-escritores tem muitas variações, todas envolvendo prioridades. A mais simples, chamada de *primeiro* problema dos leitores-escritores, requer que nenhum leitor seja mantido cm espera a menos que um escritor já tenha obtido permissão para usar o banco de dados compartilhado. F.m outras palavras^ nenhum leitor deve esperar o término de outros leitores simplesmente porque um escritor está esperando. O *segundo* problema dos leitores-escritores requer que o escritor, assim que estiver pronto, faça sua escrita o mais rápido possível. Km outras palavras, se um escritor estiver esperando para acessar o objeto, nenhum novo leitor poderá iniciar a leitura.

Observamos que uma solução para esses problemas poderá resultar em paralisação. No primeiro caso, os escritores poderão sofrer de paralisação e, no segundo caso, os afetados serão os leitores. Por isso, outras variantes do problema têm sido propostas. Na sequência apresentamos os arquivos de classe Java para uma solução do primeiro problema dos leitores-cscritorcs. Ela não trata o problema de paralisação. (Nos exercícios ao final do capítulo, uma das perguntas solicita que a solução seja alterada de modo que fique livre de paralisação.) Cada leitor alterna entre o estado suspenso e de leitura, conforme indicado na Figura 7.19. Quando um leitor desejar ler o banco de dados, ele chamará o método startRead ( ); quando tiver terminado a leitura, ele chamará endReadí). Cada escritor (Figura 7.20) opera de forma similar.

```
public class Reader extends Thread
i
    public Reader(irit r, Oatabase <Jb) (
        readerNum • r;
        server = db;
    t

    public void run( ) (
        1nt c;

        while (true) {
            System.out.println("reader • * readerNum *
              " is sleeping");

            Database.napping( );
            System.out.printlnCreader • * readerNum •
              • wants to read");
            c - server.startRead( );

            // você tem acesso de leitura no banco de dados
            System.out.println("reader • t readerNum +
              " is reading, Count =" * c);
            Oatabase.napping( );

            c = server.endRead( );
            System.out.println("reader • » readerNum +
              " is done reading. Count •" + c);
    )
)

    private Database server;
    privat& int readerNum;
1
```

Figura 7.19 Um leitor.

Os métodos chamados por cada leitor e escritor são definidos na classe Database na Figura 7.21. A variável readerCount controla o número de leitores. O semáforo mutex é usado para garantir a exclusão mútua quando readerCount é atualizado. O semáforo db funciona como um semáforo de exclusão mútua para os escritores. Também é utilizado pelos leitores para evitar que os escritores entrem no banco de dados enquanto este estiver sendo lido. O primeiro leitor realiza uma operação P( ) em db, evitando, assim, que algum escritor entre no banco de dados. O leitor final realiza uma operação V ( ) em db. Observe que, se um escritor estiver ativo no banco de dados e *n* leitores estiverem esperando, então um leitor será colocado na fila em db, e *n* - 1 leitores serão colocados na fila em mutex. Observe também que, quando um escritor executa db.V( ), é possível retomar a execução dos leitores em espera ou de um único escritor em espera. A selcção é feita pelo escalonador.

```
public class Writer extends Thread
1
    public Writer(int w, Database db) {
        writerNum • w;
        server • db;
    )

    public voíd run( ) {
        while (true) {
            System.out.println(Mwríter • + writerNum +
              " is sleeping.");
              Database.napping( );

            System.out.println("writer " + writerNum •
              " wants to write.');
            server.startWrite( );

            // você tem acesso de escrita no banco de dados
            System.out.println("writer " + writerNum •
              " is writing.");
            Database.napptng( );

            server.endWrite( ):
            SyStem.out.println{"writer * + writerNum •
              " is done writing.");
        Í
    |

    private Database server;
    private int writerNum;
)
```

Figura 7.20 Um escritor.

```
public class Database
1
    public Databasef ) (
      readerCoimt ° 0;
      mutex = new Semaphore(l);
      db • new Semaphore(l);
    J

    //os leitores e escritores chamam este método para "cochilar"
    public static void nappingf ) {
      int sleepTime » (int){NAP TIME " Hath.random( ));
      try {
          Thread.sleep(sleepTime*1000);
      }
      catch(InterruptedException e) { )
    )

    public int startRead( ) {
      //Figura 7.22
    )

    public int endRead( ) (
      //Figura 7.22
    )

    public void startWriteí ) {
      //Figura 7.23
    J
```

Figura 7.21 O hanco de dados para o problema dos leitores-cscritores.

```
    public void endWrite( ) (
      //Figura 7.23


    private static final int NUM_OF_READERS - 3;
    private static final int NUM-OF_WRITERS = 2;
    private static final int NAP TIME - 15;

    private int readerCount;
    private Semaphore mutex;
    private Semaphore db;
Í
```

Figura 7.21 Continuação

```
public int startReadí ) {
    mutex.P( );
    ••readerCount;

    //sç sou o primeiro leitor, informe todos
    //os outros que o banco de dados está sendo lido
    if (readerCount •• 1)
      db.P( );

    mutex.V( );

    return readerCount;
J

public int endRead( ) {
    mutex.P( ):
    —readerCount;

    //$e sou o último leitor, informe todos
    //os outros que o banco de dados não está mais sendo lido
    if (readerCount -= 0)
      db.V( );

    mutex.V( );

    return readerCount;
I
```

Figura 7.22 Métodos chamados pelos leitores.

7.6.3 O problema do jantar dos filósofos

Considere cinco filósofos que passam suas vidas meditando e comendo. Os filósofos compartilham uma mesa redonda comum cercada por cinco cadeiras, cada qual pertencente a um filósofo. No centro da mesa está uma tigela de arroz, e a mesa está posta com cinco pauzinhos *(hashi)* (Figura 7.24). Quando um filósofo medita, não interage com seus colegas. De vez em quando, um dos filósofos fica com fome c tenta pegar os dois pauzinhos mais próximos de si (os pauzinhos que estão entre ele e seus colegas da esquerda e direita). Um filósofo só pode pegar um pauzinho de cada vez. Obviamente, nio poderá pegar um que já esteja na mão de um colega. Quando o filósofo com fome está de posse dos dois pauzinhos ao mesmo tempo, ele poderá comer sem soltá-los. Quando termina de comer, o filósofo solta os pauzinhos e volta a meditar.

O problema do jantar dos filósofos é considerado um problema clássico de sincronização, não por causa de sua importância prática, nem porque os cientistas da computação não gostam de filósofos, mas porque ele *é* um exemplo de uma vasta classe de problemas de controle de concorrência. E uma representação simples da necessidade de alocar vários recursos entre vários processos, sem incorrer em deadlocks ou paralisação.

```
publíc void startWrtteí ) {
    db.P( );
|
public void endWrite( ) (
    db.V( );
)
```

Figura 7.23 Métodos chamados pelos escritores.

Figura 7.24 O jantar dos filósofos.

Uma solução simples consiste em representar cada pauzinho por um semáforo. Um filósofo tenta agarrar o pauzinho executando uma operação P naquele semáforo; o filósofo solta o pauzinho executando a operação V nos semáforos apropriados. Assim, os dados compartilhados são:

```
Semaphore chopStick[ ] • new Semaphore[5];
```

em que todos os elementos de chopStick são inicializadoscomo 1. A estrutura do filósofo/é apresentada na Figura 7.25.

```
whtle(tnie) (
    //pega o pauzinho da esquerda
    chop$tick[i).P( );
    //pega o pauzinho da direita
    chopStick[(i + 1) % 5].P( );
    eating( );
    //devolve o pauzinho da esquerda
    chopStick[i].V( );
    //devolve o pauzinho da direita
    chopStick[(i • 1) H 5].V( );
    thinkingí );
|
```

Figura 7.25 A estrutura do filósofo /'.

Embora essa solução garanta que dois filósofos vizinhos não estarão comendo simultaneamente, ela deve ser rejeitada porque tem a possibilidade de criar um deadlock. Vamos supor que todos os cinco filósofos fiquem com fome ao mesmo tempo, e cada um pegue o pauzinho à sua esquerda. Todos os elementos de chopStlck agora serão iguais a 0. Quando cada filósofo tentar pegar o pauzinho à sua direita, ficará esperando para sempre.

Várias soluções possíveis para o impasse são listadas a seguir. Essas soluções evitam o deadlock, impondo restrições aos filósofos:

- Permitir a presença de no máximo quatro filósofos ao mesmo tempo à mesa.
- Permitir que um filósofo pegue o seu pauzinho apenas se os dois pauzinhos estiverem disponíveis (observe que o filósofo deverá pegá-los em uma seção crítica).
- Usar uma solução assimétrica; por exemplo, um filósofo ímpar pega primeiro o pauzinho à sua esquerda e depois o da sua direita, enquanto um filósofo par pega o da direita e depois o da esquerda.

Na SeçãO 7.7, apresentamos uma solução ao problema do jantar dos filósofos que não incorre em deadlocks. Qualquer solução satisfatória ao problema do jantar dos filósofos deverá evitar a possibilidade de um dos filósofos morrer de fome. Uma solução livre de impasse, ou de deadlock, não elimina necessariamente a possibilidade de paralisação.

7.7 • Monitores

Embora os semáforos forneçam um mecanismo conveniente e eficaz para a sincronização de processos, seu uso incorreto poderá resultar em erros de *sincronismo* difíceis de detectar, já que esses erros só acontecem se ocorrerem determinadas sequencias de execução, e essas sequências nem sempre ocorrem.

Vimos um exemplo desses tipos de erros no uso de contadores na nossa solução ao problema do produtor-consumidor (Seçâo 7.1). Nesse exemplo, 0 problema dp *sincronismàpcontcizia* apenas raramente e ainda assim o valor do contador parecia ser razoável - errado em apenas 1 unidade. Entretanto, essa solução obviamente não é aceitável. Por esse motivo, os semáforos foram introduzidos.

Infelizmente, esses erros de sincronismo ainda ocorrem com o uso de semáforos. Para ilustrar como, vamos revisar a solução de semáforo para o problema de seçâo crítica. Todos os processos compartilham uma variável de semáforo mutex, queé inicializada em 1. Cada processo deve executar mutex.P( ) antes de entrar na seçâo crítica c mutex. V ( ) depois disso. Se essa sequência não for observada, dois processos podem estar em suas scções críticas ao mesmo tempo. Vamos examinar as várias dificuldades que poderão resultar disso. Observe que essas dificuldades surgirão mesmo se um *único* processo não tiver um bom comportamento. Essa situação poderá ser resultado de um erro honesto de programação ou de um programador *nâo-cooperativo,* ou melhor, mal-intencionado.

- Vamos supor que um processo troque a ordem na qual são executadas as operações P( ) e V ( ) no semáforo *mutex,* resultando na seguinte execução:

```
mutex.V( );
cr1ticalSection( );
mutex.P( );
```

  Nessa situação, vários processos podem estar executando em suas seçõçs críticas ao mesmo tempo, violando o requisito de exclusão mútua. Esse erro só poderá ser descoberto se vários processos estiverem ativos simultaneamente em suas seções críticas. Observe que essa situação nem sempre pode ser reproduzida.

- Vamos supor que um processo substitua mutex.V( ) por mutex.P( ). Ou seja, ele executa

```
mutex.p( );
crítica)Section( );
mutex.P( };
```

  Nesse caso, ocorrerá um deadlock.

- Vamos supor que um processo omita mutex.P( ), ou mutex.V( ), ou ambos. Nesse caso, a exclusão mútua será violada ou ocorrerá um deadlock.

Esses exemplos ilustram que vários tipos de erros podem ser gerados facilmente quando os programadores utilizam semáforos de forma incorreta para resolver o problema de seção crítica. Problemas semelhantes podem surgir em outros modelos de sincronização que discutimos na Seção 7.6.

Para lidar com tais erros, pesquisadores desenvolveram estruturas cm linguagem de alto nível. Nesta scção, vamos descrever uma estrutura de sincronização de alto nível - o tipo *monitor.*

Lembre-se que um típo, ou tipo de dados abstrato, encapsula dados privados com métodos públicos para realizar operações com os dados. Um monitor apresenta uma série de operações definidas pelo programador que recebem exclusão mútua no monitor. O tipo monitor também contêm a declaração de variáveis cujos valores definem o estado de uma instância desse tipo, juntamente com o corpo dos procedimentos ou funções que operam essas variáveis. O pseudocódigo semelhante a Java que descreve a sintaxe de um monitor é:

```
monitor    nome-do-monitor
{
    //declarações de variável
    pubtic entry pl(...) {
      • **
    }
    public entry p2(...){
      *--
    }
Í
```

A implementação interna de um tipo monitor não pode ser acessada diretamente pelos vários threads. Um procedimento definido em um monitor só poderá acessar as variáveis que estiverem declaradas localmente no monitor e qualquer parâmetro formal que for passado para os procedimentos. O encapsulamento fornecido pelo tipo monitor também limita o acesso às variáveis locais apenas pelos procedimentos locais.

A estrutura do monitor proíbe o acesso concorrente a todos os procedimentos definidos no monitor. Portanto, apenas um thread (ou processo) de cada vez pode estar ativo no monitor em determinado momento. Consequentemente, *o* programador não precisa codificar essa sincronização explicitamente; ela está incorporada no tipo monitor.

Variáveis do tipo condi tion desempenham um papel especial nos monitores, por conta de operações especiais que podem ser chamadas sobre elas: wai t e si gnal. Um programador que precise escrever seu próprio esquema de sincronização personalizado pode definir uma ou mais variáveis do tipo condition:

```
condition x,y;
```

A operação x.wai t; significa que o thread que chama essa operação ficará suspenso ate" que outro thread chame

```
x.signal;
```

A operação x.signal retoma exatamente um thread. Se nenhum thread estiver suspenso, a operação signa J não tem efeito; ou seja, o estado de x é como se a operação nunca tivesse ocorrido (Figura 7.26). Compare esse esquema com a operação V com semáforos, que sempre afota o estado do semáforo.

Agora vamos supor que, quando a operação x.signal for chamada por um thread P, exista um thread *Q* suspenso associado com a condição x. Claramente, se o thread Q suspenso pode retomar sua execução, o thread *P* de sinalização deve esperar. Caso contrário, *P* e *Q* ficarão ativos ao mesmo tempo no monitor. Observe, no entanto, que os dois threads podem conccitualmcnte continuar com sua execução. Existem duas possibilidades:

1. *Signal-und-Waií* (sinalizar e esperar) - Pespera até *Q* sair do monitor, ou espera por outra condição.
2. *Signal-and-Conti/iue* (sinalizar e continuar) - *Q* espera até *P* sair do monitor, ou espera por outra condição.

Existem argumentos razoáveis em favor da adoção das duas opções. Como *P* já estava executando no monitor, *Signal-and-Continue* parece mais razoável. No entanto, se deixarmos *P* continuar, então, quando *Q* tiver sido retomado, a condição lógica pela qual *Q* estava esperando pode não ser mais válida. *Signai-and-Wait* foi defendido por Hoare, principalmente porque o argumento anterior a seu favor se traduz diretamente em

Figura 7.26 Monitor com variáveis de condição.

regras de prova simples e elegantes. Uni meio-termo entre essas duas opções foi adotado na linguagem Concurrent Pascal (Pascal Concorrente). Quando o thread P executa a operação signal, ele imediatamente sai do monitor. Portanto, *Q* é imediatamente retomado. Esse modelo c menos poderoso do que o modelo de Hoarc, porque um thread não pode sinalizar mais de uma ve*/. durante unia única chamada de procedimento.

Vamos ilustrar esses conceitos apresentando uma solução livre de deadlocks para o problema do jantar dos filósofos. Lembre-se de que um filósofo pode pegar seus pauzinhos apenas se ambos estiverem disponíveis. Para codificar essa solução, ê preciso fazer a distinção entre três estados nos quais o filósofo pode se encontrar. Para isso, apresentamos as seguintes estruturas de dados:

```
int[ ] state - new int[5];
static final tnt THINKING • 0;
static final int HUNGRY • 1;
static final int EATING = 2;
```

O filósofo; só pode definir a variável state[i] = EATING se seus dois vizinhos não estiverem comendo, ou seja, a condição (state[(i * 4) % 5] !=EATING) and (state[(i + 1) % 5] I=EATING) é verdadeira.

Também precisamos declarar

```
condit1on[ ] self = new condttion[5];
```

onde o filósofo / pode se atrasar quando estiver com fome, mas não puder obter os pauzinhos necessários.

Agora estamos em condições de descrever nossa solução para o problema do jantar dos filósofos. A distribuição dos pauzinhos é controlada pelo monitor dp, que é uma instância do tipo monitor dini ngPhi 1 osophers, cuja definição usando um pseudocódigo semelhante à Java é apresentada na Figura 7.27. Cada filósofo, antes de começar a comer, deve chamar a operação pi ckUp( ). Isso poderá resultar na suspensão do thread do filósofo. Após a conclusão bem-sucedida da operação, o filósofo pode comer. Depois de comer, o filósofo chama a operação putDown( ), e começa a pensar. Assim, o filósofo / deve chamar as operações pickUpt) e putDown( ) na seguinte sequência:

```
dp.piCkUp(i);
eat( );
dp.putDQwn(i);
```

Ê fácil mostrar que essa solução garante que dois filósofos vizinhos não estarão comendo ao mesmo tempo *t* que não haverá deadlocks. Observamos, no entanto, que é possível que um filósofo morra de fome. Não vamos apresentar uma solução para esse problema, pedindo, em vez disso, que você desenvolva uma na seção de exercícios mais adiante.

```
monitor diningPhilosophers {
    int[ ] state = new int[5];
    static final int THINKING - 0;
    static fina» int HUNGRY = 1;
    static final int EATING • 2;
    condition[ ] self • new condition[5];

    public diningPhilosophers (
      for (int i = 0; i < 5; i++)
        state[i] • THINKING;
    |

    public entry pickUp(ínt 0 (
      statefi] » HUNGRY;
      test(i);
      if (statefi] ! • EATING)
        self[i].wait;
    |

    public entry putOown(int i) {
      state[1] = THINKING;
      // testar vizinhos à esquerda e à direita
      test((i • 4) % 5);
      test((i + 1) % 5);
    I

    private test(int i) I
      if { (state[{i + 4) % S] != EATING) &&
           (statefi] " HUNGRY) S&
           (state[(i • 1) * 5] !* EATING) ) {
             state[i) = EATING;
             self[f].signal;

      1
    )
\
```

Figura 7.27 Uma solução com monitor para o problema do jantar dos filósofos.

7.8 • Sincronização em Java

Nesta seção, descrevemos como Java sincroniza a atividade dos threads, permitindo ao programador desenvolver soluções generalizadas garantindo exclusão mútua entre threads. Uma aplicação que garanta a consistência de dados mesmo quando está sendo acessada de forma concorrente por múltiplos threads é considerada segura para thread *{thread safe).*

7.8.1 Buffer limitado

A solução de memória compartilhada ao problema do buffer limitado descrita no Capítulo 4 tem dois problemas. Em primeiro lugar, tanto o produto quanto o consumidor utilizam laço de espera ocupada se o buffer estiver cheio ou vazio. Em segundo lugar, conforme indicado na Seção 7.1, a condição de corrida na variável count ê compartilhada pelo produtor e consumidor. Esta seção aborda esses e outros problemas e desenvolve uma solução utilizando mecanismos de sincronização Java.

7.8.1.1 Espera ocupada

A espera ocupada foi apresentada na Seção 7.5.2, quando examinamos uma implementação das operações de semáforo P e V. Nesta seção, descreveremos como um processo poderia se bloquear como uma alternativa à espera ocupada. Uma forma de conseguir esse bloco de operações em Java seria fazer com que o thread chamasse o método Thread.yield( ). Lembre-se da Seçâo 7.3.1 que, quando um thread chama o método yield( ), o thread permanece no estado Executável, mas permite que a JVM selecione para executar outro thread Executável de igual prioridade. O método yi el d ( ) faz uso mais eficaz da CPU do que a espera ocupada. No entanto, nesta instância, usar uma ou outra opção poderia levar a um deadlock.

Aqui está uma situação que poderia causar um deadlock. Lembre-se de que a JVM escalona threads usando um algoritmo baseado em prioridades. A JVM garante que o thread com prioridade mais alta dentre todos os threads no estado Executável executará antes da execução de um thread de prioridade mais baixa. Se o produtor tiver uma prioridade mais alta do que a do consumidor e o buffer estiver cheio, o produtor entrará no laçowhilee ficará em espera ocupada ou chamará yield ( ) para outro thread Executável de igual prioridade, enquanto espera que count seja decrementado abaixo de BUFFER_SIZE. Enquanto o consumidor tiver prioridade mais baixa do que a do produtor, ele não conseguirá ser escalonado para execução pela JVM e, portanto, nunca conseguirá consumir um item e liberar espaço do buffer para o produtor. Nessa situação, o produtor e o consumidor ficam em deadlock. O produtor fica esperando que o consumidor libere espaço de buffer c o consumidor fica esperando ser escalonado pela JVM. Mais adiante veremos que existe uma alternativa melhor do que a espera ocupada ou método yiel d( ) durante a espera pela ocorrência de uma condição desejada.

7.8.1.2 Condição de corrida

Na Seção 7.1, vimos um exemplo das consequências da condição de corrida *(racecondition)* na variável compartilhada count. A Figura 7.28 ilustra como Java evita as condições de corrida gerenciando os acessos concorrentes a dados compartilhados.

Essa situação introduz uma nova palavra reservada: synchronized. Todo objeto Java tem associado a ele um único bloco de operações. O objeto que é uma instância da classe BoundedBuf fer tem um bloco de operações associado a ele. Normalmente, quando um objeto está sendo referenciado (ou seja, seus métodos estão sendo chamados), o bloco de operações é ignorado. Quando um método é declarado como synchroni zed, no entanto, a chamada ao método requer a posse do bloco de operações para o objeto. Se o bloco de operações estiver nas mãos de outro thread, o thread que chama o método synchroni zed é bloqueado e colocado no *conjunto de entrada (entry set)* para o bloco de operações do objeto. O conjunto de entrada representa o conjunto de threads que espera o bloco de operações ficar disponível. Se o bloco de operações estiver disponível quando um método synchronized for chamado, o thread que chama se torna proprietário do bloco de operações do objeto e poderá entrar no método. O bloco de operações é liberado quando o thread sai do método. Sc o conjunto de entradas do bloco de operações não estiver vazio quando o bloco de operações for liberado, a JVM seleciona um thread arbitrário do seu conjunto como o novo proprietário do bloco de operações. A Figura 7.29 ilustra como o conjunto de entradas funciona.

Se o produtor chamar o método enter ( ), conforme indicado na Figura 7.28 e o bloco de operações para o objeto estiver disponível, o produtor torna-se o proprietário do bloco de operações; ele então pode entrar no método, onde poderá alterar o valor de count e outros dados compartilhados. Se o consumidor tentar chamar o método remove ( ) enquanto o produtor possuir o bloco de operações, o consumidor bloqueará porque o bloco de operações não está disponível. Quando produtor sair do método enter{), ele liberará o bloco de operações. O consumidor agora pode adquirir o bloco de operações e entrar no método remove.

A primeira vista, esta abordagem parece pelo menos resolver o problema de ter uma condição de corrida na variável count. Como os métodos enter( ) e removei) foram declarados como synchronized, garantimos que apenas um thread pode estar ativo em um desses métodos de cada vez. No entanto, a propriedade do bloco de operações levou a outro problema. Vamos supor que o buffer está cheio e que o consumidor está suspenso. Se o produtor chamar o método enter( ), ele poderá continuar porque o bloco de operações está disponível. Quando o produtor chama o método enter( ), ele verifica que o buffer está cheio e executa o

método yiel d( ). O produtor ainda detém o bloco de operações para o objeto. Quando o consumidor acordar e tentar chamar o método removei) (que acabaria liberando espaço do buffer para o produtor), ele bloqueará porque não detém o bloco de operações para o objeto. Assim, vemos outro exemplo de deadlock. Tanto o produtor quanto o consumidor não conseguem prosseguir porque (1) o produtor está bloqueado esperando que o consumidor libere espaço no buffer e (2) o consumidor está bloqueado esperando que o produtor libere o bloco de operações.

```
public synchronized void enter(Object item) {
    while (count == BUFFER SIZE)
      Thread.yie1d( );

    »+count;
    buffer[in] - item;
    in - (in + 1) * BUFFER SIZE;
)
public synchronized Object remove( ) {
    Object item;

    while (count == 0)
      Thread.yield( );

    --count;
    item • bufferfout];
    out • (out + 1) % 8UFFER_SIZE;

    return item;
```

Figura 7.28 Métodos sincronizados enter( ) e removei).

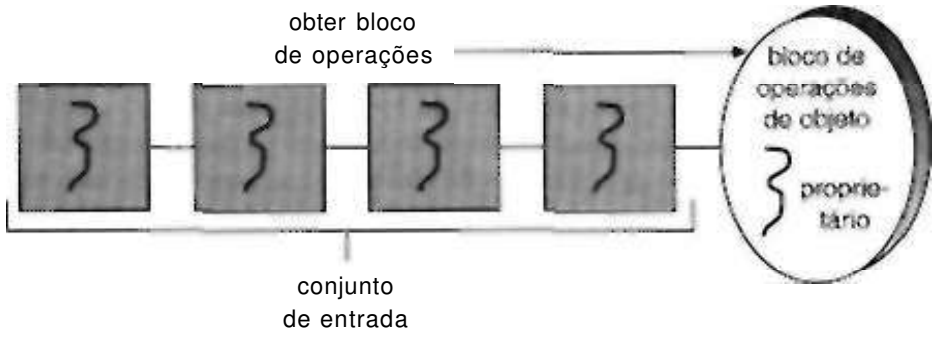


Figura 7.29 Conjunto de entrada.

Assim, quando cada método é declarado synchroni zed, é possível evitar a condição de corrida nas variáveis compartilhadas. No entanto, a presença do laço yield( ) levou a outro tipo de deadlock possível.

AFigura7.30abordaolaçoyield( ) introduzindo dois novos métodos Java: wait( )cnotify( ).Além de ter um bloco de operações, cada objeto também tem um conjunto de espera *(u/ait set)* associado a ele. Esse conjunto de espera consisie em um conjunto de threads e inicialmente está vazio. Quando um thread entra cm um método synchroni zed, ele detém o bloco de operações para o objeto. No entanto, esse thread pode determinar que não consegue prosseguir porque determinada condição não foi atendida. Isso acontecerá se o produtor chamar o método enter( ) e o buffer estiver cheio. Idealmente, o thread liberará o bloco de operações Q esperará até que a condição que permita a sua continuação seja atendida, lissa solução teria evitado a situação de deadlock descrita anteriormente.

```
public synchronized void enter(Object item) (
    while (count •- BUFFER^SIZE) í
      try {
        wait( );
      Í
      catch(InterruptedException e) 1 }
    I
    t+count;
    buffer[in] • item;
    in * (in + 1) % BUFFER SIZE;

    notifyí );
}

public synchronized Object renwve( ) (
    Object item;

    while (count <•- 0) |
      try {
        «ait( );
      1
      catch (InterruptedException e) { )
    \
    --count;
    item • buffer[out);
    out " (out + 1) % BUFFERJIZE;

    nottfy( );

    return item;
```

Figura 7.30 Métodos enter( ) c removei) usando wait( ) e notify( ).

Quando um thread chama o método wait( ), acontece o seguinte:

1. O thread libera o bloco de operações para o objeto.
2. O estado do thread é definido como Bloqueado.
3. O thread é colocado no conjunto de espera para o objeto.

Considere o exemplo na Figura 7.30. Se o produtor chamar o método enter( ) e verificar que o buffer está cheio, ele chamará o método wai t( ). Essa chamada libera o bloco de operações, bloqueia o produtor e coloca o produtor no conjunto de espera para o objeto. A liberação do bloco de operações permite que o consumidor entre no método remove( ), onde ele libera espaço no buffer para o produto. A Figura 7.3 1 ilustra a entrada e os conjuntos de espera para um bloco de operações.

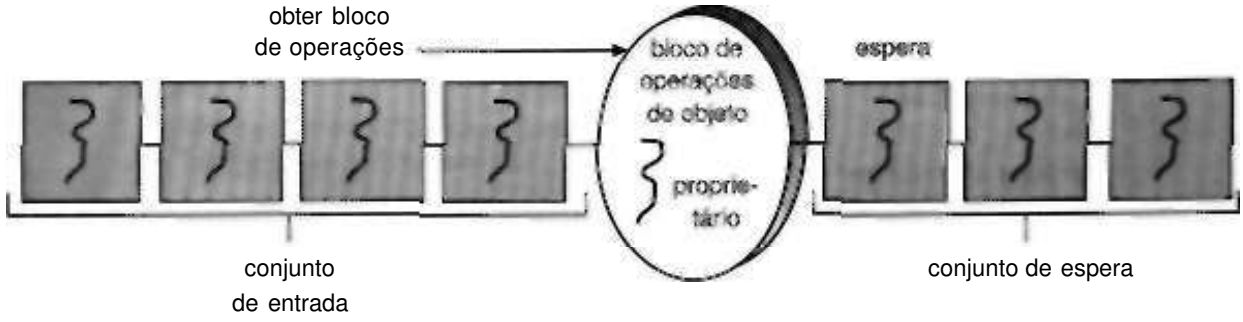


Figura 7.31 Conjuntos de entrada e de espera.

Como o thread consumidor sinaliza que o produtor pode prosseguir? Normalmente, quando um thread sai de um método synchroni zed, a ação default c o thread que está saindo liberar apenas o bloco de operações associado com o objeto, possivelmente removendo um thread do conjunto de entrada e dando a ele a posse do bloco de operações. No entanto, no final dos métodos enter( ) eremove{), existe uma chamada ao método notify(J.A chamada a nofityí):

1. Kscolhe um thread arbitrário T da lista de threads no conjunto de espera.
2. Passa T do conjunto de espera para o conjunto de entrada.
3. Ajusta o estado de T de Bloqueado para Kxecutável.

T agora pode competir pelo bloco de operações com os outros threads. Assim que T tiver obtido controle do bloco de operações novamente, ele retornará da chamada de wait( ), onde poderá verificar o valor de count novamente.

Para descrever os métodos wait( ) e notifyt) nos termos do programa mostrado na Figura 7.30, consideramos que o buffer está cheio e o bloco de operações para o objeto está disponível.

- O produtor chama o método enter( ), verifica se o bloco de operações está disponível e entra no método. Assim que estiver no método, o produtor determina que o buffer está chirio e chama o método wai t ( ). Essa chamada (I) libera o bloco de operações para o objeto e (2) ajusta o estado do produtor como Bloqueado, colocando o produtor no conjunto de espera para o objeto.
- Por fim, o consumidor chama o método remove( ) porque o bloco de operações para o objeto agora está disponível. O consumidor remove um item do buffer e chama nof i fy( ). Observe que o consumidor ainda detém o bloco de operações para o objeto.
- A chamada a nof i ty ( ) remove o produtor do conjunto de espera para o objeto, move o produtor para o conjunto de entrada e ajusta o estado do produtor para Kxecutável.
- O consumidor sai do método remove ( ). Ao sair desse método libera o bloco de operações para o objeto.

```
public class BoundedBuffer
(
    public BoundedBuffer( } {
      //o buffer está inicialmente vazio
      count • 0;
      in "0;
      Out • 0;
      buffer = new Object[BUFfER SIZE];
    1
    //o produto»- e o consumidor chamam este método para 'cochilar"
    public statíc void napp»ng( ) {
      int sleepTime • (int) (NAP TIME • Hath.random( ) );
      try {
        Thread.5leep{sleeplime*l000);
      )
      catch(IntçrruptedExceptÍon e) ( )
    1
    public synchronized void enter(0bject item) 1
      //Figura 7.33
    )
    public synchronized Object remove( ) {
      //Figura 7.34
    1
    private static final int NAf»_TIME • 5;
    private Static final int BUFFER_SIZE = 5;
    private int count, in, out;
    private Object[ ] buffer;
í
```

Figura 7.32 Buffer limitado.

```
public synchronized void enter(Object item) (
    while (count « BUFFER^SIZE) {
      try {
        wait( );
      |
      catch (InterruptedException e) { )
    )
    //adiciona um item ao buffer
    ••count;
    buffertin] - item;
    in - {in + 1) % BUFFERJIZE;

    if (count « BUFF£R_SI?£)
      System.out.println("F»roducer Entered • + item *
        • Buffer FULL");

    else
      System.out.println("f»roducer Entered " • item +
        " Buffer Size • • + count);

    notify( );
I
```

Figura 7.33 O método enter(h

- O produtor tenta readquirir o bloco de operações c é bem-sucedido, retomando a execução no retorno da chamada a wai t ( ). O produtor testa o laço whi 1 e, observa que existe espaço disponível no buffer e continua com o restante do método enter( ). Como não hà thread no conjunto de espera para o objeto, a chamada a notifyf) *é* ignorada. Quando o produtor sai do método, ele libera o bloco de operações para o objeto.

```
public synchronized Object removei ) {
    Object item;

    while (count « 0) {
      try (
        wait( );
      |
      catch (InterruptedException e) ( |
    |
    //remove um item do buffer
    --count;
    item • buffer[out];
    out • (out + 1) H BUFFER_SIZE;

    If (count •• 0)
      System.out.println('Consumer Consumed " * item •
        u Buffer EHPTY");
    else
      System.out.println("Consumer Consumed " • item *
        " Buffer Size = " • count);

    notify( );

    return item;
i
```

Figura 7.34 O método removei).

7.8.2 Solução completa

A seguir apresentamos a solução completa com múltiplos thrcads, que utiliza memória compartilhada, para o problema do buffer limitado. A classe BoundedBuffer mostrada na Figura 7.32 implementa os métodos synchronized enter( ) e removei). Essa classe pode ser substituída pela classe BoundedBuffer que foi utilizada na solução baseada em semáforo para esse problema, apresentada na Seção 7.6.1.

7.8.3 O problema dos lcitores-escritores

Agora podemos oferecer uma solução ao primeiro problema dos leitorcs-escritorcs usando a sincronização Java. Os métodos chamados porcada leitor e escritor são definidos na classe Database na Figura 7.35. A variável readerCount controla o número de leitores e dbReading é definido como true, se o banco de dados estiver sendo lido no momento, e como faise, caso contrário. dbWri ting indica se o banco de dados está sendo acessado por um escritor. Os métodos startReadí), endRead( ), startWrite e endWrite são declarados como synchroni zed para garantir exclusão mútua para as variáveis compartilhadas.

Quando um escritor deseja começar a escrever, primeiro ele verifica se o banco de dados está sendo lido ou escrito no momento. Se o banco de dados estiver sendo lido ou escrito, o escritor vai para o conjunto de espera para o objeto. Caso contrário, ele define dbWri ti ng como true. Quando um escritor terminar, ele define dbWri ting como fal se. Quando um leitor chamar startReadí), ele verifica primeiro se o banco de dados está sendo escrito no momento. Se o banco de dados estiver disponível, o leitor define dbRead i ng como true se ele for o primeiro leitor. O último leitor que chamar endRead definirá dbReading como false.

```
public class Database {
    public Database( ) {
        readerCount * 0;
        dbReading • false;
        dbWriting = false;
    I
    //leitores-escritores chamam este método para "cochilar"
    public static void napping( ) {
        int sleepTime - (int)(NAP_TIHE *
          Hath.random( ));
        try {
          Thread.sleep(sleepTimenoOO);
        )
        catch(IntermptedExc(?ption e) { }
    1
    public synchronized int startReadí ) I
        //Figura 7.36
    )
    public synchronúed int endReadf ) {
        //Figura 7.36
    1
    public synchronized void startWrite( ) (
        //Figura 7.37
    í
    public synchronized void endHrite( ) {
        //Figura 7.37
    )
    private static final int NAP_TIHE • 5;
    private int readerCount;
    //flags para indicar se o banco de dados
    //está sendo lido ou escrito
    private boolean dbReading;
    private boolean dbWriting;
)
```

Figura 7.35 O banco de dados.

7.8.4 Notificações múltiplas

Como descrito na Scção 7.8, a chamada a notify( ) seleciona um thread arbitrário da lista de threads no conjunto de espera para um objeto. Essa abordagem funciona bem quando existe no máximo um thread no conjunto de espera, mas considere o caso cm que existem múltiplos threads no conjunto de espera e mais de uma condição pela qual esperar. Talvez o thread cuja condição ainda não tenha sido atendida receba a notificação. Por exemplo, vamos supor que existam cinco threads *{Tl, Tl,* T.í, *T4, TS)* c uma variável compartilhada turn indicando que thread será usado. Quando um thread desejar trabalhar, ele chamará o método doWork( ) apresentado na Figura 7.38. Apenas o thread cujo número corresponder ao valor de turn poderá prosseguir; todos os demais threads deverão esperar a sua vez.

```
Public svnchronized int startRead( ) {
    while (dbWriting == true) {
      try (
        wait{ );
      |
      catch(InterruptedException e) 1 i
    Í

    ••readerCount;

    //se sou o primeiro leitor, informe
    //todos os outros que o banco de dados está sendo lido
    if (readerCount •• 1)
      dbHeading • true;

    return readerCount;
I

public synchronized int endRead( ) (
    --readerCount;

    //se sou o último leitor, informe todos os
    //outros que o banco de dados nâo está mais sendo lido
    if (readerCount == 0)
      dbReading = false;

    notifyA11( );

    return readerCount;
```

Figura 7.36 Os métodos chamados pelos leitores.

Considere o seguinte:

- turn » 3.
- *Tí,T2c T4* estão no conjunto de espera para o objeto.
- T.í no momento está no método doWork( ).

Quando o thread 73 tiver acabado, ele define turn como 4 (indicando que é a vez de *T4)* e chama notify( ). Achamadaa notify( ) seleciona um thread arbitrário do conjunto de espera. Se72 receber a notificação, ele retoma a execução da chamada a wait( ) e testa a condição no laço while. *Tl* verifica que não é a sua vez, por isso chama o método wa i t ( ) novamente. Por fim, T.í e *TS* chamarão os métodos doWork e também wai t ( ), já que não é a vez de 73 nem de *TS.* Agora, todos os cinco threads estão bloqueados no conjunto de espera para o objeto. Assim, temos um outro deadlock que precisa ser resolvido.

```
public synchronized void startWrite( ) {
    while (dbReading == true || dbWrftfng == true) {
      try [
          wait( );
      |
      catch(InterruptedException e) { )
    }
    //assim que não houver mais leitores-escritores
    //indicar que o banco de dados está sendo gravado
    dbWriting • true;
}

public synchronized void endWrite( ) (
    dbWriting * false;

    notifyAl1( );
)
```

Figura 7.37 Métodos chamados pelos escritores.

Como a chamada a not i fy( ) seleeiona um único thread aleatoriamente do conjunto de espera, o desenvolvedor não tem controle sobre a escolha do thread. Felizmente, Java fornece um mecanismo que permite que todos os threads no conjunto de espera sejam notificados. O método noti fyAl 1 ( ) é semelhante a noti fy( ), exceto pelo fato de que *lodos* os threads em espera são removidos do conjunto de espera e colocados no conjunto de entrada. Se a chamada a noti fy( ) em doWork( ) for substituída por uma chamada a nofityAll ( ),quando 73 terminar e ajustar turn para 4, ele chamará noti fyAl 1 ( ). Essa chamada tem o efeito de remover 77, 72 e *T4* do conjunto de espera. Os três threads podem então disputar o bloco de operações do objeto novamente e, por fim, 77 e 72 chamam o método wait( ) e apenas *T4* prossegue com o método doWorkí).

O método noti fyAl 1 ( ) é um mecanismo que ativa todos os threads em espera e permite que os threads decidam entre si quem deve executar em seguida. Em geral, noti fyAl 1 ( ) é uma operação mais cara do que noti fy( ) porque ativa todos os threads, mas é considerada uma estratégia mais conservadora, que funciona melhor quando diversos threads podem estar no conjunto de espera para um determinado objeto.

```
public void someMethod( ) 1
    synchronized(this) {
      //restante do método
    }
>
//pnum é o número do thread
//que deseja realizar algum trabalho
public synchronized void doWork(int pnum) {
    while (turn != pnum) {
      try (
          waitf );
      }
      catch(InterruptedException e) { )
    i

    //realiza algum trabalho durante um período . . .
    //OK, acabou. Agora indique ao próximo thread em espera
    //que è a sua vez de trabalhar.

    if (turn < 5)
      **turn;
    else
      turn • 1;

    notify{ );
\
```

Figura 7.38 O método doWork.

7.8.5 Sincronização de bloco

Além de declarar métodos como synchronized, Java também permite que blocos de código sejam declarados como synchronized, conforme ilustrado na Figura 7.39.

O acesso ao método criticai Section( ) requer a posse do bloco de operações paramutexLock. Declarar um método someMethod( ) como synchronized equivale a

```
Object mutexLock = new Object( );

public void someMethod( ) {
    nonCriticalSection( );

    synchronized(mutexLock) {
      cnticalSection( );
    )

    nonCritica1Section( );
)
```

Figura 7.39 Sincronização de bloco.

O intervalo de tempo entre o momento cm que um bloco de operações c atribuído e o momento em que ele é liberado é definido como o escopo do bloco de operações. Java fornece sincronização de bloco porque um método synchronized que tenha apenas uma pequena percentagem do seu código manipulando dados com parti lll a d os poderá resultar em um escopo grande demais. Nesse caso, pode ser melhor sincronizar o bloco de código que manipula os dados compartilhados do que sincronizar o método inteiro. Ksseprojeto resulta em um menor escopo de bloco de operações.

Também podemos usar os métodos wait( ) e notifyí } em um bloco sincronizado. A única diferença é que eles devem ser usados com o mesmo objeto que está sendo utilizado para sincronização. Essa abordagem é apresentada na Figura 7.40.

7.8.6 Semáforos Java

Java não fornece um semáforo, mas podemos construir um rapidamente utilizando mecanismos de sincronização default. Declarar os métodos P( ) [T] e V( ) como synchroni zed garante que cada operação seja executada de forma atómica. A classe Semaphore apresentada na Figura 7.41 implementa um semáforo de contagem básico. Ao final do capítulo, incluímos um exercício para modificar a classe Semaphore de forma a fazê-Ia atuar como um semáforo binário.

```
Object mutexLock • new Object( );
* s •
synçhronized(mutexLock) i
    try (
        mutexLock.wait( );
    )
    catch (InterruptedException e) { }
)

synchronízed(mutexLock) (
    mutexLock.notify{ );
|
```

Figura 7.40 Sincronização de bloco utilizando wait( )e notifyí).

```
public class Semaphore
{
    public Seroaphoreí ) {
      value • 0;
    }
    public Sewiaphore(Int v) (
      value = v;
    }
    public synctironized void P( ) {
      while (value <• 0) (
        try 1
          «ait( );
        )
        catch (InterruptedException e) ( |
      }
      value--;
    |
    public synchronized void V( ) {
      •tvalue;
      notify( );
    Í
    private int value;
í
```

Figura 7.41 Implementação de semáforo em Java.

7.8.7 Regras de sincronização

A palavra reservada synchroni zed é uma estrutura simples. É importante conhecer apenas algumas regras sobre seu comportamento.

1. Um tliread que detém o bloco de operações para um objeto pode entrar em outro método (ou bloco) synchroni zed para O mesmo objeto.
2. Um thread pode aninhar chamadas ao método synchroni zed para diferentes objetos. Assim, um thread pode ter a posse do bloco de operações para vários objetos diferentes ao mesmo tempo.
3. Se um método não for declarado como synchroni zed, ele poderá ser chamado independentemente da posse do bloco de operações, mesmo quando outro método synchron í zed para o mesmo objeto estiver executando.
4. Se o conjunto de espera para um objeto estiver vazio, uma chamada a notify( ) ou a notifyAll ( ) não terá efeito.

7.8.8 Monitores Java

Muitas linguagens de programação incorporaram a ideia do monitor (discutido na Seção 7.7), incluindo Concurrent Pascal, Mesa, Ne Ws e Java. Muitas outras linguagens de programação modernas forneceram algum tipo de suporte à concorrência utilizando um mecanismo semelhante aos monitores. Agora, vamos discutir a relação dos monitores no sentido estrito com os monitores Java.

Nesta seção, introduzimos a sincronização Java utilizando um bloco de operações de objeto. De muitas maneiras, o bloco de operações atua como um monitor. Todo objeto Java tem um monitor associado. Um thrc.id pode adquirir o monitor de um objeto entrando em um método synchroni zed ou bloqueando. No entanto, Java não fornece suporte a variáveis de condição nomeadas. Com monitores, as operações de wait c si gnal podem ser aplicadas a variáveis de condição com nome, permitindo que um thread espere determinada condição ou seja notificado quando uma condição tiver sido alcançada. Cada monitor Java tem apenas uma variável de condição sem nome associada a ele. As operações de wait ( ), notify( ) e notifyAll( ) po-

dem se aplicar apenas a essa variável de condição única. Quando um thread Java é ativado via noti fy( ) ou noti fyAl 1 ( ), ele não recebe informações sobre o motivo da ativação. Cabe ao thread reativado verificar por conta própria se a condição pela qual estava esperando foi atendida. Os monitores Java utilizam a abordagem sinalizar-e-conrinuar: quando um thread é sinalizado com o método noti fy( ), ele pode adquirir o bloco de operações para 0 monitor apenas quando o thread de notificação sair do método ou bloco synchroni?ed.

7.9 • Sincronização de sistemas operacionais

A seguir descrevemos os mecanismos de sincronização fornecidos pelos sistemas operacionais Solaris e Windows NT.

7.9.1 Sincronização no Solaris 2

O Solaris 2 foi projetado para suportar a computação de tempo real, multithreading e múltiplos processadores. Para controlar o acesso a seções críticas, o Solaris 2 fornece mutex adaptativos, variáveis de condição, semáforos e blocos de operações de leitura e escrita.

Um mutex adaptativo protege o acesso a cada item de dados crítico. Em um sistema multiprocessador, um mutex adaptativo começa comi) um semáforo padrão implementado como um *spinhck.* Sc os dados estiverem bloqueados e, portanto, já cm uso, o mutex adaptativo tem duas ações possíveis. Se o bloco de operações for mantido por um thread que está em execução no momento em outra CPU, o thread gira em espera ocupada enquanto estiver esperando que o bloco de operações fique disponível, porque o thread que está mantendo o bloco de operações deverá estar quase concluído. Se o thread que estiver mantendo o bloco de operações não estiver no estado de execução no momento, o thread hloqueará, sendo suspenso até ser ativado pela liberação do bloco de operações. Ele é suspenso para que evite a espera ocupada quando o bloco de operações não puder ser liberado de modo relativamente rápido. Um bloco de operações mantido por um thread suspenso tende a estar nessa categoria. Em um sistema uniprocessador, o thread que detém o bloco de operações nunca estará executando se o bloco de operações estiver sendo testado por outro thread, porque somente um thread pode executar de cada vez. Portanto, em um sistema uniprocessador, os threads sempre ficam em estado suspenso em vez de girar se encontrarem um bloco de operações. Usamos o método de mutex adaptativo para proteger apenas os dados que são acessados por segmentos curtos de código. Ou seja, um mutex é usado se um bloco de operações for mantido por menos do que algumas centenas de instruções. Se o segmento de código for maior do que isso, a espera com giro será muito ineficiente. Para segmentos de código mais longos, serão utilizados variáveis de condição e semáforos. Se o bloco de operações desejado já estiver sendo mantido, o thread chama uma operação de espera e é suspenso. Quando um thread libera o bloco de operações, ele emite um sinal para o próximo thread suspenso na fila. O custo extra de suspender e tornar a ativar um thread e das trocas de contexto associadas é menor do que o custo de desperdiçar várias centenas de instruções esperando em um spinlock.

Observe que os mecanismos de bloco de operações usados pelo kernel também são implementados para threads de usuário, por isso os mesmos tipos de blocos de operações estão disponíveis dentro e fora do kernel. Uma diferença crucial de implementação ocorre na área de herança de prioridades. As rotinas de bloco de operações do kernel seguem os métodos de herança de prioridades do kernel utilizados pelo escalonador, conforme descrito na Seção 6.5. Os mecanismos de bloco de operações de threads usuário não fornecem essa funcionalidade, pois é específica do kernel.

Os blocos de operações de leitura e escrita são usados para proteger dados que são acessados com frequência, mas geralmente apenas como somente leitura. Nessas circunstâncias, os blocos de operações de leitura e escrita são mais eficientes do que os semáforos, porque múltiplos threads podem estar lendo dados de forma concorrente, enquanto os semáforos sempre iriam serializar o acesso aos dados. Os blocos de operações de leitura e escrita têm uma implementação relativamente cara, por isso são utilizados apenas cm segmentos longos de código.

Para otimizar o desempenho do Solaris, os desenvolvedores ajustaram e refinaram os métodos de bloco de operações. Como os blocos de operações são usados com frequência c geralmente são utilizados para funções críticas de kernel. grandes ganhe >s de desempenho podem ser alcançados ajustando sua implementação e uso.

7.9.2 Sincronização no Windows NT

0 Windows NT é um kernel com múltiplos threads que também fornece suporte para aplicações de tempo real e múltiplos processadores. O NT fornece vários tipos de objetos de sincronização para lidar com exclusão mútua, incluindo mutexes, scções críticas, semáforos e objetos de evento. Os dados compartilhados são protegidos exigindo que um thrcad obtenha a posse de um mutex para acessar os dados e libere a posse quando tiver terminado. Um objeto de seção crítica tem um comportamento semelhante a um mutex, exceto pelo fato de que ele somente pode ser usado para sincronização entre aqueles threads que pertencerem ao mesmo processo. Os eventos são objetos de sincronização que podem ser utilizados como as variáveis de condição, ou seja, eles podem notificar um thrcad em espera quando uma condição desejada ocorrer.

7.10 • Resumo

Dado um conjunto de processos ou threads sequenciais cooperativos que compartilham dados, a exclusão mútua deve ser fornecida para evitar a ocorrência de uma condição de corrida em relação aos dados compartilhados. Uma solução é garantir que uma seção crítica de código esteja sendo usada apenas por um processo ou thrcad de cada vez. Existem diferentes soluções de software para resolver o problema de seção crítica, partindo do pressuposto de que apenas o bloco de operações de memória está disponível.

A principal desvantagem das soluções de software é que não são fáceis de generalizar para múltiplos threads ou para problemas mais complexos do que a seção crítica. Os semáforos superam essa dificuldade. Podemos usar semáforos para resolver vários problemas de sincronização e podemos implementá-los de forma eficiente, especialmente se houver suporte de hardware disponível para operações atómicas.

Vários problemas de sincronização distintas (tais como buffer limitado, dos leitores-cscritorcs e do jantar dos filósofos) são importantes, basicamente porque são exemplos de uma vasta classe de problemas de controle de concorrência. Esses problemas são usados para testar praticamente todos os novos esquemas de sincronização propostos.

Java fornece um mecanismo para coordenar as atividades de múltiplos threads quando estiverem acessando dados compartilhados, através das instruções synchronized, wait( ), notifyí)e notifyAll ( ). A sincronização Java é fornecida no nível da linguagem, sendo considerada um exemplo de um mecanismo de sincronização de nível mais alto chamado monitor. Além de Java, muitas linguagens forneceram suporte a monitores, incluindo Concurrent Pascal c Mesa.

Os sistemas operacionais também suportam a sincronização de threads. Por exemplo, 0 Solaris 2 e o Windows NT fornecem mecanismos tais como semáforos, mutexes c variáveis de condição para controlar o acesso a dados compartilhados.

• Exercícios

7.1 A primeira solução de software correta conhecida para o problema de seção crítica para dois threads foi desenvolvida por Dekker; ela está apresentada na Figura 7.42. Os dois threads, T_0 c 7";, coordenam a atividade compartilhando um objeto da classe Dekker. Mostre que o algoritmo satisfaz a todos os três requisitos do problema de seção crítica.

7.2 No Capítulo 5, demos uma solução de multithreading ao problema do buffer limitado que utilizava a troca de mensagens. A classe MessageQueue não é considerada segura para threads, o que significa que uma condição de corrida é possível quando múltiplos threads tentarem acessar a fila de forma concorrente. Modifique a classe MessageQueue utilizando a sincronização Java para que seja segura.

7.3 Crie uma classe BinarySemaphore que implemente um semáforo binário.

7.4 A instrução wait( ) em todos os exemplos de programa Java era parte do laço wh i le. Explique porque é preciso sempre utilizar uma instrução while quando estiver usando wait( ) e por que a instrução if nunca seria utilizada.

7.5 A solução para o problema dos leitores-escritores não impede que os escritores em espera sofram de paralisação. Se o banco de dados estiver sendo lido no momento e houver um escritor, os leito-

res subsequentes terão permissão para ler o banco de dados antes que um escritor possa escrever. Modifique a solução de modo a não haver paralisação dos escritores que estão esperando.

7.6 Uma solução baseada em monitor ao problema do jantar dos filósofos, escrito em pseudocódigo semelhante a Java e utilizando variáveis de condição, foi apresentada na Seção 7.7. Desenvolva uma solução para o mesmo problema utilizando sincronização Java.

7.7 A solução apresentada para o problema do jantar dos filósofos não impede que um filósofo morra de fome. Por exemplo, dois filósofos, digamos, o filósofo, e o filósofo^, poderiam alternar entre as atividades de comer e pensar, fazendo com que o filósofo^ nunca conseguisse comer. Usando a sincronização Java, desenvolva uma solução para o problema que impeça um filósofo de morrer de fome.

7.8 Na Seção 7.4, mencionamos que desabilitar interrupções com frequência poderia afetar o clock do sistema. Explique o motivo para isso e como esses efeitos podem ser minimizados.

```
public class Dekker extends MutualExclusion                    ,

    public Dekker( J (
        flag[0] = false;
        flagfl] - false;
        turn = TURN 0;
    1

    public void enteringCriticalSection(int tj {
        (nt Other;

        other » 1 - t;
        fUg[t) * true;

        while (flag[other] •- true) (
          if (turn =- other) {
            flagtt] = false;
            while (turn »• other)
                Thread.yield( );

            flag[t] - true;
          I
        )
    )

    public void leavingCriticalSectionfint t) (
        turn • 1 - t;
        flaglt] = false;


    private volatile int turn;
    private volatile boolean[ ] flag B new boolean[2];
|
```

Figura 7.42 Algoritmo de Dekker para exclusão mútua.

7.9 Neste capítulo, utilizamos a instrução synchroni zed com os métodos de *instância.* Chamar um método de instância requer associar o método com um objeto. Entrar no método synchroni zed exige ter apose do bloco de operações do objeto. Os métodos *estáticos* são diferentes porque não exigem associação com um objeto quando são chamados. Explique como é possível declarar os métodos estáticos como synchroni zed.

7.10 O *Problema de barbeiro dorminhoco.* Uma barbearia consiste em uma sala de espera com *n* cadeiras, e um salão para o barbeiro com uma cadeira de barbear. Se não houver clientes para atender, o barbeiro vai dormir. Se entrar um cliente na barbearia c todas as cadeiras estiverem ocupadas, o cliente não fica.

Se o barbeiro estiver ocupado, mas houver cadeiras disponíveis, o cliente vai esperar em uma das cadeiras livres. Se o barbeiro estiver dormindo, o cliente vai acordá-lo. Kscreva um programa para coordenar o barbeiro e os clientes utilizando a sincronização Java.

7.11 í > *problema dos fumantes.* Considere um sistema com três processos *fumante* e um processo *agente.* (iada fumante está continuamente enrolando um cigarro e fumando-o em seguida. Mas para enrolar e fumar um cigarro, o fumante precisa de três ingredientes: fumo, papel e fósforos. Um dos processos fumante tem papel, outro tem fumo e o terceiro tem os fósforos. O agente tem um suprimento infinito de todos os três materiais. O agente coloca dois dos ingredientes na mesa. O fumante que tem o outro ingrediente faz o cigarro e o fuma, sinalizando o agente na conclusão. O agente coloca outros dois dos ingredientes e o ciclo se repete. Kscreva um programa que sincronize o agente c os fumantes usando a sincronização Java.

7.12 Explique por que o Solaris 2 e o Windows NT implementam múltiplos mecanismos de bloco de operações. Descreva as circunstâncias nas quais eles utilizam spinlocks. mutexes, semáforos, mutex adaptativo e variáveis de condição. Km cada caso. explique por que o mecanismo é necessário.

Notas bibliográficas

Os algoritmos de exclusão mútua I e 2 para duas tarefas foram discutidos no trabalho clássico de Dijkstra [1965a). O algoritmo de Dekker (Exercício 7.1) - a primeira solução de software correia para o problema de exclusão mútua de dois processos - foi desenvolvido pelo matemático holandês T. Dekker. Esse algoritmo também foi discutido por Dijkstra [1965a]. Uma solução mais simples para o problema de exclusão mútua de dois processos foi apresentada mais tarde por Peterson [198 11 (algoritmo 3).

Dijkstra 11965a] apresentou a primeira solução ao problema de exclusão mútua para» processos. Essa solução, no entanto, não tem um limite superior na quantidade de tempo que determinado processo precisa esperar para entrar na seção crítica. Knuth [1966] apresentou o primeiro algoritmo com um limite; seu limite era 2" rodadas. DeBruijn 11967) refinou o algoritmo de Knuth reduzindo o tempo de espera para *ir* rodadas e, depois disso, Kisenberg e McGuire 11972] conseguiram reduzir o tempo para o menor limite de *n* - 1 rodadas. Lamport 11974| apresentou um esquema diferente para resolver o problema de exclusão mútua - o algoritmo do padeiro; ele também requer *n* - 1 rodadas, mas *c* mais fácil de programar c entender. Burns |19781 desenvolveu o algoritmo de solução de hardware que satisfaz o requisito de espera limitada.

Discussões gerais relativas ao problema de exclusão mútua foram apresentadas por Lamport |1986, 1991]. Uma série de algoritmos para exclusão mútua foi apresentada por Raynal [1986).

Informações sobre soluções de hardware disponíveis para a sincronização de processos podem ser encontradas em Patterson e Hennessy 11998).

O conceito de semáforo foi sugerido por Dijkstra | I965a|. Patil [19711 analisou se os semáforos são capazes de resolver todos os problemas de sincronização possíveis. Parnas 11975] discutiu alguns dos problemas nos argumentos de Patil. Kosarajul 1973) deu seguimento ao trabalho de Patil produzindo um problema que não pode ser resolvido pelas operações *watt csignal.* I.ipton [1974) discutiu a limitação das várias primitivas de sincronização.

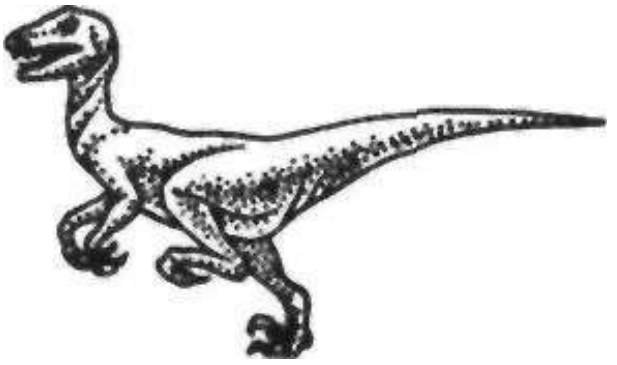
Os problemas clássicos de coordenação de processos que descrevemos aqui são paradigmas de uma vasta classe de problemas de controle de concorrência. O problema do buffer limitado, o problema do jantar dos filósofos e o problema do barbeiro dorminhoco (Exercício 7.10) foram sugeridos por Dijkstra 11965a, 1971], O problema dos fumantes (Exercício 7.11) foi desenvolvido por Patil [19711. O problema dos leitores-escritores foi sugerido por Courtois e colegas [19711. A questão de leituras e escritas concorrentes foi discutida por Lamport 11977), assim como o problema de sincronização de processos independentes 11976).

O conceito de monitor foi desenvolvido por Brinch Hansen 119731. Uma descrição completa do monitor foi feita por Hoare 11974|. Kessels [1977) propôs uma extensão do monitor para permitir a sinalização automática. Um trabalho descrevendo as classificações dos monitores foi publicado por Bulir e colegas 11995J. Discussões gerais relativas à programação concorrente foram oferecidas por Bcn-Ari [1991] e Burns e Davies [19931-

Detalhes relativos a como Java sincroniza os threads podem ser encontrados em Oaks e Wong 11999|, Lea (1997], e Gosling e colegas [1996]. Hartley (19981 faz inúmeras referencias a programação concorrente e multithreading em Java. O Java Report [19981 tratou dos tópicos de multithreading avançado c sincronização em Java.

As primitivas de sincronização para o Windows NT foram discutidas por Solomon [1998) e Pham e Garg [1996|. Detalhes dos mecanismos de blocos de operações utilizados no Solaris 2 são apresentados por Khanna e colegas [1992], Powell e colegas [1991] e especialmente Bykholt e colegas [1992]. Outras referências para a sincronização do Solaris 2 incluem Vahalia [1996] e Graham [1995|.

Capítulo 8



DEADLOCKS

Quando vários processos competem por um número finito de recursos, poderá ocorrer o seguinte: um processo solicita um recurso e o recurso não está disponível no momento. Nesse caso, o processo entra em estado de espera. Pode acontecer que os processos em espera nunca mais mudem de estado, porque os recursos que cies solicitaram estão sendo mantidos por outros processos em espera. Essa situação é chamada de deadlock OU impasse. Já discutimos essa questão rapidamente no Capítulo 7, em relação aos semáforos.

Talvez a melhor forma de ilustrar um deadlock seja com um exemplo de uma lei aprovada pela assembleia do estado norte-americano de Kansas no início deste século. Dizia, em parte: "Quando dois trens se aproximarem um do outro em um cruzamento, ambos deverão parar completamente e nenhum dos dois deverá ser acionado até que o outro tenha partido."

Neste capítulo, ilustramos o deadlock no nível do sistema operacional e em aplicações Java com multithreading. Também são descritos os métodos que os sistemas operacionais e os programadores podem utilizar para lidar com o problema dos deadlocks.

8.1 • Modelo de sistema

Um sistema consiste em um número finito de recursos a serem distribuídos entre processos concorrentes. Os recursos são divididos em vários tipos, cada qual consistindo em múltiplas instâncias idênticas. O espaço na memória, ciclos de CPU, arquivos, blocos de operações de objetos e dispositivos de l/O (tais como impressoras e unidades de fita) são exemplos de tipos de recursos. Se um sistema tiver duas CPUs, então o tipo de recurso *CPU* terá duas instâncias. Da mesma forma, o tipo de recurso *impressora* poderá ter cinco instâncias.

Se um processo solicitar uma instância de um tipo de recurso, a alocação de *qualquer* instância do tipo satisfará a requisição. Se isso não acontecer, então as instâncias não são idênticas, e as classes de tipo de recurso não foram definidas adequadamente. Por exemplo, um sistema pode ter duas impressoras. Essas duas impressoras podem ser definidas para estar na mesma classe de recurso se for indiferente qual impressora vai imprimir wm determinado resultado. No entanto, se uma impressora estiver no nono andar e a outra no subsolo, as pessoas do nono andar talvez não considerem as duas impressoras equivalentes, e talvez seja preciso definir uma classe separada de recursos para cada impressora.

Um processo deve solicitar um recurso antes de usá-lo, e deverá liberar o recurso após o uso. Da mesma forma, um processo pode solicitar tantos recursos quantos achar necessários para realizar sua tarefa. Obviamente, o número de recursos solicitados não pode exceder o número total de recursos disponíveis no sistema. Em outras palavras, um processo não pode solicitar três impressoras se o sistema tiver apenas duas. Se um pedido como esse for feito, ele será rejeitado pelo sistema.

No modo de operação normal, um processo somente poderá usar um recurso na seguinte sequencia:

1. *Pedido:* Se o pedido não puder ser satisfeito imediatamente (por exemplo, o recurso está sendo usado por outro processo), então o processo solicitante deve esperar até que possa obter o recurso.

2. *Uso:* O processo pode operar no recurso (por exemplo, se o recurso for uma impressora, o processo poderá imprimir na impressora).
3. *Liberação: O* processo libera o recurso.

O pedido e a liberação de recursos são chamadas ao sistema, como explicado no Capítulo 3. Exemplos de chamadas ao sistema incluem requeste rei ease device, openeclose file, e ali ocateefree memory. O pedido e a liberação de recursos que não são gerenciados pelo sistema operacional podem ser realizados através das operações P c V nos semáforos ou através da aquisição e liberação de um bloco de operações de um objeto Java através da palavra reservada synchronized. Para cada uso de um recurso gerenciado pelo kernel por um processo ou thread, o sistema operacional verifica para ter certeza de que o processo solicitou e recebeu acesso ao recurso. Uma tabela do sistema registra se cada recurso está livre ou alocado e, para cada recurso que está alocado, a que processo. Se um processo solicitar um recurso que está alocado a outro processo no momento, ele poderá ser acrescentado a uma fila de processos que estão esperando por esse recurso.

Um conjunto de processos está em estado de deadlock quando todos os processos no conjunto estão esperando por um evento que pode ser causado apenas por outro processo no conjunto. Os eventos com os quais estamos preocupados aqui são a aquisição c liberação de recursos. Os recursos podem ser recursos físicos (por exemplo, impressoras, unidades de fita, espaço de memória e ciclos de CPU) ou recursos lógicos (por exemplo, arquivos, semáforos, bloco de operações de objetos c monitores). Outros tipos de eventos também podem resultar em deadlocks. Por exemplo, na Seção 7.8, vimos um tipo de deadlock no qual um thread entrava em um método synchronized e realizava um laço de espera ocupada antes de liberar o bloco de operações.

Para ilustrar o estado de deadlock, consideramos um sistema que tenha três unidades de fita. Vamos supor que existam três processos, cada qual mantendo uma dessas unidades de fita. Se nesse instante cada processo solicitar outra unidade de fita, os três processos estarão em estado de deadlock. Cada um está esperando pelo evento de liberação da unidade de fita, que pode ser causado apenas por um dos demais processos em espera. Esse exemplo ilustra um impasse de processos que estão competindo pelo mesmo tipo de recursos.

Os deadlocks também podem envolver diferentes tipos de recursos. Por exemplo, considere um sistema que tenha uma impressora e uma unidade de fita. Vamos supor que 0 processo /', esteja de posse da unidade de fita e que o processo P_r esteja de posse da impressora. Se $P,$ solicitar a impressora e P, solicitar a unidade de fita, ocorrerá um deadlock.

Um programador que está desenvolvendo aplicações com múltiplos threads deverá prestar muita atenção a esse problema: os programas com múltiplos threads são bons candidatos a deadlock porque provavelmente existem vários threads competindo pelos recursos compartilhados (tais como blocos de operações de objetos).

8.2 • Caracterização de deadlocks

Os deadlocks são indesejáveis. Em um deadlock, os processos nunca terminam sua execução e os recursos do sistema ficam comprometidos, impedindo que outros jobs iniciem. Antes de discutirmos os vários métodos para tratar o problema de deadlocks, vamos descrever suas características.

8.2.1 Condições necessárias

Uma situação de deadlock pode ocorrer se as quatro condições seguintes ocorrerem ao mesmo tempo em um sistema:

1. *Exclusão mútua:* Pelo menos um recurso deverá ser mantido em modo não-compartilhado; ou seja, apenas um processo de cada vez pode usar esse recurso. Se outro processo solicitar esse recurso, o processo solicitante deverá ser retardado até que o recurso tenha sido liberado.
2. *Posse e espera:* Deve haver um processo que esteja mantendo pelo menos um recurso e esteja esperando para obter recursos adicionais que estejam sendo mantidos por outros processos no momento.
3. *Não-preempção:* Os recursos não podem sofrer preempção; ou seja, um recurso só pode ser liberado voluntariamente pelo processo que o mantém, depois que esse processo tiver completado sua tarefa.

4. *Espera circular:* Deve haver um conjunto $\{P_0, P],\ldots, P_{n}\}$ de processos em espera, de modo que Po esteja esperando por um recurso que é mantido por *P\, P* esteja esperando por um recurso que é mantido por P_2,.-., P_M esteja esperando por um recurso que é mantido por P,, e P,, esteja esperando por um recurso que é mantido por P_y.

Enfatizamos que as quatro condições devem ser verdadeiras para que haja um deadlock. A condição de espera circular implica na condição de posse e espera, de modo que as quatro condições não são completamente independentes.

8.2.2 Grafo de alocação de recursos

Os deadlocks podem ser mais bem descritos em termos de um grafo orientado chamado grafo de alocação de recursos do sistema. Fste grafo consiste em um conjunto de vértices V e um conjunto de arestas/l. O conjunto de vértices Vé dividido em dois tipos diferentes de nós, P = U'|,P_2,.... P_n), sendo esse conjunto composto por todos os processos ativos no sistema, e R = *{RQ,* R,,.... *R„* composto por todos os tipos de recursos no sistema.

Uma aresta direcionada do processo P, para o tipo de recurso R, é denotada por *P, -> R,;* significa que o processo P, solicitou uma instância do tipo de recurso R; e, no momento, está esperando por aquele recurso. Uma aresta direcionada do tipo de recurso R, para o processo P_t é denotada por R; -> *P-* significa que uma instância do tipo de recurso R, foi alocada para o processo P,. Uma aresta direcionada *P, -» R, é* denominada aresta de pedido; uma aresta direcionada *R,-> P^* denominada aresta de atribuição.

Em termos gráficos, cada processo P, é representado como um círculo e cada tipo de recurso R_f é representado como um quadrado. Como o tipo de recurso R, pode ter mais de uma instância, representamos cada instância por um ponto no quadrado. Observe que uma aresta de pedido aponta apenas para o quadrado R_T, enquanto uma aresta de atribuição deve designar um dos pontos no quadrado.

Quando o processo P, solicita uma instância do tipo de recurso R„ uma aresta de pedido é inserida no grafo de alocação de recursos. Quando esse pedido tiver sido atendido, a aresta de pedido é transformada *instantaneamente* em uma aresta de atribuição. Quando o processo não precisar mais acessar o recurso, ele o liberará e, como resultado, a aresta de atribuição será excluída.

O grafo de alocação de recursos mostrado na Figura 8.1 representa a seguinte situação.

- Os conjuntos P, *Re A:*
  - *o* $P = \{P_{n}, P_{,}, P_{,}\}$
  - o $R = \{R_u R_{i,} R_{ii} R_4\}$
  - *o* $A = \{P_{l}\text{-}*R_h P_2\text{-}*R\text{^}R_l\text{^}P_{z>} R_l{}'*P_{li} R_i\text{->}P_K R_i\text{-}*P_i\}$
- Instâncias de recursos:
  - ° Uma instância do tipo de recurso R,
  - ° Duas instâncias do tipo de recurso R,
  - ° Uma instância do tipo de recurso R,
  - *o* Três instâncias do tipo de recurso R_4
- Estados do processo:
  - ° O processo P, está mantendo uma instância do tipo de recurso Rj e está esperando por uma instância do tipo de recurso R,.
  - o O processo P_2 está mantendo uma instância de R, e R, c está esperando por uma instância do tipo de recurso R_J
  - ° *O* processo Pi está mantendo uma instância de R_t.

Dada a definição de um grafo de alocação de recursos, é possível demonstrar que, se o grafo não contiver ciclos, nenhum processo no sistema está em deadlock. Se, por outro lado, o grafo contiver um ciclo, poderá haver deadlock.

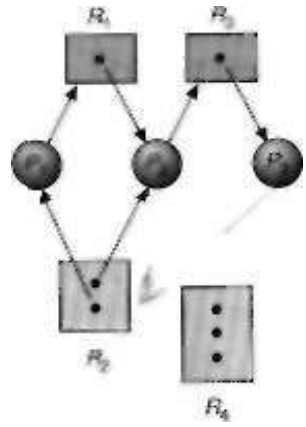


Figura 8.1 Grafo de alocação de recursos.

Se cada tipo de recurso tiver exatamentç uma instância, um ciclo implicará a ocorrência de um deadlock. Se o ciclo envolver apenas um conjunto de tipos de recursos, cada um com apenas uma única instância, então ocorreu um deadlock* Cada processo envolvido no ciclo está em um deadlock. Nesse caso, um ciclo no grafo é uma condição necessária e suficiente para a existência de um deadlock.

Se cada tipo de recurso tiver várias instâncias, então um ciclo não implica necessariamente que ocorreu um deadlock. Nesse caso, um ciclo no grafo é uma condição necessária mas não suficiente para a existência de um deadlock.

Para ilustrar esse conceito, vamos voltar para o grafo de alocação de recursos representado na Figura 8.1. Vamos supor que P_t solicite uma instância do tipo de recurso R_2. Como nenhuma instância de recurso está disponível no momento, uma aresta de pedido P, -* R_2*é* acrescentada ao grafo (Figura 8.2). Neste ponto, existem dois ciclos mínimos no sistema:

K, *Ri* -* *Pi* -> *Ri* -> Pi
P_2 -> R, -» P, -*R_Z->P_2

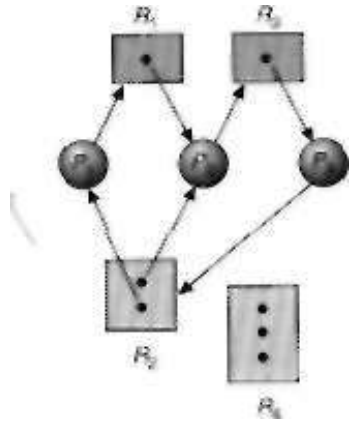


Figura 8.2 Grafo de alocação de recursos com uni deadlock.

Os processos P|, P, e P(estão em deadlock. O processo P_2 está esperando pelo recurso R(, que é mantido pelo processo Pj. O processo P_{iy} por outro lado, está esperando que o processo P| ou P_2 libere o recurso R_2. Além disso, o processo P, está esperando que o processo P_2 libere o recurso R,.

Agora considere o grafo de alocação de recursos da Figura 8.3. Neste exemplo, também temos um ciclo:

P_l-*R_i-*Pi-*R_1-* P,

No entanto, não existe deadlock. Observe que o processo P_4 pode liberar sua instância do tipo de recurso R_2. Esse recurso pode então ser alocado para P_{it} quebrando o ciclo.

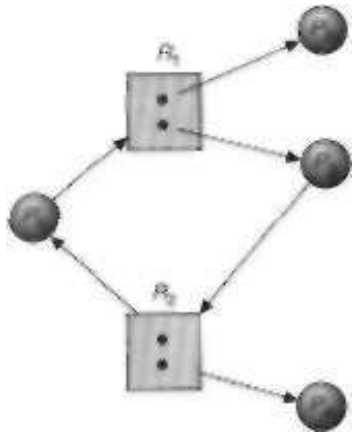


Figura 8.3 Grafo de alocaçâo de recursos com um ciclo mas sem deadlock.

Km resumo, se um grafo de alocaçâo de recursos não tiver um ciclo, o sistema *não* estará cm deadlock. Por outro lado, se houver um ciclo, o sistema poderá ou não estar cm deadlock. É importante ter essa observação em mente ao tratar com o problema de deadlocks.

Antes de prosseguirmos com uma discussão sobre como tratar os deadlocks, vamos ver como um deadlock pode ocorrer em um programa Java com múltiplos threads, como indicado na Figura 8.4.

```
class Mutex { J

class A extends Thread
(
    public A{Hutex f, Hutex s) (
      first • f;
      second • s;
    |
    public vold run( ) {
      synchronized (first) {
          //faça alguma coisa
          synchronized (second) {
            //faça outra coisa
          )
      )
    )
    private Mutex fir$t, second;
|
class B extends Thread
{
    public B(Mutex f, Mutex s) (
      first • f;
      second • s;
    |
    public void run( ) f
      synchronized (second) {
          //faça alguma coisa
          synchronized (first) (
            //faça outra coisa
          )
      |
    ¡
    private Mutex first, second;
}
public class DeadlockExample
i
    //Figura 8.5
)
```

Figura 8.4 Kxemplo de deadlock.

Nesse exemplo, O threadA tenta obter os blocos de operações de objetos na seguinte ordem: (1) mutexX, (2) mutexY, enquanto threadB tenta fazer o mesmo usando a ordem (I)mutexYe (2)mutexX. O deadlock é possível no seguinte cenário:

threadA -> mutexY -» threadB -* mutexX -» threadA

Observe que, embora o deadlock seja possível, não teria ocorrido se threadA tivesse conseguido obter e liberar os blocos de operações para mutexX e mutexY antes que o threadB tentasse obter os blocos de operações. Esse exemplo ilustra o problema: ele é difícil identificar e testar, e uma aplicação pode entrar em deadlock apenas em certas circunstâncias.

8.3 • Métodos para tratar de deadlocks

Basicamente, existem três métodos distintos para tratar do problema dos deadlocks:

1. Podemos usar um protocolo para garantir que o sistema *nunca* entre em estado de deadlock.
2. Podemos permitir que o sistema entre em estado de deadlock e, em seguida, se recupere.
3. Podemos ignorar o problema e fingir que os deadlocks nunca ocorrem no sistema.

A terceira solução é utilizada pela maioria dos sistemas operacionais, incluindo o UNIX. AJVM também não faz nada para gerenciar deadlocks. Cabe ao desenvolvedor de aplicações escrever programas que tratem dos deadlocks. Vamos a aresta rapidamente cada método. Em seguida, nas Seções 8.4 a 8.7, apresentamos algoritmos detalhados.

Para garanrir que os deadlocks nunca ocorrerão, o sistema poderá usar um esquema de prevenção de deadlocks ou de impedimento de deadlocks. A prevenção de deadlocks *é* um conjunto de métodos utilizados para garantir que pelo menos uma das condições necessárias (Seção 8.2. i) não seja válida. Esses métodos previnem deadlocks limitando a maneira como os pedidos de recursos podem ser feitos. A Seção 8.4 discute esses métodos.

```
public statlc voíd main($tring args[ ]) {
    Hutex (TxitexX * new Mutex{ );
    Mutex mutexY * new Hutex( );

    A threadA = new A(mutexX, mutexY);
    B threadB = new B(mutexX, mutexY);

    threadA.startí );
    threadB.start( );
}
```

Figura 8.5 Criando os threads.

Impedimento de deadlock, por outro lado, requer que o sistema operacional receba informações adicionais relativas a quais recursos um processo solicitará e utilizará durante sua vida útil. Com esse conhecimento adicional, o sistema operacional poderá decidir, para cada pedido, se o processo deverá esperar ou não. Cada pedido exige que o sistema considere os recursos disponíveis no momento, os recursos alocados para cada processo no momento e os pedidos e liberações futuros de cada processo, para decidir se o pedido atual pode ser satisfeito ou deve ser postergado. Esses esquemas são discutidos na Seção 8.5.

Se um sistema não empregar um algoritmo de prevenção de deadlocks ou de impedimento deadlock, poderá ocorrer um deadlock. Nesse ambiente, o sistema pode fornecer um algoritmo que examine o estado do sistema para determinar se um deadlock ocorreu e um algoritmo para recuperação do deadlock (se um deadlock de fato tiver ocorrido). Essas questões são arestadas nas Seçóes 8.6 e 8.7.

Se um sistema não garantir que um deadlock nunca vai ocorrer e, além disso, não oferecer um mecanismo detecção e recuperação de deadlocks, então o sistema poderá chegar a um estado de deadlock sem ter como reconhecer o que aconteceu. Nesse caso, o deadlock não-detectado resultará na deterioração do desempenho do sistema, porque os recursos estarão sendo mantidos por processos que não podem ser executa-

dos, c porque mais e mais processos, à medida que eles fazem pedidos por recursos, entram no estado de deadlock. Por íim, o sistema vai parar de funcionar e precisará ser reiniciado manualmente.

Embora esse método não pareça ser uma forma viável de resolver o problema de deadlock, ainda assim é utilizado em alguns sistemas operacionais. Em muitos sistemas, os dcadlocks ocorrem com pouca frequência (digamos, uma vez por ano); assim, é mais barato usar esse método do que passar pelos dispendiosos processos de prevenção c impedimento de dcadlocks, ou os métodos de detecção e recuperação de dcadlocks que precisam ser utilizados constantemente. Além disso, existem circunstâncias em que o sistema está em um estado paralisado sem estar cm deadlock: considere um processo de tempo real que está executando na sua prioridade mais alta (ou qualquer processo que esteja executando em um escalonador não-preemptivo) e que nunca retorna o controle para sistema operacional. Assim, os sistemas precisam ter métodos de recuperação manual para condições de nào-deadlock c podem simplesmente utilizar essas mesmas técnicas para recuperação de dcadlocks.

Como observado antes, a JVM não faz nada para gerenciar dcadlocks, cabe ao desenvolvedor de aplicações escrever programas que sejam livres de deadlocks. Nesta seção, vamos analisar um exemplo que ilustra como o deadlock é possível usando métodos selecionados da API básica Java c como o programador pode desenvolver programas que tratem deadlocks de forma apropriada.

No Capítulo 5, apresentamos os threadsjava c a API que permite aos usuários criar e manipular threads. Os métodos suspend ( ) e resume ( ) tornaram-se obsoletos cm Java 2 porque poderiam levar a deadlocks. O método suspendi) suspende a execução de um thread em execução no momento. O método resume( ) retoma a execução de um thread suspenso. Uma vez suspenso um thread, a única forma dele continuar é se outro thread o retomar. Além disso, um thread suspenso continua a manter todos os blocos de operações enquanto estiver bloqueado. O deadlock é possível se um thread suspenso mantiver um bloco de operações em um objeto e o thread que poderá retomá-lo exigir a propriedade do bloco de operações antes de retomar o thread suspenso.

O método resume ( ) não pode levar a um estado de deadlock, mas como é usado cm associação ao método suspendi), ele também tornou-se obsoleto. stop( ) tornou-se obsoleto também, mas não porque ele pode levar a um deadlock. Diferentemente de quando um thread é suspenso, quando um thread é interrompido, ele libera todos os blocos de operações que possui. No entanto, os blocos de operações são normalmente usados no seguinte cenário: (1) adquirir o bloco de operações, (2) acessar uma estrutura de dados compartilhados e (3) liberar o bloco de operações. Se um thread estiver no meio da etapa 2 quando ocorrer a interrupção, ele liberará o bloco de operações, mas poderá deixar a estrutura de dados compartilhados em estado inconsistente. Por esse motivo, stop( ) ficou obsoleto.

O Capítulo 5 apresentou um applet com múltiplos threads que exibia a hora. Esse programa foi escrito usando os métodos obsoletos suspendi), resumeí) e stop. Quando esse applet começava, ele criava um segundo thread que tinha como resultado a hora do dia. Vamos chamar esse segundo thread de *thread de relógio* para nossos propósitos aqui. Em vez de fazer com que o thread de relógio executasse se o applet não estivesse sendo exibido na janela do navegador, o método stop ( ) para o applet suspenderia o thread de relógio e o retomaria no método start ( ) do applet. (Lcmbre-sc de que o método stop( ) de um applet é chamado quando o navegador sai da página onde está o applet. O método start ( ) de um applet é chamado quando um applet é criado pela primeira vez ou quando o browser volta para a página Web do applet.)

Esse applet de hora pode ser reescrito a fim de ser compatível com Java 2 e, portanto, sem utilizar os métodos suspendi) e resume( ). Em vez disso, ele usará uma variável booleana que indica se o thread de relógio pode executar ou não. Essa variável será definida como verdadeiro no método start ( ) do applet indicando que o thread pode executar. O método stop( ) do applet vai defini-la como falso. O thread de relógio vai verificar o valor dessa variável booleana no seu método run( ) c só prosseguirá se for verdadeiro. Como o thread do applet c o thread de relógio estarão compartilhando essa variável, o acesso a ela será controlado por meio do bloco synchronized. Esse programa está representado na Figura 8.6.

Se o thread de relógio verificar que o valor boolcano é falso, ele se suspenderá chamando o método wai t ( ) do objeto mutex. Quando o applet desejar retomar o thread de relógio, ele definirá a variável como verdadeiro e chamará not i fy ( ) no objeto mutex. Essa chamada a noti f y ( ) ativa o thread de relógio e verifica o valor da variável booleana. Vendo que agora esse valor é verdadeiro, o thread prosseguirá em seu método run{) exibindo a data e a hora.

```
import.java.applet.*;
1mport.java.awt.*;

public class ClockApplet exteds Applet implements Runnable
{
    public void run ( ){
      Thread me = Thread.currentThread( );

      •hfle (clockThread == me) (
          try {
              Thread.sleep{1000);
              repaint( );
              synchronized (mutex) í
                while (ok == false)
                  mutex.wait( );
              |
          }
          catch (InterruptedExcçption e) 1 )
      }
    )

    public void start( ) {
      // Figura 8.7
    |

    public void stop( ) {
      // Figura 8.7
    1

    public void destroy( ) (
      clockThread = null;
    |

    public void paint(Graphics g) (
     g.drawStringí new java.util.Date( ).toString( ), 10, 30);
    |

    private volatile Thread clockThread;
    private boolean ok • false;
    private Object mutex - new Object ( );
}
```

Figura 8.6 Applet que mostra a data c hora.

O método obsoleto stop( ) poderá ser removido com uma tática semelhante. O thread de relógio primeiro faz referência a si mesmo chamando Thread .currentThread ( ) no início do seu método run( ). O laço whi - ledo método run( ) será verdadeiro desde que o valor de clockThread seja igual ao valor da auto-referência. Quando o método destroy ( ) do applet for chamado (indicando, portanto, que o applet deverá ser terminado), ele fará clockThread = null. O thread de relógio interrompe a execução saindo do laço whi lee retornando de run( ).

8.4 • Prevenção de deadlocks

Como observado na Seção 8.2.1, para que ocorra um deadlock, cada uma das quatro condições necessárias deve ser válida. Ao garantir que pelo menos uma dessas condições não seja válida, é possível *prevenir* a ocorrência de um deadlock. Vamos aprofundar esse conceito, examinando cada uma das quatro condições necessárias separadamente.

```
//este método é chamado quando o applet
//é iniciado ou quando voltamos ao applet
public void start( ) {
    if (clockThread •- null) (
        ok •« true;
        clockThread • new Ihread(this);
        ClockThread.start( );
    |
    else í
        synchronized(mutex) 1
            ok • true;
            mutex.notify{ );
        í
    í
»

//este método è chamado quando salmos
//da página onde o applet está
public void stop( ) {
    if (clockThread !- null) í
        synchronized(mutex) {
          ok * false;
        )
        clockThread • null;
    |
í
```

Figura 8.7 Métodos start( ) e stop do applet.

8.4.1 Exclusão mútua

A condição de exclusão mútua deve ser válida para recursos nâo-compartilhaveis. Por exemplo, uma impressora ou método synchroni zed não podem ser compartilhados por vários processos ao mesmo tempo. Recursos compartilháveis, por outro lado, não precisam de acesso mutuamente exclusivo e, portanto, não podem estar envolvidos em um deadloek. Arquivos somente de leitura e métodos não-synchroni zed são bons exemplos de um recurso compartiIhável. Se vários processos tentarem abrir um arquivo somente de leitura ao mesmo tempo, poderão obter acesso simultâneo ao arquivo. Um processo nunca precisa esperar por um recurso compartilhável. Em geral, no entanto, não é possível prevenir deadlocks negando a condição de exclusão mútua: alguns recursos são intrinsecamente náo-compartilháveis.

8.4.2 Posse e espera

Para ter certeza de que a condição posse e espera *(hold-and-wait)* nunca vai ocorrer no sistema, é preciso garantir que, sempre que um processo solicitar um recurso, ele não mantenha outros recursos. Um protocolo que pode ser usado exige que processo solicite e receba todos os seus recursos antes de começar a execução. É possível implementar esse requisito exigindo que as chamadas ao sistema que solicitam os recursos para determinado processo precedam todas as demais chamadas ao sistema.

Um protocolo alternativo permite que um processo solicite recursos apenas quando o processo não tiver nenhum. Um processo poderá solicitar alguns recursos e depois usá-los. Antes de poder solicitar quaisquer recursos adicionais, no entanto, ele deverá liberar todos os recursos que estão sendo alocados a ele no momento.

Para ilustrar a diferença entre esses dois protocolos, consideramos um processo que copia dados de unia unidade de fita para um arquivo em disco, ordena o arquivo em disco e, em seguida, imprime os resultados em uma impressora. Se todos os recursos precisarem ser solicitados no início do processo, então o processo deverá solicitar inicialmente a unidade de fita, o arquivo cm disco e a impressora. Ele manterá a impressora durante a execução inteira, embora só precise da impressora no fim.

O segundo método permite que o processo solicite inicialmente apenas a unidade de fita c o arquivo em disco. Ele copia da unidade para o disco e, em seguida, libera a unidade de fita e o arquivo em disco. O proces-

so deverá, então, solicitar novamente o arquivo em disco e a impressora. Após copiar o arquivo em disco para a impressora, ele liberará esses dois recursos e terminará.

Existem duas desvantagens principais nesses protocolos. Em primeiro lugar, a *utilização de recursos* pode ser baixa, porque muitos dos recursos podem ser alocados mas não utilizados durante longos períodos. No caso apresentado, por exemplo, podemos liberar a unidade de fita e o arquivo em disco e solicitar novamente o arquivo em disco e a impressora, somente se tivermos certeza de que nossos dados permanecerão no arquivo em disco. Se não pudermos ter certeza de que ficarão, devemos solicitar todos os recursos no início para ambos os protocolos.

Em segundo lugar, a paralisação é uma possibilidade. Um processo que precisa de muitos recursos populares pode ter de esperar indefinidamente, porque pelo menos um dos recursos de que ele necessita está sempre alocado a algum outro processo.

Essa solução também é impraticável em Java, porque um processo solicita recursos (blocos de operações) entrando em métodos ou blocos synchroni zed. Como os recursos de blocos de operações são solicitados dessa maneira, seria difícil escrever uma aplicação que seguisse um dos protocolos apresentados.

8.4.3 Não-preempção

A terceira condição necessária é que não haja preempção de recursos que já foram alocados. Para garantir que essa condição não ocorra, podemos usar o seguinte protocolo. Se um processo que estiver de posse de alguns recursos solicitar outro recurso que não pode ser imediatamente alocado a ele (ou seja, o processo deve esperar), então todos os recursos sendo mantidos no momento são submetidos à preempção. Isto é, esses recursos são liberados implicitamente. Os recursos que sofreram preempção serão acrescentados à lista de recursos pelos quais o processo está esperando. O processo só será reiniciado quando puder obter novamente seus antigos recursos, assim como os novos que estão sendo solicitados.

Como alternativa, se um processo solicitar alguns recursos, primeiro verificamos se eles estão disponíveis. Se estiverem, eles são alocados. Se não estiverem, verificamos se estão alocados a algum outro processo que está esperando recursos adicionais. Se esse for o caso, os recursos desejados são arrancados do processo em espera e alocados ao processo solicitante. Se os recursos não estiverem disponíveis ou sendo mantidos por um processo em espera, o processo solicitante deverá esperar. Enquanto espera, alguns de seus recursos podem sofrer preempção, mas apenas se outro processo os solicitar. Um processo só poderá ser reiniciado quando ele receber os novos recursos que está solicitando e recuperar quaisquer recursos que sofreram preempção durante sua espera.

Esse protocolo é aplicado com frequência a recursos cujo estado pode ser facilmente saivo e restaurado mais tarde, tais como registradores de CPU e espaço de memória. Geralmente, não pode ser aplicado a recursos como impressora e drivers de fita magnética. Também não pode ser aplicado a objetos, porque o processo interrompido poderá deixar o objeto em um estado indeterminado.

8.4.4 Espera circular

Uma forma de garantir que a condição de espera circular jamais ocorra é impor uma ordem total sobre todos os tipos de recursos e exigir que cada processo solicite recursos em ordem ascendente de enumeração.

Seja $R = \{R_i R_!, \ldots, R_{,,}\}$ o conjunto de tipos de recursos. Vamos atribuir a cada tipo de recurso um número inteiro único, que permite a comparação entre dois recursos para determinar se um precede o outro na nossa ordenação. Formalmente, definimos uma função de um-para-um $F:R$ -+ N, onde N é o conjunto de números naturais. Por exemplo, se o conjunto de tipos R incluir unidades de fita, unidades de disco e impressoras, então a função /" poderá ser definida da seguinte maneira:

/'(unidade de fita) • 1,
F(unidade de disco) = 5,
F(Impressora) = 12.

Agora, podemos considerar o seguinte protocolo para prevenir deadlocks. Cada processo somente pode solicitar recursos apenas em ordem ascendente de enumeração, isto é, um processo poderá solicitar inicialmente qualquer número de instâncias de um tipo de recurso, digamos R_i. Depois disso, o processo poderá solicitar instâncias do tipo de recurso R_f se e somente se $h'(R_t) > l'(R_i)$. Se várias instâncias do mesmo tipo de recurso forem necessárias, deve ser emitido um *único* pedido para todos eles. Por exemplo, usando a função definida anteriormente, um processo que deseja usar a unidade de fita e a impressora ao mesmo tempo deverá primeiro solicitar a unidade de fita e depois solicitar a impressora.

De outro modo, podemos exigir que, sempre que um processo solicitar uma instância do tipo de recurso R_r ele tenha liberado quaisquer recursos R_i de modo que $F(R_i) \pounds F(R_f)$.

Podemos implementar esse esquema em uma aplicação Java definindo uma ordem entre todos os objetos no sistema. Todos os pedidos de blocos de operações para os objetos devem ser feitos em ordem ascendente. Por exemplo, se a ordem de blocos de operações no programa Java indicado na Figura 8.4 fosse

F(mutexX) = 1,
F(mutexY) = 5.

então, classe B não poderia solicitar os blocos de operações fora de ordem. Lembre-sc de que desenvolver uma ordem, ou hierarquia, por si só não impede um dcadlock. Cabe aos desenvolvedores de aplicações escrever programas que sigam a ordenação. Além disso, observe que a função *F* deve ser definida de acordo com a ordem normal de uso dos recursos em um sistema. Por exemplo, como a unidade de fita normalmente é necessária antes da impressora, seria razoável definir F(unidade de fita) < F(impressora).

8.5 • Impedimento de deadlocks

Os algoritmos de prevenção de deadlocks previnem os deadlocks limitando a forma de realizar os pedidos. As limitações garantem que pelo menos uma das condições necessárias para um deadlock não possa ocorrer e, portanto, que não haja deadlocks. Os possíveis efeitos colaterais de impedir os deadlocks por esse método são baixa utilização de dispositivos, throughput reduzido de sistema e potencial paralisação de processos.

Um método alternativo para evitar deadlocks é exigir que um usuário (processo) forneça informações adicionais sobre como os recursos serão solicitados. Por exemplo, em um sistema com uma unidade de fita e uma impressora, podemos ser informados que o processo *P* solicitará primeiro a unidade de fita e depois a impressora, antes de liberar os dois recursos. O processo *Q,* por outro lado, solicitará primeiro a impressora e depois a unidade de fita. Com o conhecimento da sequencia completa de pedidos e liberações para cada processo, podemos decidir, para cada pedido, se o processo deverá ou não esperar. Cada pedido exige que o sistema considere os recursos disponíveis no momento, os recursos alocados a cada processo no momento e os pedidos e liberações futuros de cada processo, para decidir se o pedido atual pode ser atendido ou se precisará esperar para evitar um possível deadlock futuro.

Os vários algoritmos diferem na quantidade e tipo de informações necessárias. O modelo mais simples e útil de algoritmo requer que cada processo declare o *número máximo* de recursos de cada tipo que será necessário. Considerando informações *a priori* sobre o número máximo de recursos de cada tipo que podem ser solicitados para cada processo, podemos criar um algoritmo que garanta que o sistema nunca entrará em estado de deadlock. Esse algoritmo define a abordagem de impedimento de deadlock. Um tal algoritmo examina de forma dinâmica o estado de alocação de recursos para garantir que não haja uma condição de espera circular. O estado de alocação de recursos é definido pelo número de recursos alocados e disponíveis, e as demandas máximas dos processos.

Neste livro, apresentamos um algoritmo de impedimento de deadlock para um ambiente no qual só existe uma instância de cada tipo de recurso. As notas bibliográficas fornecem informações sobre o caso mais geral, no qual o sistema tem várias instâncias de um tipo de recurso.

Se tivermos um sistema de alocação de recursos com apenas uma instância de cada tipo de recursos, poderemos usar uma variante do grafo de alocação de recursos definido na Seção 8.2.2 para evitar deadlock.

Além das arestas de pedido e atrihuição, introduzimos um novo tipo de aresta, chamada aresta de demarcação. Um aresta de demarcação P_f —• R_f indica que o processo P, pode solicitar o recurso R, em algum momento no futuro. Essa aresta assemelha-se a uma aresta de pedido em termos de direção, mas é representada por uma linha tracejada. Quando o processo *P,* solicitar um recurso R_p a aresta de demarcação $P_($ —> R, será convertido cm uma aresta de pedido. Da mesma forma, quando *R, é* liberado por P,, a aresta de atribuição R, -> *P,* é reconvertida para aresta de demarcação P, -* R_y. Observe que os recursos devem ser reivindicados *a priori* no sistema. Ou seja, antes que o processo *P,* comece a executar, todas as arestas de demarcação já devem ter aparecido no grafo de alocação de recursos. Podemos relaxar essa condição permitindo que uma aresta de demarcação *P,* —> R, seja acrescentada ao grafo apenas se todas as arestas associadas com o processo P, sejam aresta de demarcação.

Vamos supor que o processo *P,* solicite o recurso R_f. O pedido só poderá ser concedido se a conversão da aresta de pedido $P_($ -* R_f em uma aresta de atribuição R, -* *P,* não resulte na formação de um ciclo no grafo de alocação de recursos. Observe que verificamos a segurança utilizando um algoritmo de detecção de ciclos. Um algoritmo para detectar um ciclo neste grafo requer a ordem de M- operações, onde *n* é o número de processos no sistema.

Se não existirem ciclos, a alocação do recurso deixará o sistema em um estado seguro. Se um ciclo for encontrado, a alocação colocará o sistema cm um estado inseguro. Portanto, o processo *P,* terá de esperar que seus pedidos sejam satisfeitos.

Para ilustrar esse algoritmo, considere o grafo de alocação de recursos da Figura 8.8. Vamos supor que P. solicite R_2- Embora R_2 esteja livre no momento, não podemos alocá-lo para P,, já que essa ação criará um ciclo no grafo (Figura 8.9). Um ciclo indica que o sistema está cm um estado inseguro. Se Pi solicitar R_2>^{e} *Pi* solicitar R,, então ocorrera um deadlock. *j~*

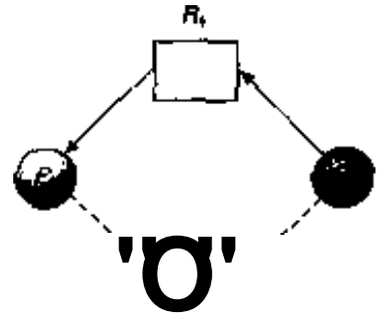


Figura 8.8 Grafo de alocação de recursos para evitar deadlock.

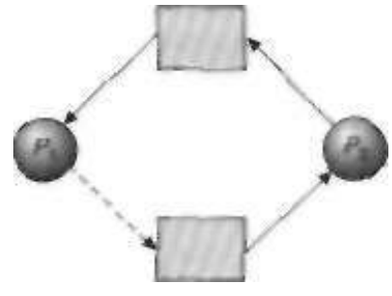


Figura 8.9 Estado inseguro em um grafo de alocação de recursos.

8.6 • Detecção de deadlocks

Se um determinado sistema não emprega algoritmo de prevenção de deadlocks ou de impedimento de deadlock, poderá ocorrer um deadlock. Nesse contexto, o sistema deverá fornecer uma das seguintes opções:

- Um algoritmo que examine o estado do sistema para determinar se ocorreu uni deadlock
- Um algoritmo para recuperar o sistema do deadlock

Na discussão a seguir, analisamos esses dois requisitos no contexto de sistemas com apenas uma instância de cada tipo de recurso. A seção Notas bibliográficas contém informações sobre 0 caso mais geral, em que o sistema possui várias instâncias de cada tipo de recurso. Neste ponto, no entanto, é importante ressaltar que um esquema de detecção *c* recuperação requer custos que incluem não apenas os custos do tempo de execução de manter as informações necessárias e executar o algoritmo de detecção, mas também as perdas potenciais inerentes à recuperação de um deadlock.

Se todos os recursos tiverem apenas uma única instância, podemos definir um algoritmo de detecção de deadlocks que utili/e uma variante do grafo de alocação de recursos, chamado grafo de espera. Esse grafo é obtido a partir do grafo de alocação de recursos, removendo os nós de tipo recurso e fundindo as arestas apropriadas.

Mais precisamente, uma aresta de *P,* a *P,* em um grafo de espera implica que o processo *P,* está esperando que o processo *P-* libere um recurso do qual *P,* necessita. Uma aresta *P,* -> /', existe em um grafo de espera se e somente se o grafo de alocação de recursos correspondente contiver duas arestas *P^R^eR^-* P,* para um recurso R_q. Por exemplo, na Figura 8.10, apresentamos um grafo de alocação de recursos e o grafo de espera correspondente.

Como antes, um deadlock existirá no sistema se e somente se 0 grafo de espera contiver um ciclo. Para detectar deadlocks o sistema precisa *manter* o grafo de espera e periodicamente *chamar um algoritmo* que procura um ciclo no grafo.

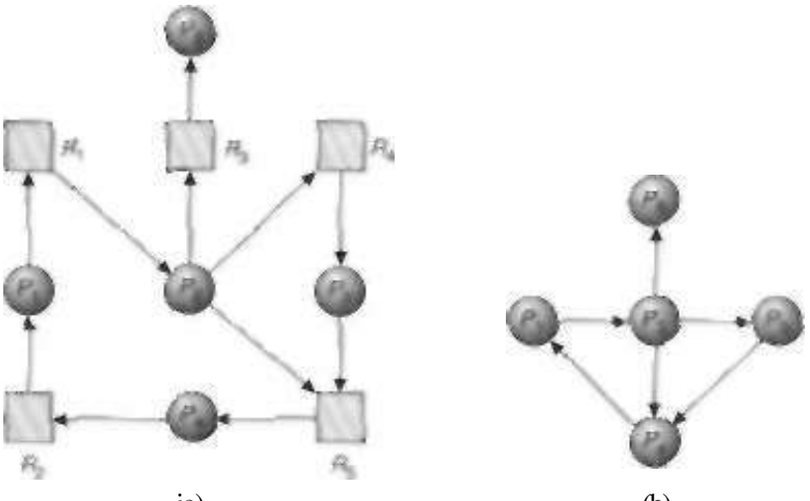
Um algoritmo para detectar um ciclo cm um grafo requer na ordem de $0(n^2)$ operações, onde *n* é o número de vértices no grafo.

Passamos agora para a análise de quando o algoritmo de detecção deve ser chamado. A resposta dependerá de dois fatores:

I. Com que *frequência* um deadlock tende a ocorrer?

I. Quantos processos serão afetados pelo deadlock quando ele acontecer?

Se os deadlocks ocorrerem com frequência, o algoritmo de detecção deverá ser chamado com frequência. Os recursos alocados a processos em deadlock ficarão ociosos até que o deadlock seja quebrado. Além disso, o número de processos envolvidos no ciclo de deadlock tenderá a crescer-



ia) (b)

Figura 8.10 (a) Grafo de alocação de recursos, (b) Grafo de espera correspondente.

Os deadlocks srt existem quando algum processo fizer um pedido que não possa ser atendido imediatamente. É possível que esse pedido seja o pedido final que conclua uma cadeia de processos em espera. No caso extremo, poderíamos chamar o algoritmo de detecção de deadlocks sempre que um pedido de alocação não puder ser atendido imediatamente. Nesse caso, identificamos não apenas o conjunto de processos em

deadlock, mas também o processo específico que "causou" o deadlock. (Na verdade, cada um dos processos em deadlock é um elo no ciclo do grafo de recursos, por isso todos eles juntos causaram o deadlock.) Se houver muitos tipos diferentes de recursos, um pedido poderá causar muitos ciclos no grafo de recursos, cada ciclo completado pelo pedido mais recente e causado por um dos processos identificáveis.

É claro que chamar o algoritmo de detecção de dcadlocks para todo pedido poderá incorrer em um custo considerável em tempo de computação. Uma alternativa menos cara seria simplesmente chamar o algoritmo em intervalos menos frequentes - por exemplo, uma vez por hora ou sempre que a utilização de CPU cair abaixo de 40%. (Um deadlock acaba reduzindo em muito o throughput do sistema e fará com que a utilização de CPU caia.) Se o algoritmo de detecção for chamado em momentos arbitrários, poderá haver muitos ciclos no grafo de recursos. Em geral, não seria possível distinguir qual dentre os muitos processos em deadlock causou o deadlock.

8.7 • Recuperação de um deadlock

Quando um algoritmo de detecção determina a existência de um deadlock, há várias alternativas- Uma possibilidade c informar o operador que ocorreu um deadlock e deixar o operador tratar o problema manualmente. A outra possibilidade é deixar que o sistema *se recupere* do deadlock automaticamente- Existem duas opções para quebrar um deadlock. Uma solução é simplesmente abortar um ou mais processos para interromper a espera circular. A segunda opção é fazer a preempção de alguns recursos de um ou mais processos em deadlock.

8.7.1 Termino de processos

Para eliminar dcadlocks abortando um processo, usamos um dos dois métodos apresentados a seguir. Em ambos os métodos, o sistema recupera todos os recursos alocados aos processos terminados.

- *Aborlar lodos os processos em deadlock:* Esse método claramente interromperá o ciclo de deadlock, mas com grande custo, já que esses processos podem ter tido um longo tempo de computação e os resultados dessas computações parciais terão de ser descartados, devendo provavelmente ser recalculados mais tarde.
- *Abortar um processo de cada vez até eliminar o ciclo de deadlock:* Esse método envolve um grande custo, já que, depois que cada processo é abortado, um algoritmo de detecção de deadlocks deve ser chamado para determinar se ainda existem processos em deadlock.

Observe que abortar um processo pode não ser fácil. Se o processo estava cm meio a uma operação de atualização de arquivo, encerrá-lo no meio da operação deixará o arquivo em estado incorreto. Terminar um thread Java faz com que o thread libere quaisquer blocos de operações que estavam sendo mantidos. Como resultado, o(s) objcto(s) poderá(ão) ficar em um estado arbitrário.

Se o método de término parcial for usado, então, para determinado conjunto de processos em deadlock, será preciso determinar que processo (ou processos) precisam ser terminados na tentativa de quebrar o deadlock. Kssa determinação é uma decisão política, semelhante à dos problemas de escalonamento de CPU. A questão é basicamente económica; devemos abortar os processos cujo término incorrerá em custo mínimo. Infelizmente, o termo *custo mínimo* não é exato. Muitos fatores podem determinar qual processo será escolhido, incluindo:

1. Qual a prioridade do processo
2. Quanto tempo o processo ficou computando e por quanto tempo ele ainda continuará executando antes de concluir sua tarefa designada
3. Quantos e que tipos de recursos o processo utilizou (por exemplo, se os recursos são de fácil preempção)
4. Quantos recursos mais o processo precisa para concluir
5. Quantos processos precisarão ser terminados
6. Se o processo é interativo ou batch

8.7.2 Preempção de recursos

Para eliminar deadlocks usando a preempção de recursos» fazemos a preempção sucessiva de alguns recursos dos processos e conferimos esses recursos a outros processos até que o ciclo de deadlock seja quebrado.

Se houver necessidade de preempção para tratar deadlocks, então três questões precisam ser analisadas:

1. *Seleção de uma vitima:* Que recursos e processos devem ser submetidos à preempção? Como no término de um processo, é preciso determinar a ordem de preempção para minimizar custos. Os fatores de custo incluem parâmetros como o número de recursos mantidos por um processo em deadlock e a quantidade de tempo que um processo em deadlock já consumiu durante sua execução.
2. *Rollback* (volta ao passado): Se efetuarmos a preempção de um recurso de um processo, o que deve ser feito com esse processo? Evidentemente, ele não poderá continuar sua execução normal; falta algum recurso necessário. Devemos retornar o processo para algum estado seguro, e reiniciá-lo a partir desse estado. Como, cm geral, é difícil determinar o que é um estado seguro, a solução mais simples é o rollback total: abortar o processo e reiniciá-lo. No entanto, é mais eficaz retornar o processo somente o necessário para quebrar o deadlock. Por outro lado, o método requer que o sistema mantenha mais informações sobre o estado de todos os processos em execução.
3. *Starvation* (paralisação): Como garantir que não haverá paralisação? Ou seja, como garantir que o processo a sofrer preempção de seus recursos não será sempre o mesmo? F.m um sistema no qual a Seleção de vítimas baseia-se principalmente nos fatores de custo, o mesmo processo pode ser sempre escolhido como vítima. Como resultado, esse processo nunca conclui a tarefa que lhe foi designada, correspondendo a uma situação de paralisação que precisa ser resolvida em qualquer sistema concreto. Evidentemente, é preciso garantir que um processo seja escolhido como vítima somente um (pequeno) número finito de vezes. A solução maiscomum é incluir o número de rollbacks no fator de custo.

8.8 • Resumo

Um estado de deadlock ocorre quando dois ou mais processos estão esperando indefinidamente por um evento que só pode ser causado por um dos processos em espera. Em princípio existem três métodos para tratar deadlocks:

1. Usar algum protocolo para garantir que o sistema nunca entre em estado de deadlock.
2. Permitir que o sistema entre em estado de deadlock e depois se recupere.
3. Ignorar o problema e fingir que os deadlocks nunca ocorrem no sistema.

A terceira solução é a usada pela maioria dos sistemas operacionais, incluindo o UNIX e a JVM. Uma situação de deadlock poderá ocorrer se e somente se quatro condições necessárias forem válidas ao mesmo tempo no sistema: exclusão mútua, posse e espera, nâo-preempção e espera circular. Para prevenir deadlocks, é preciso garantir que pelo menos uma das condições necessárias nunca seja válida.

Outro método para evitar deadlocks, menos estrito do que os algoritmos de prevenção, é ter informações *a priori* sobre como cada processo estará utilizando o recurso. Usando essas informações, é possível definir um algoritmo de impedimento de deadlock.

Se um sistema não utilizar um protocolo para garantir que os deadlocks nunca ocorrerão, um esquema de detecção e recuperação deverá ser empregado. Um algoritmo de detecção de deadlocks deve ser chamado para determinar se um deadlock ocorreu. Se o deadlock for detectado, o sistema deverá se recuperar terminando alguns dos processos em deadlock ou efetuando a preempção de recursos a partir de alguns dos processos em deadlock.

Em um sistema que seleciona vítimas para rollback principalmente com base nos fatores de custo, pode ocorrer uma situação de paralisação. Como resultado, o processo seleeionado nunca concluirá a tarefa que lhe foi designada.

Finalmente, os pesquisadores vêm discutindo o fato de que nenhuma dessas abordagens básicas por si só é apropriada para todo o espectro de problemas de alocação de recursos nos sistemas operacionais. As abordagens básicas podem ser combinadas, permitindo a Seleção separada de uma solução ótima para cada classe de recursos em um sistema.

Exercícios

8.1 I .istc três exemplos de deadlocks que não estejam relacionados com um ambiente de sistema de computação.

8.2 É possível ter um deadlock envolvendo apenas um único processo? Explique sua resposta.

8.3 Considere o deadlock de tráfego indicado na Figura 8.11.

a. Mostre que as quatro condições necessárias para o deadlock de fato estão presentes nesse exemplo.

b. Apresente uma regra simples que evite deadlocks nesse sistema.

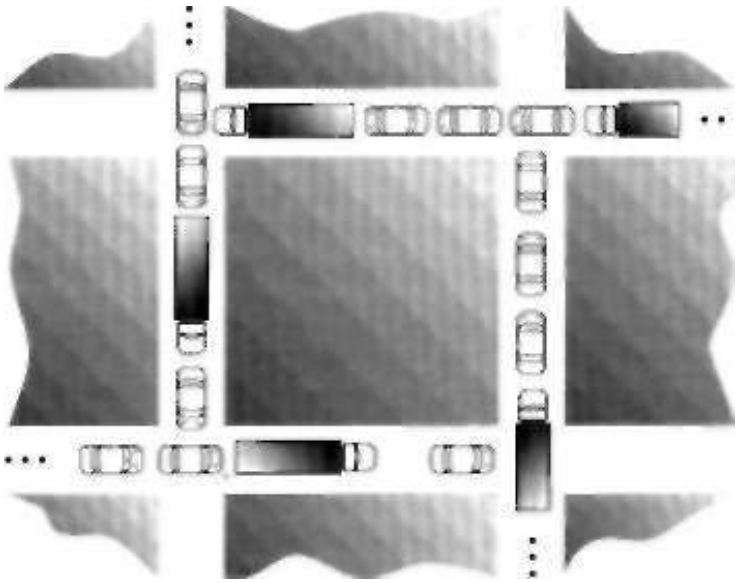


Figura 8.11 Deadlock de tráfego para o Exercício 8.3.

AA Suponha que um sistema esteja em um estado inseguro. Mostre que *é* possível que os processos concluam sua execução sem entrar cm estado de deadlock.

8.5 Uma solução possível para prevenir deadlocks *é* ter um recurso único de nível mais alto que deve ser solicitado antes de qualquer outro recurso. Por exemplo, se vários threads tentarem acessar os blocos de operações para cinco objetos Java *A* ... E_y o deadlock será possível. Podemos prevenir o deadlock adicionando um sexto objeto *F*. Sempre que um thread desejar adquirir o bloco de operações para qualquer objeto *A* ... E, deverá primeiro obter o bloco de operações para 0 objeto *F*. Essa solução é chamada de contenção: os blocos de operações para os objetos *A*... *E* estão contidos no bloco de operações para o objeto *F*. Compare esse esquema com o esquema de espera circular da Scção 8-4.4.

8.6 Escreva um programa Java que ilustre o deadlock fazendo com que métodos synchronized chamem outros métodos synchronized.

8.7 Escreva um programa Java que ilustre o deadlock fazendo com que threads separados tentem realizar operações em semáforos distintos.

8.8 Considere um sistema que consista em quatro recursos do mesmo tipo que são compartilhados por três processos, cada qual precisando no máximo de dois recursos. Mostre que o sistema está livre de deadlocks.

8.9 Considere um sistema que consista em *m* recursos do mesmo tipo, sendo compartilhados por;/ processos. Os recursos podem ser solicitados e liberados pelos processos apenas um de cada vez. Mostre que o sistema está livre de deadlocks se as duas condições seguintes forem válidas:

a. A necessidade máxima de cada processo está entre 1 e *m* recursos

b. A soma de todas as necessidades máximas é menor do que $m + n$.

8.10 Um sistema pode detectar que alguns de seus processos estão em situação de paralisação? Se a resposta for "sim"» explique como. Se a resposta for "não", explique como o sistema pode tratar o problema de paralisação.

8.11 Considere a seguinte política de alocação de recursos. Pedidos e liberações de recursos são permitidos a qualquer momento. Sc um pedido por recursos não puder ser atendido porque os recursos não estão disponíveis, vamos verificar então os processos que estão bloqueados, esperando recursos. Se tiverem os recursos desejados, esses recursos são retirados deles e passados ao processo solicitante. O vetor dos recursos pelos quais o processo está aguardando c aumentado para incluir os recursos que são retirados.

Por exemplo, considere um sistema com três tipos de recursos e o vetor *Available* iniçializado como (4,2,2). Se *PQ* solicitar (2,2, T), ele os obtém. Se *P*\ solicitar (1,0,1), ele os obtém. Então, se $P_{„}$ solicitar (0,0,1), ele será bloqueado (recurso não disponível). Se P» solicitar agora (2,0,0), ele obterá o recurso disponível (1,0,0) e aquele que foi alocado a P_0 (já que P_0 está bloqueado). O vetor *Allocation* de P_0 desce para (1,2,1) c o vetor *Need* aumenta para (1,0,1).

a. Pode haver um deadlock? Se a resposta for "sim", dê um exemplo. Se a resposta for "não", especifique que condição necessária não poderá ocorrer.
b. Pode ocorrer um bloco de operações indefinido? Explique a resposta.

8.12 Um túnel de estrada de ferro com um conjunto único de trilhos conecta duas cidadezinhas do estado de Vermont. A ferrovia pode ficar em uma situação de impasse (deadlock) se um trem indo para o sul c outro indo para o norte entrarem no túnel ao mesmo tempo (os trens não podem voltar). Escreva um programa Java que previna o deadlock usando semáforos ou sincronização Java. Inicialmente, não se preocupe com a paralisação dos trens em direção ao sul causada pelos trens seguindo para o norte (ou vice-versa) (e não se preocupe com o fato de os trens não pararem ou baterem). Assim que sua solução possa prevenir o deadlock, modifique-a de modo que não haja possibilidade de paralisação.

Notas bibliográficas

Dijkstra (1965a| foi um dos primeiros e mais influentes colaboradores na área de dcadlocks. Holt 11972] foi a primeira pessoa a formalizar a noção de dcadlocks em termos de um modelo teórico cm grafo semelhante ao apresentado neste capítulo. A situação de estagnação foi arestada por Holt 11972). Hyman 11985] forneceu o exemplo de deadlock sobre a lei do estado de Kansas.

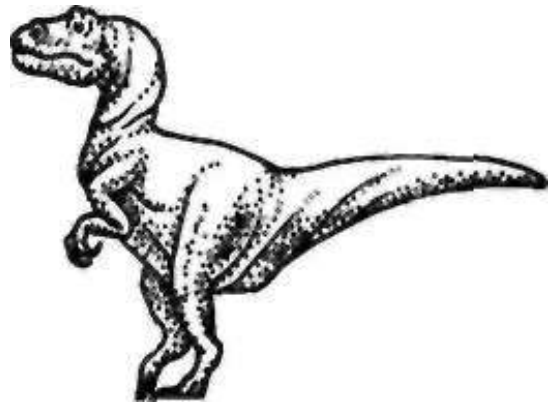
Os vários algoritmos de prevenção foram sugeridos por Havender 11968], que criou o esquema de ordenação de recursos para o sistema IBM OS/360.

Um algoritmo impedimento de deadlock (o algoritmo do banqueiro) para um único tipo de recurso foi desenvolvido por Dijkstra |1965a], e estendido para vários tipos de recursos por Habermann (1969|. Os exercícios 8.8 e 8.9 são de Holt 119711.

Um algoritmo de detecção de dcadlocks para múltiplas instâncias de um tipo de recurso foi apresentado por Coffman e colegas [1971].

Bach 11987] descreve quantos dos algoritmos no kernel do UNIX tradicional tratam dcadlocks. A descrição de como Java 2 trata os deadlocks foi obtida do trabalho de Oaks e Wong [1999].

Parte Três



GERÊNCIA DE MEMÓRIA

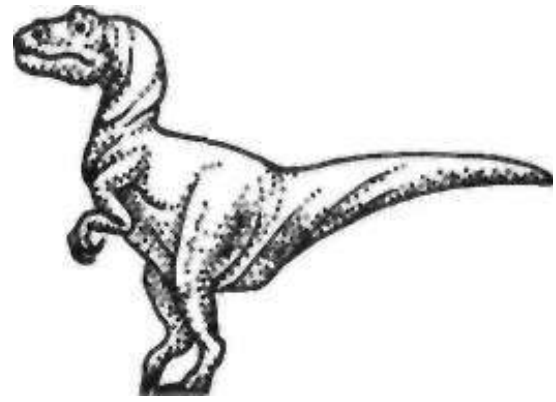
O principal objetivo de um sistema de computação é executar programas. Esses programas, juntamente com os dados que eles accssam, devem estar (pelo menos parcialmente) na memória principal durante a execução.

Para melhorar a utilização da CPU e a velocidade de resposta aos usuários, o computador deve manter vários processos na memória. Os muitos esquemas distintos de gerência de memória rcflctem várias abordagens; a eficácia de determinado algoritmo depende de cada situação específica.

Como a memória principal geralmente é pequena demais para acomodar todos os dados e programas de forma permanente, o sistema de computação deve fornecer armazenamento secundário como apoio. A maioria dos sistemas de computação modernos utilizam discos como o principal meio de armazenamento online para informações (programas e dados). O sistema de arquivos fornece o mecanismo para armazenamento online e acesso aos discos.

Os dispositivos que se acoplam a um computador variam em múltiplas dimensões. Os dispositivos transferem um caractere ou bloco de caracteres de cada vez. Podem ser acessados de forma unicamente sequencial ou aleatória. Transferem dados síncrona ou assincronamente. São dedicados ou compartilhados. Podem ser somente de leitura ou de leitura e escrita. Variam muito em termos de velocidade. De várias formas, também são os componentes mais lentos de um sistema de computação. Como a variedade é muito grande, o sistema operacional deve fornecer uma ampla gama de funcionalidades para permitir que as aplicações controlem todos os aspectos dos dispositivos.

Capítulo 9



GERÊNCIA DE MEMÓRIA

No Capítulo 6, mostramos como a CPU pode ser compartilhada por vários processos. Como resultado do escalonamento de CPU, podemos melhorar a utilização da CPU e a velocidade da resposta do computador para os seus usuários. Para alcançar esse aumento de desempenho, entretanto, é preciso manter vários processos na memória; é preciso *compartilhar* memória.

Neste capítulo, discutimos várias formas de gerenciar memória. Os algoritmos de gerência de memória variam de uma abordagem primitiva próxima à máquina a estratégias de paginação e segmentação. Cada abordagem tem suas próprias vantagens e desvantagens. A Seleção de um método de gerência de memória para um sistema específico depende de muitos fatores, especialmente do projeto de *hardware* do sistema. Como veremos, muitos algoritmos requerem suporte do hardware.

9.1 • Fundamentos

Como vimos no Capítulo 1, a memória é fundamental para a operação de um sistema de computação moderno. A memória consiste em um grande vetor de palavras ou bytes, cada qual com seu próprio endereço. A CPU busca instruções da memória de acordo com o valor do contador de programa. Kssas instruções também poderão causar carga e armazenamento em endereços de memória específicos.

Um ciclo típico de execução de instrução, por exemplo, primeiro busca uma instrução da memória. A instrução é então decodificada e poderá fazer com que operandos sejam buscados na memória. Depois que a instrução tiver sido executada sobre os operandos» os resultados poderão ser armazenados de volta na memória. Observe que a unidade de memória vê apenas um fluxo de endereços de memória; não sabe como eles são gerados (o contador de instrução, indexação, indireçáo, endereços literais e assim por diante) ou para que servem (instruções ou dados). Da mesma forma, podemos ignorar *como* um endereço de memória é gerado por um programa. Kstamos interessados apenas na sequência de endereços de memória gerados pelo programa em execução.

9.1.1 Mapeamento de endereços

Geralmente, um programa reside em disco como um arquivo executável binário. O programa deve ser levado à memória e colocado cm um processo para ser executado. Dependendo do tipo de gerência de memória em uso, o processo pode ser movido entre o disco e a memória durante sua execução. A colcção de processos no disco que está esperando para ser levada para a memória para execução forma a fila de entrada.

O procedimento normal é selecionar uni dos processos na fila de entrada c carregar esse processo na memória. A medida que o processo é executado, ele acessa instruções e dados da memória. Por fim, o processo termina e seu espaço de memória é declarado disponível.

A maioria dos sistemas permite que um processo de usuário resida cm qualquer parte da memória física. Assim, embora o espaço de endereçamento do computador comece em 00000, o primeiro endereço do processo de usuário não precisa ser 00000. Esse arranjo afeta os endereços que o programa de usuário pode usar.

Na maioria dos casos, um programa de usuário passará por várias etapas (algumas das quais podem ser opcionais) antes de ser executado (t'igura 9.1). Os endereços podem ser representados de diferentes formas durante essas etapas. Os endereços no programa-fonte são geralmente simbólicos (tais como count). Em geral, um compilador vai fazer a associação desses endereços simbólicos com endereços relocávcis (tais como "14 bytes a partir do início desse módulo"). O carregador ou linkeditor, por sua vez, vai associar esses endereços relocáveis aos endereços absolutos (como 74014). Cada associação é um mapeamento de um espaço de endereçamento para outro.

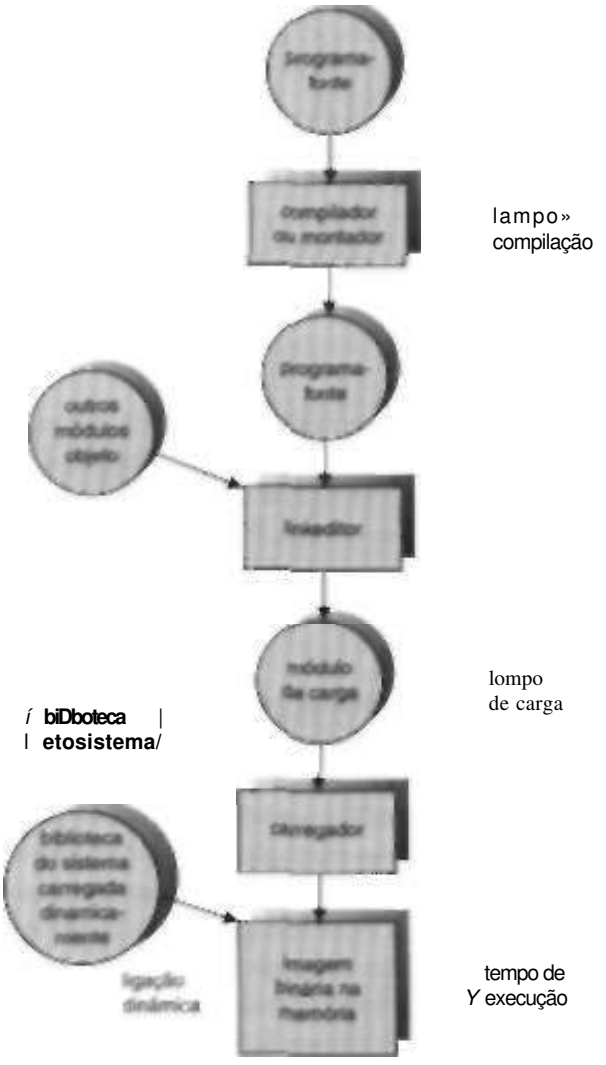


Figura 9.1 As várias etapas do processamento de um programa de usuário.

Classicamente, a associação de instruções e dados com endereços de memória pode ser feita em qualquer etapa da vida de um programa.

- *Em tempo de compilação:* Se a posição de memória onde o processo residirá for conhecida em tempo de compilação, então, um código absoluto poderá ser gerado. Por exemplo, se soubermos *a priori* que um processo de usuário reside na posição R, o código compilado iniciará naquela posição e se estenderá a partir daí. Se, mais tarde, a posição de início mudar, então será necessário recompilar o código. Os programas de formato .COM do MS-DOS são código absoluto associados em tempo de compilação.
- *Em tempo de carga:* Se durante a compilação não estiver determinado onde o processo residirá na memória, o compilador deverá gerar um código relocável. Nesse caso, a associação final c* retardada até o instante de carga. Se o endereço de início mudar, basta recarregar o código de usuário para incorporar o valor alterado.

- *Em tempo de execução:* Sc o processo puder ser movido durante sua execução de um segmento de memória para outro, a associação deverá ser retardada até o tempo de execução. Hardware especial deve estar disponível para que esse esquema funcione, como será discutido na Seção 9.1.2. A maioria dos sistemas operacionais de uso geral utiliza esse método.

Uma parte considerável deste capítulo será dedicada a mostrar como essas várias associações podem ser implementadas de forma eficaz em um sistema de computação e discutir o suporte de hardware adequado.

9.1.2 Espaço de endereçamento lógico *versus* físico

Um endereço gerado pela CPU é normalmente chamado de endereço lógico, enquanto um endereço visto pela unidade de memória, ou seja, aquele carregado no registrador de endereço da memória, é normalmente chamado de endereço físico.

Os métodos de resolução de endereço em tempo de compilação e tempo de carga resultam em um ambiente no qual os endereços lógicos e físicos são iguais. Por outro lado, o esquema de resolução de endereços em tempo de execução define um contexto no qual os endereços lógicos c físicos diferem. Nesse caso normalmente fazemos referência ao endereço lógico como endereço virtual. Usamos *endereço lógico* e *endereço virtual* indistintamente neste texto. O conjunto de todos os endereços lógicos gerados por um programa é um espaço de endereçamento lógico; o conjunto de todos os endereços físicos que correspondem a esses endereços lógicos *é* um espaço de endereçamento físico. Assim, no esquema de resolução de endereço em tempo de execução, os espaços de endereçamento lógico e físico diferem.

O mapeamento de tempo de execução dos endereços virtuais para físicos é feito pela unidade de gerência da memória (MMU - Memory-Managcment Unit), que é um dispositivo de hardware. Kxistem muitos métodos diferentes para obter tal mapeamento, conforme será discutido nas Seçõcs 9.3,9.4, 9.5 e 9.6. Por enquanto, esse mapeamento será mostrado com um esquema MMU simples, que é uma generalização do esquema de registrador básico descrito na Seção 2-5.3.

Como ilustrado na Figura 9.2, esse método requer um suporte de hardware ligeiramente diferente da configuração de hardware discutida na Seção 2.4. O registrador de base agora é chamado de registrador de relocação. O valor do registrador de relocação é *adicionado* a todo endereço gerado por um processo de usuário no momento em que ele é enviado para a memória. Por exemplo, se a base está em 14.000, então unia tentativa por parte do usuário de endereçar a posição 0 é dinamicamente relocada para a posição 14.000; um acesso à posição 346 é mapcado para a posição 14.346. O sistema operacional MS-DOS, executando na família de processadores Intel 80X86, utiliza quatro registradores de relocação quando está carregando e executando processos.

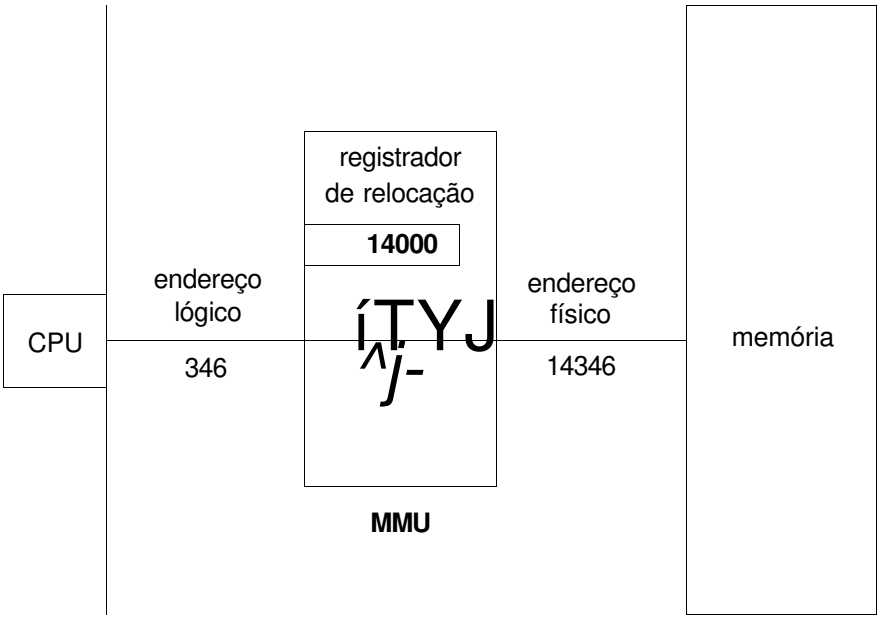


Figura 9.2 Relocação dinâmica usando um registrador de relocação.

Observe que o programa de usuário nunca vê os endereços físicos *reais.* O programa pode criar um ponteiro para a posição 346, armazená-lo na memória, manipulá-lo, compará-lo com outros endereços - tudo isso como o número 346. Somente quando ele for usado como um endereço de memória (em uma carga ou armazenamento indireto, por exemplo) ele será relocado em relação ao registrador de base. O programa de usuário lida com endereços *lógicos.* O hardware de mapeamenio de memória converte os endereços lógicos em físicos. Essa forma de associação em tempo de execução foi discutida na Seção 9.1.1. A posição final de um endereço de memória referenciado só é determinada quando a referência é feita.

Observe também que agora temos dois tipos diferentes de endereços: endereços lógicos (na faixa de 0 a *máx)* e endereços físicos (na faixa *R* + 0 a *R* + *máx* para o valor base *R). O* usuário gera apenas endereços lógicos e pensa que o processo executa nas posições 0 a *máx.* O programa de usuário fornece endereços lógicos; esses endereços lógicos devem ser mapeados em endereços físicos antes de serem usados.

O conceito de um *espaço de endereçamento lógico* que é associado a um *espaço de endereçamento físico* é central a uma gerência de memória adequada.

9.1.3 Carga dinâmica

Até agora em nossa discussão, o programa inteiro e os dados de um processo devem estar na memória física para que o processo seja executado. O tamanho de um processo é limitado ao tamanho da memória física. Para obter melhor utilização de espaço de memória, podemos usar a carga dinâmica. Com a carga dinâmica, uma rotina só é carregada quando é chamada. Todas as rotinas são mantidas em disco em um formato de carga relocável. O programa principal é carregado na memória e é executado. Quando uma rotina precisa chamar outra rotina, a rotina que está chamando primeiro verifica se a outra rotina foi carregada. Se não tiver sido carregada, o carregador relocável é chamado para carregar a rotina desejada na memória e atualizar as tabelas de endereços do programa para refletir essa alteração. Em seguida, o controle é passado para a rotina recém-carregada.

A vantagem da carga dinâmica é que uma rotina não-utilizada nunca é carregada. Esse método é particularmente útil quando há necessidade de grande quantidade de código para lidar com casos que ocorrem com pouca frequência, como rotinas de erro. Nesse caso, embora o tamanho total do programa possa ser grande, a parte que é usada de fato (e, portanto, carregada) pode ser muito menor.

A carga dinâmica não requer suporte especial do sistema operacional. E responsabilidade dos usuários projetar seus programas para aproveitar esse método. Os sistemas operacionais podem 3Judar o programador, no entanto, fornecendo rotinas de biblioteca para implementar a carga dinâmica.

9.1.4 Ligação dinâmica e bibliotecas compartilhadas

Observe que a Figura 9.1 também mostra bibliotecas com ligação dinâmica. Alguns sistemas operacionais oferecem suporte apenas à ligação estática, na qual as bibliotecas de linguagem do sistema são tratadas como qualquer outro módulo objeto e são combinados pelo carregador na imagem binária do programa. O conceito de ligação dinâmica é semelhante ao conceito de carga dinâmica. Em vez de a carga ser adiada até o tempo de execução, a ligação é adiada. Esse recurso geralmente é usado com as bibliotecas do sistema, tais como bibliotecas de sub-rotinas de linguagem. Sem esse recurso, todos os programas em um sistema precisam ter uma cópia da biblioteca de linguagem (ou pelo menos das rotinas referenciadas pelo programa) incluída na imagem executável. Esse requisito desperdiça espaço em disco e memória principal. Com a ligação dinâmica, um stub é incluído na imagem para cada referência de rotina de biblioteca. Esse stub é um pequeno trecho de código que indica como localizar a rotina de biblioteca apropriada residente na memória ou como carregar a biblioteca se a rotina ainda não estiver presente.

Quando esse stub é executado, ele verifica se a rotina necessária já está na memória. Se a rotina não estiver na memória, o programa a carrega na memória. De qualquer forma, o stub substitui a si mesmo pelo endereço da rotina e a executa. Assim, da próxima vez que aquele segmento de código for acessado, a rotina de biblioteca será executada diretamente, não incorrendo em custo para ligação dinâmica. Nesse esquema, todos os processos que utilizam uma biblioteca de linguagem executam apenas uma cópia do código de biblioteca.

Esse recurso pode ser estendido as .uualizações de biblioteca (como correções de bugs). Uma biblioteca pode ser substituída por uma nova versão e todos os programas que fazem referência à biblioteca passarão a usar automaticamente a nova versão. Sem a ligação dinâmica, todos esses programas precisariam ser linkeditados novamente para obter acesso à nova biblioteca. Para que os programas não executem acidentalmente novas versões incompatíveis das bibliotecas, as informações de versão estão incluídas no programa e na biblioteca. Mais de uma versão de uma biblioteca pode ser carregada na memória e cada programa utiliza suas informações de versão para decidir que cópia da biblioteca deverá ser utilizada. Pequenas alterações retêm o mesmo número de versão, enquanto grandes alterações incrementam o número da versão. Assim, somente os programas que são compilados com a nova versão de biblioteca são afetados pelas alterações incompatíveis incorporadas nela. Outros programas linkeditados antes que a nova biblioteca tenha sido instalada continuarão a usar a biblioteca mais antiga. Esse sistema também é chamado de bibliotecas compartilhadas.

Diferentemente da carga dinâmica, a ligação dinâmica geralmente requer ajuda do sistema operacional. Se os processos na memória estiverem protegidos uns dos outros (Seção 9.3), o sistema operacional será a única entidade que poderá verificar sç a rotina necessária está no espaço de memória de outro processo ou que poderá permitir que vários processos acessem os mesmos endereços de memória. Esse conceito será discutido em maiores detalhes quando tratarmos de paginação na Seção 9.4.5.

9.1.5 Overlays

Para que um processo possa ser maior do que a quantidade de memória alocada a ele, podemos usar overlays. A ideia do overlay é manter na memória apenas as instruções e dados que são necessários em determinado momento. Quando outras instruções são necessárias, elas são carregados no espaço que foi anteriormente ocupado por instruções que não são mais necessárias.

Como exemplo, considere um montador em dois passos. Durante ao passo I, ele constrói uma tabela de símbolos; em seguida, durante o passo 2, ele gera um código em linguagem de máquina. Podemos dividir esse montador cm código de passo 1, código de passo 2, a tabela de símbolos e as rotinas de suporte comuns usadas pelos passos 1 e 2. Vamos supor que os tamanhos desses componentes sejam os seguintes (K significa "kilobyte", que é 1.024 bytes):

Passo 1	70 K
Passo 2	KOK
Tabela de símbolos	20K
Rotinas comuns	30K

Para carregar tudo de uma vez, seria necessário ter 200K de memória. Se apenas 150K estiverem disponíveis, não será possível executar o processo. No entanto, observe que o passo 1 e o passo 2 não precisam estar na memória ao mesmo tempo. Assim, definimos dois overlays: o overlay *A* consiste na tabela de símbolos, nas rotinas comuns e no passo 1, e o overlay *B* consiste na tabela de símbolos, nas rotinas comuns e no passo 2.

Acrescentamos um driver de overlay (1 OK) e começamos com o overlay *A* na memória. Quando terminamos o passo 1, passamos para o driver de overlay, que lê o overlay fí na memória, sobrescrevendo o overlay *A* e, em seguida, transfere o controle para o passo 2. O overlay *A* precisa apenas de 120K, enquanto o overlay íí precisa de 130K (Figura 9J). Agora, podemos executar nosso montador nos 150K de memória. Ele carregará um pouco mais rápido porque menos dados precisam ser transferidos antes da execução começar. No entanto, ele executará um pouco mais lento, devido à operação de l/O extra para ler o código do overlay *B* sobre o código do overlay *A*.

O código do overlay *A* e o código do overlay *B* são mantidos no disco como imagens de memória absoluta e são lidos pelo driver de overlay conforme necessário. Algoritmos especiais de relocação e ligação são necessários para construir os overlays.

Como na carga dinâmica, os overlays não precisam de suporte especial do sistema operacional. Eles podem ser completamente implementados pelo usuário com estruturas de arquivos simples, lendo dos arquivos para a memória e, em seguida, pulando para aquela posição de memória e executando as instruções reccm-lidas. O sistema operacional só verifica que existe mais l/O do que o normal.

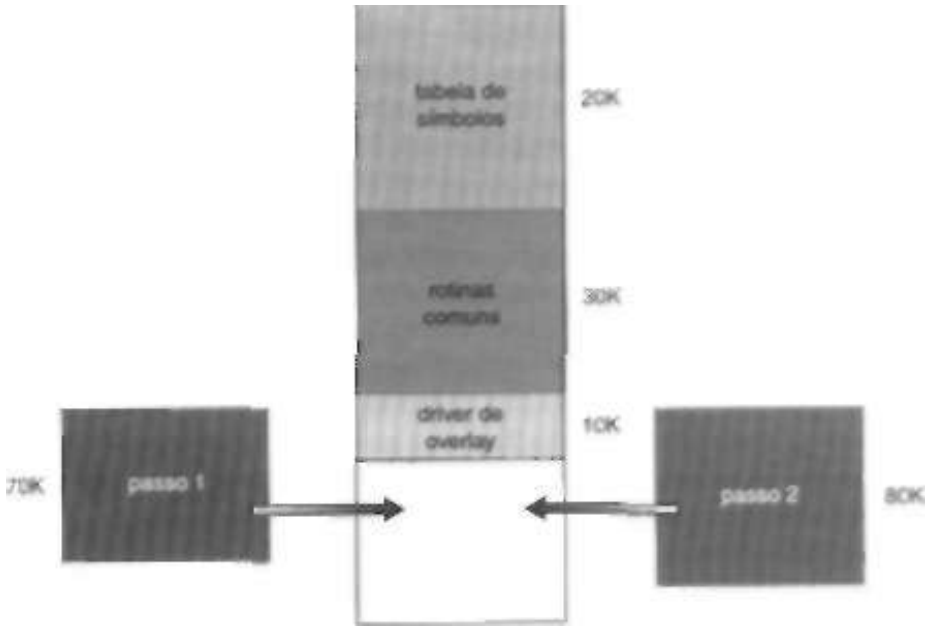


Figura 9.3 Ovcrlays para um montador de dois passos.

O programador, por outro lado, deve projetar e programar a estrutura de overlay adequadamente. Essa tarefa pode ser muito difícil, exigindo conhecimento profundo da estrutura do programa, seu código e estruturas de dados. Como o programa é, por definição, grande {os programas pequenos não precisam do overlay), obter um entendimento satisfatório do programa pode ser difícil. Por isso, o uso de overlays está limitado atualmente a microcomputadores c outros sistemas que possuem quantidades limitadas de memória física e que não tenham suporte de hardware para técnicas mais avançadas. Alguns compiladores de computador fornecem suporte de overlays ao programador para tornar a tarefa mais fácil. Técnicas automáticas para executar programas grandes em quantidades limitadas de memória física são preferíveis.

9.2 • Swapping

Um processo precisa estar na memória para ser executado. Um processo, no entanto, pode ser removido temporariamente da memória para um armazenamento auxiliar e, em seguida, retornado à memória para continuar sua execução. Por exemplo, considere um ambiente de multiprogramação com um algoritmo de escalonamento de CPU round-robin. Quando o quantum expirar, o gerenciador de memória começará a descarregar o processo que acabou de terminar (operação de *swap out)* e carregar outro processo para o espaço de memória que foi liberado (operação de *swap m)* (Figura 9.4). Enquanto isso, o escalonador de CPU alocará uma fatia de tempo para algum outro processo na memória. Quando cada processo terminar seu quantum, ele será trocado por outro processo. Idealmente, o gerenciador de memória poderá trocar processos de forma rápida o suficiente de modo que existam sempre processos na memória prontos para executar quando o escalonador de CPU desejar reescalonar a CPU. O quantum deve ser grande o suficiente para que quantidades razoáveis de cálculos sejam efetuadas entre as trocas.

Uma variante dessa regra de troca, ou swapping, *é* usada para algoritmos de escalonamento com base em prioridade. Se um processo de prioridade mais alta chegar e desejar serviço, o gerenciador de memória poderá descarregar o processo de prioridade mais baixa para que ele possa carregar e executar o processo de prioridade mais alta. Quando o processo de prioridade mais alta terminar, o processo de prioridade mais baixa poderá voltar e continuar processando. Essa variante da operação de troca às vezes é chamada de roll out, roll in.

Normalmente, um processo que é descarregado será carregado para o mesmo espaço de memória que ocupava anteriormente. Essa restrição é determinada pelo método de resolução de endereço. Se a resolução for feita no momento de carga ou montagem, o processo não poderá ser movido para posições diferentes. Se a resolução em tempo de execução estiver sendo usada, é possível passar um processo para um espaço de memória diferente, porque os endereços físicos são calculados durante o tempo de execução.

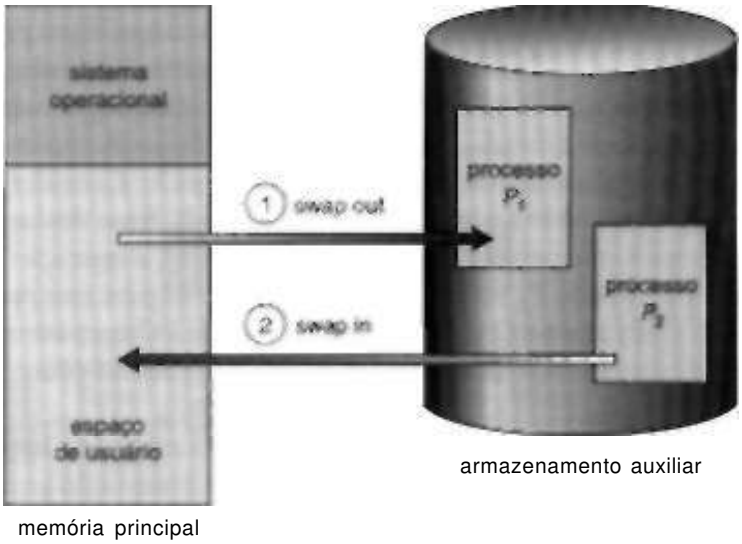


Figura 9.4 Troca de dois processos usando um disco como armazenamento auxiliar.

A troca requer um armazenamento auxiliar, geralmente constituído por um disco rápido. Ele deve ser grande o suficiente para acomodar cópias de todas as imagens de memória para todos os usuários, e deve fornecer acesso direto a essas imagens de memória. O sistema mantém uma fila de processos prontos consistindo em todos os processos cujas imagens de memória estejam no armazenamento auxiliar ou na memória principal e prontas para executar. Sempre que o escalonador de CPU executar um processo, ele chama o dispatcher. O dispatcher verifica se o próximo processo na fila está na memória. Sc o processo não estiver, e não houver região de memória livre, o dispatcher descarrega um processo que está na memória *(stvap out)* e carrega o processo desejado em seu lugar *(swap in).* Em seguida, ele recarrega os registradores da forma usual e transfere o controle para o processo selecionado.

E evidente que o tempo de troca de contexto em um sistema de troca como esse é relativamente alto. Para ter uma ideia do tempo de troca de contexto, vamos supor que o processo de usuário tem 1 megabyte e o armazenamento auxiliar é um disco rígido padrão com uma taxa de transferencia de 5 megabytes por segundo. A transferência real do processo de 1MB entre o disco e a memória leva

$$\begin{aligned} \text{1000K/5000K por segundo} &= 1/5 \text{ segundo} \\ &= 200 \text{ milissegundos} \end{aligned}$$

Considerando que não há necessidade de posicionamento da cabeça de leitura e tomando uma latência média de 8 milissegundos, o tempo de troca é de 208 milissegundos. Como devemos efetuar a descarga e a carga do processo, o tempo de troca total é de 416 milissegundos.

Para a utilização eficiente da CPU, nosso tempo de execução para cada processo deve ser longo em relação ao tempo de troca. Assim, em um algoritmo de escalonamento de CPU round-robin, por exemplo, o quantum de tempo deve ser consideravelmente maior do que 0,416 segundos.

Observe que a parte principal do tempo de troca *é o* tempo de transferência. O tempo de transferência total *é* diretamente proporcional *h quantidade* de memória trocada. Se o sistema de computador tiver 128 megabytes de memória principal e um sistema operacional residente que ocupa 5MB, o tamanho máximo do processo de usuário será 123MB. No entanto, muitos processos de usuário podem ser bem menores do que isso, digamos, 1MB. Um processo de 1MB poderia ser movido em 208 milissegundos, comparado com 24,6 segundos para a movimentação de 123MB. Portanto, seria útil saber exatamente quanta memória determinado processo de usuário *está* utilizando, c não simplesmente quanto ele *poderia* estar usando. Então, precisaríamos trocar apenas o que é efetivamente usado, reduzindo o tempo de troca. Para que esse método funcione, o usuário precisa manter o sistema informado sobre qualquer mudança nos requisitos de memória. Assim, um processo com requisitos de memória dinâmica precisará emitir chamadas ao sistema (request memory e rei ease memory) para informar o sistema operacional sobre suas mudanças de necessidade de memória.

Existem outras limitações na operação de troca. Se desejamos descarregar um processo, é preciso ter certeza de que ele está completamente inativo. Qualquer l/O pendente é particularmente preocupante. Um processo pode estar esperando por uma operação de l/O quando quisermos descarregar esse processo para liberar sua memória. No entanto, se a operação de entrada/saída estiver acossando assincronamente a memória de usuário para usar os buffers de i/O, então o processo não poderá ser descarregado. Considere que a operação de I/O foi colocada na fila porque o dispositivo estava ocupado. Se o pnKesso P_x fosse descarregado (operação de *swap ont)* e o processo /*. fosse carregado na memória (operação de *swap* /«)»a operação de l/O poderia tentar usar memória que agora pertence ao processo P>. As duas principais soluções para esse problema são: (1) nunca descarregar um processo com l/O pendente ou (2) executar operações de I/O apenas em buffers do sistema operacional. As transferencias entre o sistema operacional e a memória de processo ocorrem apenas quando um processo é carregado *(swap in).*

A suposição de que a troca requer poucos reposicionamentos na cabeça de leitura precisa de explicações adicionais. Esse tópico só será discutido no Capítulo 13, em que a estrutura do armazenamento secundário é abordada. Em geral, o espaço de swap é alocado como uma parte do disco, separada do sistema de arquivos, de modo que sua utilização é a mais rápida possível.

Atualmente, o procedimento de troca padrão é usado em poucos sistemas. Ele requer um tempo excessivo de troca e fornece pouco tempo de execução para ser uma solução aceitável de gerência de memória. Versões modificadas, no entanto, são encontradas em muitos sistemas.

Uma modificação da operação de swapping é utilizada em muitas versões do UNIX. Normalmente, o swapping ficaria desabilitado, mas seria iniciado se muitos processos estivessem executando e usando um valor limite de memória. Esse procedimento seria novamente interrompido se a carga no sistema fosse reduzida. A gerência de memória no UNIX é descrita na Scção 20.6.

Os primeiros PCs não tinbam hardware sofisticado (ou sistemas operacionais que utilizassem bem o hardware) para implementar métodos de gerência de memória mais avançados, mas eles eram usados para executar múltiplos processos grandes através de uma versão modificada de swapping. Um exemplo importante é o sistema operacional Microsoft Windows 3.1, que suporta a execução concorrente de processos na memória. Se um novo processo for carregado e não houver memória principal suficiente, um processo antigo será descarregado para o disco. Esse sistema operacional, no entanto, não oferece o procedimento de troca completo, já que o usuário, e não o cscalonador, decide o momento da preempção de determinado processo. Qualquer processo descarregado permanecerá assim (e sem executar) até que o usuário selecione esse processo para execução. Sistemas operacionais da Microsoft posteriores, como o Windows NT, aproveitam os recursos avançados de MMU agora encontrados até em PCs. Na Scção 9.6, descrevemos o hardware de gerência de memória encontrado na família de processadores Intel 386 utilizado em muitos PCs. Nessa seção, também descrevemos o tipo de gerência de memória usado na CPU por outro sistema operacional avançado para PCs, o IBM OS/2.

9.3 • Alocação contígua de memória

A memória principal deve acomodar o sistema operacional e os vários processos de usuário. A memória geralmente é dividida em partições: uma para o sistema operacional residente e outra para os processos de usuário. É possível colocar o sistema operacional na memória baixa ou na memória alta. O principal fator que afeta essa decisão é a localização do vetor de interrupção. Como o vetor de interrupção geralmente está na memória baixa, é mais comum colocar o sistema operacional na memória baixa. Assim, neste texto, vamos discutir apenas a situação na qual o sistema operacional reside na memória baixa. O desenvolvimento da outra situação é semelhante.

Como é recomendável, em geral, que haja vários processos de usuário residentes na memória ao mesmo tempo, precisamos considerar o problema de como alocar memória disponível aos vários processos que estão na fila de entrada esperando para serem carregados na memória. Antes de fazê-lo, devemos discutir a questão da proteção da memória, ou seja, proteger o sistema operacional contra os processos de usuário e proteger os processos de usuário uns dos outros. Essa proteção é fornecida com o uso de um registrador de relocação, conforme discutido na Seção 9.1.2, com um registrador de limite, conforme discutido na Seção 2.5.3. O registrador de relocação contém o valor do menor endereço físico; o registrador de limite contém a faixa de en-

dereços lógicos (por exemplo, relocação = 100.040 e limite = 74.600). Com os registradores de relocação e limite, cada endereço lógico deve ser menor do que o registrador de limite; a MMU mapeia o endereço lógico *dinamicamente,* adicionando o valor do registrador de relocação. Esse endereço mapeado é enviado para a memória (Figura 9.5).

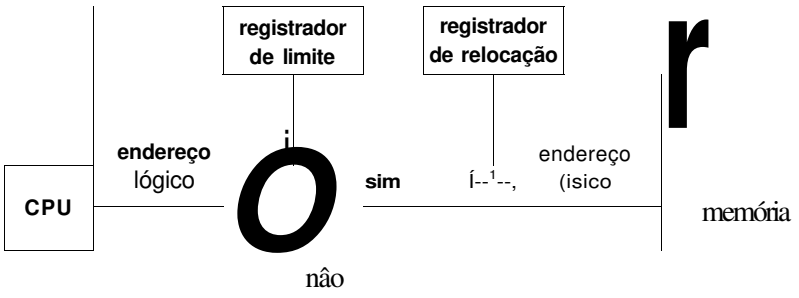


Figura 9.5 Suporte de hardware para os registradores de relocação e limite.

Quando o cscalonador de CPU sclcciona um processo para execução, o dispatcher carrega os registradores de relocação e limite com os valores corretos como parte da troca de contexto. Como todo endereço gerado pela CPU é verificado em relação a esses registradores, podemos proteger o sistema operacional e os programas e dados de outros usuários contra modificação por esse processo em execução.

Observe que o esquema do registrador de relocação fornece uma forma eficaz de permitir que o tamanho do sistema operacional mude dinamicamente. Essa flexibilidade é desejável em muitas situações. Por exemplo, o sistema operacional contem código e espaço de buffer para drivers de dispositivo. Se um driver de dispositivo (ou outro serviço de sistema operacional) não for usado com frequência, não *é* bom manter o código e dados na memória, pois o espaço poderá ser usado para outros propósitos. Tal código às vezes é chamado de código transiente do sistema operacional; ele vem e vai conforme necessário. Assim, usar esse código muda o tamanho do sistema operacional durante a execução do programa.

Um dos métodos mais simples para alocação de memória é dividir a memória cm uma série de partições de tamanho fixo. Cada partição pode conter exatamente um processo. Portanto, o grau de multiprogramaÇão está limitado pelo número de partições. Quando uma partição está livre, um processo é selecionado da fila de entrada e é carregado na partição livre. Quando o processo termina, a partição fica disponível para outro processo. Fsse método foi originalmente utilizado pelo sistema operacional IBM OS/360 (chamado MFT); não está mais em uso. O método descrito a seguir é uma generalização do esquema de partição fixa (chamado MVT); ele é usado principalmente em um ambiente batch. Observe que muitas das ideias apresentadas aqui também se aplicam em um ambiente de tempo compartilhado no qual a segmentação pura é usada para o gerência de memória (Seção 9.5).

O sistema operacional mantém uma tabela indicando que partes de memória estão disponíveis e que partes estão ocupadas. Inicialmente, toda memória está disponível para processos de usuário e é considerada como um grande bloco de memória disponível, um "buraco". Quando um processo chega e precisa de memória, procuramos um bloco de memória livre grande o suficiente para esse processo. Se encontrarmos um, alocamos apenas a quantidade necessária de memória, mantendo o restante disponível para satisfazer pedidos futuros.

À medida que os processos entram no sistema, são colocados em uma fila de entrada. O sistema operacional leva cm conta os requisitos de memória de cada processo c a quantidade de espaço de memória disponível para determinar a que processos a memória será alocada. Quando um processo recebe espaço, ele é carregado na memória e poderá competir pela CPU. Quando um processo termina, ele libera sua memória, que o sistema operacional poderá preencher com outro processo da fila de entrada.

A qualquer momento, existe uma lista de tamanhos de bloco disponíveis e a fila de entrada. O sistema operacional poderá ordenar a fila de entrada de acordo com um algoritmo de escalonamento. A memória é alocada

aos processos até que, finalmente, os requisitos de memória do próximo processo não possam ser satisfeitos; nenhum bloco de memória disponível (ou *"buraco")* é grande o suficiente para armazenar esse processo. O sistema operacional pode então esperar até que um bloco grande o suficiente esteja disponível ou percorrer a fila de entrada para ver se os requisitos menores de memória de algum outro processo podem ser atendidos.

Em geral, existe sempre um *conjunto* de blocos de memória livres, de vários tamanhos, dispersos na memória. Quando um processo chega e precisa de memória, é feita uma busca no conjunto por um bloco que seja grande o suficiente para o processo. Se o bloco for grande demais, ele será dividido em dois: uma parte é alocada ao processo que chega; a outra é devolvida para o conjunto de blocos livres. Quando um processo termina, ele libera seu bloco de memória, que é então colocado de volta no conjunto de blocos livres. Se o novo bloco de memória for adjacente aos outros, esses blocos adjacentes são reunidos para formar um bloco de memória livre maior. Neste ponto, talvez seja necessário verificar se existem processos esperando por memória e se essa memória recém-liberada e recombinada pode satisfazer as exigências de qualquer um dos processos em espera.

Esse procedimento é uma instância particular do problema de alocação de memória dinâmica geral, que consiste em como atender a um pedido de tamanho « de uma lista de blocos de memória livres. Existem muitas soluções para esse problema. O conjunto de blocos é pesquisado para determinar qual deles deve ser alocado. As estratégias de first-fit, best-fit e worst-fit são as mais comumente usadas para selecionar um bloco de memória livre do conjunto de blocos disponíveis.

- *Fif$t-fÍt* A loca o *primeiro* bloco de memória grande o suficiente. A busca pode começar no início do conjunto de blocos ou no ponto em que a pesquisa first-fit anterior terminou. Podemos interromper a busca assim que encontramos um bloco de memória grande o suficiente.
- *Best-fit: Aloca o mertorbioco* de memória grande o suficiente. É preciso procurar na lista inteira, a menos que a lista seja ordenada por tamanho. Essa estratégia gera o menor bloco de memória restante.
- *Worst-fit:* Aloca o *maior* bloco de memória. Mais uma vez, é preciso procurar na lista inteira, a menos que ela esteja classificada por tamanho. Essa estratégia gera o maior bloco de memória restante, que pode ser mais útil do que o bloco restante menor de uma abordagem best-fit.

Simulações têm mostrado que as estratégias de first-fit e best-fit são melhores do que a de worst-fit em termos de redução de tempo e utilização da memória. As duas receberam a mesma avaliação em termos de utilização de memória, mas em geral a estratégia de first-fit é mais rápida.

Os algoritmos que acabamos de descrever sofrem de fragmentação externa. A medida que os processos são carregados e descarregados da memória, o espaço livre na memória é quebrado em pequenas partes. A fragmentação externa existe quando* há espaço na memória total suficiente para atender a um pedido, mas não é contíguo; a memória é fragmentada em um grande número de pequenos blocos.

Esse problema de fragmentação pode ser grave. No pior caso, teríamos um bloco de memória livre (desperdiçada) entre cada dois processos. Se toda essa memória estivesse cm um grande bloco livre, talvez fosse possível executar um número muito maior de processos. A Seleção das estratégias de first-fit x best-fit pode afetar a quantidade de fragmentação. (A estratégia first-fit é melhor para alguns sistemas, enquanto a best-fit é melhor para outros.). Outro fator équal lado de um bloco livre éalocado. (Qual é a parte restante-aquela no topo ou aquela na base?) Independentemente dos algoritmos usados, no entanto, a fragmentação externa será um problema.

Dependendo da quantidade total de memória e do tamanho médio de um processo, a fragmentação externa pode ser um problema mais ou menos grave. A análise estatística do método first-fit, por exemplo, revela que mesmo com alguma otimização, dados N blocos alocados, outros $0{,}5N$ blocos serão perdidos devido à fragmentação. Ou seja, um terço da memória poderá ser inutilizável! Essa propriedade é chamada de regra dos 50%.

Existe outro problema que ocorre com o esquema de alocação de partições múltiplas. Considere um bloco de memória livre de 18.464 bytes. Vamos supor que o próximo processo solicite 18.462 bytes. Sealocarmos exatamente o bloco solicitado, ficaríamos com um bloco de memória livre de 2 bytes. O custo para manter o controle desse bloco de memória será consideravelmente maior do que o próprio bloco. A abordagem geral é quebrar a memória física em blocos de tamanho fixo e alocar memória em unidades de tamanhos de

bloco. Dessa forma, a memória alocada a um processo pode ser ligeiramente maior do que a memória solicitada. A diferença entre esses dois números é a fragmentação interna, ou seja, memória que é interna a uma partição, mas que não é utilizada. /»

Uma solução para o problema da fragmentação externa *è* a compactação. A meta c trocar de posição o conteúdo da memória para reunir toda memória livre em um grande bloco. A compactação nem sempre é possível. Se a relocação é estática c for feita no momento da montagem OU carga, a compactação não poderá ser feita. A compactação *só* será possível se a relocação for dinâmica e for feita em tempo de execução. Se os endereços são relocados dinamicamente, a relocação requer apenas mover o programa e os dados e, em seguida, alterar o registrador de base para refletir o novo endereço base. Quando a compactação é possível, devemos determinar seu custo. O algoritmo de compactação mais simples consiste em mover todos os processos em direção a um lado da memória; todos os blocos livres se movem na outra direçâo, gerando um grande bloco de memória disponível. Esse esquema pode ser caro.

Outra solução possível para o problema de fragmentação externa é permitir que o espaço de endereçamento lógico de um processo seja não-contíguo, possibilitando que um processo receba memória física onde ela estiver disponível. Existem duas técnicas complementares para alcançar essa solução: paginação (Seção 9.4) e segmentação (Seção 9.5). Essas duas técnicas também podem ser combinadas (Seção 9.6).

9.4 • Paginação

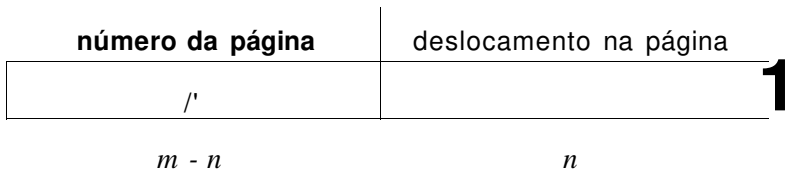
A paginação é um esquema que permite que o espaço de endereçamento físico de um processo seja não-contíguo. A paginação evita o problema de ajustar os pedaços de memória dos mais diversos tamanhos no armazenamento auxiliar, um problema sério que afetou a maioria dos esquemas de gerência de memória anteriores. Quando alguns fragmentos de código ou dados que residem na memória principal precisam ser descarregados (operação de *sivap otit),* deve haver espaço disponível no armazenamento auxiliar. Os problemas de fragmentação discutidos em relação à memória principal também prevalecem com o armazenamento auxiliar, exceto pelo fato de que o acesso é muito mais lento, por isso é impossível fazer a compactação. Devido às vantagens em relação aos métodos anteriores, a paginação em suas muitas formas é utilizada com frequência em muitos sistemas operacionais.

9.4.1 Método básico

A memória física é quebrada em blocos de tamanho fixo chamados quadros *(frames).* A memória lógica também é quebrada em blocos de tamanho igual chamados páginas. Quando um processo vai ser executado, suas páginas são carregadas em qualquer quadro de memória disponível a partir do armazenamento auxiliar. O armazenamento auxiliar é dividido cm blocos de tamanho fixo que têm o mesmo tamanho que os quadros de memória.

0 suporte de hardware para a paginação está ilustrado na Figura 9.6. Cada endereço gerado pela CPU é dividido cm duas partes: um número de página (/>) e um deslocamento de página (Í/), OU offset. O número de página é usado como um índice em uma tabela de página. A tabela de página contém o endereço base de cada página na memória física. Esse endereço base é combinado com o deslocamento de página para definir o endereço de memória física que é enviado para a unidade de memória. O modelo de paginação da memória é apresentado na Figura 9.7.

0 tamanho da página (assim como o tamanho do quadro) é definido pelo hardware. O tamanho de uma página é geralmente uma potência de 2, variando entre 512 bytes e 16 megabytes por página, dependendo da arquitetura do computador. A Seleção de uma potência de 2, como ramanho de página torna particularmente fácil a tradução de um endereço lógico em um número de página e deslocamento de página. Se o tamanho do espaço de endereçamento lógico for *2'"* e o tamanho da página for 2" unidades de endereçamento (bytes ou palavras), os *m — n* bits mais significativos de um endereço lógico designam o número da página, e os «bits menos significativos designam o deslocamento na página. Assim, o endereço lógico será o seguinte:



onde *p* é um índice na tabela de páginas e *d* é o deslocamento dentro da página.

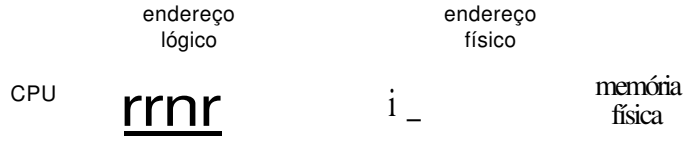


Figura 9.6 Hardware de paginação.

Como um exemplo concreto (embora minúsculo), considere a memória da Figura 9.8. Usando um tamanho de página de 4 bytes e uma memória física de 32 bytes (8 páginas), mostramos como a visão de memória do usuário pode ser mapeada na memória física. O endereço lógico 0 corresponde à página 0, com deslocamento 0. Com a indexação na tabela de página, verificamos que a página 0 está no quadro 5. Assim, o endereço lógico 0 é mapeado no endereço físico 20 (= (5 X 4) + 0). O endereço lógico 3 (página 0, deslocamento 3) *é* mapeado no endereço físico 23 (=(5 x 4) + 3). O endereço lógico 4 corresponde à página 1, deslocamento 0; de acordo com a tabela de página, a página I c mapeada no quadro 6. Assim, o endereço lógico 4 é mapeado no endereço físico 24 (= (6 x 4) + 0). O endereço lógico 13 é mapeado no endereço físico 9.

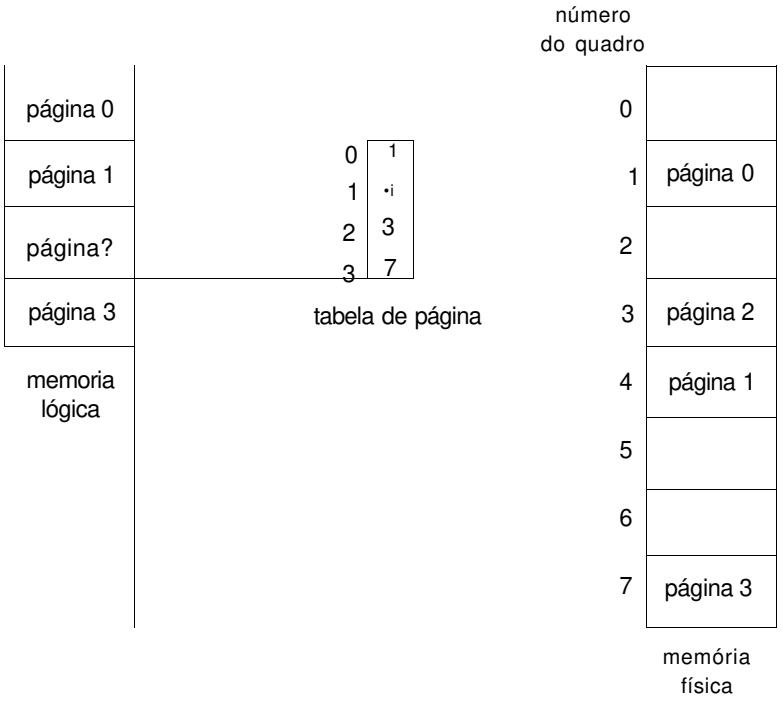


Figura 9.7 Modelo de paginação da memória lógica e física.

Observe que a paginação em si é uma fornia de relocação dinâmica. Todo endereço lógico é associado pelo hardware de paginação a algum endereço físico. O leitor atento terá observado que usar a paginação é semelhante a usar uma tabela de registradores de base (de relocação), um para cada quadro de memória.

Quando usamos um esquema de paginação, não existe fragmentação externa: *qualquer* quadro livre pode ser alocado a um processo que precisa dele. No entanto, pode haver alguma fragmentação interna. Observe que os quadros são alocados como unidades. Se os requisitos de memória de um processo não forem múltiplos do tamanho das páginas, o *último* quadro alocado talvez não fique completamente cheio. Por exemplo, se as páginas tiverem 2048 bytes, um processo de 72.766 bytes precisaria de 35 páginas mais 1086 bytes. Ele receberia 36 quadros, resultando em uma fragmentação interna de 2048 - 1086 = 962 bytes. No pior caso, um processo precisaria de *n* páginas mais um byte. Receberia *n* + 1 quadros, resultando em uma fragmentação interna de praticamente um quadro inteiro.

Se o tamanho do processo for independente do tamanho da página, esperamos que a fragmentação interna tenha em média meia página por processo. Essa consideração sugere que páginas de tamanho pequeno são desejáveis. No entanto, existe um custo envolvido em cada entrada na tabela de página, e esse custo é reduzido à medida que o tamanho das páginas aumenta. Além disso, a entrada/saída de disco é mais eficiente quando o número do dados sendo transferidos é maior (Capítulo 13). Em geral, os tamanhos de página cresceram com o tempo à medida que processos, conjuntos de dados e memória principal aumentaram de tamanho. Hoje cm dia, as páginas geralmente têm entre 2 e 8KB. Algumas CPUs e kernels até suportam múltiplos tamanhos de página. Por exemplo, o Solaris usa páginas de 4K e 8K, dependendo dos dados armazenados pelas páginas. Os pesquisadores agora estão desenvolvendo suporte dinâmico a tamanho de página variável.

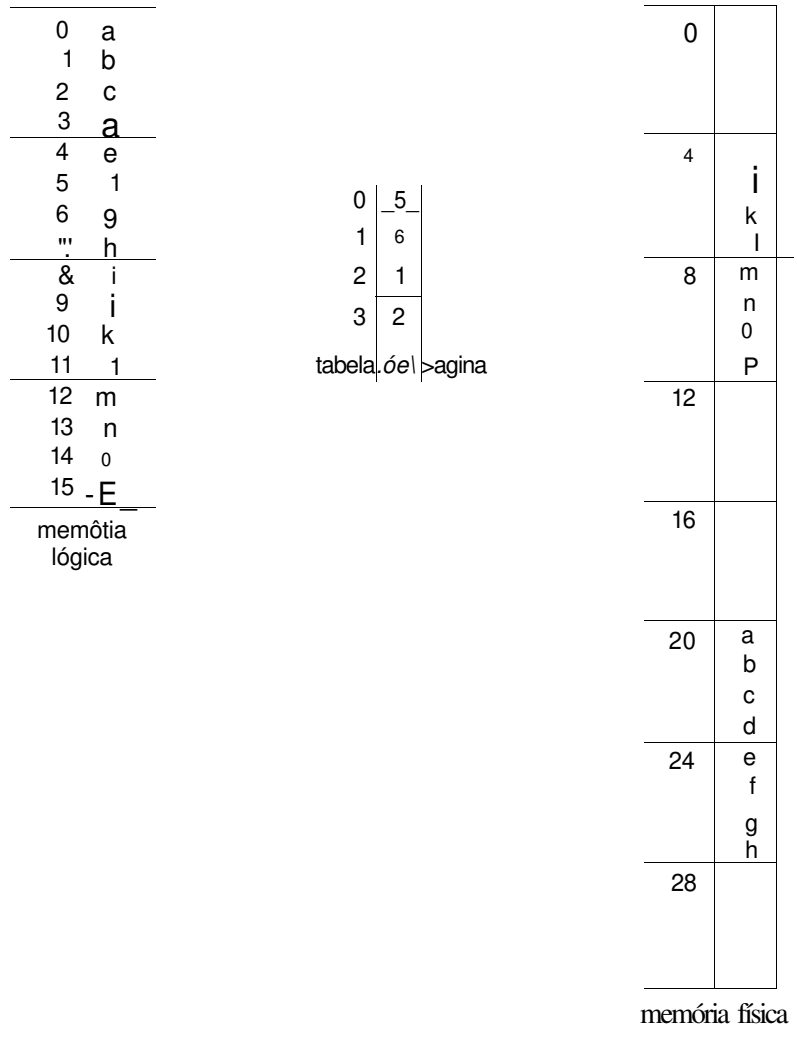


Figura 9.8 Exemplo de paginação para uma memória de 32 bytes com páginas de 4 bytes.

Cada entrada na tabela de página tem geralmente 4 bytes de comprimento, mas esse tamanho também pode variar. Uma entrada de 32 bits pode apontar para um dos 2" quadros de página física. Se um quadro tiver 4K, um sistema com entradas de 4 bytes pode endereçar 2^{16} bytes (ou 64 GB) de memória física.

Quando um processo chega no sistema para ser executado, seu tamanho, expresso em páginas, é examinado. Cada página do processo precisa de um quadro. Assim, se o processo precisar de *n* páginas, deve haver pelo menos « quadros disponíveis na memória. Sc houver H quadros disponíveis, eles são alocados a esse processo que está chegando. A primeira página do processo é carregada em um dos quadros alocados, e o número do quadro é colocado na tabela de página para esse processo. A próxima página é carregada em outro quadro e o seu número de quadro é colocado na tabela de página e assim por diante (Figura 9.9).

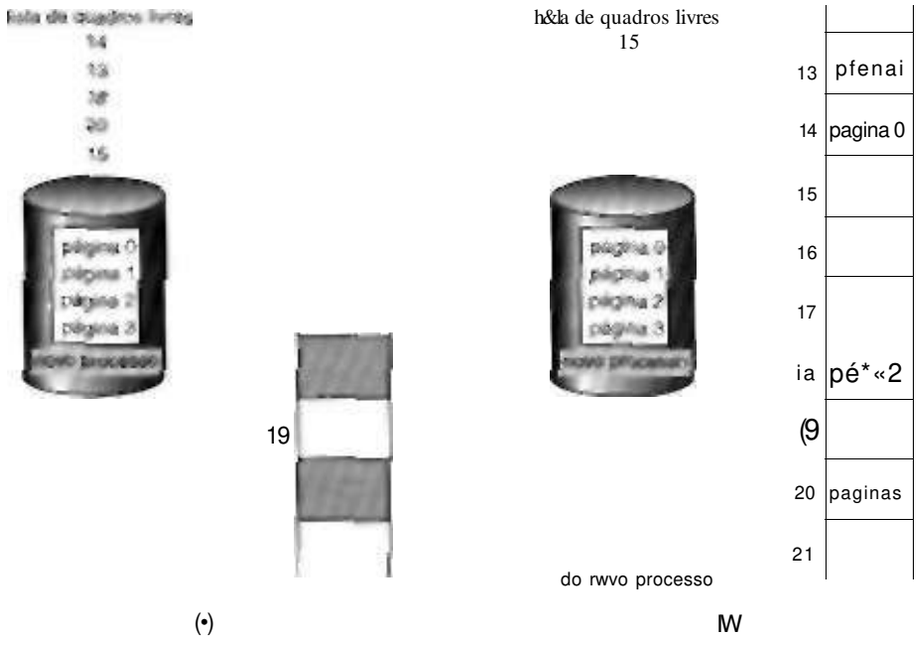


Figura 9.9 Quadros livres, (a) Antes da alocação. (b) Depois da alocação.

Um aspecto importante da paginação é a separação clara entre a visão de memória pelo usuário e a memória física real. O programa de usuário visualiza essa memória como um único espaço contíguo, quç contém apenas esse programa. Na verdade, o programa de usuário está disperso na memória física, que também contém outros programas. A diferença entre a visão da memória pelo usuário e a memória física real é reconciliada pelo hardware de tradução de endereços. Os endereços lógicos são traduzidos em endereços físicos. Esse mapeamento é oculto do usuário e é controlado pelo sistema operacional. Observe que o processo de usuário por definição não é capaz de acessar memória que não lhe pertence. Ele não tem como endereçar memória fora da sua tabela de página e a tabela inclui apenas as páginas que o processo detém.

Como o sistema operacional está gerenciando memória física, ele deve estar ciente dos detalhes de alocação da memória física: que quadros estão alocados, que quadros estão disponíveis, qual o total de quadros existentes e assim por diante. Essas informações são geralmente mantidas em uma estrutura de dados chamada tabela de quadros *{frame table).* A tabela de quadros tem uma entrada para cada quadro de página física, indicando se este está livre ou alocado e, se estiver alocado, a que página de qual processo ou processos.

Além disso, o sistema operacional deve estar ciente de que os processos de usuário operam no espaço de usuário e que todos os endereços lógicos devem ser mapeados para produzir endereços físicos. Se um usuário fizer uma chamada ao sistema (para realizar uma operação de I/O, por exemplo) e fornecer um endereço como parâmetro (um huffer, por exemplo), esse endereço deve ser mapcado para produzir o endereço físico correto. O sistema operacional mantém uma cópia da tabela de página para cada processo, da mesma forma que mantém uma cópia do contador de instruções *t* do conteúdo dos registradores. Essa cópia é usada para traduzir endereços lógicos em endereços físicos sempre que o sistema operacional precisar mapear manualmente um endereço lógico em um endereço físico. Também é usada pelo dispatcher de CPU para definir a tabela de página de hardware quando um processo deverá receber a CPU. A paginação, portanto, aumenta o tempo de troca de contexto.

9.4.2 Estrutura da tabela de página

Cada sistema operacional tem seus próprios métodos para armazenar tabelas de página. A maioria aloca uma tabela de página para cada processo. Um ponteiro para a tabela de página é armazenado com outros valores de registrador (como o contador de instruções) no bloco de controle do processo. Quando o dispatcher *é* informado que determinado processo deve ser iniciado, ele deverá recarregar os registradores de usuário e definir os valores de tabela de página de hardware corretos a partir da tabela de página de usuário armazenada.

9.4.2.1 Suporte **de hardware**

A implementação de hardware da tabela de página pode ser feita de várias formas diferentes. No caso mais simples, a tabela de página é implementada como um conjunto de registradores dedicados. Esses registradores devem ser construídos com uma lógica de altíssima velocidade para tornar a tradução do endereço de paginação eficiente. Cada acesso à memória deve passar pelo mapa de paginação, por isso a eficiência é uma consideração importante. O dispatcher de CPU recarrega esses registradores, da mesma forma que ele recarrega os demais registradores. As instruções para carregar ou modificar os registradores de tabela de página são, é claro, privilegiadas, de modo que somente o sistema operacional pode alterar o mapa da memória. O DEC PDP-11 é um exemplo desse tipo de arquitetura. O endereço consiste em 16 bits e o tamanho da página é 8K. A tabela de página consiste em oito entradas que são mantidas em registradores rápidos.

O uso de registradores para a tabela de página será satisfatório se a tabela for razoavelmente pequena (por exemplo, 256 entradas). A maioria dos computadores modernos, no entanto, permitem que a tabela de página seja muito grande (por exemplo, 1 milhão de entradas). Para essas máquinas, o uso de registradores rápidos para implementar a tabela de página não é viável. Em vez disso, a tabela de página é mantida na memória principal, e um registrador de base da tabela de páginas (PTBR) aponta para a tabela de página. Mudar as tabelas de página requer mudar apenas esse único registrador, reduzindo consideravelmente o tempo de troca de contexto.

0 problema com essa abordagem é o tempo necessário para acessar uma posição na memória do usuário. Se queremos acessar uma posição /, primeiro devemos indexá-la na tabela de página, usando o valor no PTBR deslocado pelo número de página r, Essa tarefa requer acesso à memória. Ela fornece o número do quadro, que é combinado com o deslocamento na página para gerar o endereço real, e então podemos acessar o local desejado na memória. Com esse esquema, *dois* acessos à memória são necessários para acessar um byte (um para a entrada na tabela de página, um para o byte). Assim, a velocidade de acesso à memória é diminuída por um fator de 2. Esse atraso seria intolerável na maior partç dos casos. Podemos simplesmente usar o recurso de troca!

A solução padrão para o problema e* utilizar um cache de hardware especial, pequeno e de busca rápida, chamado registradores associativos ou translation look-aside buffers (TLBs). Um grupo de registradores associativos *6* construído com memória especialmente rápida. Cada registrador consiste em duas partes: uma chave e um valor. Quando um item é apresentado aos registadores associativos, ele é comparado com todas as chaves ao mesmo tempo. Se o item for encontrado, o campo de valor correspondente será apresentado. A pesquisada é rápida: o hardware, no entanto, é caro. Geralmente, o número de entradas em um **TLB** está entre 8 c 2048.

Os registradores associativos são usados com as tabelas de página da seguinte forma. Eles contêm apenas algumas entradas da tabela de página. Quando um endereço lógico é gerado pela CPU, seu número de página é apresentado para um conjunto de registradores associativos que contem os números de página e seus números de quadro correspondentes. Se o número de página for encontrado no registradores associativos, seu número de quadro estará imediatamente disponível c será usado para acessar a memória. A tarefa completa pode levar menos de 10% mais do que levaria caso uma referência de memória não-mapeada fosse utilizada.

Se o número da página não estiver nos registradores associativos, uma referência de memória à tabela de página deverá ser feita. Quando o número do quadro for obtido, podemos usá-lo para acessar a memória (Figura 9.10). Além disso, acrescentamos o número de página e o número do quadro aos registradores associativos de modo que possam ser encontrados rapidamente na próxima referência. Se o TLB já estiver cheio de

entradas, o sistema operacional deverá escolher uma para substituição. Infelizmente, toda vez que uma nova tabela de página é selecionada (por exemplo, cada troca de contexto), o TLB deve ser apagado (operação de *flush)* para garantir que o próximo processo a ser executado não use as informações de tradução erradas. Caso contrário, haveria entradas antigas no TLB contendo endereços virtuais válidos mas endereços físicos incorretos ou inválidos deixados dos processos anteriores.

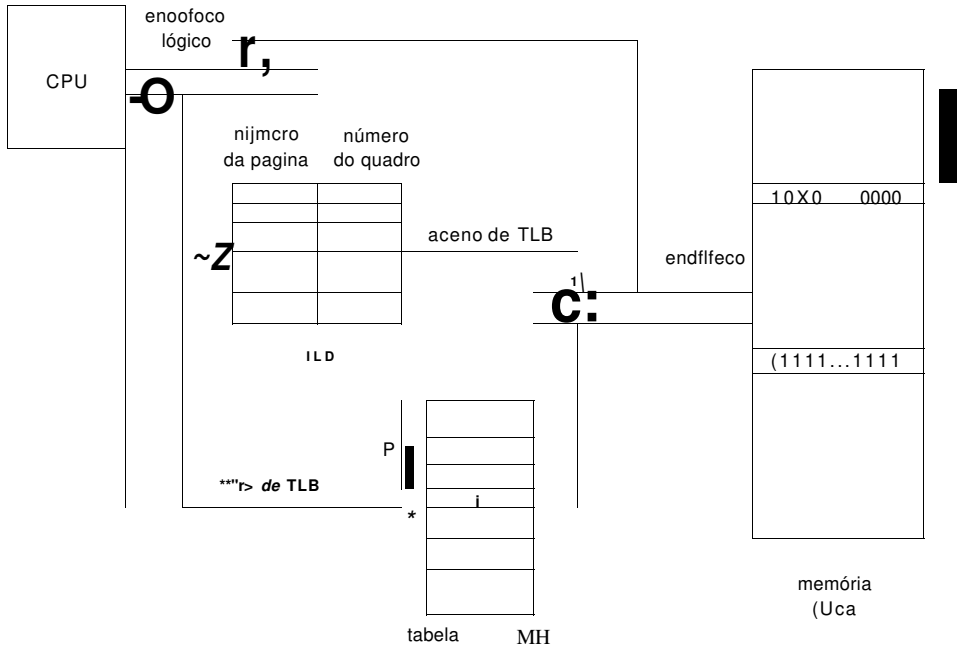


Figura 9.10 Hardware de paginação com TLB.

A percentagem de vezes em que um número de página ê encontrado nos registradores associativos é chamada taxa de acerto. Uma taxa de acerto de 80% significa que encontramos o número de página nos registradores associativos 80% das vezes. Se levar 20 nanossegundos para analisar os registradores associativos c 100 nanossegundos para acessar a memória, então um acesso à memória mapeada levará 120 nanossegundos quando o número da página estiver nos registradores associativos. Se não encontrarmos o número de página nos registradores associativos (20 nanossegundos), primeiro precisamos acessar a memória para a tabela de página e o número do quadro (100 nanossegundos), e depois acessar o byte desejado na memória (100 nanossegundos), para um total de 220 nanossegundos. Para encontrar o tempo efetivo de acesso à memória, é preciso pesar cada caso com sua probabilidade:

$$\text{tempo efetivo de acesso} = 0{,}80 \times 120 + 0{,}20 \times 220 = 140 \text{ nanossegundos}$$

Neste exemplo, sofremos uma redução de 40% no tempo de acesso à memória (de 100 a 140 nanossegundos).

Para uma taxa de acerto de 98%, temos o seguinte:

$$\text{tempo efetivo de acesso} = 0{,}98 \times 120 + 0{,}02 \times 220 = 122 \text{ nanossegundos}$$

Essa alta taxa de acerto produz um aumento de apenas 22% no tempo de acesso.

A taxa de acerto certamente está relacionada ao número de registradores associativos. Com o número de registradores variando entre 16 e 5 12, uma taxa de acerto de 80 a 98% pode ser obtida. O processador Motorola 68030 (usado nos sistemas Apple Macintosh) tem um TLB de 22 entradas. A CPU Intel 80486 (encontrada em alguns PCs) tem 32 registradores e diz ter uma taxa de acerto de 98%. Os processadores UltraSPARC I ÔC II fornecem dois TLBs separados, um para as páginas que contêm instruções e outro para as páginas que contêm dados. Cada um tem 64 entradas.

9.4.2.2 Proteção

A proteção de memória em um ambiente paginado é obtida por bits de protcção que estão associados com cada quadro. Normalmente, esses bits são mantidos na tabela de página. Um bit pode definir uma página para leitura e escrita ou somente de leitura. Toda referência à memória passa pela tabela de página para encontrar o número de quadro correto. Ao mesmo tempo que o endereço físico está sendo calculado, os bits de protcção podem ser verificados para checar se existem escritas sendo feitas em uma página somente de leitura. Uma tentativa de escrever em uma página somente de leitura causará uma exceção de hardware no sistema operacional (violação de proteção de memória).

Podemos facilmente expandir essa abordagem para fornecer um nível mais profundo de proteção. Podemos criar hardware para fornecer proteção somente de leitura, proteção de leitura c escrita ou proteção somente de execução. Ou, ao fornecer bits de proteção separados para cada tipo de acesso, podemos permitir qualquer combinação desses acessos; tentativas ilegais gerarão exceções para o sistema operacional.

Mais um bit é geralmente anexado a cada entrada na tabela de página: um bit válido-inválido. Quando esse bit é definido como "válido", seu valor indica que a página associada está no espaço de endereçamento lógico do processo e, portanto, é uma página legal (válida). Se o bit for definido como "invalido", o valor indica que a página não está no espaço de endereçamento lógico do processo. Endereços ilegais são bloqueados com o uso do bit válido-inválido. O sistema operacional define esse bit para cada página para habilitar ou desabilitar acessos a essa página. Por exemplo, em um sistema com um espaço de endereçamento de 14 bits (0 a 16.383), talvez tenhamos um programa que só deve usar endereços de 0 a 10.468. Considerando um tamanho de página de 2K, temos a situação indicada na Figura 9,11. Os endereços nas páginas 0, 1, 2, 3, 4 e 5 são mapeados normalmente através da tabela de página. Qualquer tentativa de gerar um endereço nas páginas 6 e7, no entanto, verificará que o bit válido-inválido está definido como inválido, e o computador vai causar uma exceção para o sistema operacional (referência de página inválida).

Observe que, como o programa se estende apenas até o endereço 10.468, qualquer referência além desse endereço é ilegal. No entanto, as referências à página 5 são classificadas como válidas, por isso os acessos aos endereços até 12.287 são válidos. Apenas os endereços de 12.288 a 16.383 são inválidos. Esse problema é resultado do tamanho de página de 2K e reflete a fragmentação interna da página.

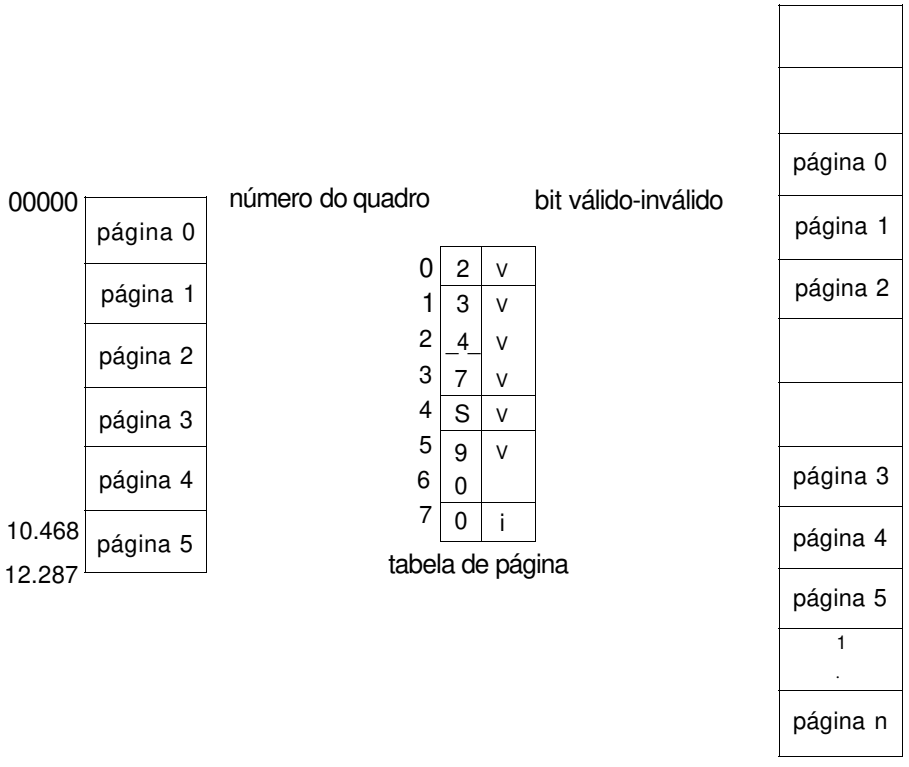


Figura 9.11 Bit válido (v) ou inválido (i) em uma tabela de página.

Raramente um processo utiliza toda sua faixa de endereços. Na verdade, muitos processos usam apenas uma pequena fração do espaço de endereçamento disponível. Seria um desperdício nesses casos criar uma tabela de página com entradas para cada página na faixa de endereços. A maior parte da tabela não seria utilizada, mas ocuparia um espaço de memória valioso. Alguns sistemas fornecem hardware, na forma de um registrador de tamanho da tabela de página (page-table length register - PTLR), para indicar o tamanho da tabela de página. Esse valor é verificado em relação a todo endereço lógico para checar se o endereço está na faixa válida para o processo. A falha desse teste gera uma exceção para o sistema operacional.

9.4.3 Paginação **multinível**

A maioria dos sistemas de computação modernos suporta um grande espaço de endereçamento lógico: (2^{i2}a $2^{é4}$). Em um ambiente como esses, a tabela de página cm si se torna excessivamente grande. Por exemplo, considere um sistema com um espaço de endereçamento lógico de 32 bits. Se o tamanho da página em tal sistema for 4K bytes (2^{12}), então uma tabela de página poderá consistir em até 1 milhão de entradas ($2^{12}/2^{12}$). Considerando que cada entrada consiste em 4 bytes, cada processo pode precisar de até 4 megaby tes de espaço de endereçamento físico somente para a tabela de página. Obviamente, não queremos alocar a tabela de página contiguamente na memória principal. Uma solução simples para este problema é dividir a tabela de página em partes menores. Existem várias formas de conseguir essa divisão.

Uma forma é usar um algoritmo de paginação de dois níveis, no qual a rabeia de página em si também é paginada (Figura 9.12). Lembre-se do nosso exemplo de máquina de ,Í2 bits com um tamanho de página de 4K bytes. Um endereço lógico *é* dividido em um número de página consistindo em 20 bits, e um deslocamento de página consistindo cm 12 bits. Como paginamos a tabela de página, o número da página é dividido em um número de 10 bits c um deslocamento de 10 bits. Assim, um endereço lógico torna-se:

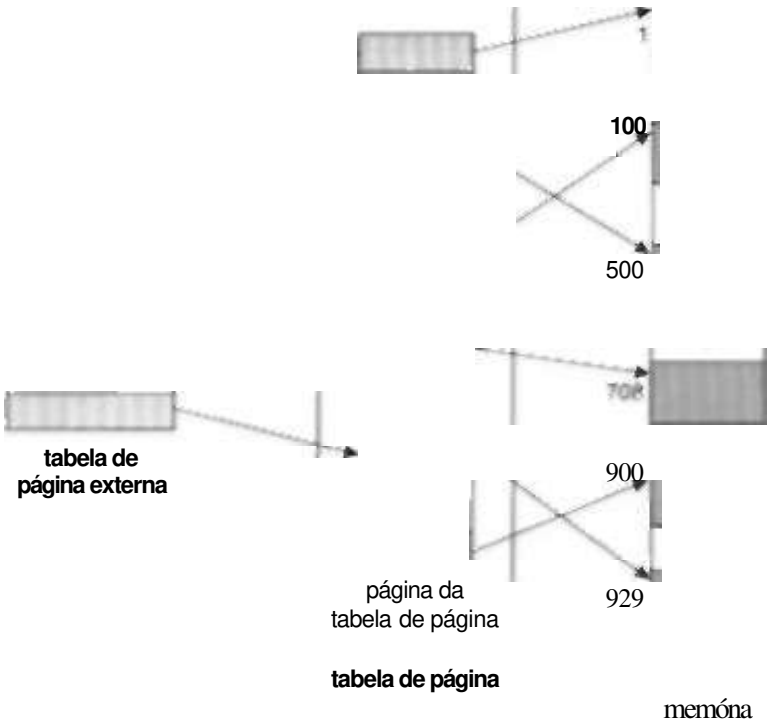


Figura 9.12 Esquema de paginação de dois níveis.

número da página		deslocamento na página
1 *	*Pi*	*d*
10	10	12

ondep, é um índice para a rabeia de página externa e/>, é o deslocamento dentro da página da tabela de página externa. O método de tradução de endereço para essa arquitetura é mostrado na Figura 9.13. A arquiterura VAX suporta a paginação de dois níveis. O VAX é uma máquina de 32 bits com tamanho de página de 512 bytes. O espaço de endereçamento lógico de um processo c dividido cm quatro seções iguais, cada qual consistindo cm 2^{10} bytes. Cada seção representa uma parte diferente do espaço de endereçamento lógico de um processo. Os primeiros 2 bits mais significativos do endereço lógico designam a seção apropriada. Os próximos 21 bits representam o número da página lógica daquela seção, e os 9 bits finais representam um deslocamento na página desejada. Particionando a tabela de página dessa forma, o sistema operacional poderá deixar partições sem uso até que um processo precise delas. Um endereço na arquitetura VAX é o seguinte:

seção	página	deslocamento
s	*P*	*d*
2	21	9

onde *s* designa o número da seção, *p* é um índice para a tabela de página e *d* é o deslocamento dentro da página.

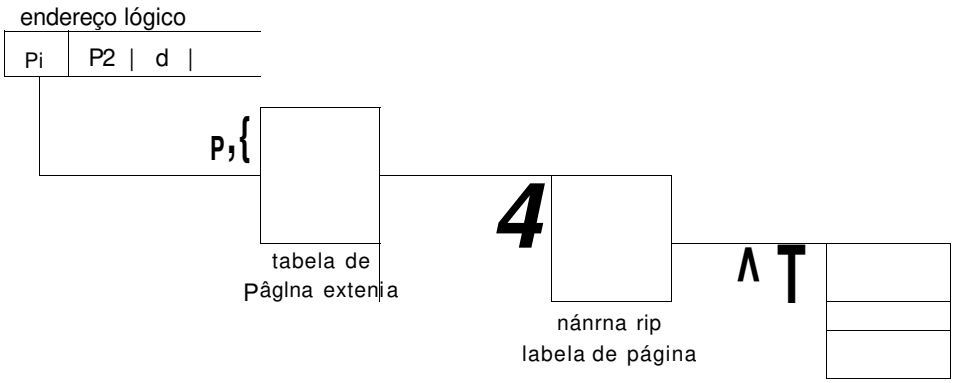


Figura 9.13 Tradução de endereço para arquitetura com paginação para dois níveis de 32 bits.

O tamanho de uma tabela de página de um nível para um processo VAX que utiliza uma seção ainda é 2^{Jt} bits ' 4 bytes por entrada = 8 megabytes. Para que o uso da memória principal seja reduzido ainda mais, o VAX pagina as tabelas de página do processo de usuário.

Para um sistema com espaço de endereçamento lógico de 64 bits, um esquema de paginação de dois níveis não é mais apropriado. Para ilustrar essa questão, vamos supor que o tamanho da página nesse sistema seja 4K bytes ($2^{1''}$). Nesse caso, a tabela de página consistirá em até $2^{<2}$ entradas. Se usarmos um esquema de paginação de dois níveis, as tabelas de página internas teriam convenientemente o tamanho de uma página contendo 2'° entradas de 4 bytes. Os endereços seriam assim:

página externa	pagina interna	deslocamento
P"	*P2*	*d*
42	10	12

A tabela de página externa consistirá em 2^{A1} entradas ou 2^{4m1} bytes. O método óbvio para evitar uma tabela tão grande é dividir a tabela de página externa em partes menores. Essa abordagem também é usada em alguns processadores de 32 bits para maior flexibilidade e eficiência.

Podemos dividir a tabela de página externa de várias formas. Podemos paginar a tabela de página externa, nos dando um esquema de paginação de três níveis. Vamos supor que a tabela de página externa seja formada por páginas de tamanho padrão (2^{10} entradas ou $2'^{2}$ bytes); um espaço de endereçamento de 64 bits ainda assim é grande:

2- **página externa**	página externa	**página interna**	**deslocamento**
py	P2	*pi*	*d*
32	10	10	12

A tabela de página externa ainda tem 2^{14} bytes.

A próxima etapa seria um esquema de paginação de quatro níveis, onde a tabela de página externa de 2" nível também é paginada. A arquitetura SPARC (com endereçamento de 32 bits) suporta um esquema de paginação de três níveis, enquanto a arquitetura Motorola 68030 de 32 bits suporta um esquema de paginação de quatro níveis.

Como a paginação multinível afeta o desempenho do sistema? Considerando que cada nível é armazenado como uma tabela separada na memória, converter um endereço lógico em um endereço físico pode fazer uso de quatro acessos à memória. Dessa forma, quintuplicamos a quantidade de tempo necessária para um acesso à memória! Armazenar em cache mais uma vez é vantajoso, no entanto, e o desempenho permanece razoável. Considerando uma taxa de acerto de cache de 98%, temos

tempo efetivo de acesso = 0,98 x 120 + 0,02 x 520
= 128 nanossegundos

Assim, mesmo com os níveis extras de pesquisa cm tabela, temos um aumento de apenas 28% no tempo de acesso à memória.

9.4.4 Tabela de página invertida

Geralmente, cada processo tem uma tabela de página associada a ele. A tabela de página contém uma entrada para cada página que o processo está usando (ou um slot para cada endereço virtual, independentemente da validade do mesmo). Essa representação em tabela é natural, já que os processos fazem referência às páginas através dos endereços virtuais das páginas. O sistema operacional deverá então traduzir essa referência em um endereço de memória física. Como a tabela é classificada por endereço virtual, o sistema operacional pode calcular onde na tabela está a entrada de endereço físico associada e usar esse valor dirctamente. Uma das desvantagens desse método é que cada tabela de página pode consistir em milhões de entradas. Essas tabelas consomem grande quantidade de memória física, que é necessária apenas para controlar como a outra memória física está sendo usada.

Para resolver esse problema, podemos usar uma tabela de página invertida. Uma tabela de página invertida tem uma entrada para cada página real (quadro) de memória. Cada entrada consiste no endereço virtual da página armazenada naquela posição de memória real, com informações sobre o processo que é proprietário da página. Assim, só existe uma tabela de página no sistema, e ela só tem uma entrada para cada página de memória física. A Figura 9.14 mostra a operação de uma tabela de página invertida. Compare-a com a Figura 9.6, que mostra uma tabela de página padrão em funcionamento. Os exemplos de sistemas usando esse algoritmo incluem o computador IBM System/38, o IBM RISC System 6000, IBM RT e as estações de trabalho Hewlett-Packard Spectrum.

Para ilustrar esse método, uma versão simplificada da implementação da tabela de página invertida utilizada no IBM RT será descrita a seguir. Cada endereço virtual no sistema consiste cm um trio

<id de processo, número de página, deslocamento>.

Cada entrada na tabela de página invertida é um par <id de processo, número de página >. Quando ocorre uma referencia de memória, parte do endereço virtual, que consiste em < id de processo, número de página>, é apresentado ao subsistema de memória. A tabela de página invertida é pesquisada para encontrar correspondências. Se uma correspondência for encontrada, digamos, na entrada /, o endereço físico </', dcslocamento> é gerado. Se não for encontrada correspondência, houve uma tentativa de acesso a endereço ilegal

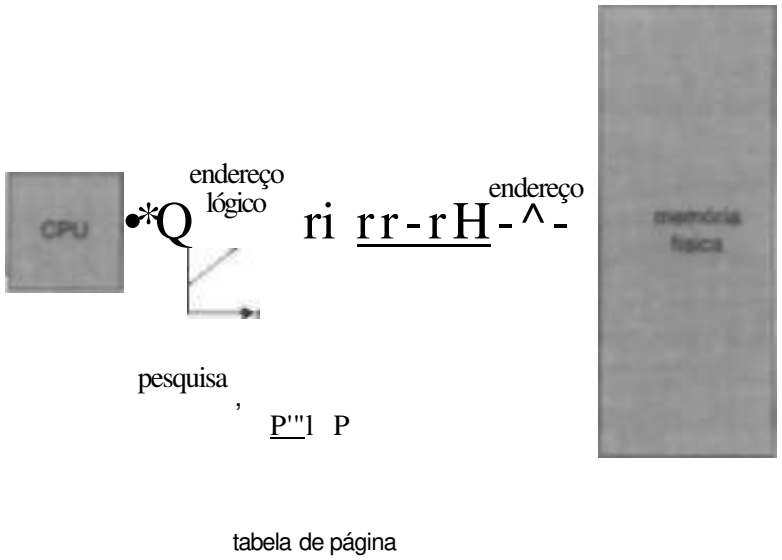


Figura 9.14 Tabela de página invertida.

Embora esse esquema reduza a quantidade de memória necessária para armazenar cada tabela de página, ele aumenta o tempo necessário para pesquisar a tabela quando ocorre uma referência de página. Como a tabela de página invertida é classificada por endereços físicos, mas as pesquisas são feitas com endereços virtuais, a tabela inteira talvez precise ser pesquisada para encontrar uma correspondência. Essa pesquisa pode demorar muito. Para aliviar o problema, usamos uma tabela de hashing para limitar a pesquisa para uma entrada ou, no máximo, algumas poucas entradas na tabela de página. K claro que cada acesso à tabela de hashing acrescenta uma referência de memória ao procedimento, por isso uma referência de memória virtual requer pelo menos duas leituras de memória real: uma para a entrada na tabela de hashing, outra para a tabela de página. Para melhorar o desempenho, utilizamos os registradores de memória associativa para manter entradas recentemente localizadas. Esses registradores são pesquisados em primeiro lugar, antes que a tabela de hashing seja consultada.

9.4.5 Páginas compartilhadas

Outra vantagem da paginação é a possibilidade de *compartilhar* código comum. Essa consideração é particularmente importante em um ambiente de tempo compartilhado. Considere um sistema que suporte 40 usuários, cada qual executando inrí editor de textos. Se o editor de textos consistir em 150K de código e 50K de espaço de dados, seriam necessários 8000 K para dar suporte a 40 usuários. Se o código for reentrante, no entanto, ele poderá ser compartilhado, como indicado na Figura 9.15. Aqui vemos um editor de três páginas (cada página com 50K de tamanho; o tamanho de página grande *é* usado para simplificar a figura) sendo compartilhado entre três processos. Cada processo tem sua própria página de dados.

O código reentrante (também chamado de código puro) é um código não-automodificável. Se o código for reentrante, ele nunca vai mudar durante a execução. Assim, dois ou mais processos podem executar o mesmo código ao mesmo tempo. Cada processo tem sua própria cópia de registradores e armazenamento de dados para manter os dados para a execução do processo. Obviamente os dados para dois processos diferentes irão variar para cada processo.

Apenas uma cópia do editor precisa ser mantida na memória física. Cada tabela de página do usuário é mapeada na mesma cópia física do editor, mas as páginas de dados são mapeadas em quadros diferentes. Portanto, para dar suporte a 40 usuários, precisamos apenas de uma cópia do editor (150K), mais 40 cópias dos 50K de espaço de dados por usuário. O espaço total necessário agora é de 2150K, em vez de X000K - uma economia significativa.

Outros programas com uso intensivo também podem ser compartilhados: compiladores, sistemas de janelas, sistemas de bancos de dados etc. Para ser compartilhável, o código precisa ser reentrante. A natureza somente de leitura do código compartilhado não deve ficar sob responsabilidade da correção do código; o sistema operacional deve implementar essa propriedade. O compartilhamento de memória entre os proces-

sos em um sistema é semelhante ao compartilhamento do espaço de endereçamento de uma tarefa por threads, conforme descrito no Capítulo 4.

Os sistemas que utilizam tabelas de página invertidas têm dificuldade em implementar memória compartilhada. A memória compartilhada é geralmente implementada como dois endereços virtuais que são mapeados cm um mesmo endereço físico. Esse método padrão não pode ser usado, no entanto, porque só existe uma entrada de página virtual para cada página física, por isso uma página física não pode conter os dois (ou mais) endereços virtuais compartilhados.

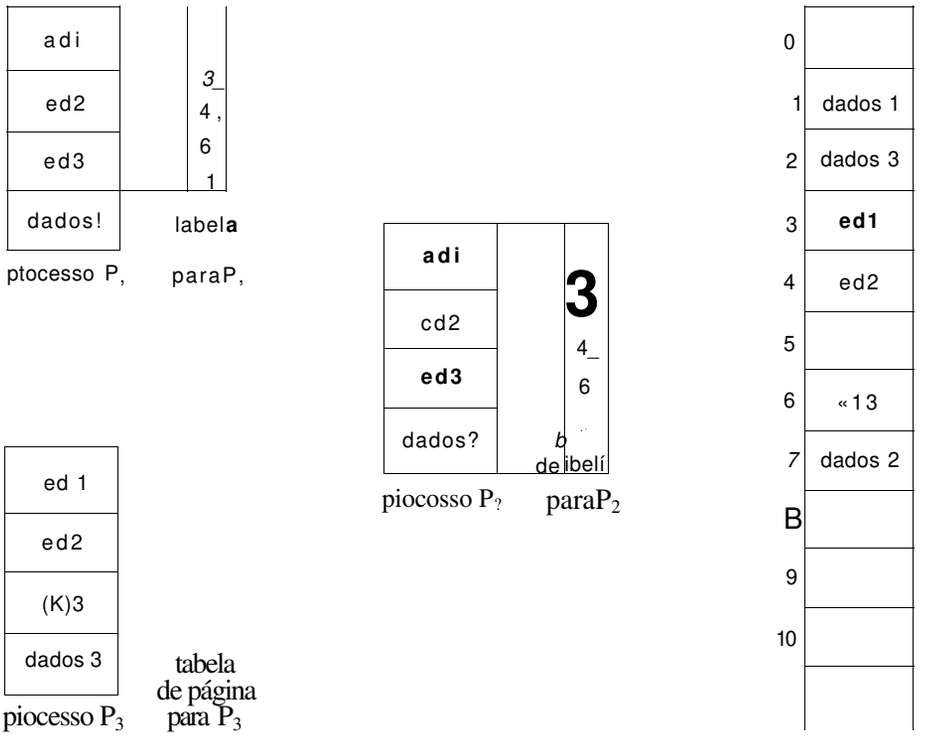


Figura 9.15 Compartilhamento de código em um ambiente de paginação.

9.5 • Segmentação

Um aspecto importante da gerência de memória que se tornou inevitável com a paginação é a separação da visão de usuário da memória e a memória física real. A visão de usuário da memória não é igual à memória física real. A visão de usuário é mapeada na memória física. O mapeamento permite a diferenciação entre memória lógica e física.

9.5.1 Método básico

Qual a visão de usuário da memória? Será que o usuário pensa que a memória £ um vetor linear de bytes, alguns contendo instruções e outros contendo dados, ou será que existe alguma outra visão de memória preferível? Existe consenso geral de que o usuário ou programador de um sistema não pensa na memória como um vetor linear de bytes. Km vez disso, o usuário prefere visualizar a memória como uma coleção de segmentos de tamanho variável, sem ordenação necessária entre os segmentos (Figura 9.16).

Considere o que você pensa sobre um programa quando 0 está escrevendo. Pensa nele como um programa principal com uma série de sub-rotinas, procedimentos, funções ou módulos. Também pode haver várias estruturas de dados: tabelas, vetores, pilhas, variáveis c assim por diante. Cada um desses módulos ou elementos de dados é referenciado por um nome. Você faz referência a "tabela de símbolos", "função *Sejrt"*$_y$ "programa principal", sem se importar com os endereços na memória que esses elementos ocupam. Você não está preocupado em saber se a tabela de símbolos é armazenada antes ou depois da função *Sqrt.* Cada um desses segmentos tem tamanho variável; o tamanho é definido intrinsecamente pelo propósito do segmento no programa. Os elementos de um segmento são identificados por seu deslocamento a partir do início do segmento: a primeira instrução do programa, a décima sétima entrada na tabela de símbolos, a quinta instrução da função *Sqrt* etc.

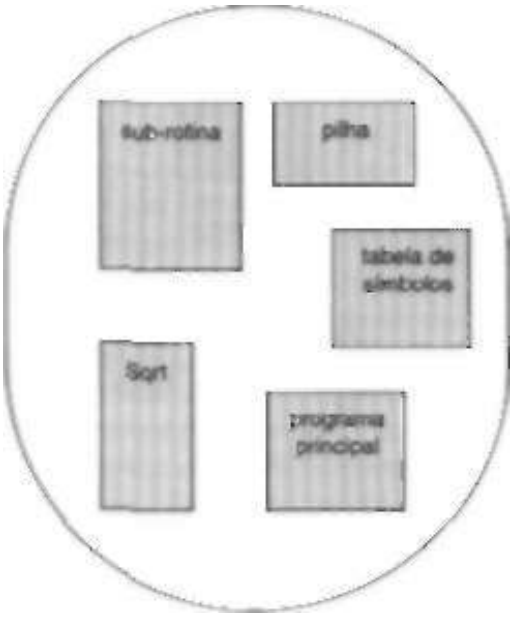


Figura 9.16 Visão de usuário de um programa.

A segmentação é um esquema de gerência de memória que oferece suporte a essa visão de usuário da memória. Um espaço de endereçamento lógico *é* uma coleção de segmentos. Cada segmento tem um nome e tamanho. Os endereços especificam o nome do segmento e o deslocamento dentro do segmento. O usuário especifica, portanto, cada endereço por duas quantidades: um nome de segmento e um deslocamento. (Compare esse esquema com o esquema de paginação, onde o usuário especificou apenas um único endereço que foi particionado pelo hardware em um número de página e um deslocamento, todos invisíveis para o programador.)

Para fins de simplicidade de implementação, os segmentos são numerados e referenciados por um número de segmento, em vez de um nome de segmento. Assim, um endereço lógico consiste em uma dupla:

< número do segmento, deslocamento».

Normalmente, o programa de usuário *é* compilado, e o compilador constrói automaticamente segmentos que refletem o programa de entrada. Um compilador Pascal pode criar segmentos separados para (1) as variáveis globais; (2) a pilha de chamada a procedimento, para armazenar parâmetros e endereços de retorno; (3) a parte de código de cada procedimento ou função c (4) as variáveis locais de cada procedimento ou função. Um compilador FORTRAN poderá criar um segmento separado para cada bloco comum. Vetores podem ser atribuídos a segmentos separados. O carregador reuniria todos esses segmentos e os atribuiria a números de segmento.

9.5.2 Hardware

Embora o usuário agora possa fazer referência a objetos no programa por um endereço bidimensional, a memória física cm si ainda é, evidentemente, uma sequência de bytes unidimensional. Dessa maneira, precisamos definir uma implementação para mapear endereços bidimensionais definidos pelo usuário em endereços físicos unidimensionais. Esse mapeamento *6* efetivado por uma tabela de segmentos. Cada entrada da tabela de segmentos possui uma base de segmento e um limite de segmento. A base de segmento contém o endereço físico de início no qual reside o segmento na memória, enquanto o limite de segmento especifica o tamanho do segmento.

O uso de uma tabela de segmentos é mostrado na Figura 9.17. Um endereço lógico consiste em duas partes: um número de segmento, 5, e um deslocamento nesse segmento, *d.* O número de segmento é usado como um índice para a tabela de segmentos. O deslocamento *d* do endereço lógico deve estar entre 0 e o limite de segmento. Se não estiver, é gerada uma exceção, para o sistema operacional (tentativa de endereçamento lógico após o final do segmento). Se esse deslocamento for legal, ele será adicionado à base do segmento para gerar o endereço na memória física do byte desejado, A tabela de segmentos é basicamente um vetor de pares de registradores base limite.

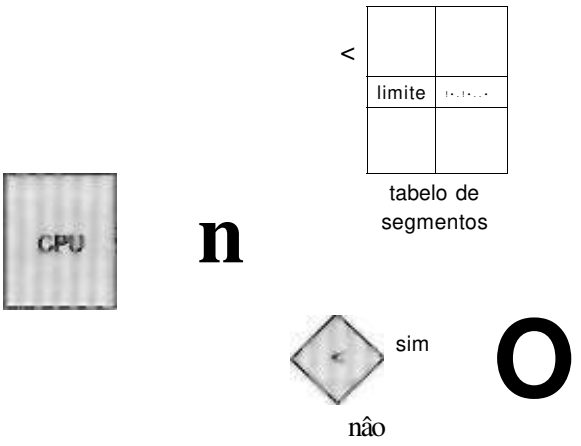


Figura 9.17 Hardware de segmentação.

Como exemplo, considere a situação apresentada na Figura 9.18. Temos cinco segmentos numerados de 0 a 4. Os segmentos são armazenados na memória física conforme indicado. A tabela de segmentos possui uma entrada separada para cada segmento, dando o endereço de início do segmento na memória física (a base) e o ramanho desse segmento (o limite). Por exemplo, o segmento 2 tem 400 bytes de comprimento, e começa na posição 4300. Assim, uma referência ao byte 53 do segmento 2 é mapeada na posição 4300 + 53 = 4353. Uma referência ao segmento 3, byte 852, é mapeada para 3200 (a base do segmento 3)+ 852 = 4052. Uma referência ao byte 1222 do segmento 0 resultaria em uma exceção ao sistema operacional, pois esse segmento só tem looo bytes de comprimento.

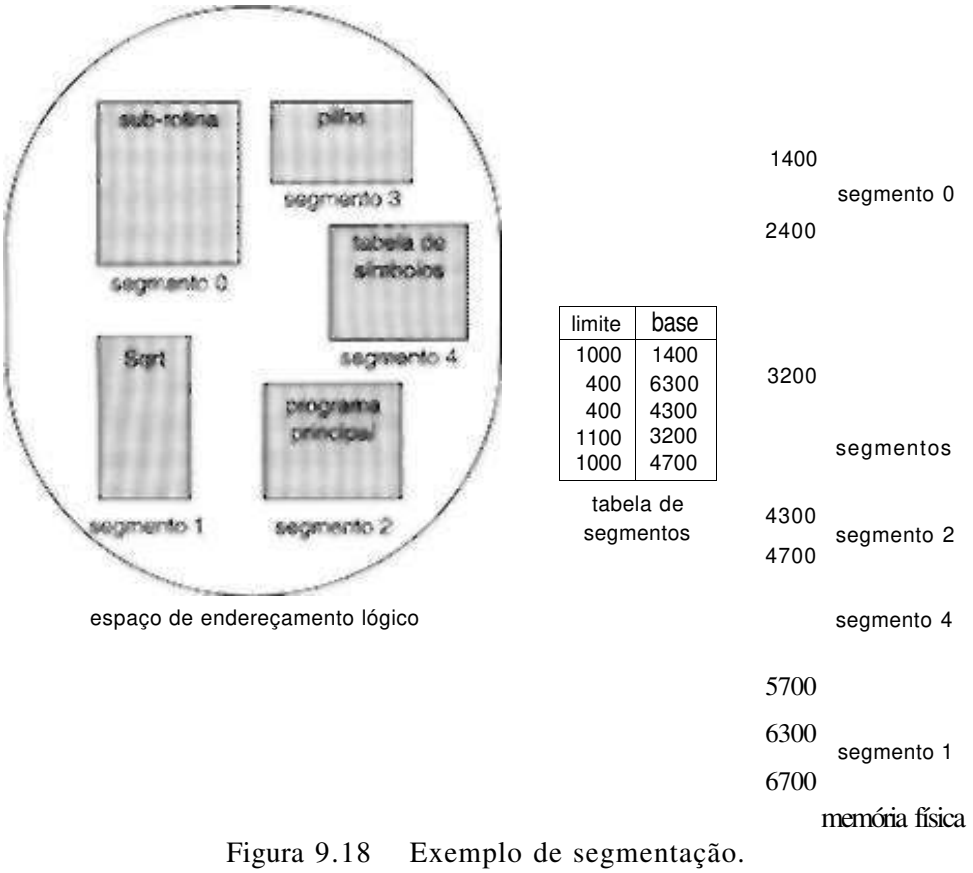


Figura 9.18 Exemplo de segmentação.

9.5.3 Proteção e compartilhamento

Uma vantagem específica da segmentação é a associação da proteção aos segmentos. Como os segmentos representam uma porção semanticamente definida do programa, é provável que todas as entradas no segmento sejam usadas da mesma forma. Portanto, temos alguns segmentos que são instruções, enquanto outros são dados. Em uma arquitetura moderna, as instruções não são automodificáveis, por isso os segmentos de instrução podem ser definidos como somente de leitura ou somente de execução. O hardware de mapeamento de memória verificará os bits de proteção associados com cada entrada na tabela de segmentos para evitar acessos ilegais à memória, tais como tentativas dèescrever em um segmento somente de leitura ou de usar um segmento somente de execução como dados. Ao cojocar um vetor no seu próprio segmento, o hardware de gerência de memória verificará automaticamente se os índices do vetor são legais e não ficam fora dos limites do vetor. Portanto, muitos erros comuns de programa serào detectados pelo hardware antes que possam causar danos graves.

Outra vantagem da segmentação envolve o *compartilhamento* de código ou dados. Cada processo tem uma tabela de segmentos associada a ele, que o dispatcher utiliza para definir a tabela de segmentos de hardware quando esse processo recebe a CPU. Os segmentos são compartilhados quando as entradas nas tabelas de segmentos de dois processos diferentes apontam para a mesma posição física (Figura 9.19).

O compartilhamento ocorre no nível do segmento. Assim, quaisquer informações poderão ser compartilhadas se forem definidas como um segmento. Vários segmentos podem ser compartilhados, portanto um programa composto por vários segmentos pode ser compartilhado.

Por exemplo, considere o uso de um editor de textos em um sistema de tempo compartilhado. Um editor completo pode ser bem grande, composto por muitos segmentos. Esses segmentos podem ser compartilhados entre todos os usuários, limitando a memória física necessária para dar suporte às tarefas de edição. Em vez de « cópias do editor, precisamos apenas de uma cópia. Para cada usuário, ainda é preciso ter segmentos únicos e separados para armazenar as variáveis locais. Esses segmentos, é claro, não seriam compartilhados.

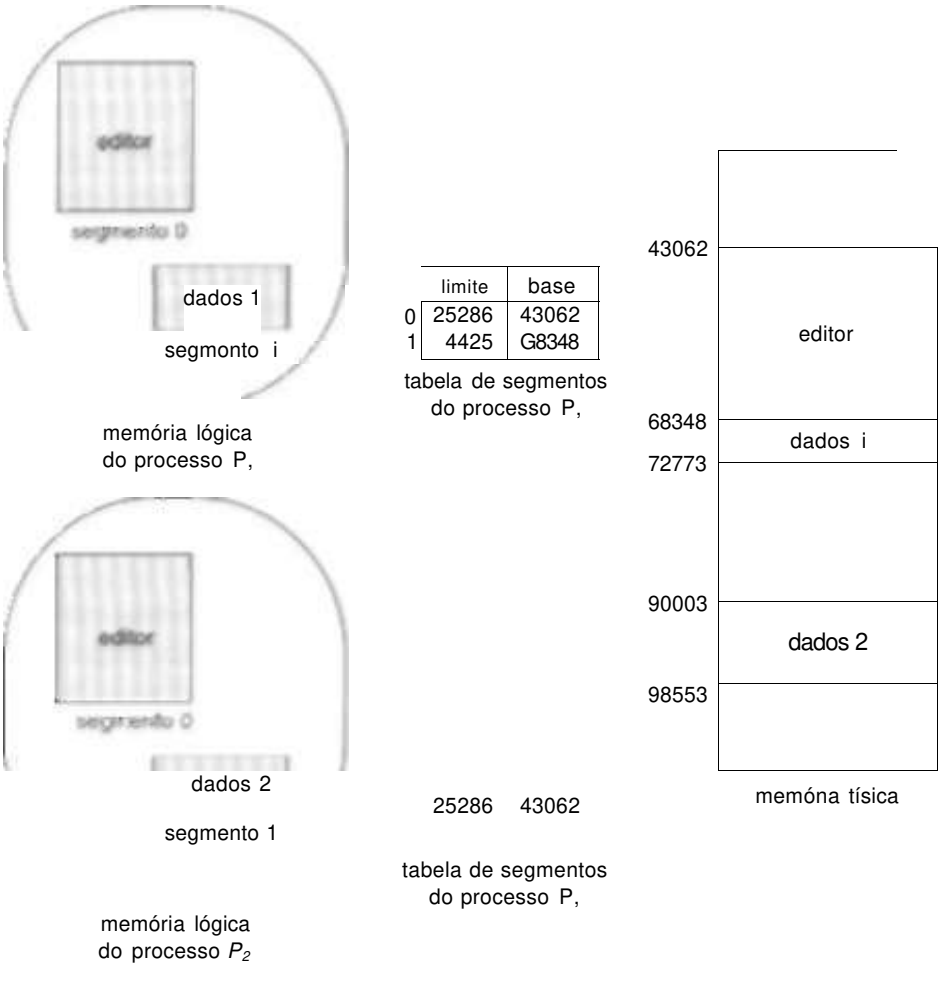


Figura 9.19 Compartilhamento de segmentos em um sistema de memória segmentada.

Também é possível compartilhar apenas partes dos programas. Por exemplo, os pacotes de sub-rotinas comuns podem ser compartilhados entre muitos usuários se forem definidos como segmentos compartilháveis somente de Leitura. Dois programas FORTRAN, por exemplo, podem usar a mesma sub-rotina *Sqrt,* mas apenas uma cópia física da rotina *Sqrt* seria necessária.

Embora esse compartilhamento aparente ser simples, existem considerações sutis. Os segmentos de código geralmente contêm referências a si mesmos. Por exemplo, um salto condicional normalmente tem um endereço de transferência. O endereço de transferência ê um número de segmento e deslocamento. O número de segmento do endereço de transferência será o número de segmento do segmento de código. Se tentarmos compartilhar esse segmento, todos os processos compartilhados deverão definir o segmento de código compartilhado como tendo o mesmo número de segmento.

Por exemplo, se quisermos compartilhar a rotina *Sqrt,* c um processo desejar torná-la o segmento 4 e outro, o segmento 17, como a rotina *Sqrt* fará referência a si mesma? Como existe apenas uma cópia física de *Sqrt,* ela deve fazer referência a si mesma de forma igual para os dois usuários - ela precisa ter um número de segmento único. A medida que o número de usuários que compartilham o segmento aumenta, também aumenta a dificuldade de encontrar um número de segmento aceitável.

Os segmentos de dados somente de leitura que não contêm ponteiros físicos podem ser compartilhados como números de segmento diferentes, assim como os segmentos de código que fazem referência a si mesmos não diretamente, mas apenas indiretamente. Por exemplo, desvios condicionais que especificam o endereço de desvio como um deslocamento do contador de programa atual ou relativo a um registrador que contém o número de segmento atual fariam com que o código evitasse a referência direta ao número de segmento amai.

9.5.4 Fragmentação

O escalonador de longo prazo deve encontrar e alocar memória para todos os segmentos de um programa de usuário. Essa situação e" semelhante à paginação exceto pelo fato de que os segmentos são de tamanho *variável* \ as páginas têm todas o mesmo tamanho. Assim, como ocorre com o esquema de partição de tamanho variável, a alocação de memória é um problema de alocação de memória dinâmica, geralmente resolvido com o algoritmo best-fit ou first-fit.

A segmentação pode então causar fragmentação externa, quando todos os blocos de memória livre são pequenos demais para acomodar um segmento. Nesse caso, o processo pode simplesmente ter de esperar até que mais memória (ou pelo menos um bloco de memória maior) se torne disponível, ou a compactação pode ser usada para criar um bloco de memória livre maior. Como a segmentação é por natureza um algoritmo de relocação dinâmica, podemos compactar memória sempre que desejarmos. Se o escalonador de CPU precisar esperar por um processo, devido a um problema de alocação de memória, poderá (ou não) percorrer a fila de CPU procurando um processo menor e de prioridade mais baixa para executar.

A fragmentação externa é realmente um problema sério para um esquema de segmentação? Será que o escalonamento de longo prazo com compactação pode ajudar? As respostas a essas perguntas dependem basicamente do tamanho médio do segmento. Por um lado, podemos definir cada processo como sendo um segmento. Essa abordagem se reduz ao esquema de partição de tamanho variável. No outro extremo, todo byte poderia ser colocado no seu próprio segmento e rclocado separadamente. Esse arranjo simplesmente elimina a fragmentação externa; no entanto, cada byte precisaria de um registrador de base para sua relocaçào, duplicando a utilização de memória! E claro que a próxima etapa lógica - segmentos pequenos, de tamanho fixo - è a paginação. Em geral, se o tamanho de segmento médio for pequeno, a fragmentação externa também será pequena. (Por analogia, considere colocar bagagem no porta-malas de um carro; ela nunca parece caber. No entanto, se você abrir as malas e colocar os itens individuais no porta-malas, tudo cabe direitinho.) Como os segmentos individuais são menores do que o processo como um todo, eles tendem a caber nos blocos de memória disponíveis.

9.6 • Segmentação com paginação

Tanto a paginação quanto a segmentação têm suas vantagens e desvantagens. Na verdade, dos dois processadores mais populares sendo utilizados no momento, a linha Motorola 68000 foi projetada com base em um

espaço de endereçamento contínuo, enquanto a família Intel 80X86 e Pentium baseia-se na segmentação. Ambos estão fundindo os modelos de memória cm direção a uma combinação de paginação e segmentação. É possível combinar esses métodos para melhorar cada um deles. Essa combinação é mais bem ilustrada pela arquitetura do Intel 386.

A versão de 32 bits do IBM OS/2 é um sistema operacional que executa sobre a arquitetura Intel 386 (e posterior). O 386 utiliza a segmentação com a paginação para a gerência de memória. O número máximo de segmentos por processo é 16K, e cada segmento pode ter até 4 gigabytes. O tamanho de página é 4K bytes. Não daremos uma descrição completa da estrutura de gerência de memória do 3 86 neste livro. Em vez disso, vamos apresentar as principais ideias.

O espaço de endereçamento lógico de um processo é dividido em duas partições. A primeira partição consiste em até 8K segmentos que são específicos (privados) desse processo. A segunda partição consiste em **até** 8K segmentos que são compartilhados entre todos os processos. As informações sobre a primeira partição são mantidas na Tabela Descritora Local (LDT - Local Dcscriptor Tablc), as informações sobre a segunda partição são mantidas na Tabela Descritora Global (GDT - Global Dcscriptor Tablc). Cada entrada nas tabelas LDT e GDT consiste cm 8 bytes, com informações detalhadas sobre determinado segmento, incluindo a posição base e o tamanho desse segmento.

O endereço lógico é um par (scletor, desloca mento), no qual o sclctor c um número de 16 bits:

5	8	/'
13	I	2

em que *$* designa o número do segmento, *g* indica se o segmento está na GDT ou LDT e *p* define a proteção. O deslocamento é um número de 32 bits que especifica a posição do byte (palavra) no segmento cm questão.

A máquina tem seis registradores de segmento, permitindo que seis segmentos sejam endereçados a **qualquer** momento por um processo. Ele tem seis registradores de microprograma de 8 bytes para armazenar os descritores correspondentes da LDT ou G D1. Esse cache evita que o 386 tenha de ler o descritor da memória para cada referencia de memória.

O endereço físico do 386 tem 32 bits de comprimento e é formado da seguinte maneira. O registrador de segmento aponta para a entrada apropriada na LDT ou GDT. As informações de base e limite sobre o segmento em questão são usadas para gerar um endereço linear. Km primeiro lugar, o limite é usado para verificar a validade do endereço. Se o endereço não for válido, uma falha de memória será gerada, resultando em uma execção ao sistema operacional. Sc for válido, o valor do deslocamento será somado ao valor hase, resultando em um endereço linear de 32 bits. Esse endereço é então traduzido em um endereço físico.

Como ressaltado anteriormente, cada segmento é paginado, e cada página tem 4K bytes. Uma tabela de página pode consistir em até 1 milhão de entradas. Como cada entrada consiste em 4 bytes, cada processo poderá precisar de até 4 megabytes de espaço de endereçamento físico só para a tabela de página. Evidentemente, não queremos alocar a tabela de página contiguamente memória principal. A solução adotada no 386 c usar um esquema de paginação de dois níveis. O endereço linear é dividido em um número de página que consiste em 20 bits, e um deslocamento de página que consiste cm 12 bits. Como paginamos a tabela de página, o número de página é dividido ainda em um ponteiro de diretório de página de 10 bits c um ponteiro de tabela de página de 10 bits. O endereço lógico é o seguinte:

número de página		deslocamento na página
Px	*Pi*	*d*
10	10	12

O esquema de tradução de endereço para essa arquitetura c semelhante ao esquema apresentado na Eigura 9.13. A tradução de endereço do Intel é mostrada cm mais detalhes na Figura 9.20. Para que a eficiência da utilização da memória física seja melhorada, as tabelas de página do Intel 386 podem ser jogados para o dis-

CO. Nesse caso, um bit válido é usado na entrada do diretório da página para indicar se a tabela para a qual a entrada está apontando está na memória ou no disco. Se a tabela estiver no disco, o sistema operacional poderá usar os outros 31 bits para especificar a posição no disco da tabela; a tabela poderá então ser levada para a memória sob demanda.

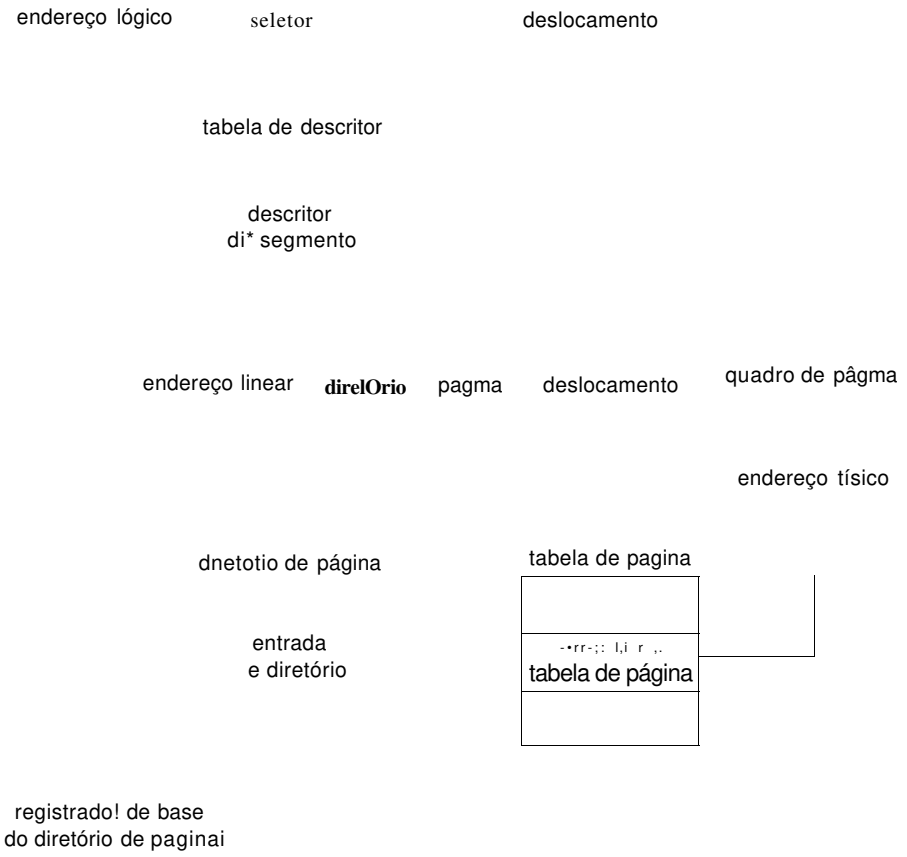


Figura 9.20 Tradução de endereço no Intel 80386.

9.7 • Resumo

Os algoritmos de gerência de memória para sistemas operacionais multiprogramados variam da abordagem de sistema monousuário simples a segmentação paginada. O maior fator determinante do método usado em um determinado sistema é o hardware existente. 'Iodo endereço de memória gerado pela CPU deve ser verificado em termos de legalidade e possivelmente ser mapeado em um endereço físico. A verificação não pode ser implementada (de modo eficiente) no software. Portanto, estamos limitados pelo hardware disponível.

Os algoritmos de gerência de memória discutidos (alocação contígua, paginação, segmentação e combinações de paginação e segmentação) diferem em muitos aspectos. A seguir estão considerações importantes a serem usadas na comparação de diferentes estratégias de gerência de memória:

- *Suporte de hardware:* Um registrador de base simples ou um par de registradores de base e de limite é suficiente para os esquemas de única e múltiplas partições, enquanto a paginação e a segmentação precisam de tabelas de mapeamento para definir o mapa de endereços.
- *Desempenho: A* medida que o algoritmo de gerência de memória se torna mais complexo, o tempo necessário para mapear um endereço lógico em um endereço físico aumenta. Para os sistemas simples, precisamos apenas comparar ou adicionar ao endereço lógico - operações que são rápidas. A paginação e a segmentação também podem ser rápidas, se a tabela for implementada em registradores rápidos. Sea tabela estiver na memória, no entanto, os acessos à memória por parte do usuário podem ter seu desempenho muito afetado. Um conjunto de registradores associativos pode reduzir essa degradação de desempenho para um nível aceitável.

- *iragmentaçâo:* Um sistema multiprogramado geralmente terá um desempenho mais eficiente se tiver um nível mais alto de multiprogramação. Para determinado conjunto de processos, podemos aumentar o nível de mui ti programação somente colocando mais processos na memória. Para realizar essa tarefa, devemos reduzir o desperdício ou a fragmentação de memória. Os sistemas com unidades de alocação de tamanho fixo, como o esquema de partição única e paginação, sofrem de fragmentação interna. Os sistemas com unidades de alocação de tamanho variável, tais como o esquema de múltiplas partições e segmentação, sofrem de fragmentação externa.
- *Relocação:* Uma solução ao problema de fragmentação externa é a compactação. A compactação implica mover um programa na memória sem que o programa perceba a diferença. Essa consideração requer que os endereços lógicos sejam relocados dinamicamente, no tempo de execução. Se os endereços forem relocados apenas no momento de carga, não será possível compactar a memória.
- *Stvapping;* Qualquer algoritmo pode ter a operação de swapping ou troca acrescentada a ele. Km intervalos determinados pelo sistema operacional, geralmente determinados pelas políticas de escalonamento de CPU, os processos são copiados da memória principal para o armazenamento auxiliar, e mais tarde são copiados de volta para a memória principal. Esse esquema permite que mais processos sejam executados do que caberiam na memória de uma só vez.
- *Compartilhamento:* Outro meio de aumentar o nível de multiprogramação é compartilhar código e dados entre diferentes usuários. Compartilhar geralmente requer que a paginação ou a segmentação sejam utilizadas, para fornecer pequenos pacotes de informação (páginas ou segmentos) que possam ser compartilhados. O compartilhamento é uma forma de executar muitos processos com uma quantidade limitada de memória, mas os programas e dados compartilhados devem ser projetados com cuidado.
- *Vroteção:* Se a paginação ou segmentação forem fornecidas, diferentes seções de um programa de usuário podem ser declaradas como somente de execução, somente de leitura ou de leitura-escrita. Essa restrição é necessária com código ou dados compartilhados e geralmente c útil em qualquer caso para fornecer verificações de tempo de execução para erros de programação comuns.

• Exercícios

9.1 Cite duas diferenças entre os endereços lógicos e físicos.

9.2 Explique a diferença entre a fragmentação interna e externa.

9.3 Descreva os seguintes algoritmos de alocação:

a. First fit
b. Best fit
c. Worstfit

9.4 Quando um processo é descarregado da memória *(roll OUt),* ele perde sua capacidade de usar a CPU (pelo menos por enquanto). Descreva outra situação na qual um processo perde sua capacidade de usar a CPU, mas onde o processo não é descarregado.

9.5 Considerando partições de memória de I00K, 500K, 200K, 300K e 600K (nessa ordem), como cada um dos algoritmos de first-fit, best-fit e worst-fit colocaria processos de 212K, 4I7K, 112Kc 426K (nessa ordem)? Que algoritmo faz uso mais eficiente da memória?

9.6 Considere um sistema no qual um programa pode ser separado em duas partes: código e dados. A CPU sabe se deseja uma instrução (busca de instrução) ou dados (busca ou armazenamento de dados). Portanto, dois pares de registradores de base limite são fornecidos: um para instruções e outro para dados. 0 par de registradores de base limite de instruções é automaticamente definido como somente de leitura, por isso os programas podem ser compartilhados entre vários usuários. Discuta as vantagens c desvantagens desse esquema.

9.7 Por que os tamanhos de página são sempre potências de 2?

9.8 Considere um espaço de endereçamento lógico de oito páginas de 1.024 palavras cada, mapeado em uma memória física de 32 quadros.

a. Quantos bits existem no endereço lógico?
b. Quantos bits existem no endereço físico?

9.9 Por que, em um sistema com paginação, um processo não pode acessar memória que não seja de sua propriedade? Como o sistema operacional poderia permitir acesso a outra memória? Por que deve fornecer acesso ou por que não deve?

9.10 Considere um sistema de paginação com a tabela de página armazenada na memória.

a. Se uma referência de memória leva 200 nanossegundos, quanto tempo leva uma referencia de memória paginada?
b. Se adicionarmos registradores associativos, e 75 % de todas as referências de tabela de página forem encontradas nos registradores associativos, qual é o tempo efetivo de referência de memória? (Considere que achar uma entrada na tabela de página nos registradores associativos ocorre em tempo zero, se a entrada estiver presente.)

9.11 Qual *é* o efeito de permitir que duas entradas em uma tabela de página apontem para o mesmo quadro de página na memória? Explique como usar esse efeito para diminuir o tempo necessário para copiar uma grande quantidade de memória de um local para outro. Qual seria 0 efeito em uma página de atualizar alguns bytes em outra?

9.12 Por que a segmentação e a paginação às vezes se combinam em um só esquema?

9.13 Descreva um mecanismo segundo o qual um segmento poderia pertencer ao espaço de endereçamento de dois processos diferentes.

9.14 Kxplique por que é mais fácil compartilhar um módulo reentrante utilizando a segmentação do que fazê-lo empregando a paginação pura.

9.15 Compartilhar segmentos entre processos sem exigir o mesmo número de segmento *é* possível em um sistema de segmentação com ligações dinâmicas.

a. Defina um sistema que permita a ligação estática c o compartilhamento de segmentos sem exigir que os números de segmento sejam iguais.
b. Descreva um esquema de paginação que permita que as páginas sejam compartilhadas sem exigir que os números de página sejam iguais.

9.16 Considere a seguinte tabela de segmentos:

Segmento	Base	Tamanho
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

Quais são os endereços físicos para os seguintes endereços lógicos?

a. 0,430
b. 1,10
c. 2,500
d. 3,400
e. **4,112**

9.17 Considere o esquema de tradução de endereços Intel apresentado na Figura 9.20.

a. Descreva todas as etapas seguidas pelo Intel 80386 na tradução de um endereço lógico em um endereço físico.

b. Quais são as vantagens para o sistema operacional de ter hardware que forneça hardware de tradução de memória tão complicado?

c. Rxiste alguma desvantagem desse sistema de tradução de endereços? Se houver, quais são? Se não, por que ele não é usado por todos os fabricantes?

9.18 No IBM/370, a proteção de memória é fornecida com o uso de *chaves.* Uma chave é um valor de 4 bits. Cada bloco de memória de 2K tem uma chave (a chave de memória) associada a ele. A CPU também tem uma chave (a chave de proteção) associada a ela. Uma operação de armazenamento só c permitida se as duas chaves forem iguais, ou se uma delas for igual a zero. Quais dos seguintes esquemas de gerência de memória podem ser usados com sucesso com esse hardware?

a. Máquina básica

b. Sistema monousuário

c. Multiprogramação com um número fixo de processos

d. Multiprogramação com um número variável de processos

e. Paginação

f. Segmentação

Notas bibliográficas

Aalocação dinâmica de memória foi discutida por Knuth [1973, Seção 2.5], que encontrou através de simulação resultados indicando que a estratégia de first-fit é geralmente superior à de besr-fit. Knuth [1973] discutiu a regra dos 50%.

O conceito de paginação pode ser creditado aos projetistas do sistema Atlas, que foi descrito por Kilburn e colegas 11961]e por Howarthe colegas [1961]. O conceiro de segmentação foi discutido pela primeira vez por Dennis [1965]. O GE 645, no qual o MULTICS foi implementado originalmente [Organick 1972], ofereceu suporte à segmentação paginada pela primeira vez.

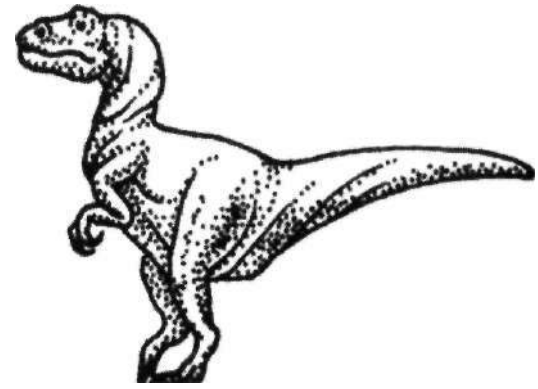
As tabelas de página invertidas foram discutidas em um artigo sobre o gerenciador de memória IBM RT por Chang e Mergen [1988].

Memórias cache, incluindo memória associativa, foram descritas e analisadas por Smith [1982]. Este trabalho também inclui uma bibliografia extensa sobre o assunto. Hennessy e Patterson [1996] discutiram os aspectos de hardware de TLBs, caches e MMUs.

A família de microprocessadores Motorola 68000 foi descrita em Motorola [1989a]. Informações sobre o hardware de paginação do 80386 podem ser encontradas em Intel l'1986].Tanenbaum [1992| também discutiu a paginação do Intel 80386. O hardware Intel 80486 também foi escrito cm uma publicação Intel [1989].

O gerência de memória para várias arquiteturas - como Pentium II, PowerPC e UltraSPARC - foi descrito porJacobeMudge[I998b|.

Capítulo 10



MEMORIA VIRTUAL

No Capítulo 9, discutimos várias estratégias de gerência de memória que são usadas nos sistemas de computação. Todas essas estratégias têm a mesma meta: manter muitos processos na memória ao mesmo tempo para permitir a multiprogramação. No entanto, eles tendem a exigir que o processo inteiro esteja na memória antes que possa ser executado.

A **memória virtual** é uma técnica que permite a execução de processos que podem não estar inteiramente na memória. A principal vantagem visível desse esquema é que os programas podem ser maiores do que a memória física. Além disso, ele abstrai a memória principal em um vetor extremamente grande e uniforme de armazenamento, separando a memória lógica conforme vista pelo usuário da memória física. Essa técnica libera os programadores da preocupação com as limitações de memória. A memória virtual não é fácil de implementar, entretanto, e pode diminuir em muito o desempenho se for utilizada sem o devido cuidado. Neste capítulo, vamos discutir a memória virtual na forma de paginação sob demanda e examinar sua complexidade e custo.

10.1 • Fundamentos

Os algoritmos de gerência de memória descritos no Capítulo 9 são necessários devido a um requisito básico: as instruções sendo executadas devem estar na memória física. A primeira abordagem para atender esse requisito é colocar todo o espaço de endereçamento lógico na memória física. Overlays e a carga dinâmica podem ajudar a diminuir essa restrição, mas geralmente requerem precauções especiais e trabalho extra por parte do programador. Essa restrição parece ser necessária e razoável, mas também é infeliz, já que limita o tamanho de um programa ao tamanho da memória física.

Na verdade, um exame dos programas reais nos mostra que, em muitos casos, o programa inteiro não é necessário. Por exemplo,

- Os programas em geral têm código para tratar condições de falta incomuns. Como esses faltas raramente ocorrem na prática (se é que ocorrem), o código quase nunca é executado.
- Vetores, listas e tabelas muitas vezes têm reservada mais memória do que na verdade precisam. Um vetor pode ser declarado 100 por 100 elementos, embora raramente seja maior do que 10 por 10 elementos. A tabela de símbolos de um montador pode ter espaço para 3000 símbolos, embora o programa médio tenha menos de 200 símbolos.
- Determinadas opções e recursos de um programa podem ser usados raramente. Por exemplo, as rotinas nos computadores do governo norte-americano que equilibram o orçamento só têm sido utilizadas recentemente.

Mesmo nos casos em que o programa inteiro é necessário, talvez não o seja de uma só vez (esse é o caso dos overlays, por exemplo).

A capacidade de executar um programa que só está parcialmente na memória pode conferir muitas vantagens:

- Um programa não ficaria mais limitado pela quantidade de memória física disponível. Os usuários poderiam escrever programas para um espaço de endereçamento virtual extremamente grande, simplificando a tarefa de programação.
- Como cada programa de usuário pode utilizar menos memória física, mais programas poderiam ser executados ao mesmo tempo, com um aumento correspondente na utilização de CPU e throughput, mas sem aumento no tempo de resposta ou de *turnaround*.
- Menos operações de I/O seriam necessárias para carregar ou fazer a movimentação de programas de usuário para a memória, de modo que cada programa de usuário poderia executar mais rapidamente.

Assim, executar um programa que não está inteiramente na memória beneficiaria o sistema e o usuário.

A memória virtual é a separação da memória lógica do usuário da memória física. Essa separação permite que uma memória virtual extremamente grande seja fornecida para os programadores quando apenas uma memória física menor esteja disponível (Figura 10.1). A memória virtual torna a tarefa de programação muito mais fácil, porque o programador não precisa se preocupar com a quantidade de memória física disponível ou com que tipo de código pode ser colocado em overlays; em vez disso, ele pode se concentrar no problema a ser programado. Nos sistemas que oferecem suporte à memória virtual, os overlays praticamente desapareceram.

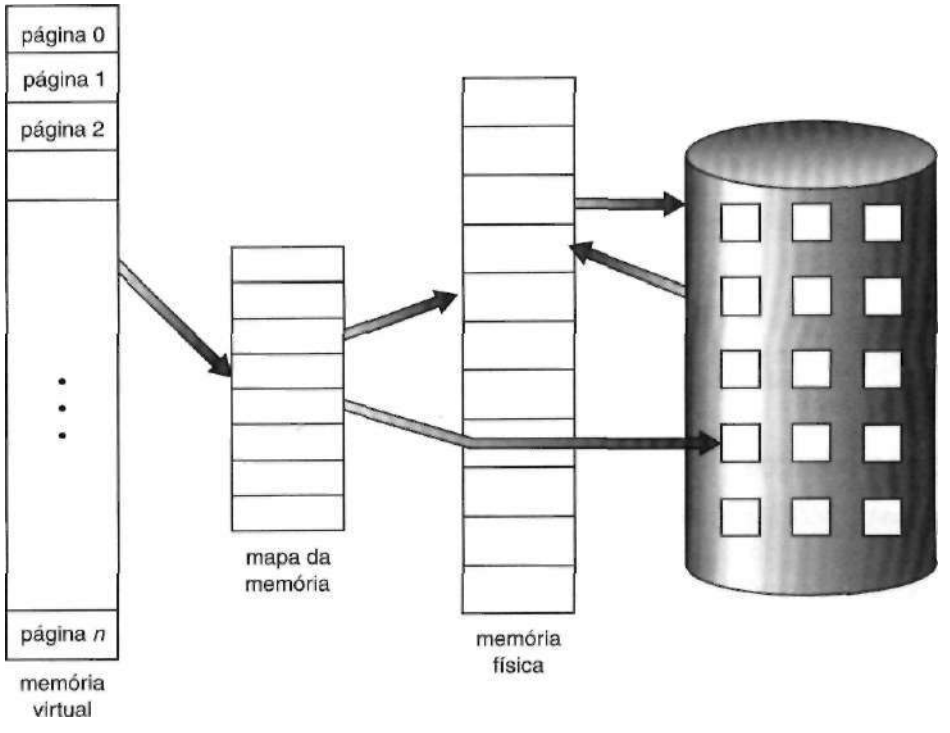


Figura 10.1 Diagrama mostrando que a memória virtual é maior do que a memória física.

A memória virtual é normalmente implementada por paginação sob demanda. Também pode ser implementada em um sistema de segmentação. Vários sistemas fornecem um esquema de segmentação paginada, no qual os segmentos são divididos em páginas. Dessa forma, a visão do usuário é a segmentação, mas o sistema operacional pode implementar essa visão com a paginação sob demanda. A segmentação sob demanda também pode ser usada para fornecer memória virtual. Os sistemas de computador da Burroughs utilizaram segmentação sob demanda. O sistema operacional IBM OS/2 também usa segmentação sob demanda. No entanto, os algoritmos de substituição de segmentos são mais complexos do que os algoritmos de substituição de páginas, porque os segmentos têm tamanho variável. Este texto não aborda a segmentação sob demanda; consulte as Notas bibliográficas para obter referências relevantes.

10.2 • Paginação sob demanda

Um sistema de paginação sob demanda é semelhante a um sistema de paginação com swapping, ou operações de troca (Figura 10.2). Os processos residem na memória secundária (que geralmente é um disco). Quando

queremos executar um processo, ele é carregado na memória *{swap in).* Entretanto, em vez de movimentar todo o processo para a memória, usamos um swapper "preguiçoso". Esse swapper nunca carrega uma página na memória, a menos que a página seja necessária. Já que agora estamos vendo um processo como uma sequência de páginas, em vez de como um grande espaço de endereçamento contíguo, o uso do termo *swap* (troca) é tecnicamente incorreto. Um *swapper* (trocador) manipula processos inteiros, enquanto um paginador *{pager)* está preocupado com as páginas individuais de um processo. Assim, usamos *paginador* em vez de *swapper* com relação à paginação sob demanda.

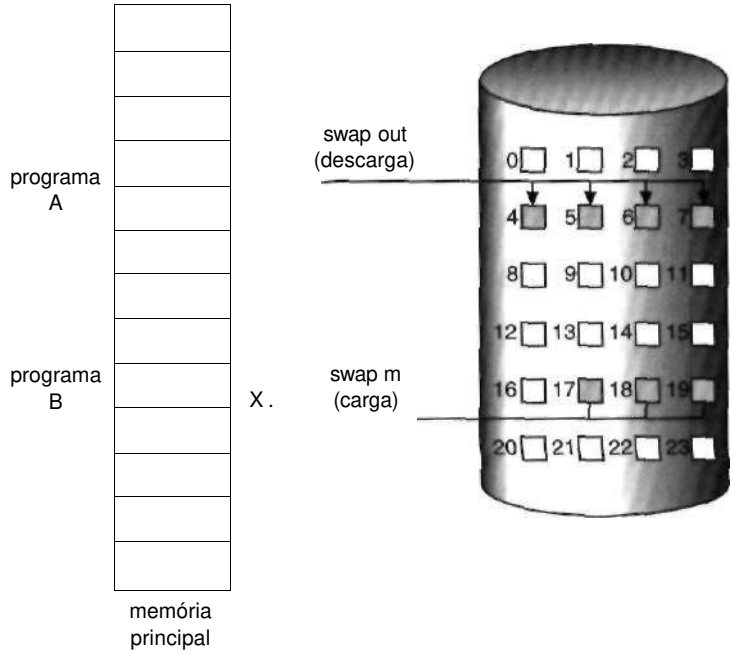


Figura **10.2** Transferência de uma memória paginada para espaço de disco contíguo.

10.2.1 Conceitos básicos

Quando um processo vai ser carregado na memória, o paginador adivinha que páginas serão usadas antes que o processo seja descarregado novamente. Em vez de carregar o processo inteiro, o paginador leva apenas as páginas necessárias para a memória. Assim, ele evita a leitura de páginas de memória que não serão utilizadas, diminuindo o tempo de troca e a quantidade de memória física necessária.

Com esse esquema, precisamos de alguma forma de suporte de hardware para fazer a distinção entre as páginas que estão na memória e as que estão no disco. O esquema do bit *válido-inválido* descrito na Seção 9.4.2 pode ser usado para esse fim. Desta vez, no entanto, quando o bit for definido para "válido", o valor indicará que a página associada é legal e está na memória. Se o bit for definido para "inválido", o valor indica que a página não é válida (ou seja, não está no espaço de endereçamento lógico do processo) ou é válida mas no momento está no disco. A entrada na tabela de página para uma página que é levada para a memória é definida normalmente, mas a entrada para uma página que não está na memória no momento é simplesmente marcada como inválida, ou contém o endereço da página no disco. Essa situação está representada na Figura 10.3.

Observe que marcar uma página como inválida não terá efeito se o processo nunca tentar acessar essa página. Portanto, se adivinhamos corretamente e carregamos todas e apenas as páginas que são de fato necessárias, o processo executará exatamente como se tivéssemos carregado todas as páginas. Enquanto um processo executa e acessa páginas que são **residentes na memória,** a execução continua normalmente.

Mas o que acontece se o processo tenta usar uma página que não foi carregada na memória? O acesso a uma página marcada como inválida causa uma interrupção de falta de página **(page-fault trap).** O hardware de paginação, ao traduzir o endereço através da tabela de página, observará que o bit inválido está ativo, gerando uma exceção para o sistema operacional. Essa exceção é resultado da falha do sistema operacional em levar a página desejada para a memória (na tentativa de minimizar o custo de transferência de disco e os re-

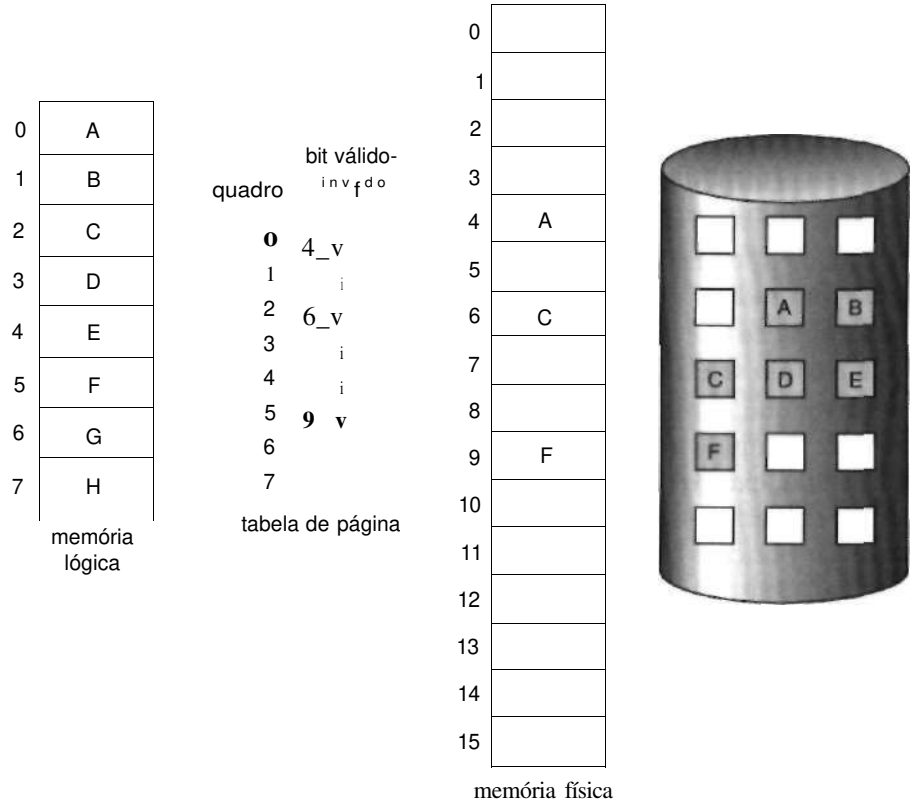


Figura 10.3 Tabela de página quando algumas páginas não estão na memória principal.

quisitos de memória), e não um falta de endereço inválido como resultado de uma tentativa de usar um endereço ilegal de memória (como um índice de vetor incorreto). Portanto, é preciso corrigir o problema. Eis o procedimento para tratar essa falta de página simples (Figura 10.4).

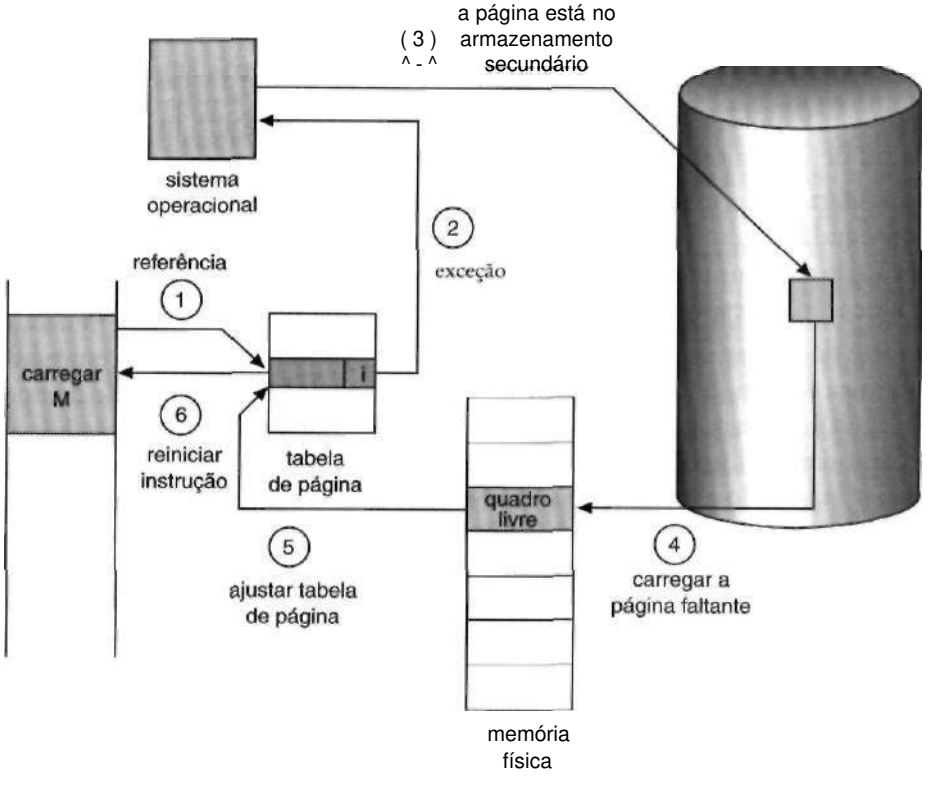


Figura 10.4 Etapas do tratamento de uma falta de página.

1. Verificamos uma tabela interna (geralmente mantida com o bloco de controle de processo) para esse processo, a fim de determinar se a referência foi um acesso à memória válido ou inválido.
2. Se a referência foi inválida, terminamos o processo. Se foi válida, mas ainda não carregamos essa página, agora podemos carregá-la.
3. Encontramos um quadro livre (escolhendo um da lista de quadros livres, por exemplo).
4. Escalonamos uma operação de disco para ler a página desejada no quadro recém-alocado.
5. Quando a leitura do disco estiver pronta, modificamos a tabela interna mantida com o processo e a tabela de página para indicar que a página agora está na memória.
6. Reiniciamos a instrução que foi interrompida pela exceção de endereço ilegal. O processo agora pode acessar a página como se ela sempre tivesse estado na memória.

É importante observar que, como salvamos o estado (registradores, código de condição, contador de instruções) do processo interrompido quando ocorre a falta de página, podemos reiniciar o processo *exatamente* no mesmo local e estado, exceto pelo fato de que a página desejada agora está na memória e é acessível. Dessa forma, é possível executar um processo, embora partes dele (ainda) não estejam na memória. Quando um processo tenta acessar posições que não estão na memória, o hardware gera exceções para o sistema operacional (falta de página). O sistema operacional lê a página desejada na memória e reinicia o processo como se a página sempre tivesse estado na memória.

No caso extremo, poderíamos começar a executar um processo *sem* nenhuma página na memória. Quando o sistema operacional define o ponteiro de instrução para a primeira instrução do processo, que está em uma página não-residente na memória, o processo imediatamente falha no acesso à página. Depois que essa página é levada para a memória, o processo continua a executar, causando faltas conforme adequado até que todas as páginas necessárias estejam na memória. A partir desse ponto, ele pode continuar a execução sem novas faltas. Esse esquema é a paginação sob demanda pura: uma página só será levada para a memória quando ela for necessária.

Teoricamente, alguns programas podem acessar várias novas páginas da memória com a execução de cada instrução (uma página para a instrução e muitas para os dados), possivelmente causando múltiplas faltas de página por instrução. Essa situação resultaria em um desempenho inaceitável do sistema. Felizmente, a análise dos processos em execução mostra que esse comportamento é muito pouco provável. Os programas tendem a ter uma localidade de referência, descrita na Seção 10.5.1, que resulta em desempenho razoável da paginação sob demanda.

O hardware para suportar a paginação sob demanda é igual ao hardware para paginação e swapping:

- *Tabela de página:* Essa tabela é capaz de marcar uma entrada como inválida através de um bit válido-inválido ou de um valor especial nos bits de proteção.
- *Memória secundária*: Essa memória mantém as páginas que não estão presentes na memória principal. A memória secundária geralmente é um disco de alta velocidade. É conhecida como o dispositivo de troca, e a seção de disco usada para esse fim é chamada de espaço de troca (swap) ou armazenamento auxiliar. A alocação do espaço de troca (swap) é discutida no Capítulo 13.

Além desse suporte de hardware, software considerável é necessário, como veremos adiante. Limitações adicionais de arquitetura devem ser impostas. Uma limitação crucial é a necessidade de poder reiniciar qualquer instrução após uma falta de página. Na maior parte dos casos, esse requisito é fácil de atender. Uma falta de página pode ocorrer em qualquer referência de memória. Se a falta de página ocorrer na busca de instrução, podemos reiniciar buscando a instrução novamente. Se a falta de página ocorrer enquanto estamos buscando um operando, devemos buscar e decodificar a instrução novamente e depois buscar o operando.

Como exemplo de pior caso, considere uma instrução de três endereços como ADD (somar) o conteúdo de A e B colocando o resultado em C. Essas são as etapas para executar a instrução:

1. Buscar e decodificar a instrução (ADD).
2. Buscar A.
3. Buscar B.
4. Somar A e B.
5. Armazenar a soma em C.

Se ocorrer uma falta quando tentarmos armazenar em C (porque C está em uma página que não está na memória no momento), teríamos de obter a página desejada, levá-la para a memória, corrigir a tabela de página e reiniciar a instrução. Esse reinício exigiria repetir o procedimento todo: buscar a instrução, decodificá-la, buscar os dois operandos e somá-los. No entanto, não há muito trabalho repetido (menos que uma instrução completa), e a repetição é necessária apenas quando ocorre uma falta de página.

A principal dificuldade ocorre quando uma instrução pode modificar várias posições de memória diferentes. Por exemplo, considere a instrução MVC (mover caractere) do IBM System 360/370, que pode mover até 256 bytes de uma posição para outra (possivelmente com sobreposição). Se um dos blocos (de origem ou destino) passar de um limite de página, poderá ocorrer uma falta de página depois que a movimentação tiver sido parcialmente efetuada. Além disso, se os blocos de origem e destino se sobrepuserem, o bloco que origem poderá ter sido modificado e, nesse caso, não será possível simplesmente reiniciar a instrução.

Esse problema pode ser resolvido de duas formas diferentes. Em uma solução, o microcódigo calcula e tenta acessar as duas pontas de ambos os blocos. Se houver possibilidade de ocorrência de uma falta de página, ela acontecerá nesta etapa, antes de qualquer modificação. O movimento pode ocorrer, já que sabemos que não haverá falta de página, uma vez que todas as páginas relevantes estão na memória. A outra solução utiliza registradores temporários para manter os valores de posições sobrescritas. Se houver uma falta de página, todos os valores antigos serão escritos novamente na memória antes que ocorra uma exceção. Essa ação restaura a memória ao seu estado antes da instrução ser iniciada, de modo que ela possa ser repetida.

Um problema semelhante de arquitetura ocorre em máquinas que utilizam modos de endereçamento especiais, incluindo os modos de autodecremento e auto-incremento (por exemplo, o PDP-11). Esses modos de endereçamento usam um registrador como um ponteiro e automaticamente decrementam ou incrementam o registrador, conforme indicado. O autodecremento decrementa automaticamente o registrador *antes* de usar seu conteúdo como endereço do operando; o auto-incremento automaticamente incrementa o registrador *depois* de usar seu conteúdo como endereço do operando. Assim, a instrução

```
MOV (R2)+,-(R3)
```

copia o conteúdo da posição apontada pelo registrador 2 para a posição apontada pelo registrador 3. O registrador 2 é incrementado (em 2 para uma palavra, já que PDP-11 é um computador com endereçamento por byte) depois de ser usado como ponteiro; o registrador 3 é decrementado (em 2) antes de ser usado como ponteiro. Agora considere o que acontecerá se houver uma falta quando estivermos tentando armazenar na posição apontada pelo registrador 3. Para reiniciar a instrução, devemos redefinir os dois registradores para os valores que tinham antes de começarmos a execução da instrução. Uma solução é criar um novo registrador especial de status para gravar o número do registrador e a quantidade modificada para cada registrador que é alterado durante a execução de uma instrução. Esse registrador de status permite que o sistema operacional *desfaça* os efeitos de uma instrução parcialmente executada que causa uma falta de página.

Esses não são absolutamente os únicos problemas de arquitetura resultantes do acréscimo da paginação a uma arquitetura existente para permitir a paginação sob demanda, mas ilustram algumas das dificuldades existentes. A paginação é acrescentada entre a CPU e a memória em um sistema de computação. Ela deve ser inteiramente transparente ao processo de usuário. Assim, as pessoas geralmente crêem que a paginação pode ser adicionada a qualquer sistema. Embora essa suposição seja verdadeira para um ambiente sem paginação sob demanda, no qual uma falta de página representa um erro fatal, ela não se aplica nos casos em que uma falta de página significa apenas que uma página adicional deve ser levada para a memória e processo retomado.

10.2.2 Desempenho da paginação sob demanda

A paginação sob demanda pode ter um efeito significativo no desempenho de um sistema de computador. Para ver o motivo, vamos calcular o tempo efetivo de acesso de uma memória paginada sob demanda. Na maior parte dos sistemas de computação, o tempo de acesso à memória, denotado *ma,* hoje em dia varia entre 10 e 200 nanossegundos. Desde que não haja falta de página, o tempo efetivo de acesso é igual ao tempo de acesso à memória. Se, no entanto, ocorrer uma falta de página, primeiro precisaremos ler a página relevante do disco e depois acessar a palavra desejada.

Seja p a probabilidade de ocorrência de uma falta de página ($0 < p < 1$). Seria esperado que p ficasse próximo a zero; ou seja, só haverá algumas poucas faltas de página. O tempo *efetivo* de acesso será então

$$\text{tempo efetivo de acesso} = (1 - p) \times ma + p \times \text{tempo de falta de página}$$

Para calcular o tempo efetivo de acesso, devemos saber quanto tempo é necessário para tratar uma falta de página. Uma falta de página causa a ocorrência da seguinte sequência de eventos:

1. Exceção para o sistema operacional.
2. Escrita dos registradores de usuário e estado do processo.
3. Determinação de que a interrupção era uma falta de página.
4. Verificação que a referência de página era legal e determinação da posição da página no disco.
5. Realização de uma leitura do disco em um quadro livre:
   a. Espera na fila por esse dispositivo até que o pedido de leitura seja atendido.
   b. Espera pelo tempo de busca e/ou latência do dispositivo.
   c. Início da transferência da página para um quadro livre.
6. Enquanto espera, alocação da CPU a algum outro usuário (escalonamento de CPU, opcional).
7. Interrupção do disco (operação de I/O concluída).
8. Escrita dos registradores e o estado de processo para o outro usuário (se a etapa 6 for executada).
9. Determinação de que a interrupção foi do disco.
10. Correção da tabela de página e das outras tabelas para mostrar que a página desejada agora está na memória.
11. Espera que a CPU seja alocada a esse processo novamente.
12. Restauração dos registradores de usuário, estado de processo e nova tabela de página, em seguida, retomada da instrução interrompida.

Nem todas essas etapas são necessárias em todos os casos. Por exemplo, estamos assumindo que, na etapa 6, a CPU está alocada a outro processo enquanto ocorre a operação de I/O. Esse arranjo permite que a multiprogramação mantenha a utilização de CPU, mas requer tempo adicional para retomar a rotina de serviço de falta de página quando a transferência de I/O estiver completa.

De qualquer modo, enfrentamos três importantes componentes do tempo de serviço de falta de página:

1. Atender a interrupção da falta de página.
2. Ler a página.
3. Reiniciar o processo.

A primeira e a terceira tarefas podem ser reduzidas, com código cuidadoso, a várias centenas de instruções. Essas tarefas podem levar de 1 a 100 microssegundos cada. O tempo de mudança de página, por outro lado, provavelmente ficará perto dos 24 milissegundos. Um disco rígido típico tem uma latência média de 8 milissegundos, uma busca de 15 milissegundos e um tempo de transferência de 1 milissegundo. Assim, o tempo total de paginação seria próximo dos 25 milissegundos, incluindo o tempo de hardware e software. Lembre-se também que estamos analisando apenas o tempo de serviço do dispositivo. Se uma fila de processos estiver esperando pelo dispositivo (outros processos que causaram falta de página), temos de somar o tempo de fila de dispositivo enquanto esperamos que o dispositivo de paginação fique livre para atender nosso pedido, aumentando ainda mais o tempo de troca.

Se usarmos um tempo médio de serviço de falta de página de 25 milissegundos e um tempo de acesso à memória de 100 nanossegundos, o tempo efetivo de acesso em nanossegundos será

$$\begin{aligned}\text{tempo efetivo de acesso} &= (1 - p) \times (100) + p \ (25 \text{ milissegundos}) \\ &\text{o } (1 - p) \times 100 + p \times 25.000.000 \\ &= 100 + 24.999.900 \times p.\end{aligned}$$

Vemos então que o tempo efetivo de acesso é diretamente proporcional à taxa de falta de página. Se um acesso em 1000 causar uma falta de página, o tempo efetivo de acesso será de 25 microssegundos. O computador reduziria sua velocidade em um fator de 250 devido à paginação sob demanda! Se quisermos menos do que 10% de degradação, precisamos de

$$110 > 100 + 25.000.000 \times p,$$
$$10 > 25.000.000 \times p,$$
$$p < 0{,}0000004.$$

Ou seja, para manter a redução de velocidade devido à paginação em um nível razoável, só podemos permitir que ocorra falta de página em menos de 1 acesso à memória a cada 2.500.000 acessos.

É importante manter a taxa de falta de página baixa em um sistema de paginação sob demanda. Caso contrário, o tempo efetivo de acesso aumenta, retardando a execução de processos de forma drástica.

Um aspecto adicional da paginação sob demanda é o manuseio e uso geral do espaço de swap. A operação de I/O no disco para o espaço de swap é geralmente mais rápida do que para o sistema de arquivos. É mais rápida porque o espaço de swap é alocado em blocos muito maiores e as pesquisas nos arquivos e os métodos de alocação indireta não são utilizados (consulte o Capítulo 13). Portanto, é possível que o sistema obtenha melhor throughput de paginação copiando uma imagem de arquivo inteira para o espaço de swap na inicialização do processo e, em seguida, realize a paginação sob demanda a partir do espaço de swap. Sistemas com um espaço de swap limitado podem empregar um esquema diferente quando arquivos binários são usados. As páginas sob demanda para esses arquivos são carregadas diretamente do sistema de arquivos. No entanto, quando a substituição de página é necessária, essas páginas podem simplesmente ser sobrescritas (porque nunca são modificadas) e lidas do sistema de arquivos novamente, se necessário. Outra opção é inicialmente demandar páginas do sistema de arquivos, mas gravá-las no espaço de swap à medida que são substituídas. Essa abordagem garante que somente as páginas necessárias sejam lidas do sistema de arquivos, mas que toda a paginação subsequente seja feita a partir do espaço de swap. Esse método parece ser bom; é usado no UNIX BSD.

10.3 • Substituição de página

Na nossa apresentação até agora, a taxa de falta de página não tem sido um problema grave, porque cada página tem no máximo uma falta, quando é referenciada pela primeira vez. Essa representação não é estritamente precisa. Considere que, se um processo de 10 páginas utiliza apenas metade delas, a paginação sob demanda economiza a I/O necessária para carregar as cinco páginas que nunca são usadas. Também podemos aumentar nosso grau de multiprogramação executando o dobro de processos. Assim, se tivéssemos 40 quadros, executaríamos oito processos, em vez dos quatro que seriam executados se cada um exigisse 10 quadros (cinco dos quais nunca usados).

Se aumentarmos o grau de multiprogramação, estaremos realizando sobrealocação de memória. Se executarmos seis processos, cada qual com 10 páginas de tamanho mas que, na verdade, só utiliza cinco páginas, teremos maior utilização de CPU e throughput, com 10 quadros não usados. No entanto, é possível que cada um desses processos, para determinado conjunto de dados, de repente tente usar todas as suas 10 páginas, resultando na necessidade de 60 quadros, quando apenas 40 estão disponíveis. Embora essa situação seja pouco provável, a probabilidade aumenta à medida que o nível de multiprogramação aumenta, de modo que o uso médio da memória é próximo à memória física disponível. (No nosso exemplo, por que parar em um nível seis de multiprogramação, quando podemos passar para um nível sete ou oito?)

Além disso, considere que a memória do sistema não é usada apenas para manter páginas de programa. Os buffers para as operações de I/O também consomem uma quantidade significativa de memória. Esse uso pode aumentar a carga dos algoritmos de substituição de memória. Decidir quanta memória alocar para I/O e para páginas de programas é um desafio e tanto. Alguns sistemas alocam uma percentagem fixa de memória para buffers de I/O, enquanto outros permitem que processos de usuário e o subsistema de I/O concorram pela memória do sistema todo.

A sobrealocação se manifesta da seguinte maneira. Enquanto um processo de usuário está executando, ocorre uma falta de página. O hardware gera uma exceção para o sistema operacional, que verifica suas tabelas internas para ver se essa falta de página é genuína e não um acesso ilegal à memória. O sistema operacional determina onde a página está residindo no disco, mas verifica que *não* existem quadros livres na lista de quadros livres: toda memória está sendo usada (Figura 10.5).

O sistema operacional tem várias opções nesse momento. Poderia terminar o processo de usuário. No entanto, a paginação por demanda é a tentativa do sistema operacional melhorar a utilização e o throughput do sistema de computador. Os usuários não precisam estar cientes de que seus processos estão executando em um sistema paginado - a paginação deveria ser logicamente transparente ao usuário. Por isso, essa opção não é a melhor.

Poderíamos descarregar um processo *(swap out),* liberando todos os seus quadros e reduzindo o nível de multiprogramação. Essa opção é boa em determinadas circunstâncias; ela será considerada na Seção 10.5. Aqui, discutimos uma possibilidade mais interessante: a **substituição de página.**

10.3.1 Esquema básico

A substituição de página assume a seguinte abordagem. Se nenhum quadro está livre, encontramos um que não está sendo usado no momento e o liberamos. Podemos liberar um quadro gravando seu conteúdo no espaço de swap e alterando a tabela de página (e todas as outras tabelas) para indicar que a página não está mais na memória (Figura 10.6). Agora podemos usar o quadro liberado para armazenar a página para a qual ocorreu a falta no processo. Modificamos a rotina de serviço de falta de página para incluir a substituição de página:

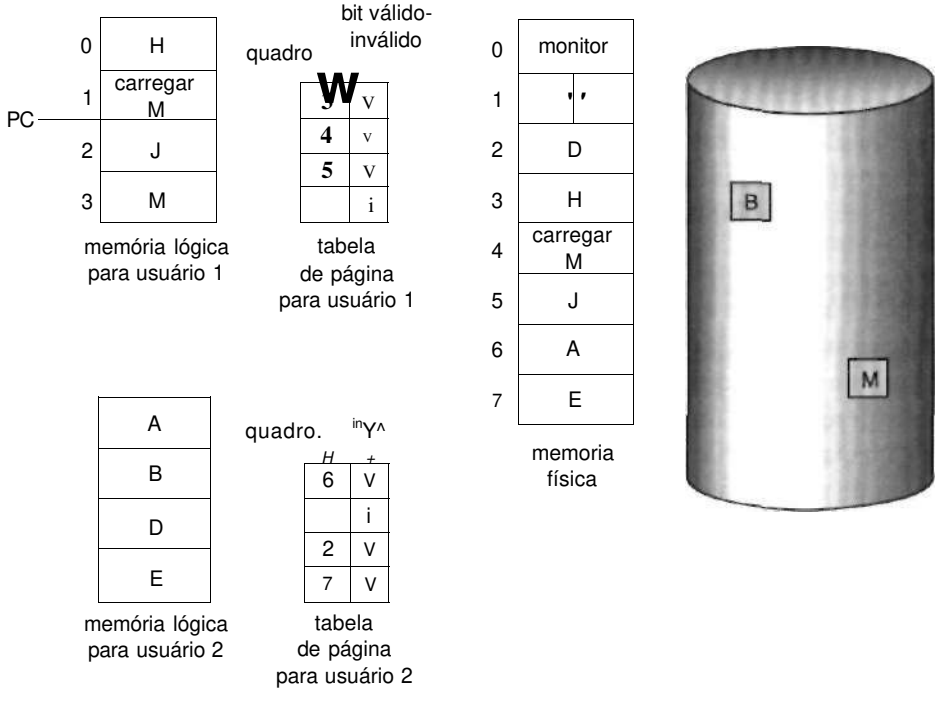


Figura 10.5 Necessidade de substituição de página.

1. Localizar a posição da página desejada no disco.
2. Encontrar um quadro livre:
   a. Se houver um quadro livre, use-o.
   b. Se não houver quadros livres, use um algoritmo de substituição de página para selecionar um quadro vítima.
   c. Gravar a página vítima no disco; alterar as tabelas de página e quadro de acordo.
3. Ler a página desejada no quadro (recém-) liberado; alterar as tabelas de página e quadro.
4. Retomar o processo de usuário.

Observe que, se nenhum quadro estiver livre, *duas* transferências de página uma descarga e uma carga são necessárias. Essa situação dobra o tempo de serviço de falta de página e aumenta o tempo de acesso efetivo proporcionalmente.

Podemos reduzir esse custo usando um **bit** de **modificação** *(bit dirty).* Cada página ou quadro pode ter um bit de modificação associado a ele no hardware. O bit de modificação para uma página é ativado pelo hardware sempre que qualquer palavra ou byte na página for alterado, indicando que a página foi modificada. Quando uma página é selecionada para substituição, examinamos seu bit de modificação. Se o bit estiver ativo, sabemos que a página foi modificada desde que foi lida do disco. Nesse caso, devemos gravar a página no disco. Se o bit de modificação não estiver ativo, no entanto, a página *não* foi modificada desde que foi carregada na memória. Portanto, se a cópia da página no disco não tiver sido sobrescrita (por outra página, por exemplo), podemos evitar gravar a página da memória no disco; ela já está lá. Essa técnica também se aplica a páginas somente de leitura (por exemplo, páginas de código binário). Tais páginas não podem ser modificadas; assim, podem ser descartadas quando desejado. Esse esquema pode reduzir significativamente o tempo necessário para atender uma falta de página, já que reduz o tempo de I/O pela metade *se* a página não foi modificada.

A substituição de página é essencial para a paginação sob demanda. Ela completa a separação entre a memória física e a memória lógica. Com esse mecanismo, os programadores poderão receber uma imensa memória virtual em uma pequena memória física. Com a paginação que não é sob demanda, os endereços de usuário são mapeados em endereços físicos, por isso os dois conjuntos de endereços podem ser diferentes. No entanto, todas as páginas de um processo ainda devem estar na memória física. Com a paginação sob demanda, o tamanho do espaço de endereçamento lógico não é mais limitado pela memória física. Se tivermos um processo com 20 páginas, poderemos executá-los em 10 quadros simplesmente usando a paginação sob demanda e usando um algoritmo de substituição para encontrar um quadro livre sempre que necessário. Se uma página que foi modificada será substituída, seu conteúdo vai ser copiado para o disco. Uma referência posterior àquela página causará uma falta de página. Nesse momento, a página será levada de volta à memória, substituindo talvez alguma outra página no processo.

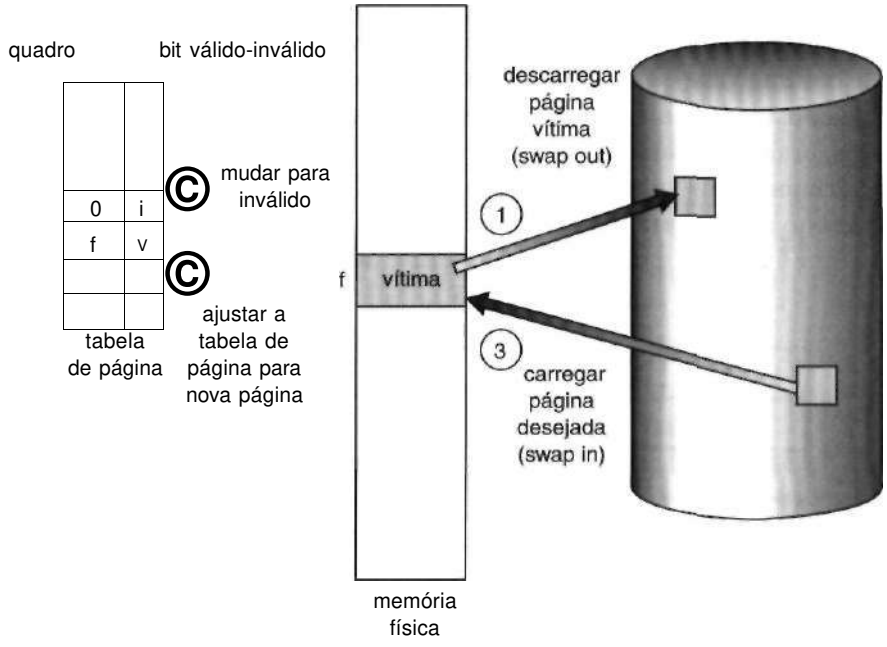


Figura **10.6** Substituição de página.

É preciso resolver dois principais problemas para implementar a paginação sob demanda: temos de desenvolver um **frame-allocation algorithm** (algoritmo de alocação de quadros) e um **page-replacement algorithm** (algoritmo de substituição de página). Se tivermos vários processos na memória, precisamos decidir quantos quadros serão alocados a cada processo. Além disso, quando for necessária uma substituição de página, precisamos selecionar os quadros a serem substituídos. Criar algoritmos apropriados para resolver esses problemas é uma tarefa importante, porque a operação de I/O de disco

é muito cara. Até mesmo ligeiras melhorias nos métodos de paginação sob demanda geram grandes ganhos no desempenho do sistema.

Existem muitos algoritmos distintos de substituição de página. Provavelmente todo sistema operacional tem seu próprio esquema de substituição exclusivo. Como selecionar um algoritmo de substituição específico? Em geral, queremos aquele com a menor taxa de falta de página.

Avaliamos um algoritmo executando-o sobre uma determinada série de referências de memória e calculando o número de falta de página. A série de referências de memória é chamada string de referência. Podemos gerar strings de referência artificialmente (por um gerador de números aleatórios, por exemplo) ou podemos monitorar um determinado sistema e registrar o endereço de cada referência de memória. A última opção gera um grande volume de dados (na ordem de 1 milhão de endereços por segundo). Para reduzir o número de dados, usamos dois fatos.

Em primeiro lugar, para determinado tamanho de página (e o tamanho de página é geralmente fixado pelo hardware ou sistema), precisamos considerar somente o número da página, em vez do endereço todo. Em segundo lugar, se tivermos uma referência a uma página *p,* qualquer referência *imediatamente* posterior à página *p* nunca causará uma falta de página. A página *p* estará na memória após a primeira referência; as referências imediatamente posteriores não terão falta.

Por exemplo, se monitorarmos determinado processo, poderemos registrar a seguinte sequência de endereços:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105,

que, a 100 bytes por página, é reduzido à seguinte string de referência

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1.

Para determinar o número de faltas de página para uma dada string de referência e um algoritmo de substituição de página, também precisamos conhecer o número de quadros de página disponíveis. Obviamente, à medida que o número de quadros disponíveis aumenta, diminui o número de falta de página. Para a string de referência considerada anteriormente, por exemplo, se tivéssemos três ou mais quadros, teríamos apenas três faltas, uma falta para a primeira referência a cada página. Por outro lado, com apenas um quadro disponível, teríamos uma substituição com cada referência, resultando em 11 faltas. Em geral, esperamos uma curva como a da Figura 10.7. A medida que o número de quadros aumenta, cai o número de faltas de página para um número mínimo. Obviamente, adicionar memória física aumenta o número de quadros.

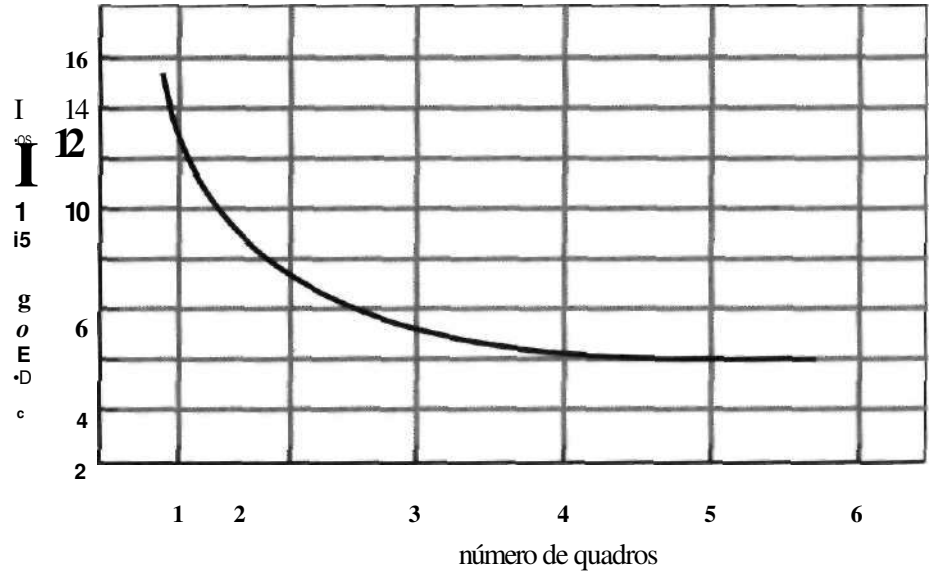


Figura 10.7 Gráfico de falta de página *versus* número de quadros.

Para ilustrar os algoritmos de substituição de página, iremos usar a string de referência

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

para uma memória com três quadros.

10.3.2 Substituição de página FIFO

O algoritmo de substituição de página mais simples é um algoritmo FIFO. Esse algoritmo associa a cada página o instante em que a página foi levada para a memória. Quando é preciso substituir uma página, a página mais antiga é escolhida. Observe que não é estritamente necessário registrar o instante em que a página é trazida. Podemos criar uma fila FIFO para manter todas as páginas na memória. Substituímos a página que estiver no início da fila. Quando uma página é movida para a memória, ela é inserida no final da fila.

Para nosso string de referência do exemplo, os três primeiros quadros estão inicialmente vazios. As primeiras três referências (7, 0,1) causam faltas de página e são carregadas nos quadros vazios. A próxima referência (2) substitui a página 7, porque a página 7 foi carregada em primeiro lugar. Como 0 é a próxima referência e já está na memória, não existe falta para essa referência. A primeira referência a 3 resulta na substituição da página 0, já que ela foi a primeira das três páginas na memória (0,1 e 2) a ser carregada. Devido **a** essa substituição, a próxima referência, a 0, falhará. A página 1 é substituída pela página 0. Esse processo continua como indicado na Figura 10.8. Toda vez que ocorre uma falta, mostramos que páginas estão nos nossos três quadros. Existem ao todo 15 faltas.

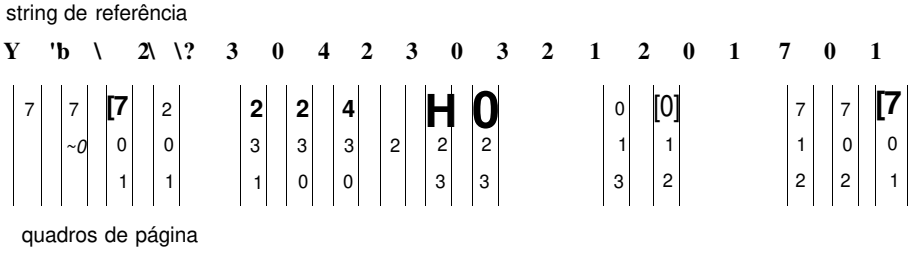


Figura 10.8 Algoritmo de substituição de página FIFO.

O algoritmo de substituição de página FIFO é fácil de entender e programar. No entanto, seu desempenho nem sempre é bom. A página substituída pode ser um módulo de inicialização que foi usado há muito tempo e não é mais necessário. Por outro lado, ela pode conter uma variável muito usada que foi inicializada cedo e está em uso constante.

Observe que, mesmo se optarmos por selecionar uma página que está em uso ativo, tudo funciona corretamente. Depois de descarregarmos uma página ativa para trazer uma nova página, ocorre uma falta quase que imediatamente para recuperar **a** página ativa. Alguma outra página precisará ser substituída para trazer a página ativa de volta para a memória. Assim, uma opção de substituição ruim aumenta a taxa de falta de página e diminui velocidade de execução de um processo, mas não causa execução incorreta.

Para ilustrar os problemas possíveis com um algoritmo de substituição de página FIFO, considere o string de referência

1,2,3,4,1,2,5,1,2,3,4,5.

A Figura 10.9 mostra a curva de falta de página *versus* o número de quadros disponíveis. Observe que o número de faltas para quatro quadros (10) é *maior* do que o número de faltas para três quadros (nove)! Esse resultado tão inesperado é chamado de **anomalia de Belady:** para alguns algoritmos de substituição de página, **a** taxa de falta de página pode *aumentar* à medida que o número de quadros alocados aumenta. Poderíamos esperar que conferir mais memória a um processo aumentaria o seu desempenho. Em pesquisas iniciais, os pesquisadores verificaram que essa suposição nem sempre era verdadeira. A anomalia de Belady foi descoberta como resultado da pesquisa.

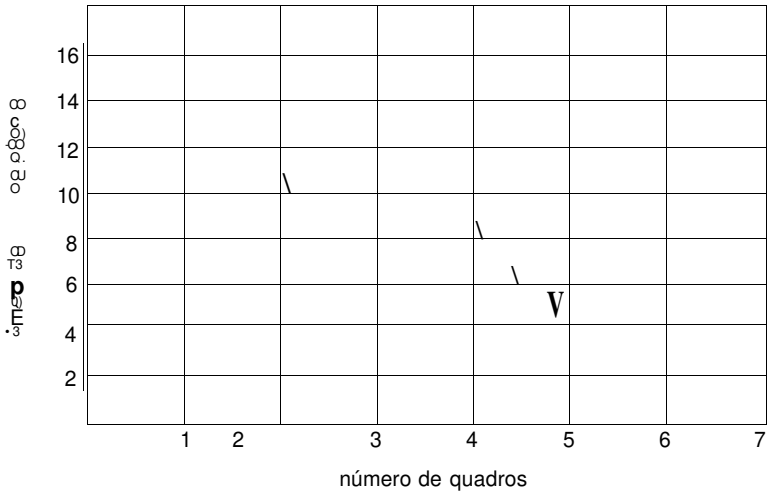


Figura 10.9 Curva de falta de página para substituição FIFO em uma string de referência.

10.3.3 Algoritmo de substituição ótimo

Um resultado da descoberta da anomalia de Belady foi a pesquisa por um algoritmo de substituição ótimo. Um algoritmo de substituição de página ótimo tem a menor taxa de falta de página de todos os algoritmos e nunca sofrerá da anomalia de Belady. Ele existe e foi chamado de OPT ou MIN. Sua estrutura é a seguinte:

Substituir a página que não será usada pelo período mais longo.

O uso desse algoritmo de substituição de página garante a menor taxa de falta de página possível para um número fixo de quadros.

Por exemplo, em nossa amostra de string de referência, o algoritmo de substituição de página ótimo geraria nove faltas de página, conforme indicado na Figura 10.10. As três primeiras referências causam faltas que preencherão os três quadros vazios. A referência à página 2 substitui a página 7, porque 7 ainda não estará sendo usada até a referência 18, enquanto a página 0 será usada em 5, e a página 1 em 14. A referência à página 3 substitui a página 1, já que a página 1 será a última das três páginas na memória a ser referenciada novamente. Com apenas nove faltas de página, a substituição ótima é bem melhor do que um algoritmo FIFO, que tinha 15 faltas. (Se ignorarmos os três primeiros, que todos os algoritmos apresentarão, a substituição ótima é duas vezes melhor do que a substituição FIFO.) Na verdade, nenhum algoritmo de substituição pode processar esse string de referência em três quadros com menos de nove faltas.

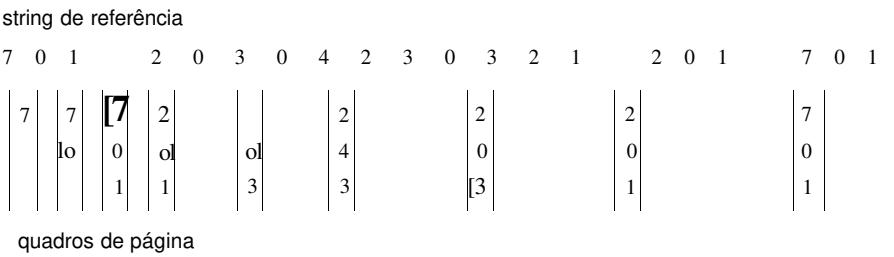


Figura 10.10 Algoritmo de substituição de página ótimo.

Infelizmente, o algoritmo de referência de página ótimo é difícil de implementar, porque requer conhecimento futuro do string de referência. (Encontramos uma situação semelhante com o algoritmo de escalonamento de CPU SJF na Seção 6.3.2.) Como resultado, o algoritmo ótimo é usado principalmente para estudos comparativos. Por exemplo, pode ser útil saber que, embora um novo algoritmo não seja ótimo, ele está a 12,3% do algoritmo ótimo na pior das hipóteses e dentro de 4,7% na média.

10.3.4 Algoritmo de substituição LRU

Se o algoritmo ótimo não for viável, talvez uma aproximação desse algoritmo seja possível. A principal distinção entre os algoritmos FIFO e OPT (além de olhar para a frente ou para trás no tempo) é que o algoritmo FIFO utiliza a data em que a página foi levada para a memória; o algoritmo OPT usa a data em que uma página deverá ser *usada.* Se usarmos o passado recente como uma aproximação do futuro próximo, substituiremos a página que *não foi usada* por mais tempo (Figura 10.11). Essa abordagem é o algoritmo **least** recently **used** (LRU).

A substituição LRU associa a cada página a data que essa página foi usada pela última vez. Quando uma página precisa ser substituída, o algoritmo LRU seleciona a página que não foi usada pelo maior período de tempo. Essa estratégia é o algoritmo de substituição de página ótimo olhando para trás no tempo, ao invés de para a frente. (Estranhamente, se S^R for o inverso de um string de referência 5, a taxa de falta de página para o algoritmo OPT em *S é* igual à taxa de falta de página para o algoritmo OPT em S^R. Da mesma forma, a taxa de falta de página para o algoritmo LRU em 5 é igual à taxa de falta de página para o algoritmo LRU em S^R.*)*

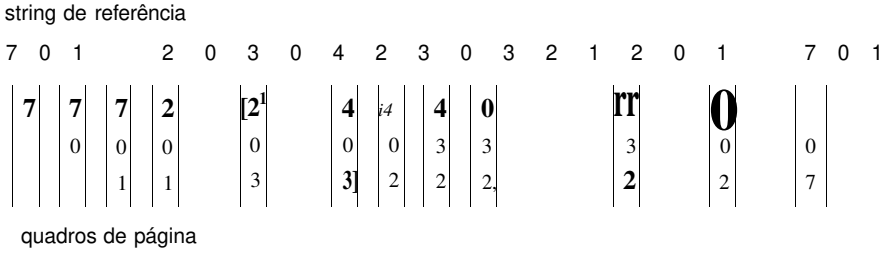


Figura 10.11 Algoritmo de substituição de página LRU.

O resultado da aplicação do algoritmo de substituição LRU ao nosso exemplo de string de referência está representado na Figura 10.11. O algoritmo LRU produz 12 faltas. Observe que os cinco primeiros faltas são iguais ao algoritmo de substituição ótimo. Quando ocorre **a** referência à página 4, no entanto, a substituição LRU verifica que, dos três quadros na memória, a página 2 foi usada menos recentemente. A página mais recentemente usada é a página 0 e logo antes dela a página 3 foi usada. Assim, o algoritmo LRU substitui a página 2, sem saber que a página 2 está prestes a ser usada. Quando há a falta para a página 2, o algoritmo LRU substitui a página 3, já que, das três primeiras páginas na memória {0, 3,4}, a página 3 é a que foi usada menos recentemente. Apesar desses problemas, a substituição LRU com 12 faltas ainda é melhor do que a substituição FIFO com 15 faltas.

A política LRU muitas vezes é usada como um algoritmo de substituição de página, sendo considerada boa. O principal problema é *como* implementar a substituição LRU. Um algoritmo de substituição de página LRU pode exigir ajuda substancial do hardware. O problema é determinar uma ordem para os quadros definida pela data da última utilização. Duas implementações são viáveis:

- *Contadores:* No caso mais simples, associamos a cada entrada na tabela de página um campo data-de-uso e adicionamos à CPU um relógio lógico ou contador. O relógio é incrementado para cada referência de memória. Sempre que uma referência de página é feita, o conteúdo do registrador do relógio é copiado para o campo de data-de-uso na entrada na tabela de página para essa página. Dessa forma, sempre temos a "data" da última referência a cada página. Substituímos a página pelo menor valor de data. Esse esquema requer uma busca na tabela de página para encontrar a página LRU, e uma escrita na memória (para o campo de data-de-uso na tabela de página) para cada acesso à memória. As datas também devem ser mantidas quando as tabelas de página são alteradas (devido ao escalonamento de CPU). O overflow do relógio deve ser considerado.
- *Pilha:* Outra abordagem à implementação da substituição LRU é manter uma **pilha** de números de página. Sempre que uma página é referenciada, ela é removida da pilha e colocada no topo. Dessa forma, o topo da pilha sempre será a página mais recentemente usada e a base da pilha é a página LRU (Figura 10.12). Como as entradas devem ser removidas do meio da pilha, ela é mais bem implementada por uma lista duplamente encadeada, com um ponteiro de início e fim. Remover uma página e colocá-la

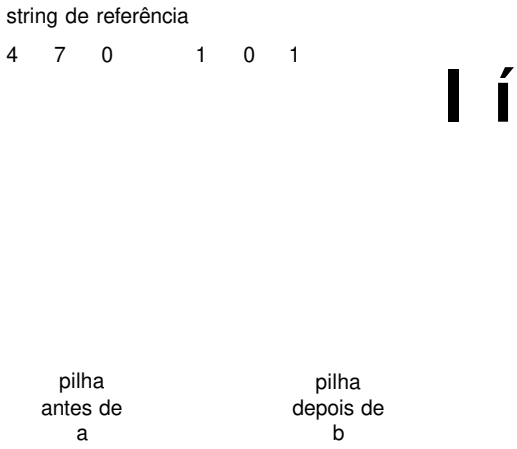


Figura 10.12 Uso de uma pilha para registrar as referências de página mais recentes.

no topo da pilha requer mudar seis ponteiros no pior dos casos. Cada atualização é um pouco mais cara, mas não existe busca para uma substituição; o ponteiro de fim aponta para o fim da pilha, que é a página LRU. Essa abordagem é particularmente apropriada para implementações de substituição LRU via software ou microcódigo.

Nem a substituição ótima nem a LRU sofrem da anomalia de Belady. Existe uma classe de algoritmos de substituição de página, chamada **algoritmos** de **pilha,** que nunca apresentam a anomalia de Belady. Um algoritmo de pilha é um algoritmo para o qual pode ser demonstrado que o conjunto de páginas na memória para n quadros é sempre um *subconjunto* do conjunto de páginas que estariam na memória com $n + 1$ quadros. Para a substituição LRU, o conjunto de páginas na memória seria as n páginas referenciadas mais recentemente. Se o número de quadros aumentar, essas n páginas ainda serão as referenciadas mais recentemente e também estarão na memória.

Observe que nenhuma implementação de LRU seria concebível sem assistência de hardware além dos registradores TLB padrão. A atualização dos campos do relógio ou da pilha deve ser feita para *cada* referência de memória. Se fôssemos utilizar uma interrupção para toda referência, para permitir que o software atualizasse essas estruturas de dados, isso tornaria toda referência de memória mais lenta por um fator de pelo menos 10, tornando lentos todos os processos de usuário por um fator de 10. Poucos sistemas poderiam tolerar esse nível de custo para a gerência de memória.

10.3.5 Aproximações do algoritmo LRU

Poucos sistemas de computador fornecem suporte de hardware suficiente para a verdadeira substituição de página LRU. Alguns sistemas não oferecem suporte de hardware e outros algoritmos de substituição de página (tais como o algoritmo FIFO) devem ser usados, todavia, vários sistemas provêem alguma ajuda, na forma de um bit de referência. O bit de referência para uma página é ativado, pelo hardware, sempre que a página é referenciada (leitura ou escrita para qualquer byte na página). Os bits de referência estão associados com cada entrada na tabela de página.

Inicialmente, todos os bits são limpos (definidos como 0) pelo sistema operacional. À medida que o processo de usuário executa, o bit associado com cada página referenciada é ativado (para 1) pelo hardware. Após algum tempo, podemos determinar quais páginas foram usadas e quais não foram usadas examinando os bits de referência. Não sabemos a *ordem* de uso, mas sabemos quais páginas foram usadas e quais não foram. Essas informações de ordenação parcial levam a muitos algoritmos de substituição de página que se aproximam da substituição LRU.

10.3.5.1 Algoritmo dos bits de referência adicionais

Podemos obter informações adicionais sobre a ordenação registrando os bits de referência em intervalos regulares. Podemos manter um byte de 8 bits para cada página em uma tabela na memória. Em intervalos regu-

lares (digamos, a cada 100 milissegundos), uma interrupção do temporizador transfere o controle para o sistema operacional. O sistema operacional passa o bit de referência de cada página para o bit mais significativo de seu byte de 8 bits, deslocando os outros bits para a direita 1 bit, descartando o bit menos significativo. Esses registradores de deslocamento de 8 bits contêm o fundamentos do uso da página para os últimos 8 períodos de tempo. Se o registrador contiver 00000000, a página não foi usada nesses períodos; uma página que é usada pelo menos uma vez em cada período teria um valor de registrador de 11111111.

Uma página com o valor do registrador fundamentos de 11000100 foi usada mais recentemente do que outra com 01110111. Se interpretarmos esses bytes de 8 bits como inteiros sem sinal, a página com o menor número é a página LRU e pode ser substituída. Observe, no entanto, que os números não têm garantia de serem exclusivos. Podemos descarregar *(swap out)* todas as páginas com o menor valor ou usar uma Seleção FIFO dentre elas.

O número de bits de fundamentos pode variar, é claro, e será selecionado (dependendo do hardware disponível) para tornar a atualização a mais rápida possível. Em casos extremos, o número pode ser reduzido para zero, deixando apenas o bit de referência. Esse algoritmo é chamado **algoritmo de substituição de página de segunda chance.**

10.3.5.2 Algoritmo de segunda chance

O algoritmo básico de algoritmo de segunda chance é um algoritmo de substituição FIFO. Quando uma página tiver sido selecionada, no entanto, inspecionamos seu bit de referência. Se o valor for 0, continuamos com a substituição de página. Se o bit de referência for 1, no entanto, essa página recebe uma segunda chance e passamos para a Seleção da próxima página FIFO. Quando a página obtiver uma segunda chance, seu bit de referência é limpo e o tempo de chegada é ajustado para a data atual. Assim, uma página que recebe uma segunda chance só será substituída quando todas as outras páginas forem substituídas (ou receberem uma segunda chance). Além disso, se uma página for usada com frequência suficiente para manter seu bit de referência ativo, ela nunca será substituída.

Uma forma de implementar o algoritmo de segunda chance (às vezes chamado de algoritmo do clock) é uma fila circular. Um ponteiro indica que página deve ser substituída a seguir. Quando um quadro é necessário, o ponteiro avança até encontrar uma página com um bit de referência 0. A medida que avança, ele limpa os bits de referência (Figura 10.13). Uma vez encontrada uma página vítima, a página é substituída e a nova

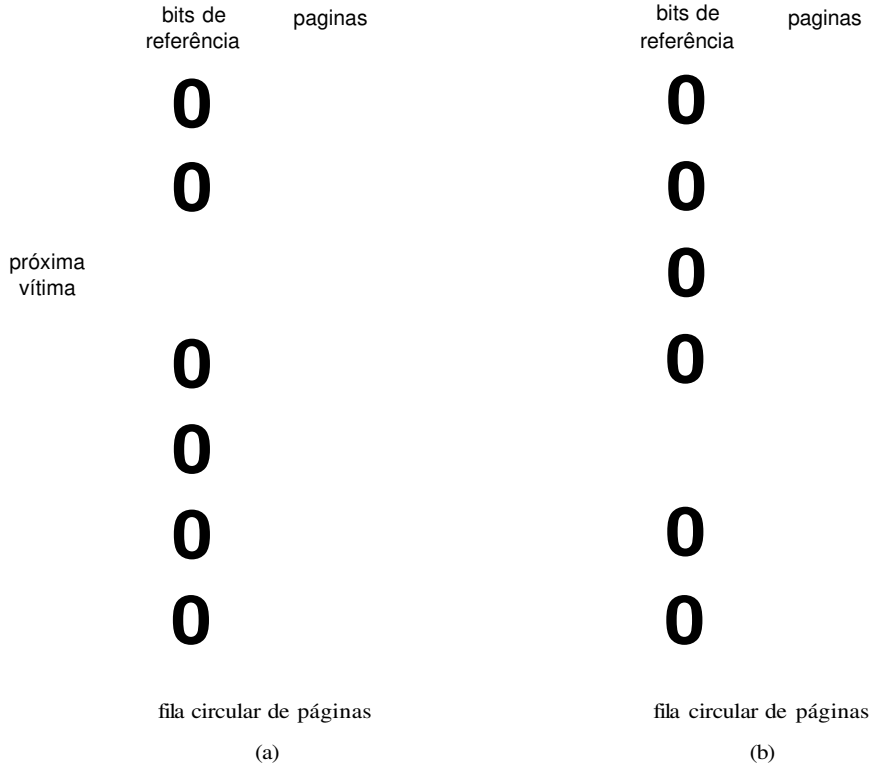


Figura 10.13 Algoritmo de substituição de página da segunda chance (ou do clock).

página é inserida na fila circular naquela posição. Observe que, no pior caso, quando todos os bits forem definidos, o ponteiro faz um ciclo em toda a fila, dando a cada página uma segunda chance. Ele limpa todos os bits de referência antes de selecionar a próxima página para substituição. A substituição de segunda chance se degenera em substituição FIFO se todos os bits estiverem ativos.

10.3.5.3 Algoritmo da segunda chance melhorado

iodemos melhorar o algoritmo de segunda chance considerando o bit de referência e o bit de modificação (Seção 10.3) como um par ordenado. Com esses dois bits, temos as seguintes quatro classes possíveis:

1. (0,0) - não foi usada nem modificada recentemente - melhor página para substituir
2. (0,1) - não foi usada recentemente mas foi modificada - opção não tão boa, porque a página precisará ser gravada antes de ser substituída
3. (1,0) - recentemente usada mas limpa - provavelmente logo será usada novamente
4. (1,1)- recentemente usada e modificada - provavelmente logo será usada novamente e a página precisará ser gravada em disco antes de ser substituída

Quando há necessidade de substituição de página, cada página está em uma dessas quatro classes. Usamos o mesmo esquema que o algoritmo do clock, mas em vez de examinar se a página para a qual estamos apontando tem o bit de referência ativo 1, examinamos a classe à qual a página pertence. Substituímos a primeira página encontrada na menor classe não-vazia. Observe que talvez seja preciso percorrer a fila circular várias vezes antes de achar uma página para ser substituída.

Esse algoritmo é usado no esquema de gerência de memória virtual do Macintosh. A principal diferença entre esse algoritmo e o algoritmo de clock mais simples é que aqui damos preferência a manter em memória páginas que foram modificadas, para reduzir o número de operações de I/O necessárias.

10.3.6 Algoritmos de substituição baseadas em contagem

Existem muitos outros algoritmos que podem ser usados para substituição de página. Por exemplo, podemos manter um contador do número de referências que foram feitas a cada página e desenvolver os dois esquemas seguintes:

- O algoritmo de substituição de página menos frequentemente usada **Least Frequently Used** (LFU) requer que a página com a menor contagem seja substituída. O motivo para essa Seleção é que uma página muito usada deve ter uma contagem de referência alta. Esse algoritmo enfrenta o seguinte problema: uma página é muito usada durante a fase inicial de um processo, mas depois nunca mais é usada. Como ela foi muito utilizada, possui uma alta contagem e permanece na memória embora não seja mais necessária. Uma solução é deslocar as contagens para a direita em 1 bit em intervalos regulares, formando uma contagem de uso médio com decaimento exponencial.
- O algoritmo de substituição de página mais frequentemente usada **Most Frequently Used (MFU)** baseia-se no argumento de que a página com menor contagem provavelmente acaba de chegar à memória e ainda deverá ser utilizada.

Como esperado, os algoritmos de substituição MFU e LFU não são comuns. A implementação desses algoritmos é cara, e eles não se aproximam da substituição OPT.

10.3.7 Algoritmo do buffer de páginas

Muitas vezes, outros procedimentos são utilizados além de um algoritmo de substituição de página específico. Por exemplo, os sistemas em geral mantêm um **pool** de quadros livres. Quando ocorre uma falta de página, um quadro vítima é escolhido como antes. No entanto, a página desejada é lida em um quadro livre do pool antes que a vítima seja gravada. Esse procedimento permite que o processo seja reiniciado assim que possível, sem esperar que a página vítima seja gravada. Quando a vítima for gravada mais tarde, o quadro será adicionado ao pool de quadros livres.

Uma expansão dessa ideia é manter uma lista de páginas modificadas. Sempre que o dispositivo de paginação estiver ocioso, uma página modificada será selecionada e gravada no disco. Seu bit de modificação será então desativado. Esse esquema aumenta a probabilidade de que uma página esteja limpa quando for selecionada para substituição e que não precisará ser gravada.

Outra modificação é manter um pool de quadros livres, lembrando de que página estava em cada quadro. Como o conteúdo do quadro não é modificado quando um quadro é gravado no disco, a página antiga pode ser reutilizada diretamente do pool de quadros livres se for necessária antes que o quadro seja reutilizado. Nenhuma operação de I/O é necessária nesse caso. Quando ocorre uma falta de página, primeiro verificamos se a página desejada está no pool de quadros livres. Se não estiver, devemos selecionar um quadro livre e carregá-la nele.

Essa técnica é usada no sistema VAX/VMS, com um algoritmo de substituição FIFO. Quando o algoritmo FIFO substitui por engano uma página que ainda está em uso ativo, essa página é rapidamente recuperada do buffer de quadros livres e nenhuma operação de I/O é necessária. O buffer de quadros livres fornece proteção contra o algoritmo relativamente fraco, mas simples, de substituição FIFO. Esse método é necessário porque as primeiras versões de VAX não implementavam corretamente o bit de referência.

10.4 • Alocação de quadros

Como alocamos a quantidade fixa de memória livre entre os vários processos? Se tivermos 93 quadros livres e dois processos, quantos quadros cada processo obterá?

O caso mais simples de memória virtual é o sistema monousuário. Considere um sistema monousuário com 128K de memória composta por páginas de IK. Portanto, existem 128 quadros. O sistema operacional pode usar 35K, deixando 93 quadros para o processo de usuário. Na paginação sob demanda pura, todos os 93 quadros seriam inicialmente colocados na lista de quadros livres. Quando um processo de usuário começasse a execução, ele geraria uma sequência de falta de página. As primeiras 93 faltas de página obteriam quadros livres da lista de quadros livres. Quando a lista tivesse sido esgotada, um algoritmo de substituição de página seria usado para selecionar uma das 93 páginas na memória para ser substituída pela nonagésima quarta e assim por diante. Quando o processo terminasse, os 93 quadros mais uma vez seriam colocados na lista de quadros livres.

Existem muitas variações dessa estratégia simples. Podemos solicitar que o sistema operacional aloque todo seu espaço de tabela e buffer a partir da lista de quadros livres. Quando esse espaço não estiver em uso pelo sistema operacional, poderá ser utilizado como suporte à paginação de usuário. Podemos tentar manter três quadros livres reservados na lista de quadros livres o tempo todo. Assim, quando ocorrer uma falta de página, haverá um quadro livre disponível para paginação. Enquanto está ocorrendo a troca de página, uma substituição pode ser selecionada, sendo então gravada no disco, à medida que o processo de usuário continua executando.

Outras variantes também são possíveis, mas a estratégia básica é clara: o processo de usuário recebe qualquer quadro livre.

Um problema diferente surge quando a paginação sob demanda é combinada com a multiprogramação. A multiprogramação coloca dois processos (ou mais) na memória ao mesmo tempo.

10.4.1 Número mínimo de quadros

Existem, é claro, várias limitações às nossas estratégias para a alocação de quadros. Não é possível alocar mais do que o número total de quadros disponíveis (a menos que exista compartilhamento de página). Existe também um número mínimo de quadros que pode ser alocado. Obviamente, à medida que o número de quadros alocados para cada processo diminui, a taxa de falta de página aumenta, tornando a execução do processo mais lenta.

Além das propriedades de desempenho indesejadas da alocação de apenas alguns quadros, existe um número mínimo de quadros que precisam ser alocados. Esse número mínimo é definido pela arquitetura do conjunto de instruções. Lembre-se de que, quando ocorre uma falta de página antes que a execução de uma

instrução esteja concluída, a instrução deverá ser reiniciada. Consequentemente, é preciso ter quadros suficientes para manter todas as páginas distintas que qualquer instrução única pode referenciar.

Por exemplo, considere uma máquina na qual todas as instruções de referência de memória têm apenas um endereço de memória. Assim, precisamos de pelo menos um quadro para a instrução e um quadro para a referência de memória. Além disso, se o endereçamento indireto de um nível for permitido (por exemplo, uma instrução de carga na página 16 pode fazer referência a um endereço na página 0, que é uma referência indireta à página 23), então a paginação requer pelo menos três quadros por processo. Pense sobre o que poderia acontecer se um processo tivesse apenas dois quadros.

O número mínimo de quadros é definido pela arquitetura do computador. Por exemplo, a instrução *move* do PDP-11 é maior que uma palavra para alguns modos de endereçamento, portanto a instrução em si pode ocupar duas páginas. Além disso, cada um dos seus dois operandos podem ser referências indiretas, para um total de seis quadros. O pior caso para o IBM 370 é provavelmente a instrução MVC. Como a instrução é de memória para memória, utiliza 6 bytes e pode ocupar duas páginas. O bloco de caracteres a ser movido e a área para a qual ele será movido também podem ocupar duas páginas. Essa situação exigiria seis quadros. (Na verdade, o pior caso ocorre quando a instrução MVC é o operando para uma instrução EXECUTE que ocupa um limite de página; nesse caso, oito quadros são necessários.)

O cenário de pior caso ocorre em arquiteturas que permitem vários níveis de indireção (por exemplo, cada palavra de 16 bits poderia conter um endereço de 15 bits e um indicador indireto de 1 bit). Teoricamente, uma instrução simples de carga poderia referenciar um endereço indireto que poderia referenciar um endereço indireto (em outra página) que também poderia referenciar um endereço indireto (em outra página) e assim por diante, até que todas as páginas na memória virtual tivessem sido tocadas. Assim, no pior caso, toda a memória virtual deve estar na memória física. Para superar essa dificuldade, é preciso colocar um limite nos níveis de indireção (por exemplo, limitar uma instrução para, no máximo 16 níveis de indireção). Quando ocorre a primeira indireção, um contador é ajustado em 16; o contador é então diminuído para cada indireção sucessiva nessa instrução. Se o contador for decrementado até 0, ocorre uma exceção (indireção excessiva). Essa limitação reduz o número máximo de referências de memória por instrução para 17, exigindo o mesmo número de quadros.

O número mínimo de quadros por processo é definido pela arquitetura, enquanto o número máximo é definido pela quantidade de memória física disponível. Entre esses extremos, ainda temos muitas opções em termos de alocação de quadros.

10.4.2 Algoritmos de alocação

A forma mais fácil de dividir *m* quadros entre *n* processos é dar a todos uma parcela igual de *min* quadros. Por exemplo, se houver 93 quadros e cinco processos, cada processo obterá 18 quadros. Os 3 quadros restantes podem ser usados como um pool de buffers de quadros livres. O esquema é chamado de **alocação igual,** 1

Uma alternativa é reconhecer que os vários processos precisarão de diferentes quantidades de memória. Considere um sistema com um tamanho de quadro de 1K. Se um pequeno processo secundário de 10K e um banco de dados interativo de 127K forem os dois únicos processos executando em um sistema com 62 quadros livres, não faz muito sentido dar a cada processo 31 quadros. O processo auxiliar não precisa de mais de 10 quadros, por isso os outros 21 são desperdiçados.

Para resolver esse problema, podemos usar a **alocação proporcional.** Alocamos a memória disponível para cada processo de acordo com seu tamanho. Considerando que o tamanho da memória virtual para o processo p_t é s,, temos

$$S = \text{I s,}$$

Assim, se o número total de quadros disponíveis for m, alocamos a_f quadros ao processo /?,, onde a_x é aproximadamente

$$cij = Sj/S \text{ x } m.$$

É claro que devemos ajustar cada tf, para ser um inteiro maior que o número mínimo de quadros exigidos pelo conjunto de instruções, com soma não ultrapassando *m.*

Para a alocação proporcional, dividiríamos 62 quadros entre dois processos, um de 10 páginas e um de 127 páginas, alocando respectivamente, 4 quadros e 57 quadros, já que

$$10/137 \times 62 \ll 4$$
$$127/137 \times 62 \ll 57.$$

Dessa forma, os dois processos compartilham os quadros disponíveis de acordo com suas "necessidades", em vez de igualmente.

Em ambas as formas de alocação, é claro, a alocação para cada processo pode variar de acordo com o nível de multiprogramação. Se o nível de multiprogramação aumentar, cada processo perderá alguns quadros para fornecer a memória necessária para o novo processo. Por outro lado, se o nível de multiprogramação diminuir, os quadros que foram alocados para o processo que partiu agora podem ser divididas entre os processos restantes.

Observe que, com a alocação igual ou proporcional, um processo de alta prioridade é tratado da mesma forma que um processo de baixa prioridade. Por definição, no entanto, talvez o processo de prioridade mais alta devesse receber mais memória para acelerar sua execução, em detrimento dos processos de baixa prioridade.

Uma abordagem seria usar um esquema de alocação proporcional no qual a razão de quadros depende não do tamanho relativo dos processos, mas das prioridades dos processos ou de uma combinação de tamanho e prioridade.

10.4.3 Alocação global *versus* local

Outro fator importante na forma em que os quadros são alocados aos vários processos é a substituição de página. Com vários processos competindo por quadros, é possível classificar os algoritmos de substituição de página em duas amplas categorias: **substituição global** e **substituição local.** A substituição global permite que um processo selecione um quadro de substituição do conjunto de todos os quadros, mesmo se esse quadro estiver alocando no momento a algum outro processo; um processo pode tirar um quadro de outro. A substituição local requer que cada processo selecione somente a partir de seu próprio conjunto de quadros alocados.

Por exemplo, considere um esquema de alocação no qual os processos de alta prioridade possam selecionar quadros dos processos de baixa prioridade para substituição. Um processo pode selecionar uma substituição a partir de seus próprios quadros ou de quadros de qualquer outro processo de prioridade mais baixa. Essa abordagem permite que um processo de alta prioridade aumente sua alocação de quadros à custa do processo de baixa prioridade.

Com uma estratégia de substituição local, o número de quadros alocados para um processo não muda. Com a substituição global, pode acontecer de um processo selecionar apenas quadros alocados a outros processos, aumentando assim o número de quadros alocados a ele (considerando que outros processos não escolham *seus* quadros para substituição).

Um problema com um algoritmo de substituição global é que um processo não pode controlar sua própria taxa de falta de página. O conjunto de páginas na memória para um processo depende não só do comportamento de paginação desse processo, mas também do comportamento de paginação de outros processos. Portanto, o mesmo processo pode ter um desempenho bem diferente (levando 0,5 segundos para uma execução e 10,3 segundos para a próxima) devido a circunstâncias totalmente externas. Esse não é o caso com um algoritmo de substituição local. Na substituição local, o conjunto de páginas na memória para um processo é afetado pelo comportamento de paginação apenas daquele processo. De sua parte, a substituição local poderá limitar um processo não disponibilizando outras páginas de memória menos usadas. Assim, a substituição global geralmente resulta em maior throughput do sistema e, portanto, é o método mais comum.

10.5 • Thrashing

Se o número de quadros alocados para um processo de baixa prioridade cair abaixo do número mínimo exigido pela arquitetura do computador, devemos suspender a execução do processo. Em seguida, devemos

descarregar o restante de suas páginas, liberando todos os quadros alocados. Essa medida introduz um nível de swapping de escalonamento intermediário de CPU.

Na verdade, analise qualquer processo que não tenha quadros "suficientes". Embora seja tecnicamente possível reduzir o número de quadros alocados para o mínimo, existe um número (maior) de páginas em uso ativo. Se o processo não tiver esse número de quadros, ele causará uma falta de página rapidamente. Nesse momento, ele deverá substituir alguma página. No entanto, como todas as páginas estão em uso ativo, ele deverá substituir uma página que logo será necessária novamente. Consequentemente, ele causará uma falha novamente, várias e várias vezes seguidas. O processo continua a falhar, substituindo páginas para em seguida falhar, levando-as de volta.

Essa alta atividade de paginação é chamada de **thrashing.** Um processo está em estado de **thrashing** se estiver despendendo mais tempo paginando do que executando.

10.5.1 Causa do thrashing

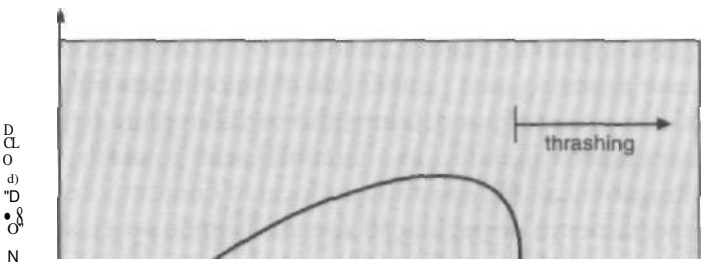
O thrashing resulta em graves problemas de desempenho. Considere o seguinte cenário, que é baseado no comportamento real dos primeiros sistemas de paginação.

O sistema operacional monitora a utilização de CPU. Se a utilização de CPU for muito baixa, aumentamos o grau de multiprogramação introduzindo um novo processo no sistema. Um algoritmo de substituição de página global é usado; ele substitui as páginas independentemente do processo ao qual elas pertencem. Agora, vamos supor que um processo entre em uma nova fase de execução e precise de mais quadros. Ele começa a gerar faltas de página e retira quadros de outros processos. Entretanto, esses processos precisam dessas páginas e acabam falhando, pegando quadros de outros processos. Esses processos com falta de página devem utilizar o dispositivo de paginação para carregar e descarregar páginas. A medida que elas entram na fila para o dispositivo de paginação, a fila de processos prontos vai esvaziando. Enquanto os processos esperam pelo dispositivo de paginação, a utilização de CPU diminui.

O escalonador de CPU vê a redução da utilização de CPU e *aumenta* o grau de multiprogramação como resultado disso. O novo processo tenta iniciar pegando quadros dos processos em execução, causando mais faltas de página e uma fila maior para o dispositivo de paginação. Consequentemente, a utilização de CPU cai ainda mais, e o escalonador de CPU tenta aumentar ainda mais o grau de multiprogramação. Ocorreu o thrashing e o throughput do sistema desaba. A taxa de falta de página aumenta drasticamente. Como resultado, o tempo efetivo de acesso à memória aumenta. Nada está sendo realizado, porque os processos estão utilizando todo seu tempo na paginação.

Esse fenómeno é ilustrado na Figura 10.14, na qual a utilização de CPU é traçada em relação ao grau de multiprogramação. A medida que o grau de multiprogramação aumenta, a utilização de CPU também aumenta, embora mais lentamente, até que um valor máximo seja alcançado. Se o grau de multiprogramação for aumentado além desse ponto, ocorre o thrashing e a utilização de CPU cai drasticamente. Nesse ponto, para aumentar a utilização de CPU e acabar com o thrashing, é preciso *diminuir* o grau de multiprogramação.

Podemos limitar os efeitos do thrashing usando um **algoritmo de substituição local** (ou por **prioridade).** Com a substituição local, se um processo entrar em thrashing, ele não poderá roubar quadros de outro pro-



grau de multiprogramação

Figura 10.14 Thrashing.

cesso e fazer com que este também entre em thrashing. As páginas são substituídas com relação ao processo do qual fazem parte. No entanto, se processos estiverem em thrashing, eles ficarão na fila do dispositivo de paginação a maior parte do tempo. O tempo de serviço médio para uma falta de página aumentará devido à fila média mais longa para o dispositivo de paginação. Assim, o tempo de acesso efetivo aumentará mesmo para um processo que não esteja em thrashing.

Para evitar o thrashing, é preciso fornecer a um processo o número de quadros que ele precisa. No entanto, como saber quantos quadros ele "precisa"? Existem várias técnicas. A estratégia do conjunto de trabalho *{working-set),* discutida na Seção 10.5.2, começa analisando quantos quadros um processo está de fato utilizando. Essa abordagem define o modelo de localidade da execução de processos.

O modelo de localidade afirma que, à medida que um processo executa, ele se move de localidade a localidade. Uma localidade é um conjunto de páginas que são usadas juntas ativamente (Figura 10.15). Um programa geralmente é formado por várias localidades diferentes, que podem se sobrepor.

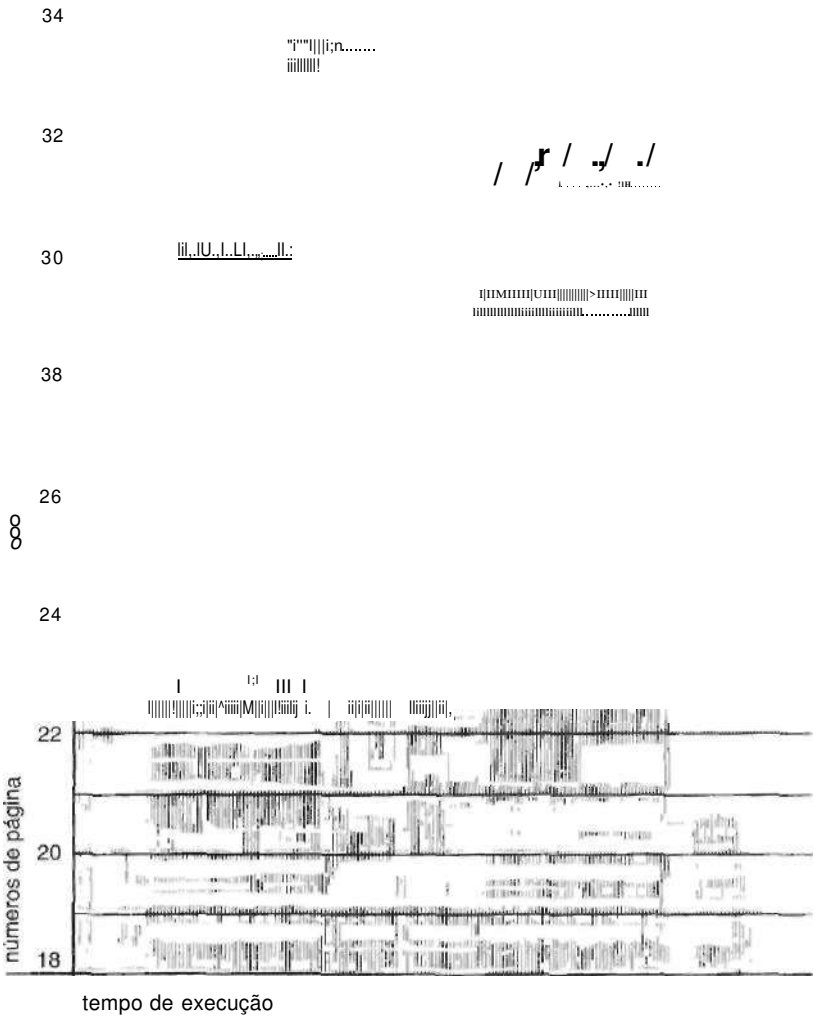


Figura 10.15 Localidade em um padrão de referência de memória.

Por exemplo, quando uma sub-rotina é chamada, ela define uma nova localidade. Nessa localidade, as referências de memória são feitas às instruções da sub-rotina, suas variáveis locais e um subconjunto das variáveis globais. Quando sai de uma sub-rotina, o processo deixa essa localidade, já que as variáveis locais e instruções da sub-rotina não estão mais em uso ativo. E possível retornar a essa localidade depois. Assim, vemos que as localidades são definidas pela estrutura do programa e suas estruturas de dados. O modelo de localidade afirma que todos os programas apresentarão essa estrutura básica de referência de memória. Observe que o modelo de localidade é o princípio não-declarado por trás das discussões de cache feitas até agora neste livro. Se os acessos aos tipos de dados fossem aleatórios e não tivessem um padrão, o cache seria inútil.

Vamos supor que tenhamos alocado quadros suficientes para um processo a fim de acomodar sua localidade atual. Haveria falta de página para as páginas nessa localidade até que todas as páginas estivessem na memória; em seguida, não haveria mais falta até haver uma mudança de localidade. Se alocarmos menos quadros do que o tamanho da localidade atual, o processo entrará em estado de thrashing, já que não pode manter na memória todas as páginas que estão sendo ativamente utilizadas.

10.5.2 O modelo de conjunto de trabalho (working-set model)

O modelo de conjunto de trabalho baseia-se na suposição de localidade. Esse modelo utiliza um parâmetro, A, para definir a janela de conjunto de trabalho. A ideia é examinar as A referências de página mais recentes. O conjunto de páginas nas A referências de página mais recentes é o **conjunto de trabalho** (Figura 10.16). Se uma página estiver em uso ativo, ela estará no conjunto de trabalho. Se não estiver mais sendo usada, ela sairá do conjunto de trabalho A unidades de tempo depois de sua última referência. Assim, o conjunto de trabalho é uma aproximação da localidade do programa.

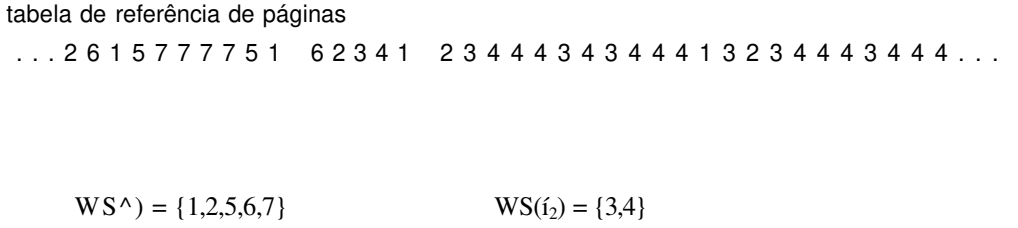


Figura 10.16 Modelo de conjunto de trabalho.

Por exemplo, considerando a sequência de referências de memória indicadas na Figura 10.16, se A = 10, então o conjunto de trabalho no instante t_1 será {1,2,5,6,7}. No instante $t^$ o conjunto de trabalho mudou para {3,4}.

A exatidão do conjunto de trabalho depende da Seleção de A. Se A for muito pequeno, ele não abrangerá a localidade toda; se A for grande demais, ele poderá se sobrepor a várias localidades. No caso extremo, se A for infinito, o conjunto de trabalho será o conjunto de páginas tocadas durante a execução do processo.

A propriedade mais importante do conjunto de trabalho é seu tamanho. Se calcularmos o tamanho do conjunto de trabalho, W5S,, *(working-set sizé)* para cada processo no sistema, podemos considerar

$$D = I\ WSS_h$$

onde *D* é a demanda total por quadros. Cada processo está usado ativamente as páginas do seu conjunto de trabalho. Assim, o processo *i* precisa de WSS, quadros. Se a demanda total for maior do que o número total de quadros disponíveis (D > *m),* ocorrerá thrashing, porque alguns processo não terão quadros suficientes.

O uso do modelo de conjunto de trabalho fica então simples. O sistema operacional monitora o conjunto de página de cada processo e aloca para esse conjunto um número suficiente de quadros para que tenha o tamanho do seu conjunto de trabalho. Se houver quadros extras suficientes, outro processo poderá ser iniciado. Se a soma dos tamanhos do conjunto de trabalho aumentar, excedendo o número total de quadros disponíveis, o sistema operacional selecionará um processo para suspender. As páginas do processo são descarregadas e seus quadros são relocados para outros processos. O processo suspenso pode ser retomado depois.

Essa estratégia de conjunto de trabalho evita o thrashing ao mesmo tempo em que mantém o grau de multiprogramação o mais alto possível. Assim, ele otimiza a utilização de CPU.

A dificuldade com o modelo de conjunto de trabalho é rastrear o conjunto de trabalho. A janela de conjunto de trabalho é uma janela em movimento. A cada referência de memória, uma nova referência aparece de um lado e a referência mais antiga é descartada do outro lado. Uma página está no conjunto de trabalho se for referenciada em qualquer ponto da janela de conjunto de trabalho. Podemos aproximar o modelo de conjunto de trabalho com uma interrupção de tempo em intervalos fixos e um bit de referência.

Por exemplo, assuma que A é 10.000 referências e que podemos causar uma interrupção de tempo **a** cada 5.000 referências. Quando obtivermos uma interrupção de tempo, poderemos copiar e limpar os valores de bit de referência para cada página. Assim, se ocorrer uma falta de página, poderemos examinar o bit de referência atual e os 2 bits na memória para determinar se uma página foi usada nas últimas 10.000 a 15.000 referências. Se tiver sido usada, pelo menos um desses bits estará ativo. Se não tiver sido usada, esses bits estarão desativados. Essas páginas com pelo menos um bit ativo serão consideradas como estando no conjunto de trabalho. Observe que esse arranjo não é inteiramente exato, porque não sabemos onde, dentro do intervalo de 5.000, ocorreu uma referência. Podemos reduzir a incerteza aumentando o número de bits de histórico e a frequência de interrupções (por exemplo, 10 bits e interrupções a cada 1000 referências). No entanto, o custo de serviço dessas interrupções mais frequentes será proporcionalmente mais alto.

10.5.3 Frequência de falta de páginas

O modelo de conjunto de trabalho funciona bem e o conhecimento do conjunto de trabalho poderá ser útil para a pré-paginação (Seção 10.7.1), mas parece uma forma desajeitada de controlar o thrashing. Uma estratégia que utiliza a **frequência de falta de páginas (PFF - page-fault frequency)** tem uma abordagem mais direta.

O problema específico é como evitar o thrashing. O thrashing tem uma alta taxa de falta de página. Assim, é preciso controlar a taxa de falta de página. Quando ela é muito alta, sabemos que o processo precisa de mais quadros. Da mesma forma, se a taxa de falta de página for baixa demais, o processo talvez tenha um número excessivo de quadros. Podemos estabelecer limites superior e inferior na taxa de falta de página desejada (Figura 10.17). Se a taxa de falta de página real exceder o limite superior, alocamos outro quadro a esse processo; se a taxa ficar abaixo do limite inferior, removemos um quadro do processo. Assim, podemos medir **e** controlar diretamente a taxa de falta de página para evitar o thrashing.

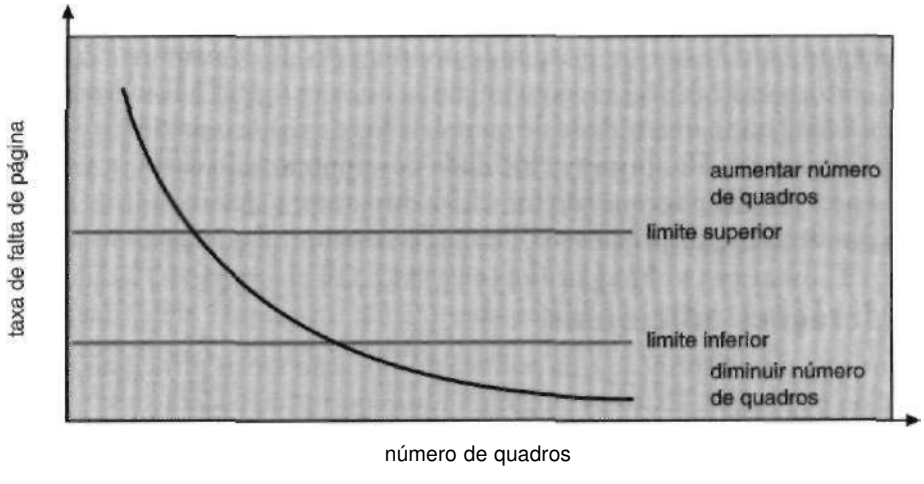


Figura 10.17 Frequência de falta de páginas.

Como ocorre com a estratégia de conjunto de trabalho, talvez seja necessário suspender um processo. Se a taxa de falta de página aumentar e não houver quadros livres disponíveis, será preciso selecionar algum processo e suspendê-lo. Os quadros liberados serão então distribuídos aos processos com altas taxas de falta de página.

10.6 • Exemplos de sistemas operacionais

Nesta seção vamos descrever como o Windows NT e o Solaris 2 implementam a memória virtual.

10.6.1 Windows NT

O Windows NT implementa a memória virtual utilizando a paginação sob demanda com **clustering.** O clustering trata faltas de página carregando não apenas a página faltante, mas também várias páginas em torno da página faltante. Quando um processo é criado, a ele é atribuído um **conjunto de trabalho mínimo e máximo.** O

conjunto de trabalho mínimo é o número de páginas que o processo com certeza tem na memória. Se houver memória suficiente disponível, um processo poderá receber o mesmo número de páginas que seu **conjunto de trabalho máximo.** O gerenciador de memória virtual mantém uma lista de quadros de página livres. Associado a essa lista está um valor limite que é usado para indicar se existe memória livre suficiente disponível ou não. Se ocorrer uma falta de página para determinado processo que esteja abaixo de seu conjunto de trabalho máximo, o gerenciador de memória virtual aloca uma página dessa lista de páginas livres. Se um processo estiver no seu conjunto de trabalho máximo e incorrer em uma falta de página, ele deverá selecionar uma página para substituição usando uma regra local FIFO de substituição de página. Quando a quantidade de memória livre ficar abaixo do limite, o gerenciador de memória virtual utilizará uma tática chamada **ajuste automático de conjunto de trabalho** para restaurar o valor acima do limite. Esse ajuste automático funciona avaliando o número de páginas alocado aos processos. Se um processo tiver mais páginas que seu **conjunto de trabalho mínimo,** o gerenciador de memória virtual utiliza um algoritmo FIFO para remover páginas até que o processo alcance seu **conjunto de trabalho mínimo.** Um processo que está no **conjunto de trabalho mínimo** poderá receber páginas da lista de quadros de página livres assim que houver memória livre disponível.

10.6.2 Solaris 2

Quando um processo incorre em uma falta de página, o kernel atribui uma página ao processo com falta da lista de páginas livres mantidas por ele. Portanto, é imperativo que o kernel tenha uma quantidade suficiente de memória livre disponível. Associados a essa lista de páginas livre estão dois parâmetros - *minfree* e *lotsfree* - que são, respectivamente, limiares inferior e superior para a memória livre disponível. Quatro vezes por segundo, o kernel verifica a quantidade de memória livre. Se essa quantidade ficar abaixo de *minfree^* um processo chamado **pageout** começa. O processo de pageout é igual ao algoritmo de segunda chance descrito na Seção 10.3.5.2; também é chamado de **algoritmo do clock de dois ponteiros.** Ele funciona da seguinte forma: o primeiro ponteiro do clock percorre todas as páginas na memória ajustando o bit de referência para 0. Em um momento posterior, o segundo ponteiro do clock examina o bit de referência para as páginas na memória, retornando as páginas cujo bit ainda está em 0 para a lista livre. O processo de pageout continua a execução até que a quantidade de memória livre exceda o parâmetro *lotsfree.* Além disso, o processo de pageout é dinâmico. Ele ajusta a velocidade dos ponteiros do clock se a memória disponível ficar muito baixa. Se o processo de pageout não for capaz de manter a quantidade de memória livre em *lotsfree,* o kernel começa a descarregar processos, liberando, assim, todas as páginas alocadas ao processo.

10.7 • Considerações adicionais

A Seleção de um algoritmo de substituição e da política de alocação são as principais decisões a serem tomadas para um sistema de paginação. Existem também muitas outras considerações.

10.7.1 Pré-paginação

Uma propriedade óbvia de um sistema de paginação sob demanda puro é o grande número de faltas de página que ocorrem quando um processo é iniciado. Essa situação é o resultado de tentar carregar a localidade inicial na memória. A mesma situação pode ocorrer em outros momentos. Por exemplo, quando um processo descarregado da memória é retomado, todas as suas páginas estão no disco e cada uma deve ser levada por sua própria falta de página. A pré-paginação é uma tentativa de evitar esse alto nível de paginação inicial. A estratégia é levar para a memória de uma vez todas as páginas que serão necessárias.

Em um sistema que utiliza o modelo de conjunto de trabalho, por exemplo, mantemos com cada processo uma lista das páginas no conjunto de trabalho. Se precisarmos suspender um processo (devido a uma espera de I/O ou falta de quadros livres), lembramos do conjunto de trabalho para esse processo. Quando o processo precisar ser retomado (conclusão de I/O ou quadros livres suficientes), automaticamente levamos para a memória seu conjunto de trabalho inteiro antes de reiniciar o processo.

A pré-paginação pode ser uma vantagem em alguns casos. A questão é simplesmente se o custo de usar a pré-paginação é menor do que o custo de efetuar o serviço das faltas de página correspondentes. Pode ser que muitas das páginas levadas de volta para a memória pela pré-paginação não sejam utilizadas.

Vamos supor que s páginas sejam pré-paginadas e uma fração *a* dessas s páginas sejam realmente usadas (0 < *a* < 1). A questão é se o custo das faltas de página economizadas *s*a* é maior ou menor do que o custo de pré-paginação de páginas s*(1 -*a)* desnecessárias. Se *a* for próximo a zero, a pré-paginação perde; se *a* for próximo a um, a pré-paginação vence.

10.7.2 Tamanho da página

Os projetistas de um sistema operacional para uma máquina existente raramente têm opção com relação ao tamanho de página. No entanto, quando novas máquinas estão sendo projetadas, uma decisão relativa ao melhor tamanho de página deve ser tomada. Como é de se esperar, não existe um único tamanho de página ideal. Em vez disso, existe um conjunto de fatores que justificam vários tamanhos. Os tamanhos de página são invariavelmente potências de 2, geralmente variando de 4.096 (2^{12}) para 4.194.304 (2^{22}) bytes.

Como é feita a Seleção de um tamanho de página? Uma preocupação é o tamanho da tabela de página. Para determinado espaço de memória virtual, diminuir o tamanho da página aumenta o número de páginas e, portanto, o tamanho da tabela de página. Para uma memória virtual de 4 megabytes (2^{22}), haveria 4.096 páginas de 1.024 bytes, mas apenas 512 páginas de 8.192 bytes. Como cada processo ativo deve ter sua própria cópia da tabela de página, um tamanho de página grande é desejável.

Por outro lado, a memória é mais bem utilizada com páginas menores. Se um processo receber memória começando na posição 00000, continuando até chegar ao valor necessário, provavelmente não terminará exatamente em um limite de página. Assim, uma parte da página final deve ser alocada (porque as páginas são unidades de alocação) mas não usada (fragmentação interna). Considerando a independência do tamanho do processo e da página, poderíamos esperar que, em média, metade da página final de cada processo será desperdiçada. Essa perda seria apenas de 256 bytes para uma página de 512 bytes, mas seria de 4.096 bytes para uma página de 8.192 bytes. Para minimizar a fragmentação interna, precisamos de um tamanho de página pequeno.

Outro problema é o tempo necessário para ler ou gravar uma página. O tempo de I/O é composto por tempos de busca, latência e transferência. O tempo de transferência é proporcional à quantidade transferida (ou seja, o tamanho da página) - um fato que pareceria ser a favor de um tamanho de página pequeno. Lembre-se do Capítulo 2, no entanto, que o tempo de busca e a latência normalmente são muito maiores que o tempo de transferência. A uma taxa de transferência de 2 megabytes por segundo, são necessários apenas 0,2 milissegundos para transferir 512 bytes. A latência, por outro lado, talvez seja de 8 milissegundos e o tempo de busca, 20 milissegundos. Do tempo de I/O total (28,2 milissegundos), portanto, 1% pode ser atribuído à transferência propriamente dita. Duplicar o tamanho da página aumenta o tempo de I/O para apenas 28,4 milissegundos. São necessários 28,4 milissegundos para ler uma única página de 1,024 bytes, mas 56,4 milissegundos para ler a mesma quantidade como duas páginas de 512 bytes cada. Portanto, querer minimizar o tempo de I/O justifica um tamanho de página maior.

Com um tamanho de página menor, no entanto, o tempo total de I/O deve ser reduzido, já que a localidade será melhor. Um tamanho de página menor permite que cada página faça a correspondência da localidade do programa de forma mais precisa. Por exemplo, considere um processo com 200K de tamanho, dos quais apenas metade (100K) são de fato usadas em uma execução. Se tivermos apenas uma página grande, devemos levar a página inteira, um total de 200K transferidos e alocados. Se tivermos páginas de apenas 1 byte, podemos levar apenas os 100K que são de fato usados, resultando em apenas 100K sendo transferidos e alocados.

Com um tamanho de página menor, teríamos melhor resolução, permitindo isolar apenas a memória que realmente é necessária. Com um tamanho de página maior, é preciso alocar e transferir não apenas o que é necessário, mas também todo o resto que esteja na página, quer seja necessário ou não. Portanto, uma página de tamanho menor deve resultar em menos I/O e em menos memória total alocada.

Por outro lado, você percebeu que com um tamanho de página de 1 byte, teríamos uma falta de página *para cada* byte? Um processo de 200K, usando apenas metade dessa memória, geraria apenas uma falta de página com um tamanho de página de 200K, mas 102.400 faltas de página para um tamanho de 1 byte. Cada falta de página gera a grande quantidade de trabalho necessário para processar a interrupção, salvar registradores, substituir uma página, entrar na fila para um dispositivo de paginação e atualizar as tabelas. Para minimizar o número de faltas de página, precisamos ter um tamanho de página grande.

A tendência histórica é ter páginas grandes. Na verdade, a primeira edição do livro OSC (Operating Systems Concepts, 1983) usava 4.096 bytes como limite superior no tamanho de página, e esse valor era o tamanho de página mais comum em 1990.0 Intel 80386 tem um tamanho de página de 4K; o Pentium II aceita tamanhos de página de 4K ou 4MB; o UltraSPARC aceita tamanhos de página de 8K, 64K, 512K ou 4MB. A evolução para tamanhos de página maiores é provavelmente resultado do aumento mais rápido das velocidades de CPU e capacidade de memória principal em relação às velocidades de disco. As faltas de página são mais caras hoje, em termos de desempenho geral do sistema, do que anteriormente. Portanto, é vantajoso aumentar o tamanho das páginas para reduzir a sua frequência. Obviamente, existe mais fragmentação interna como resultado.

Existem outros fatores a serem considerados (como o relacionamento entre o tamanho de página e o tamanho do setor no dispositivo de paginação). O problema não tem uma resposta ideal. Alguns fatores (fragmentação interna, localidade) favorecem um tamanho de página pequeno, enquanto outros (tamanho da tabela, tempo de I/O) favorecem um tamanho de página grande.

10.7.3 Tabela de página invertida

Na Seção 9.4.4, o conceito de tabela de página invertida foi apresentado. O propósito dessa forma de gerência de página é reduzir a quantidade total de memória física necessária para rastrear as traduções de endereços virtuais em físicos. Essa economia é obtida com a criação de uma tabela que tem uma entrada por página de memória física, indexada pelo par <id-processo, número-página>.

Como elas mantêm informações sobre que página da memória virtual está armazenada em cada quadro físico, as tabelas de página invertida reduzem a quantidade de memória física necessária para armazenar essas informações. No entanto, a tabela de página invertida não contém informações completas sobre o espaço de endereçamento lógico de um processo, e essas informações são exigidas se uma página referenciada não estiver na memória no momento. A paginação sob demanda requer essas informações para processar faltas de página. Para essas informações estarem disponíveis, uma tabela de página externa (uma por processo) deve ser mantida. Cada tabela dessas é similar à tabela de página tradicional por processo, contendo informações de onde cada página virtual está localizada.

Será que as tabelas de página externas negam a utilidade das tabelas de página invertidas? Como essas tabelas só são referenciadas quando ocorre uma falta de página, elas não precisam estar prontamente disponíveis. Em vez disso, são paginadas de e para a memória, conforme necessário. Infelizmente, uma falta de página pode fazer com que o gerenciador de memória virtual cause outra falta de página ao paginar a tabela de página externa necessária para localizar a página virtual no armazenamento auxiliar. Esse caso especial requer tratamento cuidadoso no kernel e um atraso no processamento da pesquisa de página.

10.7.4 Estrutura de programas

A paginação sob demanda é projetada para ser transparente ao programa de usuário. Em muitos casos, o usuário não tem conhecimento da natureza paginada da memória. Em outros casos, no entanto, o desempenho do sistema pode ser melhorado se o usuário (ou compilador) estiver a par da paginação sob demanda subjacente.

Vamos analisar um exemplo inventado mas informativo. Suponha que as páginas tem 128 palavras de tamanho. Considere um programa Java cuja função é inicializar em 0 cada elemento de um vetor de 128 por 128. O seguinte código é típico:

```
int A[ ][ ] = new int [128] [128];

for (int j = 0; j < 128; j++)
    for (int i = 0; i < 128; i++)
      A[i][j] - 0;
```

Observe que o vetor é armazenado em linhas. Ou seja, o vetor é armazenado como A[0] [0], A[0] [1], ... A[0][127], A[1][0], A[1][1]. . . . A [127] [127]. Para páginas de 128 palavras, cada linha ocupa uma pá-

gina. Assim, o código precedente zera uma palavra em cada página, depois outra palavra em cada página e assim por diante. Se o sistema operacional alocar menos do que 128 quadros para o programa todo, sua execução resultará em 128 X 128 = 16.384 faltas de página. Alterar o código para

```
int A[ ][ ] = new int [128] [128];

for (int i = 0; i < 128; i++)
    for (int j = 0; j < 128; j++)
      A[i][j] = 0;
```

por outro lado, zera todas as palavras em uma página antes de iniciar a próxima página, reduzindo o número de faltas de página para 128.

Uma cuidadosa Seleção de estruturas de dados e estruturas de programação pode aumentar a localidade e, portanto, diminuir a taxa de falta de página e o número de páginas no conjunto de trabalho. Uma pilha tem boa localidade, pois o acesso é sempre feito ao topo. Uma tabela de hashing, por outro lado, é projetada para dispersar referências, produzindo localidade ruim. É claro que a localidade da referência é apenas uma medida da eficiência do uso de uma estrutura de dados. Outros fatores de peso incluem velocidade de pesquisa, número total de referências de memória e o número total de páginas acessadas.

Em uma etapa posterior, o compilador e o carregador podem ter um efeito significativo na paginação. Separar código e dados e gerar código reentrante significa que as páginas de código só podem ser lidas e nunca serão modificadas. As páginas limpas não precisam ser descarregadas para serem substituídas. O carregador pode evitar colocar rotinas nos limites de página, mantendo cada rotina completamente em uma página. As rotinas que chamam umas às outras muitas vezes podem ser agrupadas na mesma página. Essa compactação é uma variante do problema de empacotamento da pesquisa operacional: tente compactar os segmentos de carga de tamanho variável em páginas de tamanho fixo de modo que as referências entre páginas sejam minimizadas. Tal abordagem é particularmente útil para páginas de tamanho grande.

A opção da linguagem de programação também pode afetar a paginação. Por exemplo, C e C++ utilizam ponteiros com frequência, e os ponteiros tendem a randomizar o acesso à memória. Compare essas linguagens com Java, que não fornece ponteiros. Os programas Java terão melhor localidade de referência do que os programas C ou C++ em um sistema de memória virtual.

10.7.5 Bloco de operações de I/O

Quando a paginação sob demanda é usada, às vezes é preciso permitir que algumas das páginas sejam travadas na memória. Uma situação assim ocorre quando a operação de I/O é feita de ou para a memória de usuário (virtual). A entrada/saída é muitas vezes implementada por um processador de I/O separado. Por exemplo, uma controladora de fita magnética geralmente recebe o número de bytes para transferir e um endereço de memória para o buffer (Figura 10.18). Quando a transferência é concluída, a CPU é interrompida.

Devemos ter certeza de que a seguinte sequência de eventos não ocorra: um processo emite um pedido de I/O e é colocado em uma fila para aquele dispositivo de I/O. Enquanto isso, a CPU é passada para outro processo. Esses processos causam faltas de página e, usando um algoritmo de substituição global, um deles substitui a página que contém o buffer de memória para o processo em espera. Essas páginas são então descarregadas. Algum tempo depois, quando o pedido de I/O avança para o início da fila de dispositivo, a I/O ocorre para o endereço especificado. No entanto, esse quadro agora está sendo usado para uma página diferente que pertence a outro processo.

Existem duas soluções comuns a esse problema. Uma solução é nunca executar I/O para a memória de usuário. Em vez disso, os dados são sempre copiados entre a memória do sistema e a memória de usuário. I/O ocorre apenas entre a memória do sistema e o dispositivo de I/O. Para gravar um bloco na fita, primeiro copiamos o bloco para a memória do sistema e depois o gravamos para a fita.

Essa cópia extra pode resultar em custo inaceitavelmente alto. Outra solução é permitir que as páginas sejam travadas na memória. Um bit de trava é associado com todo quadro. Se o quadro estiver travado, ele não poderá ser selecionado para substituição. Nessa abordagem, para gravar um bloco na fita, travamos na me-

Figura 10.18 O motivo pelo qual os quadros usados para I/O devem estar na memória.

mória as páginas que contêm o bloco. O sistema pode continuar como sempre. As páginas travadas não podem ser substituídas. Quando a I/O é concluída, as páginas são destravadas.

Frequentemente, parte ou todo o kernel do sistema operacional é travado na memória. A maioria dos sistemas operacionais não tolera uma falta de página causada pelo kernel. Considere o resultado da rotina de substituição de página que causa a falta de página.

Outro uso para o bit de trava envolve a substituição normal de páginas. Considere a seguinte sequência de eventos. Um processo de baixa prioridade falha. Quando um quadro de substituição é selecionado, o sistema de paginação lê a página necessária na memória. Pronto para continuar, o processo de baixa prioridade entra na fila de processos prontos e espera pela CPU. Como é um processo de baixa prioridade, pode não ser selecionado pelo escalonador por algum tempo. Enquanto o processo de baixa prioridade espera, um processo de alta prioridade falha. Procurando por uma substituição, o sistema de paginação vê uma página que está na memória mas que ainda não foi referenciada nem modificada: é a página que o processo de baixa prioridade acabou de trazer. Essa página parece uma substituição perfeita: está limpa e não precisará ser gravada, e aparentemente não foi usada por um longo período.

Se o processo de alta prioridade pode ou não substituir o de baixa prioridade envolve uma decisão política. Afinal de contas, estamos simplesmente adiando o processo de baixa prioridade em benefício do processo de alta prioridade. Por outro lado, estamos desperdiçando o esforço de trazer a página do processo de baixa prioridade. Se decidirmos evitar a substituição de uma página recém-trazida até que ela possa ser usada pelo menos uma vez, podemos usar o bit de trava para implementar esse mecanismo. Quando a página for selecionada para substituição, seu bit de trava será ativado; permanecerá ativado até que o processo com falta de página receba a CPU novamente.

Usar um bit de trava, no entanto, pode ser perigoso. O bit de trava pode ser ativado, mas nunca desativado. Se essa situação ocorrer (devido a um bug no sistema operacional, por exemplo), o quadro travado se torna inútil. O Sistema Operacional do Macintosh oferece um mecanismo de bloco de operações de página porque é um sistema monousuário e o uso excessivo do bloco de operações afeta apenas o usuário que efetua o bloco de operações. Os sistemas multiusuários devem confiar menos nos usuários. Por exemplo, o Solaris permite "indicações" de bloco de operações, mas é livre para desconsiderar essas indicações se o pool de quadros livres se tornar pequeno demais ou se um processo individual solicitar o bloco de operações de um número excessivo de páginas na memória.

10.7.6 Processamento de tempo real

As discussões neste capítulo giraram em torno de fornecer a melhor utilização geral de um sistema de computador otimizando o uso da memória. Usando a memória para dados ativos e movendo os dados inativos para o disco, é possível aumentar o throughput geral do sistema. Entretanto, processos individuais podem sofrer como resultado disso, porque agora causam mais faltas de página durante sua execução.

Considere um processo ou thread de tempo real, como descrito no Capítulo 4. Esse processo espera obter controle da CPU e executar até sua conclusão com um mínimo de atrasos. A memória virtual é a antítese da computação de tempo real, porque pode introduzir atrasos inesperados e longos na execução de um processo, enquanto as páginas são levadas para a memória. Portanto, os sistemas de tempo real quase nunca têm memória virtual.

No caso do Solaris 2, os desenvolvedores da Sun Microsystems quiseram oferecer computação de tempo real e de tempo compartilhado em um único sistema. Para resolver o problema de falta de página, incluíram no Solaris 2 um recurso que permite a um processo informar ao sistema quais páginas são importantes para aquele processo. Além de permitir as dicas sobre o uso da página, o sistema operacional permite que usuários privilegiados solicitem o bloco de operações de páginas na memória. Se houver abuso desse mecanismo, ele pode bloquear todos os outros processos fora do sistema. Ele é necessário para permitir que os processos de tempo real tenham latência de dispatch baixa e limitada (previsível).

10.8 • Resumo

É desejável a possibilidade de executar um processo cujo espaço de endereçamento lógico seja maior do que o espaço de endereçamento físico disponível. O programador pode tornar esse processo executável reestruturando-o usando overlays, mas essa é geralmente uma tarefa de programação difícil. A memória virtual é uma técnica que permite o mapeamento de um espaço de endereçamento lógico grande em uma memória física menor. A memória virtual permite que processos extremamente grandes sejam executados, e que o grau de multiprogramação aumente, melhorando a utilização de CPU. Além disso, ela libera os programadores de aplicações da preocupação com a disponibilidade de memória.

A paginação sob demanda pura nunca leva para a memória uma página até ela ser referenciada. A primeira referência causa uma falta de página para o monitor residente do sistema operacional. O sistema operacional consulta uma tabela interna para determinar onde a página está localizada no armazenamento auxiliar. Em seguida, encontra um quadro livre e lê a página do armazenamento auxiliar. A tabela de página é atualizada para refletir essa mudança, e a instrução que causou a falta de página é reiniciada. Essa abordagem permite que um processo execute mesmo que sua imagem de memória completa não esteja na memória principal de uma vez. Desde que a taxa de falta de página seja razoavelmente baixa, o desempenho será aceitável.

Podemos usar a paginação sob demanda para reduzir o número de quadros alocados a um processo. Esse arranjo pode aumentar o grau de multiprogramação (permitindo que mais processos estejam disponíveis para execução ao mesmo tempo) e, ao menos em teoria, aumenta a utilização de CPU do sistema. Permite também que os processos sejam executados mesmo que suas exigências de memória superem a memória física total disponível. Tais processos executam na memória virtual.

Se os requisitos de memória total excederem a memória física, pode ser necessário substituir as páginas da memória para liberar quadros para novas páginas. Vários algoritmos de substituição de página são usados. A substituição de página FIFO é fácil de programar, mas sofre da anomalia de Belady. A substituição de página ótima requer conhecimento futuro. A substituição LRU é uma aproximação da substituição ótima, mas mesmo ela pode ser difícil de implementar. A maioria dos algoritmos de substituição de página, tais como o de segunda chance, são aproximações da substituição LRU.

Além de um algoritmo de substituição de página, é necessária uma política de alocação de quadros. A alocação pode ser fixa, sugerindo a substituição de página local, ou dinâmica, sugerindo a substituição global. O modelo de conjunto de trabalho assume que os processos executam em localidades. O conjunto de trabalho é o conjunto de páginas na localidade atual. Da mesma forma, cada processo deve receber um número de quadros suficientes para seu conjunto de trabalho atual.

Se um processo não tiver memória suficiente para seu conjunto de trabalho, ele entrará em thrashing. Fornecer quadros suficientes para cada processo a fim de evitar o thrashing pode exigir swapping e o escalonamento de processos.

Além de exigir a resolução dos principais problemas de substituição de página e alocação de quadros, o projeto adequado de um sistema de paginação requer que consideremos o tamanho de página, I/O, travamento, pré-paginação, estrutura dos programas e outros tópicos. A memória virtual pode ser considerada um nível de uma hierarquia de níveis de armazenamento em um sistema de computação. Cada nível tem seu próprio tempo de acesso, tamanho e parâmetros de custo. Um exemplo completo de um sistema de memória virtual funcional e híbrido está apresentado no capítulo sobre o Mach, que está disponível no nosso Website (http://wvm.bell4abs.com/topic/books/os-book).

• Exercícios

10.1 Em que circunstâncias ocorrem faltas de página? Descreva as ações tomadas pelo sistema operacional quando ocorre uma falta de página.

10.2 Suponha que você possui um string de referência de página para um processo com *m* quadros (inicialmente vazios). O string de referência de página tem tamanho *p*\ *n* números de página diferentes ocorrem nele. Responda essas perguntas para qualquer algoritmo de substituição de página:

a. Qual é o limite inferior no número de faltas de página?
b. Qual é o limite superior no número de faltas de página?

10.3 Determinado computador fornece a seus usuários um espaço de memória virtual de 2^{32} bytes. O computador tem 2^{18} bytes de memória física. A memória virtual é implementada por paginação e o tamanho de página é 4.096 bytes. Um processo de usuário gera o endereço virtual 11123456. Explique como o sistema estabelece a posição física correspondente. Faça a distinção entre as operações de software e hardware.

10.4 Quais das seguintes técnicas e estruturas de programação são "adequadas" para um ambiente de paginação sob demanda? Quais são "inadequadas"? Explique suas respostas.

a. Pilha
b. Tabela de símbolos com hashing
c. Busca sequencial
d. Busca binária
e. Código puro
f. Operações de vetor
g. Indireção

10.5 Considere que temos uma memória com paginação sob demanda. A tabela de página é mantida em registradores. São necessários 8 milissegundos para o serviço de uma falta de página se um quadro vazio estiver disponível ou se a página substituída não estiver modificada, e 20 milissegundos se a página substituída estiver modificada. O tempo de acesso à memória é 100 nanossegundos.

Vamos supor que a página a ser substituída esteja modificada 70% das vezes. Qual é a taxa de falta de página aceitável máxima para o tempo efetivo de acesso de no máximo 200 nanossegundos?

10.6 Considere os seguintes algoritmos de substituição de página. Classifique os algoritmos em uma escala de cinco pontos de "ruim" a "perfeito" de acordo com sua taxa de falta de página. Separe os algoritmos que sofrem da anomalia de Belady dos que não sofrem.

a. Substituição LRU
b. Substituição FIFO
c. Substituição ótima
d. Substituição de segunda chance

10.7 Quando a memória virtual é implementada em um sistema de computação, ela incorre em certos custos e benefícios. Liste esses custos e benefícios. É possível que os custos excedam os benefícios. Explique que medidas podem ser tomadas para garantir que esse desequilíbrio não ocorra.

10.8 Um sistema operacional suporta uma memória virtual paginada, usando um processador central com um tempo de ciclo de 1 microssegundo. É necessário 1 microssegundo a mais para acessar uma outra página (que não a atual). As páginas têm 1.000 palavras, e o dispositivo de paginação é um disco que gira a 3.000 rotações por minuto e transfere 1 milhão de palavras por segundo. As seguintes medidas estatísticas foram obtidas do sistema:

- 1% de todas as instruções executadas acessaram uma página que não a página atual.
- Das instruções que acessaram outra página, 80% acessaram uma página já na memória.
- Quando uma nova página era exigida, a página substituída estava modificada 50% das vezes.

Calcule o tempo efetivo de instrução do sistema, considerando que o sistema está executando apenas um processo e que o processador está ocioso durante as transferências de disco.

10.9 Considere um sistema de paginação sob demanda com as seguintes medidas de tempo de utilização:

Utilização de CPU	20%
Disco de paginação	97,7%
Outros dispositivos de I/O	5%

Para cada um dos itens a seguir, informe se a utilização de CPU será (ou poderá ser) melhorada ou não. Explique suas respostas.

a. Instalar uma CPU mais rápida.
b. Instalar um disco de paginação maior.
c. Aumentar o grau de multiprogramação.
d. Diminuir o grau de multiprogramação.
e. Instalar mais memória principal.
f. Instalar um disco rígido mais rápido, ou múltiplas controladoras com múltiplos discos rígidos.
g. Adicionar pré-paginação aos algoritmos de busca de página.
h. Aumentar o tamanho da página.

10.10 Considere o vetor bidimensional A:

```
int A[ ][ ] = new int[100] [100];
```

onde A [0] [0] está na posição 200, em um sistema paginado com páginas de tamanho 200. Um processo pequeno está na página 0 (posições 0 a 199) para manipular a matriz; assim, toda a busca de instruções será a partir da página 0.

Para três quadros de página, quantas faltas de página são geradas pelos seguintes laços de inicialização do vetor, usando a substituição LRU e assumindo que o quadro de página 1 tem o processo e que os outros dois estão inicialmente vazios:

a.
```
for (int j = 0; j < 100; j++)
    for (int i = 0; i < 100; i++)
      A[i][j] = 0;
```

b.
```
for (int i = 0; i < 100; i++)
    for (int j = 0; j < 100; j++)
      A[i][j] - 0;
```

10.11 Considere o seguinte string de referência de página:

1,2,3,4,2,1,5,6,2,1,2,3,7,6,3,2,1,2,3,6

Quantas faltas de página ocorreriam para os seguintes algoritmos de substituição, considerando um, dois, três, quatro, cinco, seis ou sete quadros? Lembre-se de que todos os quadros estão inicialmente vazios, de modo que suas primeiras páginas todas custarão uma falta cada.

- Substituição LRU
- Substituição FIFO
- Substituição ótima

10.12 Vamos supor que você deseja utilizar um algoritmo de paginação que exija um bit de referência (como a substituição de segunda chance ou o modelo de conjunto de trabalho), mas o hardware não fornece um. Explique como você simularia um bit de referência mesmo se ele não fosse fornecido pelo hardware, ou explique por que não é possível fazê-lo. Se for possível, calcule seu custo.

10.13 Você elaborou um novo algoritmo de substituição de página que, na sua opinião, pode ser ótimo. Em alguns casos de teste extremos, ocorre a anomalia de Belady. O novo algoritmo é ótimo? Explique.

10.14 Vamos supor que a sua política de substituição (em um sistema paginado) seja examinar cada página regularmente e descartar essa página se ela não tiver sido usada desde o último exame. Quais seriam as vantagens e desvantagens da utilização dessa política em vez da substituição LRU ou de segunda chance?

10.15 A segmentação é semelhante à paginação, mas utiliza "páginas" de tamanho variável. Defina dois algoritmos de substituição de segmentos com base nos esquemas de substituição de página FIFO e LRU. Lembre-se de que, como os segmentos não são do mesmo tamanho, o segmento que é escolhido para substituição pode não ser grande o suficiente para deixar um número suficiente de posições consecutivas para o segmento necessário. Considere as estratégias para os sistemas nos quais os segmentos não possam ser relocados e as de sistemas onde a relocação seja possível.

10.16 Um algoritmo de substituição de página deve minimizar o número de faltas de página. Podemos fazer essa redução distribuindo páginas muito utilizadas de maneira uniforme em toda a memória, em vez de fazer com que venham a competir por um pequeno número de quadros de página. Podemos associar com cada quadro de página um contador do número de páginas que estão relacionadas com esse quadro. Em seguida, para substituir uma página, pesquisamos o quadro de página com o menor contador.

a. Defina um algoritmo de substituição de página usando essa ideia básica. Aborde especificamente os seguintes problemas (1) qual é o valor inicial dos contadores, (2) quando os contadores são incrementados, (3) quando os contadores são decrementados e (4) como a página a ser substituída é selecionada.

b. Quantas faltas de página ocorrem no seu algoritmo para o seguinte string de referência, para quatro quadros de página?

1,2,3,4,5,3,4,1,6,7,8,7,8,9,7,8,9,5,4,5,4,2

c. Qual é o número mínimo de faltas de página para uma estratégia de substituição de página ótima para o string de referência do item b, com quatro quadros de página?

10.17 Considere um sistema de paginação sob demanda com um disco de paginação que tenha um tempo de acesso e transferência médios de 20 milissegundos. Os endereços são traduzidos por meio de uma tabela de página na memória principal, com tempo de acesso de 1 microssegundo por acesso à memória. Assim, cada referência de memória através da tabela de página envolve dois acessos. Para melhorar esse tempo, acrescentamos memória associativa que reduz o tempo de acesso a uma referência de memória, se a entrada na tabela de página estiver na memória associativa.

Suponha que 80% dos acessos estão na memória associativa e que, do restante, 10% (ou 2% do total) causam faltas de página. Qual é o tempo efetivo de acesso à memória?

10.18 Considere um sistema de computação de paginação sob demanda no qual o grau de multiprogramação está fixado no momento em quatro. O sistema foi medido recentemente para determinar a utilização da CPU e do disco de paginação. Os resultados são uma das seguintes alternativas. Para cada caso, o que está acontecendo? É possível aumentar o grau de multiprogramação para aumentar a utilização de CPU? A paginação está ajudando a melhorar o desempenho?

a. Utilização de CPU, 13%; utilização de disco, 97%

b. Utilização de CPU, 87%; utilização de disco, 3%

c. Utilização de CPU, 13%; utilização de disco, 3%

10.19 Temos um sistema operacional para uma máquina que utiliza os registradores de base e de limite, mas a máquina foi modificada para fornecer uma tabela de página. É possível configurar as tabelas de página para simular os registradores de base e de limite? Como podemos fazer isso, ou por que não podemos?

10.20 Qual é a causa do thrashing? Como o sistema detecta o thrashing? Uma vez detectado, o que o sistema pode fazer para eliminar esse problema?

10.21 Escreva um programa Java que implemente os algoritmos de substituição de página FIFO e LRU apresentados neste capítulo. Em primeiro lugar, gere um string de referência de página aleatório no qual os números de página variem de 0 a 9. Aplique o string a cada algoritmo e registre o número de faltas de página incorridas por cada algoritmo. Implemente os algoritmos de substituição de modo que o número de quadros de página possa variar de 1 a 7. Suponha que a paginação sob demanda é usada.

Notas bibliográficas

A paginação sob demanda foi usada pela primeira vez no sistema Atlas, implementado no computador MUSE da Manchester University por volta de 1960 [Kilburn et ai. 1961]. Outro sistema pioneiro de paginação sob demanda foi o MULTICS, implementado no sistema GE 645 [Organick 1972].

Belady e colegas [1969] foram os primeiros pesquisadores a observar que a estratégia de substituição FIFO poderia ter a anomalia cunhada por Belady. Mattson e colegas [1970] demonstraram que os algoritmos de pilha não são sujeitos à anomalia de Belady.

O algoritmo de substituição ótima foi apresentado por Belady [1966]. Foi comprovado como sendo ótimo por Mattson e colegas [1970]. O algoritmo ótimo de Belady é para uma alocaçâo fixa; Prieve e Fabry [1976] apresentaram um algoritmo ótimo para situações nas quais a alocaçâo pode variar.

O algoritmo de clock melhorado foi discutido por Carr e Hennessy [1981]; ele é usado no esquema de gerência de memória virtual do Macintosh e foi descrito por Goldman [1989].

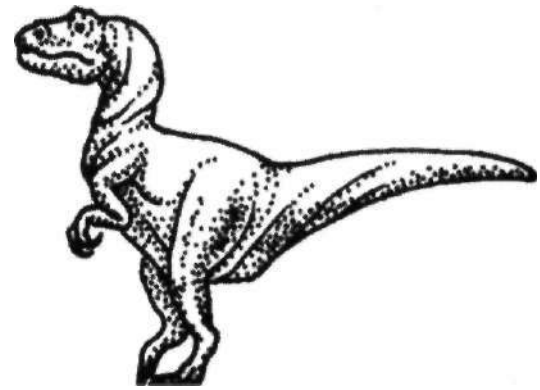
O modelo de conjunto de trabalho foi desenvolvido por Denning [1968]. As discussões relativas ao modelo foram apresentadas por Denning [1980].

O esquema para monitorar a taxa de falta de página foi desenvolvido por Wulf [1969], que aplicou com sucesso essa técnica ao sistema de computação Burroughs B5500. Gupta e Franklin [1978] forneceram uma comparação de desempenho entre o esquema de conjunto de trabalho e o esquema de substituição de frequência de falta de página.

Solomon [1998] descreve como o Windows NT implementa a memória virtual. Graham [1995] e Vahalia [1996] discutem a memória virtual no Solaris. Detalhes do OS/2 foram descritos por Iacobucci [1988].

Boas discussões estão disponíveis para o hardware de paginação Intel 80386 [Intel 1986] e o hardware Motorola 68030 [Motorola 1989b]. A gerência de memória virtual no sistema operacional VAX/VMS foi discutido por Levy e Lipman [1982]. Discussões relativas a sistemas operacionais de estação de trabalho e memória virtual foram apresentadas por Hagmman [1989]. Uma comparação de uma implementação de memória virtual nas arquiteturas MIPS, PowerPC e Pentium pode ser encontrada em Jacob e Mudge [1998a]. Um artigo complementar (Jacob e Mudge [1998b]) descreve o suporte de hardware necessário para a implementação da memória virtual em seis arquiteturas diferentes, incluindo a UltraSPARC.

Capítulo 11



SISTEMAS DE ARQUIVOS

Para a maior parte dos usuários, o sistema de arquivos é o aspecto mais visível de um sistema operacional. Ele fornece o mecanismo para armazenamento online e acesso a dados e programas que pertencem ao sistema operacional e a todos os usuários do sistema de computação. O sistema de **arquivos** consiste em duas partes distintas: uma coleção de arquivos, cada qual armazenando dados correlatos e uma **estrutura de diretório,** que organiza e fornece informações sobre todos os arquivos no sistema. Neste capítulo, consideramos vários aspectos dos arquivos e a diversidade de estruturas de diretório. Também discutimos formas de proporcionar a proteção de arquivos, que é necessária em um ambiente no qual múltiplos usuários têm acesso aos arquivos e onde é geralmente desejável controlar quem acessa os arquivos e de que forma. Além disso, discutimos o armazenamento e o acesso de arquivos no meio do armazenamento secundário mais comum, o disco. Exploramos formas de alocar espaço em disco, recuperar espaço livre, rastrear as posições de dados e interfacear as outras partes do sistema operacional com o armazenamento secundário.

11.1 • Conceito de arquivo

Os computadores podem armazenar informações em vários meios de armazenamento diferentes, tais como discos magnéticos, fitas magnéticas e discos óticos. Para que o sistema de computação seja conveniente de usar, o sistema operacional fornece uma visão lógica uniforme do armazenamento de informações. O sistema operacional abstrai das propriedades físicas de seus dispositivos de armazenamento para definir uma unidade de armazenamento lógica, *o arquivo.* Os arquivos são mapeados, pelo sistema operacional, em dispositivos físicos. Esses dispositivos de armazenamento geralmente são *não-voláteis,* de modo que o conteúdo persiste nos casos de falta de energia e reinicialização do sistema.

Um arquivo é uma coleção de informações correlatas que recebe um nome e é gravado no armazenamento secundário. Da perspectiva do usuário, um arquivo é a menor unidade alocável de armazenamento secundário lógico; ou seja, os dados só podem ser gravados no armazenamento secundário se estiverem em um arquivo. Em geral, os arquivos representam programas (tanto fonte quanto objeto) e dados. Os arquivos de dados podem ser numéricos, alfabéticos, alfanuméricos ou binários. Os arquivos podem ter forma livre, como arquivos de texto, ou podem ter uma formatação rígida. Em geral, um arquivo é uma sequência de bit, bytes, linhas ou registros cujo significado é definido pelo criador e usuário do arquivo. O conceito de arquivo é, portanto, extremamente geral.

As informações em um arquivo são definidas pelo seu criador. Muitos tipos diferentes de informações podem ser armazenadas em um arquivo: programas-fonte, programas-objeto, programas executáveis, dados numéricos, texto, registros de folha de pagamento, imagens gráficas, gravações de áudio, entre outros. Um arquivo tem uma **estrutura** determinada, definida de acordo com seu tipo. Um **arquivo de texto** é uma sequência de caracteres organizados em linhas (e possivelmente páginas); um **arquivo-fonte** é uma sequência de sub-rotinas e funções, cada qual sendo organizada como declarações seguidas por instruções executáveis; um arquivo **objeto** é uma sequência de bytes organizados em blocos compreensíveis pelo linkeditor do siste-

ma; um **arquivo executável** é uma série de seções de código que o carregador pode levar para a memória e executar. A estrutura interna dos arquivos é discutida na Seção 11.1.5.

11.1.1 Atributos de arquivo

Um arquivo recebe um nome, para conveniência de seus usuários humanos, e é referenciado pelo seu nome. Um nome geralmente é um string de caracteres, tais como "exemplo.c". Alguns sistemas fazem distinção entre caracteres maiúsculos e minúsculos nos nomes, enquanto outros sistemas consideram os dois equivalentes. Quando um arquivo recebe um nome, ele se torna independente do processo, do usuário e mesmo do sistema que o criou. Por exemplo, um usuário pode criar o arquivo "exemplo.c", enquanto outro pode editar esse arquivo especificando seu nome. O proprietário do arquivo poderá gravar o arquivo em um disquete ou fita magnética e lê-lo em outro sistema, onde ainda poderia continuar sendo chamado de "exemplo.c".

Um arquivo tem determinados atributos, que variam de um sistema operacional para outro, mas geralmente consistem em:

- *Nome:* O nome simbólico de arquivo é a única informação mantida no formato legível pelo homem.
- *Tipo:* Essas informações são necessárias para os sistemas que suportam tipos diferentes.
- *Posição:* Essas informações são um ponteiro para um dispositivo e para a posição do arquivo no dispositivo.
- *Tamanho:* O tamanho atual do arquivo (em bytes, palavras ou blocos) e possivelmente o tamanho máximo permitido estão incluídos neste atributo.
- *Proteção:* Informações de controle de acesso que controlam quem pode realizar as operações de leitura, escrita, execução etc.
- *Hora, data* e *identificação de usuário:* Essas informações podem ser mantidas para (1) criação, (2) última modificação e (3) último uso. Esses dados podem ser úteis para proteção, segurança e monitoração de uso.

As informações sobre todos os arquivos são mantidas na estrutura de diretório que também reside no armazenamento secundário. De 16 a mais de 1000 bytes podem ser necessários para gravar essas informações para cada arquivo. Em um sistema com muitos arquivos, o tamanho do diretório em si pode ser de megabytes. Já que os diretórios, como os arquivos, devem ser não-voláteis, eles precisam ser armazenados no dispositivo e levados para a memória gradativamente, conforme necessário. A organização da estrutura de diretório será discutida na Seção 11.3.

11.1.2 Operações com arquivos

Um arquivo é um **tipo abstrato de dados.** Para definir corretamente os arquivos, precisamos considerar as operações que podem ser realizadas sobre eles. O sistema operacional fornece chamadas ao sistema para criar, escrever, ler, reposicionar, excluir e truncar arquivos. Vamos considerar o que o sistema operacional deve fazer para cada uma das seis operações básicas. Depois então, será fácil ver como implementar operações semelhantes, por exemplo, renomear um arquivo.

- *Para criar um arquivo:* Um arquivo é criado em duas etapas. Primeiro, deve haver espaço no sistema de arquivos para o arquivo. O Capítulo 11 discute como alocar espaço para o arquivo. Em segundo lugar, uma entrada para o novo arquivo deve ser feita no diretório. A entrada no diretório registra o nome do arquivo e sua localização no sistema de arquivos.
- *Para escrever em um arquivo:* Para escrever em um arquivo, fazemos uma chamada ao sistema especificando o nome do arquivo e as informações a serem escritas nele. Dado o nome do arquivo, o sistema pesquisa o diretório para encontrar a sua posição. O sistema deve manter um ponteiro de escrita para a posição no arquivo onde ocorrerá a próxima escrita. O ponteiro de escrita deve ser atualizado sempre que ocorrer uma escrita.
- *Para ler um arquivo:* Para ler um arquivo, é preciso usar uma chamada ao sistema que especifique o nome do arquivo e onde (na memória) o próximo bloco do arquivo deverá ser colocado. Mais uma

vez, o diretório é pesquisado para encontrar a entrada de diretório associada, e o sistema mantém um **ponteiro de leitura** para a posição no arquivo na qual a próxima leitura deverá ocorrer. Assim que a leitura ocorrer, o ponteiro de leitura será atualizado. Como, em geral, um arquivo está sendo gravado ou lido, a maioria dos sistemas mantém apenas um **ponteiro de posição atual do arquivo.** As operações de leitura e escrita utilizam esse mesmo ponteiro, economizando espaço e reduzindo a complexidade do sistema.

- *Para reposicionar dentro do arquivo:* O diretório é pesquisado buscando a entrada apropriada, e a posição do arquivo atual é ajustada para um determinado valor. Reposicionar dentro de um arquivo não precisa envolver operações de I/O. Essa operação de arquivo também é chamada de **busca no arquivo.**
- *Para excluir um arquivo:* Para excluir um arquivo, pesquisamos no diretório o arquivo identificado pelo nome. Ao encontrar a entrada de diretório associada, liberamos todo o espaço de arquivo (para que ele possa ser reutilizado por outros arquivos) e apagamos a entrada do diretório.
- *Para truncar um arquivo:* Existem momentos em que o usuário deseja que os atributos de um arquivo permaneçam os mesmos, mas quer apagar o conteúdo do arquivo. Em vez de forçar o usuário a excluir **o** arquivo e, em seguida, recriá-lo, essa função permite que todos os atributos permaneçam inalterados (exceto pelo tamanho do arquivo), mas que o arquivo seja reajustado para o tamanho zero.

Essas seis operações básicas certamente compreendem o conjunto mínimo exigido de operações com arquivos. Outras operações comuns incluem **anexar** *{append)* novas informações ao final de um arquivo existente e **renomear** um arquivo existente. Essas operações primitivas podem então ser combinadas para realizar outras operações com arquivos. Por exemplo, podemos criar uma **cópia** de um arquivo, ou copiar o arquivo para outro dispositivo de I/O, como uma impressora ou monitor, criando um novo arquivo e, em seguida, lendo do antigo e escrevendo no novo. Também queremos ter operações que permitam que o usuário obtenha e defina os vários atributos de um arquivo. Por exemplo, talvez tenhamos uma operação que permita ao usuário determinar o status de um arquivo, tais como o tamanho do arquivo, e outra operação que permita ao usuário definir os atributos de arquivo, tais como o proprietário do arquivo.

Boa parte das operações de arquivo mencionadas envolvem pesquisar o diretório pela entrada associada com o arquivo nomeado. Para evitar essa pesquisa constante, muitos sistemas vão inicialmente **abrir** um arquivo quando ele for usado ativamente. O sistema operacional mantém uma pequena tabela contendo informações sobre todos os arquivos abertos: a **tabela de arquivos abertos** *(open-file tablé).* Quando uma operação de arquivo é solicitada, um índice para essa tabela é usado, de modo que não há necessidade de pesquisa. Quando o arquivo não estiver mais sendo utilizado ativamente, ele será **fechado** pelo processo, e o sistema operacional removerá sua entrada da tabela de arquivos abertos.

Alguns sistemas abrem implicitamente um arquivo quando a primeira referência a ele é feita. O arquivo é automaticamente fechado quando o job ou o programa que abriu o arquivo termina. A maioria dos sistemas, no entanto, exigem que um arquivo seja aberto explicitamente pelo programador com uma chamada ao sistema (open) antes que ele possa ser usado. A operação open pega um nome de arquivo e pesquisa no diretório, copiando a entrada de diretório na tabela de arquivos abertos, supondo que as proteções de arquivo permitem esse acesso. A chamada ao sistema open geralmente retornará um ponteiro à entrada na tabela de arquivos abertos. Esse ponteiro, em vez do nome do arquivo em si, é usado em todas as operações de I/O, evitando pesquisas adicionais e simplificando a interface de chamada ao sistema.

A implementação das operações open e cl ose em um ambiente multiusuário, como o UNIX, é mais complicada. Nesse sistema, vários usuários podem abrir o arquivo ao mesmo tempo. Em geral, o sistema de arquivos utiliza dois níveis de tabelas internas. Existe uma tabela local, por processo, de todos os arquivos que o processo já abriu. Armazenadas nessa tabela estão informações relativas ao uso do arquivo pelo processo. Por exemplo, o ponteiro atual de arquivo para cada arquivo é encontrado aqui, indicando a posição no arquivo que a próxima chamada read ou escrever afetará.

Cada entrada na tabela local, por sua vez, aponta para uma tabela global de arquivos abertos *(systemwide open-file tablé).* A tabela global, que abrange todo o sistema, contém informações independentes de processo, tais como a posição do arquivo no disco, as datas de acesso e o tamanho do arquivo. Assim que um arquivo for aberto por um processo, outro processo executando uma chamada open simplesmente resultará no acréscimo

de uma nova entrada à tabela local de arquivos abertos do processo com um novo ponteiro de arquivo atual e um ponteiro para a entrada apropriada na tabela global do sistema. Em geral, a tabela global de arquivos abertos também tem um **contador de aberturas** *(open count)* associado a cada arquivo, indicando o número de processos que têm o arquivo aberto. Cada chamada dose diminui esse contador e, quando o contador chegar a zero, o arquivo não estará mais em uso, e a entrada será removida da tabela global de arquivos abertos.

Em resumo, existem várias informações diferentes relacionadas com um arquivo aberto.

- *Ponteiro de arquivo:* Em sistemas que não incluem um deslocamento *(offset)* de arquivo como parte das chamadas ao sistema read e escrever, será preciso rastrear a última posição de leitura e escrita como um ponteiro de posição atual no arquivo. Esse ponteiro é exclusivo para cada processo que opera no arquivo e, portanto, deve ser mantido separado dos atributos de arquivo no disco.
- *Contador de aberturas de arquivo:* A medida que os arquivos são fechados, o sistema operacional deverá reutilizar suas entradas na tabela de arquivos abertos, ou poderá acabar ficando sem espaço na tabela. Como múltiplos processos podem abrir um arquivo, o sistema deverá esperar que o último processo feche o arquivo antes de remover a entrada na tabela de arquivos abertos. Esse contador rastreia o número de procedimentos de abertura e fechamento, e chega a zero na última operação de fechamento. O sistema poderá então remover a entrada.
- *Posição do arquivo no disco:* A maior parte das operações de arquivo exigem que o sistema modifique os dados no arquivo. As informações necessárias para localizar o arquivo no disco são mantidas na memória para evitar ter de lê-las do disco para cada operação.

Alguns sistemas operacionais fornecem recursos para bloquear seções de um arquivo aberto para acesso de múltiplos processos, para compartilhar seções de um arquivo entre vários processos e até mesmo para mapear as seções de um arquivo na memória em sistemas de memória virtual. Esta última função é chamada de **mapeamento de um arquivo em memória;** ela permite que uma parte do espaço de endereçamento virtual seja associado logicamente com uma seção de um arquivo. As operações de leitura e escrita a essa região da memória são então tratadas como operações de leitura e escrita no arquivo, simplificando em muito a utilização de arquivos. Fechar o arquivo resulta em escrever todos os dados mapeados na memória de volta no disco, removendo-os da memória virtual do processo. Múltiplos processos poderão mapear o mesmo arquivo na memória virtual de cada um, para permitir o compartilhamento de dados. As operações de escrita de qualquer processo modificam os dados na memória virtual e podem ser vistas por todos os outros processos que mapeiam a mesma seção do arquivo. Considerando nosso conhecimento de memória virtual adquirido no Capítulo 10, deve ser fácil entender como o compartilhamento de seções mapeadas na memória é implementado. O mapa de memória virtual de cada processo que participa do compartilhamento aponta para a mesma página de memória física - a página que mantém uma cópia do bloco de disco. Esse compartilhamento de memória é ilustrado na Figura 11.1. Para que o acesso aos dados compartilhados seja coordenado, os processos envolvidos podem usar um dos mecanismos para obter a exclusão mútua descritos no Capítulo 7.

11.1.3 Tipos de arquivos >

Uma consideração importante no projeto de um sistema de arquivos, e de todo sistema operacional, é se o sistema deverá reconhecer e oferecer suporte a tipos de arquivo. Se um sistema operacional reconhecer o tipo de um arquivo, ele poderá operar com o arquivo de forma razoável. Por exemplo, um erro comum ocorre quando um usuário tenta imprimir a forma objeto-binária de um programa. Essa tentativa normalmente gera lixo, mas a impressão desse arquivo poderá ser evitada *se* o sistema operacional tiver sido informado que o arquivo é um programa objeto-binário.

Uma técnica comum para implementar os tipos de arquivo é incluir o tipo como parte do nome do arquivo. O nome é dividido em duas partes - um nome e uma extensão, geralmente separada por um caractere de ponto (Figura 11.2). Dessa forma, o usuário e o sistema operacional podem saber imediatamente a partir do nome qual é o tipo de arquivo em questão. Por exemplo, no MS-DOS, um nome pode consistir em até oito caracteres seguido por um ponto e terminado por uma extensão de até três caracteres. O sistema utiliza a extensão para indicar o tipo de arquivo e o tipo de operações que podem ser realizadas com aquele arquivo. Por exemplo, apenas arquivos com extensão ".com", ".exe" ou ".bat" podem ser executados. Os arquivos

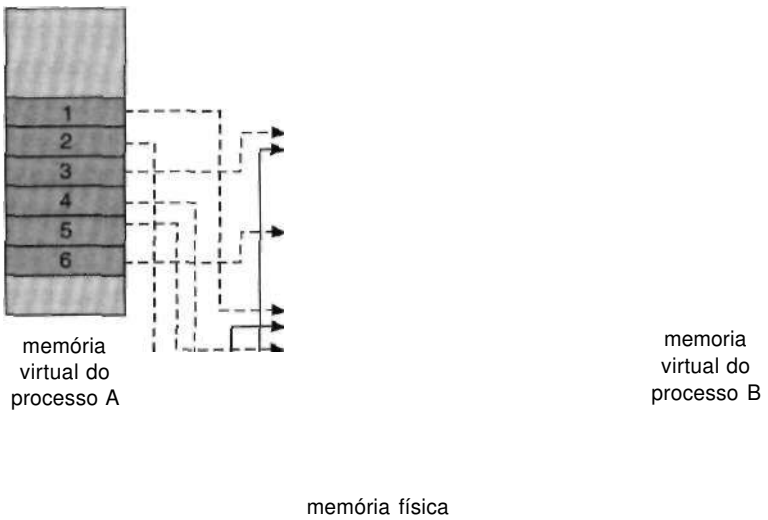


Figura 11.1 Arquivos mapeados em memória.

".com" e ".exe" são duas formas de arquivos binários executáveis, enquanto um arquivo ".bat" é um arquivo **batch** (lote) que contém comandos para o sistema operacional em formato ASCII. O MS-DOS reconhece apenas algumas extensões de arquivos, mas os programas aplicativos as utilizam para indicar os tipos de arquivos nos quais estão interessados. Por exemplo, um processador de textos pode criar arquivos com a extensão ".doe". O sistema operacional deverá então responder à Seleção desse arquivo (uma chamada open, por exemplo) chamando o processador de textos e dando a ele o arquivo como entrada. Essas extensões podem ser suprimidas se o sistema operacional o permitir, mas podem economizar tempo de digitação e ajudar os usuários a entenderem o tipo de cada arquivo. Alguns sistemas operacionais incluem suporte a extensão, enquanto outros deixam a cargo das aplicações mudar seu comportamento com base na extensão. Neste último caso, as extensões podem ser consideradas dicas para as aplicações que operam com elas.

tipo de arquivo	extensão comum	função
executável	exe, com, bin, ou nada	programa de linguagem de máquina pronto para executar
objeto	obj, o	linguagem de máquina, compilado, sem linkedição
código-fonte	c, cc, pas, java, asm, a	código fonte em várias linguagens
batch	bat, sh	comandos para o interpretador de comandos
texto	txt, doe	dados textuais, documentos
processador de textos	wpd, tex, doe, etc	vários formatos de processador de textos
biblioteca	lib, a, DLL	bibliotecas de rotinas para programadores
impressão ou visualização	ps, dvi, gif	arquivo ASCII ou binário em um formato para impressão ou visualização
arquivo compactado	are, zip, tar	arquivos correlatos agrupados em um arquivo único, às vezes compactado, para fins de arquivamento ou armazenamento

Figura **11.2** Tipos de arquivo comuns.

Outro exemplo da utilidade dos tipos de arquivo vem do sistema operacional TOPS-20. Se o usuário tentar executar um programa objeto cujo arquivo-fonte tenha sido modificado (editado) desde que o arquivo objeto foi gerado, o arquivo-fonte será recompilado automaticamente. Essa função garante que o usuário sempre executará um arquivo objeto atualizado. Caso contrário, o usuário poderia perder um tempo precioso executando o arquivo objeto antigo. Observe que, para que essa função seja possível, o sistema operacional deverá fazer a distinção entre o arquivo-fonte e o arquivo objeto para verificar a hora em que cada arquivo foi modificado pela última vez ou criado, e para determinar a linguagem do programa fonte (para que possa utilizar o compilador correto).

Considere o sistema operacional do Apple Macintosh. Nesse sistema, cada arquivo tem um tipo, como "text" ou "pict". Cada arquivo também tem um atributo de criador, contendo o nome do programa que o criou. Esse atributo é definido pelo sistema operacional durante a chamada create, portanto, seu uso é imposto e suportado pelo sistema. Por exemplo, um arquivo produzido por um processador de textos tem o nome do processador como seu criador. Quando o usuário abre esse arquivo, clicando duas vezes com o mouse no ícone que representa o arquivo, o processador de textos será chamado automaticamente, e o arquivo será carregado, pronto para edição.

O sistema UNIX não oferece esse recurso porque utiliza um número mágico bruto armazenado no início de alguns arquivos para indicar basicamente o tipo de arquivo: programa executável, arquivo batch (conhecido como *script de shell),* arquivo PostScript e assim por diante. Nem todos os arquivos têm números mágicos, por isso os recursos do sistema não podem ser baseados unicamente nesse tipo de informação. O UNIX também não registra o nome do programa de criação. Ele permite a existência de dicas sobre as extensões de nome de arquivo, mas essas extensões não são impostas nem implementadas pelo sistema operacional; basicamente, são usadas para orientar os usuários na determinação do tipo de conteúdo do arquivo.

11.1.4 Estrutura de arquivos

Também é possível utilizar tipos de arquivo para indicar a estrutura interna do arquivo. Como mencionado na Seção 11.1.3, os arquivos-fonte e objeto têm estruturas que correspondem às expectativas dos programas que os lêem. Além disso, certos arquivos devem se adequar a uma determinada estrutura que seja compreendida pelo sistema operacional. Por exemplo, o sistema operacional poderá exigir que um arquivo executável tenha uma estrutura específica de modo que possa determinar onde na memória o arquivo deverá ser carregado e qual a posição da primeira instrução. Alguns sistemas operacionais estendem essa ideia a um conjunto de estruturas de arquivos suportadas pelo sistema, com conjuntos de operações especiais para manipular arquivos com essas estruturas. Por exemplo, o popular sistema operacional VMS da DEC tem um sistema de arquivos que oferece suporte a múltiplas estruturas de arquivos. Ele define três estruturas de arquivos.

Nossa discussão aponta para uma das desvantagens do suporte por parte do sistema operacional a múltiplas estruturas de arquivos: o sistema operacional resultante é pesado demais. Se o sistema operacional definir cinco estruturas de arquivo diferentes, ele precisará conter o código para suportar essas estruturas de arquivos. Além disso, cada arquivo poderá precisar ser definido como um dos tipos de arquivo aceitos pelo sistema operacional. Graves problemas poderão resultar de novas aplicações que exijam que as informações sejam estruturadas de formas não suportadas pelo sistema operacional.

Por exemplo, vamos supor que um sistema ofereça suporte a dois tipos de arquivos: arquivos de texto (compostos de caracteres ASCII separados por um caractere carriage-return e um avanço de linha) e arquivos executáveis binários. Agora, se nós (como usuários) desejarmos definir um arquivo criptografado para proteger nossos arquivos contra leitura por pessoas não-autorizadas, talvez nenhum dos tipos de arquivo seja adequado. O arquivo criptografado não consiste em linhas de texto ASCII, mas em bits (aparentemente) aleatórios. Embora possa parecer ser um arquivo binário, ele não é executável. Como resultado, poderemos ter de contornar ou usar indevidamente o mecanismo de tipos de arquivo do sistema operacional, ou modificar ou abandonar nosso esquema de criptografia.

Alguns sistemas operacionais impõem (e suportam) um número mínimo de estruturas de arquivo. Essa abordagem tem sido adotada no UNIX, MS-DOS e outros. O UNIX considera cada arquivo uma sequência de bytes com oito bits; não há interpretação desses bytes pelo sistema operacional. Esse esquema fornece flexibilidade máxima, mas pouco suporte. Cada programa aplicativo deverá incluir seu próprio código para in-

terpretar um arquivo de entrada na estrutura adequada. No entanto, todos os sistemas operacionais deverão suportar pelo menos uma estrutura - a de um arquivo executável - para que o sistema possa carregar e executar programas.

Outro exemplo de um sistema operacional que oferece suporte a um número mínimo de estruturas de arquivo é o Sistema Operacional Macintosh, que espera que os arquivos contenham duas partes: **o ramo de recursos** e **o ramo de dados.** O ramo de recursos contém as informações de interesse para o usuário. Por exemplo, ele mantém as identificações de qualquer botão exibido pelo programa. Um usuário estrangeiro talvez queira renomear esses botões no seu próprio idioma, e o Sistema Operacional Macintosh fornece as ferramentas para permitir a modificação dos dados no ramo de recursos. O ramo de dados contém código de programa ou dados: o conteúdo tradicional dos arquivos. Para realizar a mesma tarefa em um sistema UNIX ou MS-DOS, o programador precisaria alterar e recompilar o código-fonte, a menos que tivesse criado seu próprio arquivo de dados alterável pelo usuário. A lição desse exemplo é que é útil para um sistema operacional oferecer suporte a estruturas que serão usadas com frequência e que pouparão trabalho do programador. Poucas estruturas tornam a programação inconveniente, enquanto o excesso de estruturas pode aumentar em muito o tamanho dos sistemas operacionais e causar confusão ao programador.

11.1.5 Estrutura interna dos arquivos

Internamente, localizar um deslocamento em um arquivo pode ser complicado para o sistema operacional. Lembre-se do Capítulo 2 que os sistemas de disco geralmente têm um tamanho de bloco bem definido determinado pelo tamanho de um setor.|,Todas as operações de I/O de disco são realizadas em unidades de um bloco (registro físico), e todos os blocos têm o mesmo tamanho.jÉ pouco provável que o tamanho do registro físico seja exatamente igual ao tamanho do registro lógico desejado.| Os registros lógicos podem até variar em tamanho. **Agrupar** alguns registros lógicos em blocos físicos é uma solução comum para esse problema./

Por exemplo, o sistema operacional UNIX define todos os arquivos simplesmente como um fluxo de bytes. Cada byte é individualmente endereçável pelo seu deslocamento a partir do início (ou fim) do arquivo. Nesse caso, o registro lógico é 1 byte. O sistema de arquivos agrupa e desagrupa automaticamente os bytes em blocos de disco físico (digamos 512 bytes por bloco), conforme necessário.

j O tamanho do registro lógico, o tamanho do bloco físico e a técnica de agrupamento determinam quantos registros lógicos existem em cada bloco físico. O agrupamento pode ser feito pelo programa aplicativo do iusuário ou pelo sistema operacional. /

*V^* Em ambos os casos, o arquivo pode ser considerado uma sequência de blocos. Todas as funções de I/O básicas operam em termos de blocos. A conversão dos registros lógicos em blocos físicos é um problema de software relativamente simples.

Observe que o espaço de disco sendo sempre alocado em blocos tem como resultado o possível desperdício de alguma parte do último bloco de cada arquivo. Se cada bloco tiver 512 bytes, um arquivo de 1.949 bytes ocuparia quatro blocos (2.048 bytes); os 99 bytes finais seriam desperdiçados. Os bytes desperdiçados alocados para manter tudo em unidades de blocos (em vez de bytes) é a **fragmentação interna.** Todos os sistemas de arquivos sofrem de fragmentação interna; quanto maior o tamanho do bloco, maior a fragmentação interna.

11.1.6 Semântica de consistência

A semântica de consistência é um critério importante para avaliar qualquer sistema de arquivos que suporte o compartilhamento de arquivos. ;É uma caracterização do sistema que especifica a semântica de vários usuários acessando ao mesmo tempo um arquivo compartilhado. Especificamente, essa semântica deve especificar quando as modificações de dados de um usuário são observáveis por outros usuários. Existem várias semânticas de consistência diferentes. Descrevemos aquela utilizada no UNIX.

O sistema de arquivos do UNIX (consulte o Capítulo 17) utiliza a seguinte semântica de consistência:

- As operações de escrita em um arquivo aberto por um usuário são imediatamente visíveis a outros usuários que tenham esse arquivo aberto ao mesmo tempo.

- Existe um modo de compartilhamento no qual os usuários compartilham o ponteiro da posição atual no arquivo. Assim, o avanço do ponteiro por um usuário afeta todos os usuários compartilhados. Aqui, um arquivo tem uma única imagem que se intercala em todos os acessos, independentemente da sua origem.

Essas semânticas às vezes se prestam a uma implementação na qual um arquivo é associado a uma imagem física única que é acessada como um recurso exclusivo. A disputa por essa imagem única resulta em atrasos nos processos de usuário.

11.2 • Métodos de acesso

Os arquivos armazenam informações. Quando são usadas, essas informações devem ser acessadas e lidas na memória do computador. Existem várias formas das informações no arquivo a serem acessadas. Alguns sistemas fornecem apenas um método de acesso aos arquivos. Em outros sistemas, tais como os da IBM, muitos métodos de acesso são aceitos, e escolher o método correto para determinada aplicação é um problema importante de projeto.

11.2.1 Acesso sequencial

O método de acesso mais simples é o acesso sequencial. As informações no arquivo são processadas em ordem, um registro após o outro. Esse modo de acesso é de longe o mais comum, por exemplo, os editores e compiladores geralmente acessam arquivos desse modo.

A maior parte das operações com um arquivo são de leitura e escrita. Uma operação de leitura lê a próxima porção do arquivo e automaticamente avança um ponteiro do arquivo, que indica a posição de I/O. Da mesma forma, uma operação de escrita acrescenta informações ao final do arquivo e avança até o fim do material recém-escrito (o novo fim do arquivo). Esse ponteiro de arquivo pode ser retornado para o início e, em alguns sistemas, um programa pode avançar ou voltar n registros, para determinado inteiro n (talvez apenas para $n = 1$). O acesso sequencial está apresentado na Figura 11.3. O acesso sequencial baseia-se em um modelo de fita de um arquivo e funciona tanto em dispositivos de acesso sequencial quanto em dispositivos de acesso aleatório.

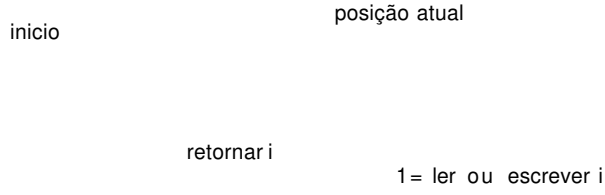


Figura 11.3 Arquivo de acesso sequencial.

11.2.2 Acesso direto

Outro método é o acesso direto (ou acesso relativo). Um arquivo é formado por registros lógicos de tamanho fixo que permitem que os programas leiam e escrevam registros rapidamente sem uma ordem específica. O método de acesso direto baseia-se em um modelo de disco de um arquivo; os discos permitem o acesso aleatório a qualquer bloco do arquivo. Para o acesso direto, o arquivo é visto como uma sequência numerada de blocos ou registros. Um arquivo de acesso direto permite que blocos arbitrários sejam lidos ou escritos. Assim, podemos ler o bloco 14, em seguida ler o bloco 53 e depois escrever o bloco 7. Não existem restrições quanto à ordem de leitura ou escrita para um arquivo de acesso direto.

Os arquivos de acesso direto são úteis para acesso imediato a grandes quantidades de informação. Os bancos de dados muitas vezes têm essa necessidade. Quando uma consulta relativa a determinado assunto surge, identificamos (calculamos) o bloco que contém a resposta e, em seguida, lemos esse bloco diretamente para fornecer as informações desejadas.

Por exemplo, em um sistema de reserva de passagens aéreas, podemos armazenar todas as informações sobre determinado vôo (por exemplo, o vôo 713) no bloco identificado pelo número do vôo. Assim, o nume-

ro dos assentos disponíveis para o vôo 713 é armazenando no bloco 713 do arquivo de reserva. Para armazenar informações sobre um conjunto maior, tais como pessoas, podemos calcular uma função de hashing nos nomes das pessoas ou pesquisar um pequeno índice na memória para determinar um bloco para leitura e pesquisa.

As operações de arquivo devem ser modificadas para incluir o número de bloco como parâmetro. Assim, temos read w, onde *néo* número do bloco, em vez de read next, e escrever «, em vez de escrever next. Uma abordagem alternativa é manter read next e escrever next, como fazemos com o acesso sequencial e adicionar uma operação, posi ti on f i 1 **e** to *n,* onde *néo* número de bloco. Em seguida, para efetuar um read *n* fazemos position file to *n* e read next.

O número de bloco fornecido pelo usuário ao sistema operacional normalmente é um **número de bloco relativo,** que é um índice relativo ao início do arquivo. Assim, o primeiro bloco relativo do arquivo é 0, o seguinte é 1, e assim por diante, embora o endereço absoluto real de disco do bloco talvez seja 14703 para o primeiro bloco, e 3192 para o segundo. O uso de números de bloco relativos permite que o sistema operacional decida onde o arquivo deve ser colocado (chamado de problema de alocação, conforme discutido mais tarde neste capítulo), e ajuda a evitar que o usuário acesse partes do sistema de arquivos que talvez não façam parte do seu arquivo. Alguns sistemas começam seus números de bloco relativos em 0; outros em 1.

Dado um tamanho de registro lógico L, um pedido para o registro *N é* transformado em um pedido de I/O para L bytes começando na posição *L* * (N - 1) dentro do arquivo (considerando que o primeiro registro é *N* = 1). Como os registros lógicos são de tamanho fixo, também é fácil ler, escrever ou excluir um registro.

Nem todos os sistemas operacionais suportam acesso direto e sequencial aos arquivos. Alguns sistemas permitem apenas acesso sequencial aos arquivos; outros permitem apenas acesso direto. Alguns sistemas exigem que um arquivo seja definido como sequencial ou direto quando ele é criado; esse arquivo pode ser acessado apenas de forma consistente com sua declaração. Observe, por outro lado, que é fácil simular o acesso sequencial sobre um arquivo de acesso direto. Se mantivermos uma variável *cp {current position),* que define nossa posição atual, podemos simular operações sequenciais, como mostrado na Figura 11.4. Por outro lado, é extremamente ineficiente e pouco prático simular um arquivo de acesso direto sobre um arquivo de acesso sequencial.

11.2.3 Outros métodos de acesso

Outros métodos de acesso podem ser desenvolvidos com base no método de acesso direto. Esses métodos adicionais geralmente envolvem a construção de um **índice** para o arquivo. O índice, como um índice na parte posterior de um livro, contém ponteiros aos vários blocos. Para encontrar um registro no arquivo, primeiro pesquisamos o índice e depois usamos o ponteiro para acessar o arquivo diretamente e encontrar o registro desejado.

Por exemplo, um arquivo de preços de varejo pode listar os códigos de produtos (UPCs - Universal Product Codes) para itens, com seus preços associados. Cada registro consiste em um UPC de 10 dígitos e um preço de 6 dígitos, perfazendo um registro de 16 bytes. Se nosso disco tiver 1.024 bytes por bloco, podemos armazenar 64 registros por bloco. Um arquivo de 120.000 registros ocuparia em torno de 2.000 blocos (2 milhões de bytes). Ao manter o arquivo ordenado por UPC, podemos definir um índice consistindo no primeiro UPC em cada bloco. Esse índice teria 2.000 entradas de 10 dígitos cada, ou 20.000 bytes, **e** assim poderia ser mantido na memória. Para encontrar o preço de determinado item, podemos fazer uma pesquisa (binária) no índice. A partir dessa pesquisa, sabemos exatamente que bloco contém o registro desejado e acessamos esse bloco. Essa estrutura nos permite pesquisar um arquivo grande efetuando poucas operações de entrada e saída.

Com arquivos grandes, o arquivo de índice propriamente dito pode ficar grande demais para ser mantido na memória. Uma solução é criar um índice para o arquivo de índice. O arquivo de índice principal contém ponteiros aos arquivos de índice secundários, que apontam para os itens de dados reais.

Por exemplo, o método de acesso sequencial indexado da IBM (ISAM - Indexed Sequential Access Mode) utiliza um pequeno índice mestre que aponta para blocos de disco de um índice secundário. Os blocos de índice secundário apontam para os blocos reais do arquivo. O arquivo é mantido ordenado por uma chave definida. Para encontrar determinado item, fazemos primeiro uma pesquisa binária do índice mestre, que

fornece o número do bloco do índice secundário. Esse bloco é lido e mais uma vez uma pesquisa binária é usada para encontrar o bloco que contém o registro desejado. Finalmente, este bloco é pesquisado sequencialmente. Dessa forma, qualquer registro pode ser localizado a partir de sua chave no máximo em duas operações de leitura de acesso direto. A Figura 11.5 mostra uma situação semelhante implementada pelo mecanismo de arquivo de índice e arquivo relativo ao sistema VMS.

acesso sequencial	implementação com acesso direto
reset	*cp = 0;*
read next	*read cp;* *cp = cp + 1;*
escrever next	*escrever cp;* *cp = cp + 1;*

Figura 11.4 Simulação de acesso sequencial em um arquivo de acesso direto.

11.3 • Estrutura de diretório

Os sistemas de arquivos dos computadores podem ser enormes. Alguns sistemas armazenam milhares de arquivos em centenas de gigabytes de disco. Para gerenciar todos esses dados, precisamos organizá-los. Essa organização geralmente é feita em duas partes. Em primeiro lugar, o sistema de arquivos é quebrado em partições, também chamadas de minidiscos no mundo da IBM ou volumes nos PCs e Macintosh. Em geral, cada disco em um sistema contém pelo menos uma partição, que é uma estrutura de baixo nível na qual residem arquivos e diretórios. Alguns sistemas utilizam partições para fornecer várias áreas separadas em um mesmo disco, tratando cada uma como um dispositivo de armazenamento separado, enquanto outros sistemas permitem que as partições sejam maiores do que um disco, de forma que possam agrupar os discos em uma única estrutura lógica. O usuário então precisa se preocupar apenas com a estrutura lógica de arquivos e diretórios; ele pode ignorar completamente os problemas de alocação de espaço físico para os arquivos. Por esse motivo, as partições podem ser consideradas discos virtuais.

Em segundo lugar, cada partição contém informações sobre os arquivos dentro dela. Essas informações são mantidas em entradas em um diretório de dispositivo ou índice de volume. O diretório de dispositivo (mais conhecido simplesmente como *diretório)* registra informações - tais como nome, posição, tamanho e tipo - para todos os arquivos naquela partição. A Figura 11.6 mostra a organização típica de um sistema de arquivos.

O diretório pode ser visto como uma tabela de símbolos que traduz nomes de arquivos em entradas de diretório. Se considerarmos essa visão, fica aparente que o diretório em si pode ser organizado de muitas for-

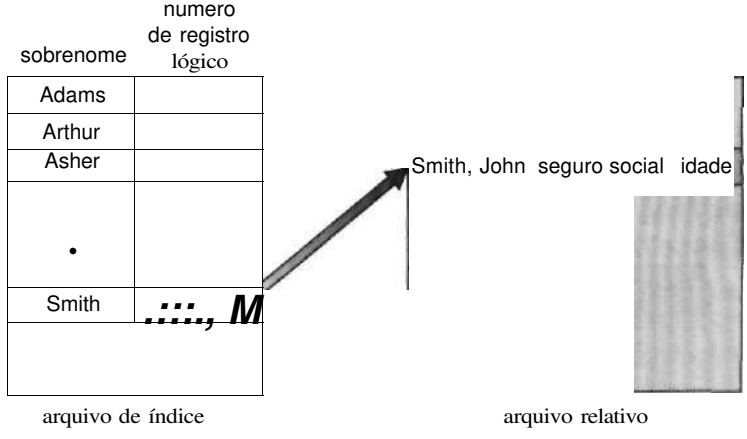


Figura 11.5 Exemplo de arquivo índice e arquivo relativo.

mas. Queremos ser capazes de inserir e excluir entradas, pesquisar uma entrada por nome e listar todas as entradas no diretório. Na Seção 11.8, discutimos as estruturas de dados apropriadas que podem ser usadas na implementação da estrutura de diretório. Nesta seção, examinamos vários esquemas para definir a estrutura lógica do sistema de diretórios. Ao considerar determinada estrutura de diretório, precisamos ter em mente que operações deverão ser realizadas em um diretório:

- *Pesquisar arquivos:* É preciso poder pesquisar uma estrutura de diretório para encontrar determinado arquivo. Como os arquivos têm nome simbólico e nomes semelhantes podem indicar uma relação entre os arquivos, talvez surja a necessidade de encontrar todos os arquivos cujos nomes correspondam a um determinado padrão.
- *Criar um arquivo:* É preciso poder criar novos arquivos e adicioná-los ao diretório.

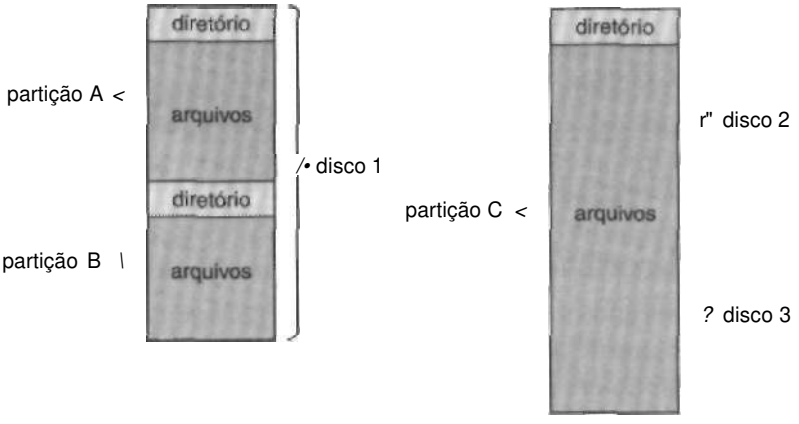


Figura 11.6 Uma organização típica de sistema de arquivos.

- *Excluir um arquivo:* Quando um arquivo não é mais necessário, precisamos ter a capacidade de removê-lo do diretório.
- *Listar um diretório:* E preciso poder listar os arquivos em um diretório e o conteúdo da entrada no diretório para cada arquivo na lista.
- *Renomear um arquivo:* Como o nome de um arquivo representa seu conteúdo para os usuários, precisamos ter a capacidade de alterar o nome quando o conteúdo ou uso do arquivo mudar. Renomear um arquivo talvez permita também que sua posição dentro da estrutura de diretório seja alterada.
- *Percorrer o sistema de arquivos:* É útil poder acessar todos os diretórios e todos os arquivos dentro de uma estrutura de diretório. Para fins de confiabilidade, é boa ideia salvar o conteúdo e a estrutura do sistema de arquivos inteiro em intervalos regulares. Essa operação geralmente consiste em copiar todos os arquivos para uma fita magnética. Essa técnica fornece uma cópia backup em caso de falha do sistema ou se o arquivo simplesmente não estiver mais em uso. Nesse caso, o arquivo pode ser copiado para fita, e o espaço em disco desse arquivo pode ser liberado para reutilização por outro arquivo.

Nas Seções 11.3.1 a 11.3.5, descrevemos os esquemas mais comuns para definir a estrutura lógica de um diretório.

11.3.1 Diretório de nível único

A estrutura de diretório mais simples é a de diretório de nível único. Todos os arquivos estão contidos no mesmo diretório, que é de fácil suporte e compreensão (Figura 11.7).

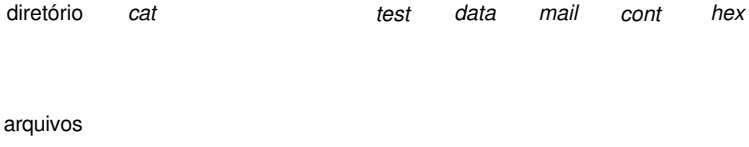


Figura 11.7 Diretório de nível único.

Um diretório de nível único tem limitações significativas, no entanto, quando o número de arquivos aumenta ou quando existe mais de um usuário. Como todos os arquivos estão no mesmo diretório, eles devem ter nomes exclusivos. Se tivermos dois usuários que chamem seu arquivo de dados de *teste,* a regra de nome exclusivo será violada. (Por exemplo, em uma turma de programação, 23 alunos deram o nome de *progl* ao programa para sua segunda tarefa; outros 11 o chamaram de *tarefai.)* Embora os nomes de arquivos sejam geralmente selecionados para refletir o conteúdo do arquivo, eles normalmente são limitados em tamanho. O sistema operacional MS-DOS permite apenas nomes de arquivos com 11 caracteres; o UNIX permite 255 caracteres.

Mesmo com um único usuário, à medida que o número de arquivos aumenta, fica difícil lembrar dos nomes de todos os arquivos, de modo a criar apenas arquivos com nomes exclusivos. Não é incomum um usuário ter centenas de arquivos em um sistema de computação e um número igual de arquivos adicionais em outro sistema. Em um ambiente assim, manter o controle de tantos arquivos é uma tarefa e tanto.

11.3.2 Diretório de dois níveis

A principal desvantagem de um diretório de nível único é a confusão de nomes de arquivos criada pelos diferentes usuários. A solução padrão é criar um diretório *separado* para cada usuário.

Na estrutura de diretório de dois níveis, cada usuário tem seu próprio **diretório de arquivos de usuário** (UFD - User File Directory). Cada UFD tem uma estrutura semelhante, mas lista apenas os arquivos de um único usuário. Quando um job de usuário começa ou quando o usuário efetua o logon, o diretório **de** arquivos **mestre** (MFD - Master File Directory) é pesquisado. O MFD é indexado por nome de usuário ou número de conta, e cada entrada aponta para o UFD daquele usuário (Figura 11.8).

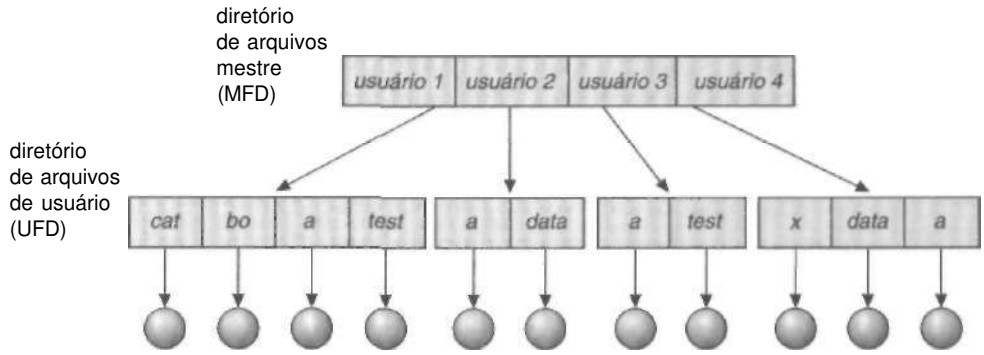


Figura **11.8** Estrutura de diretório de dois níveis.

Quando um usuário fizer referência a determinado arquivo, apenas seu próprio UFD será pesquisado. Assim, diferentes usuários podem ter arquivos com o mesmo nome, desde que todos os nomes de arquivo em cada diretório de usuário sejam exclusivos.

Para criar um arquivo para um usuário, o sistema operacional pesquisa apenas o UFD daquele usuário para determinar se existe outro arquivo com esse nome. Para excluir um arquivo, o sistema operacional restringe a sua busca ao UFD local; assim, ele não poderá excluir acidentalmente o arquivo de outro usuário que tenha o mesmo nome.

Os diretórios de usuário propriamente ditos devem ser criados e excluídos conforme necessário. Um programa especial de sistema é executado com as informações apropriadas de nome de usuário e conta. O programa cria um novo diretório de arquivos de usuário e acrescenta uma entrada para ele no diretório de arquivos mestre. A execução desse programa deve ser restrita a administradores de sistema. A alocação de espaço em disco para os diretórios de usuário pode ser realizada com as técnicas discutidas na Seção 11.6 para os próprios arquivos.

A estrutura de diretório de dois níveis resolve o problema de colisão de nomes, mas ainda assim tem problemas. Essa estrutura efetivamente isola um usuário do outro. Esse isolamento é uma vantagem quando os usuários são completamente independentes, mas é uma desvantagem quando os usuários desejam cooperar em alguma tarefa e acessar os arquivos uns dos outros. Alguns sistemas simplesmente não permitem que os arquivos locais de um usuário sejam acessados por outros usuários.

Se o acesso for permitido, um usuário deve ser capaz de indicar o nome de um arquivo no diretório de outro usuário. Para indicar determinado arquivo de forma exclusiva em um diretório de dois níveis, é preciso fornecer o nome do usuário e o nome do arquivo. Um diretório de dois níveis pode ser considerado uma árvore - ou uma árvore invertida- de altura 2. A raiz da árvore é o diretório de arquivos mestre. Seus descendentes diretos são os UFDs. Os descendentes dos diretórios de arquivo de usuário são os próprios arquivos. Os arquivos são as folhas da árvore. Especificar um nome de usuário e um nome de arquivo define um caminho na árvore a partir da raiz (o diretório de arquivos mestre) até a folha (o arquivo especificado). Assim, um nome de usuário e um nome de arquivo definem um **nome de caminho.** Todo arquivo no sistema tem um nome de caminho. Para indicar um arquivo de forma exclusiva, um usuário deve saber o nome de caminho do arquivo desejado.

Por exemplo, se o usuário A desejar acessar seu próprio arquivo de teste chamado test, ele simplesmente poderá fazer referência a test. Para acessar o arquivo de teste do usuário B (com o nome de entrada de usuário userb), no entanto, ele talvez tenha de fazer referência a /userb/test. Cada sistema tem sua própria sintaxe para nomear arquivos em diretórios diferentes do próprio diretório do usuário.

Existe uma sintaxe adicional para especificar a partição de um arquivo. Por exemplo, no MS-DOS uma partição é especificada por uma letra seguida de dois-pontos. Assim, uma especificação de arquivo pode ser "C:\userb\test". Alguns sistemas vão mais além e separam a partição, as partes de nome do diretório e nome do arquivo da especificação. Por exemplo, no VMS, o arquivo "login.com" pode ser especificado como: "u:[sst.jdeck]login.com;1", onde "u" é o nome da partição, "sst" é o nome do diretório, "jdeck" é o nome do subdiretório e " 1" é o número da versão. Outros sistemas simplesmente tratam o nome da partição como parte do nome do diretório. O primeiro nome dado é o da partição e o resto é o diretório e o arquivo. Por exemplo, "/u/pbg/test" pode especificar a partição "u", diretório "pbg" e o arquivo "test".

Um caso especial dessa situação ocorre com relação aos arquivos de sistema. Esses programas fornecidos como parte do sistema (carregadores, montadores, compiladores, utilitários, bibliotecas, entre outros) geralmente são definidos como arquivos. Quando os comandos apropriados são dados ao sistema operacional, esses arquivos são lidos pelo carregador e executados. Muitos interpretadores de comando atuam simplesmente tratando o comando como o nome de um arquivo a ser carregado e executado. Da forma em que o sistema de diretório está definido no momento, esse nome de arquivo seria pesquisado no diretório de arquivos de usuário atual. Uma solução seria copiar os arquivos do sistema para cada diretório de arquivos de usuário. No entanto, copiar todos os arquivos do sistema seria um enorme desperdício de espaço. (Se os arquivos do sistema exigem 5 megabytes, então o suporte a 12 usuários exigiria 5 X 12 = 60 megabytes apenas para cópias dos arquivos de sistema.)

A solução padrão é complicar um pouco o procedimento de pesquisa. Um diretório especial de usuário é definido para conter os arquivos do sistema (por exemplo, usuário 0). Sempre que um nome de arquivo for dado para ser carregado, o sistema operacional primeiro pesquisa o diretório de arquivos de usuário local. Se o arquivo for encontrado, ele será usado. Se não for encontrado, o sistema pesquisará automaticamente o diretório especial de usuário que contém os arquivos do sistema. A sequência de diretórios pesquisados quando um arquivo é referenciado é denominada **caminho de pesquisa.** Essa ideia pode ser estendida, de modo que o caminho de pesquisa contenha uma lista ilimitada de diretórios a serem pesquisados quando um nome de comando é fornecido. Esse método é o utilizado com mais frequência no UNIX e no MS-DOS.

11.3.3 Diretórios estruturados em árvore

Assim que tivermos visto como visualizar um diretório de dois níveis como uma árvore de dois níveis, a generalização natural é estender a estrutura de diretório em uma árvore de altura arbitrária (Figura 11.9). Essa generalização permite que os usuários criem seus próprios subdiretórios e organizem seus arquivos adequadamente. O sistema MS-DOS, por exemplo, é estruturado como uma árvore. Na verdade, uma árvore é a estrutura de diretório mais comum. A árvore tem um diretório raiz. Todo arquivo no sistema tem um nome de caminho exclusivo.

Um **diretório** (ou subdiretório) contém um conjunto de arquivos ou subdiretórios. Um diretório é simplesmente outro arquivo, mas é tratado de modo especial. Todos os diretórios têm o mesmo formato interno. Um bit em cada entrada de diretório define a entrada como um arquivo (0) ou um subdiretório (1). Chamadas ao sistema especiais criam e excluem diretórios.

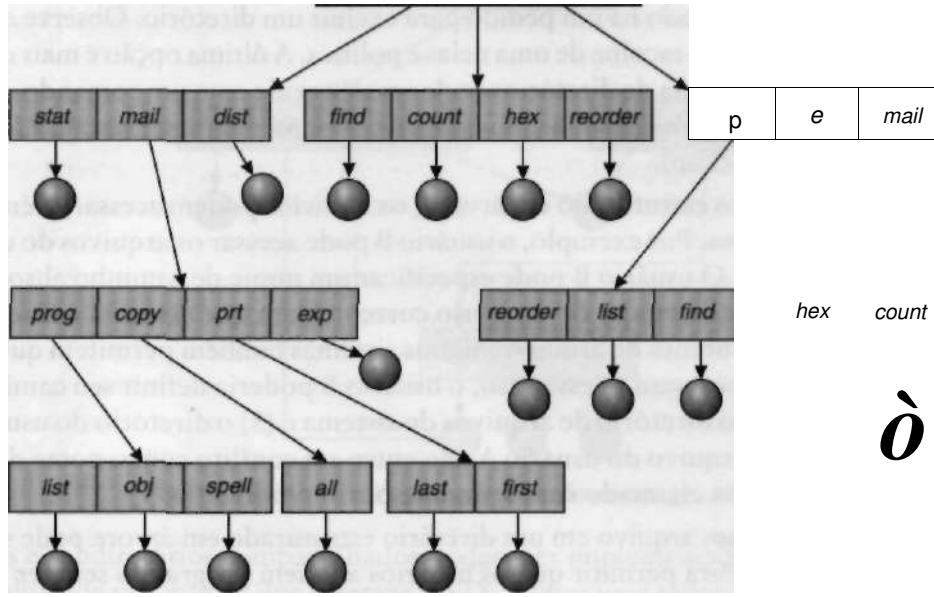


Figura 11.9 Diretório estruturado em árvore.

Em uso normal, cada usuário tem um **diretório corrente.** O diretório corrente deve conter a maior parte dos arquivos que são de interesse ao usuário. Quando for feita referência a um arquivo, o diretório corrente será pesquisado. Se houver necessidade de um arquivo que não esteja no diretório corrente, o usuário deverá especificar um nome de caminho ou alterar o diretório corrente para que seja o diretório contendo esse arquivo. Para que o usuário possa mudar o diretório corrente para outro diretório, uma chamada ao sistema é feita utilizando o nome do diretório como um parâmetro e usando esse parâmetro para redefinir o diretório corrente. De uma chamada ao sistema change di rectory à próxima, todas as chamadas open pesquisam o diretório corrente pelo arquivo especificado.

O diretório corrente inicial de um usuário é designado quando o job de usuário começa ou o usuário efetua logon. O sistema operacional pesquisa o arquivo de contabilização (ou alguma outra posição predefinida) para encontrar uma entrada para esse usuário (para fins de contabilização). No arquivo de contabilização, existe um ponteiro (ou nome) para o diretório inicial do usuário. Esse ponteiro é copiado para uma variável local do usuário, que especifica o diretório corrente inicial do usuário.

Os nomes de caminho podem ser de dois tipos: absolutos ou relativos. Um **nome de caminho absoluto** começa na raiz e segue um caminho até o arquivo especificado, fornecendo os nomes de diretório no caminho. Um **nome de caminho relativo** define um caminho a partir do diretório corrente. Por exemplo, no sistema de arquivos estruturado em árvore da Figura 11.9, se o diretório corrente for *root/spell/mail,* então o nome de caminho relativo *prt/first* refere-se ao mesmo arquivo que o nome de caminho absoluto *root/ spell/ mail/prt/first.*

Permitir que o usuário defina seus próprios subdiretórios possibilita que ele imponha uma estrutura para os seus arquivos. Essa estrutura poderá resultar em diretórios separados para os arquivos associados a diferentes tópicos (por exemplo, criamos um subdiretório para conter o texto deste livro) ou diferentes formas de informação (por exemplo, o diretório *programs* pode conter programas fonte; o diretório bin pode armazenar todos os binários).

Uma importante decisão de política em uma estrutura de diretório em árvore é como tratar a exclusão de um diretório. Se o diretório estiver vazio, sua entrada no diretório que o contém pode ser simplesmente excluída. No entanto, vamos supor que o diretório a ser excluído não esteja vazio, mas contenha vários arquivos ou possivelmente subdiretórios. Uma de duas abordagens pode ocorrer. Alguns sistemas, como o MS-DOS, só excluem um diretório quando ele estiver vazio. Assim, para excluir um diretório, o usuário precisará primeiro excluir (ou mover) todos os arquivos daquele diretório. Se houver subdiretórios, o usuário deverá aplicar esse procedimento de forma recursiva, excluindo-os também, o que pode envolver muito trabalho.

Uma alternativa - permitida pelo comando rm do UNIX - é fornecer uma opção que exclua também todos os arquivos e subdiretórios, quando há um pedido para excluir um diretório. Observe que as duas abordagens são de fácil implementação, a escolha de uma delas é política. A última opção é mais conveniente, mas mais arriscada, porque toda a estrutura de diretórios pode ser removida com um comando. Se esse comando for emitido por engano, um grande número de arquivos e diretórios precisaria ser restaurado das fitas de backup (considerando que exista backup).

Com um sistema de diretórios estruturado em árvore, os usuários podem acessar, além dos seus arquivos, os arquivos de outros usuários. Por exemplo, o usuário B pode acessar os arquivos do usuário A especificando seus nomes de caminho. O usuário B pode especificar um nome de caminho absoluto ou relativo. Como alternativa, o usuário B pode mudar de diretório corrente para que esse seja o diretório do usuário A, e acessar os arquivos por seus nomes de arquivo. Alguns sistemas também permitem que um usuário defina seus próprios caminhos de pesquisa. Nesse caso, o usuário B poderia definir seu caminho de pesquisa como (1) seu diretório local, (2) o diretório de arquivos do sistema e (3) o diretório do usuário A, nessa ordem. Desde que o nome de um arquivo do usuário A não entre em conflito com o nome de um arquivo ou arquivo de sistema local, ele seria chamado simplesmente por seu nome.

Observe que um caminho a um arquivo em um diretório estruturado em árvore pode ser maior do que em um diretório de dois níveis. Para permitir que os usuários acessem programas sem ter de lembrar esses longos nomes de caminho, o sistema Macintosh automatiza a pesquisa para programas executáveis. Ele mantém um arquivo, chamado "Desktop File", que contém o nome e a posição de todos os programas executáveis. Quando um novo disco rígido ou disquete é acrescentado ao sistema, ou a rede é acessada, o sistema operacional percorre a estrutura de diretório, procurando programas executáveis no dispositivo e registrando as informações pertinentes. Esse mecanismo suporta a funcionalidade de execução com clique duplo descrita anteriormente. Clicar duas vezes em um arquivo faz com que seu atributo de criador seja lido e que o "Desktop File" seja pesquisado por uma entrada coincidente. Assim que essa entrada for encontrada, o programa executável apropriado será iniciado, com o arquivo clicado como entrada.

11.3.4 Diretórios em grafos acíclicos

Considere dois programadores que estejam trabalhando em um projeto conjunto. Os arquivos associados a esse projeto podem ser armazenados em um subdiretório, separando-os de outros projetos e dos arquivos dos dois programadores. No entanto, como os dois programadores são igualmente responsáveis pelo projeto, ambos querem que o subdiretório esteja localizado em seu próprio diretório. O subdiretório comum deveria ser compartilhado. Um diretório ou arquivo compartilhado existirá no sistema de arquivos em dois (ou mais) locais de uma vez. Observe que um arquivo (ou diretório) compartilhado não é igual a manter duas cópias do arquivo. Com duas cópias, cada programador pode visualizar a cópia em vez do original, mas se um programador alterar o arquivo, as alterações não aparecerão na cópia do outro. Com um arquivo compartilhado, existe apenas um arquivo real, de modo que qualquer alteração feita por uma pessoa seria imediatamente vista pela outra. Essa forma de compartilhamento é particularmente importante para subdiretórios compartilhados; um novo arquivo criado por uma pessoa aparecerá automaticamente em todos os subdiretórios compartilhados.

Uma estrutura de árvore proíbe o compartilhamento de arquivos ou diretórios. Um grafo acíclico, que é um grafo sem ciclos, permite que os diretórios tenham subdiretórios e arquivos compartilhados (Figura 11.10). O mesmo arquivo ou subdiretório poderá estar em dois diretórios diferentes. Um grafo acíclico é uma generalização natural do esquema de diretório estruturado em árvore. Quando várias pessoas estão trabalhando como uma equipe, todos os arquivos a serem compartilhados podem ser colocados juntos em um diretório. Os diretórios de arquivos de usuário de todos os membros da equipe contêm esse diretório de arquivos compartilhados como um subdiretório. Mesmo quando existe um único usuário, a sua organização de arquivos pode exigir que alguns arquivos sejam colocados em vários subdiretórios diferentes. Por exemplo, um programa escrito para determinado projeto deve estar no diretório de todos os programas e no diretório daquele projeto.

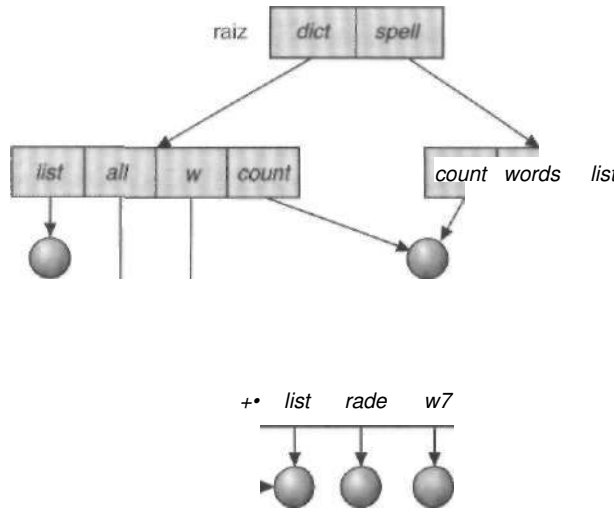


Figura 11.10 Estrutura de diretório em grafo acíclico

Os arquivos e subdiretórios compartilhados podem ser implementados de várias formas. Uma maneira :omum, exemplificada por muitos dos sistemas UNIX, é criar uma nova entrada de diretório chamada **link.** 'Um link é, na verdade, um ponteiro para outro arquivo ou subdiretório. Por exemplo, um link pode ser implementado como um nome de caminho absoluto ou relativo - chamado **link simbólico.** Quando uma referência é feita a um arquivo, o diretório é pesquisado. Se a entrada do diretório for marcada como um link, o nome do arquivo (ou diretório) real será fornecido. O link é resolvido usando o nome de caminho para localizar o arquivo real. Os links são facilmente identificáveis pelo seu formato na entrada do diretório (ou por ter um tipo especial nos sistemas que suportam tipos); eles servem como ponteiros indiretos com nome. O sistema operacional ignora esses links quando estiver percorrendo as árvores de diretório, para preservar a estrutura acíclica do sistema.

- A outra abordagem para implementar arquivos compartilhados é simplesmente duplicar todas as informações sobre eles nos dois diretórios compartilhados. Assim, as duas entradas são idênticas. Um link é claramente diferente da entrada de diretório original; portanto, as duas não são iguais. Entradas duplicadas, **no** entanto, tornam o original e a cópia indistinguíveis. O principal problema com entradas de diretório duplicadas é manter a consistência se o arquivo for modificado.

Uma estrutura de diretório em grafo acíclico é mais flexível do que uma estrutura de árvore simples, mas também é mais complexa. Observe que um arquivo agora pode ter múltiplos nomes de caminho absolutos. Consequentemente, diferentes nomes de arquivo podem fazer referência ao mesmo arquivo. Essa situação é semelhante ao problema de nomes alternativos para as linguagens de programação. Se estivermos tentando percorrer todo o sistema de arquivos (para encontrar um arquivo, acumular estatísticas sobre todos os arquivos ou copiar todos os arquivos para fins de backup), esse problema se torna significativo, já que não queremos percorrer as estruturas compartilhadas mais de uma vez.

Outro problema envolve a exclusão. Quando o espaço alocado a um arquivo compartilhado pode ser desalocado e reutilizado? Uma possibilidade é remover o arquivo sempre que alguém o excluir, mas essa ação poderá deixar ponteiros pendentes para o arquivo agora inexistente. Pior, se os outros ponteiros de arquivo contiverem endereços reais de disco, e o espaço for reutilizado depois para outros arquivos, esses ponteiros pendentes podem apontar para o meio de outros arquivos.

Em um sistema no qual o compartilhamento é implementado por links simbólicos, essa situação é um pouco mais fácil de resolver. A exclusão de um link não precisa afetar o arquivo original; somente o link é removido. Se a entrada de arquivo propriamente for excluída, o espaço para o arquivo será desalocado, deixando os links pendentes. Podemos pesquisar esses links e removê-los também, mas a menos que uma lista dos links associados seja mantida em cada arquivo, essa pesquisa pode ser cara. Como alternativa, podemos deixar os links até que ocorra uma tentativa de usá-los. Nesse momento, podemos determinar que o arquivo com o nome dado pelo link não existe e, portanto, não resolverá o nome do link; o acesso é tratado como qualquer outro nome de arquivo ilegal. (Nesse caso, o projetista do sistema deve considerar cuidadosamente o que fazer quando um arquivo é excluído ou outro arquivo com o mesmo nome é criado, antes que um link

simbólico ao arquivo original seja usado.) No caso do UNIX, os links simbólicos são mantidos quando um arquivo é excluído, e cabe ao usuário verificar se o arquivo original foi eliminado ou substituído.

Outra abordagem à exclusão é preservar o arquivo até que todas as referências a ele sejam excluídas. Para implementar essa abordagem, devemos ter algum mecanismo para determinar que a última referência ao arquivo foi excluída. Poderíamos manter uma lista de todas as referências a um arquivo (entradas de diretório ou links simbólicos). Quando um link ou uma cópia da entrada do diretório for estabelecido, uma nova entrada será adicionada à lista de referências do arquivo. Quando um link ou entrada de diretório for excluído, removemos sua entrada na lista. O arquivo será excluído quando sua lista de referências de arquivo estiver vazia.

O problema com esta abordagem é o tamanho variável e potencialmente grande da lista de referências de arquivo. No entanto, não precisamos manter a lista inteira - precisamos manter apenas um contador do número de referências. Um novo link ou entrada de diretório incrementa o contador de referências; excluir um link ou entrada decrementa o contador. Quando o contador for 0, o arquivo pode ser excluído; não existem referências restantes. O sistema operacional UNIX utiliza essa abordagem para links não-simbólicos, ou hard links, mantendo um contador de referências no bloco de informações de arquivo (ou inode, consulte a Seção 20.7.2). Ao proibir efetivamente múltiplas referências aos diretórios, mantemos uma estrutura de grafo acíclico.

Para que seus usuários evitem esses problemas, alguns sistemas não permitem diretórios ou links compartilhados. Por exemplo, no MS-DOS, a estrutura de diretório é uma estrutura em árvore, em vez de um grafo acíclico, evitando assim os problemas associados à exclusão de arquivos em uma estrutura de diretório em grafo acíclico.

11.3.5 Diretório em grafo genérico

Um problema grave com o uso de uma estrutura de grafo acíclico é que é difícil garantir que não existam ciclos. Se começarmos com um diretório dè\dois níveis e permitirmos que os usuários criem subdiretórios, teremos como resultado um diretório estruturado em árvore. Deve ser razoavelmente fácil verificar que adicionar novos arquivos e subdiretórios a um orretório estruturado em árvore existente preserva a natureza estruturada em árvore. No entanto, quando adicionamos links a um diretório estruturado em árvore, a estrutura é destruída, resultando em uma estrutura de^grafo simples (Figura 11.11).

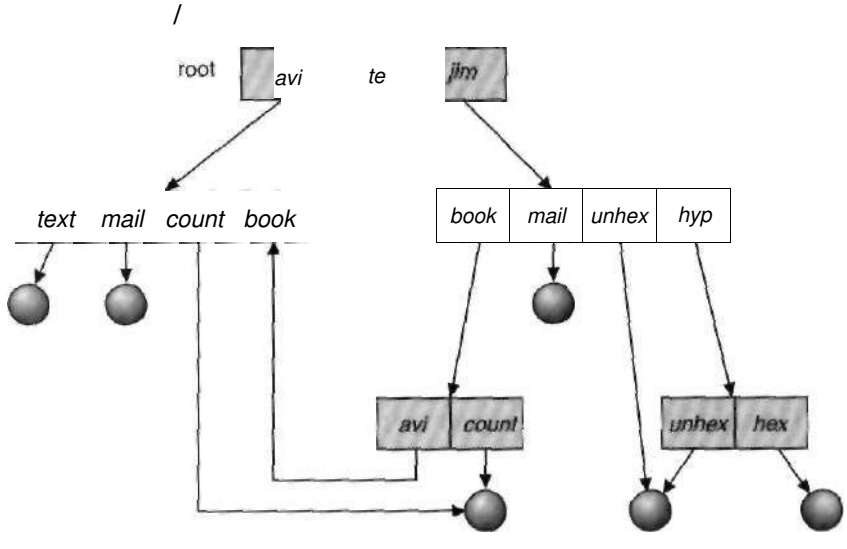


Figura 11.11 Diretório em grafo genérico.

A principal vantagem de um grafo acíclico é a relativa simplicidade dos algoritmos para percorrer o grafo e determinar quando não existem mais referências a um arquivo. Devemos evitar percorrer seções compartilhadas de um grafo duas vezes, principalmente por motivos de desempenho. Se tivermos acabado de pesquisar um subdiretório compartilhado importante para encontrar determinado arquivo, a segunda pesquisa será uma perda de tempo.

Se for permitida a existência de ciclos no diretório, devemos evitar também pesquisar qualquer componente duas vezes, por motivos de correção assim como desempenho. Um algoritmo mal projetado pode re-

sultar em um laço infinito fazendo a pesquisa continuamente no ciclo, sem nunca terminar. Uma solução é limitar arbitrariamente o número de diretórios que podem ser acessados durante a pesquisa.

Enfrentamos um problema semelhante quando tentamos determinar quando um arquivo pode ser excluído. Como ocorre nas estruturas de diretório em grafo genérico, um valor de 0 no contador de referências significa que não há mais referências no arquivo ou diretório e, portanto, que o arquivo pode ser excluído. No entanto, também é possível que o contador de referências seja diferente de zero, quando existirem ciclos, mesmo quando não for rriaisjpóssível fazer referência a um diretório ou arquivo. Essa anomalia resulta da possibilidade de auto-referênqá (um ciclo) na estrutura de diretório. Nesse caso, geralmente é necessário usar um esquema de coleta de/ixp para determinar quando a referência final foi excluída e quando o espaço de disco pode ser realocado. A coleta de lixo envolve percorrer todo o sistema de arquivos, marcando tudo que pode ser acessado. Em seguida, uma segunda passada coleta tudo que não está marcado em uma lista de espaço livre. (Um procedimento de marcação semelhante pode ser usado para garantir que uma busca ou pesquisa cobrirá tudo no sistema de arquivos uma única vez.) A coleta de lixo para um sistema de arquivos baseado em disco, no entanto, é extremamente demorada e, assim, raramente é tentada.

A coleta de lixo é necessária apenas por causa dos possíveis ciclos no grafo. Assim, trabalhar com uma estrutura de grafo acíclico é muito mais fácil. A dificuldade é evitar ciclos à medida que novos links são adicionados à estrutura. Como sabemos quando um novo link completará um ciclo? Existem algoritmos para detectar ciclos em grafos; no entanto, eles são computacionalmente caros, especialmente quando o grafo está em disco. Geralmente, por esse motivo, as estruturas de diretórios em árvore são mais comuns do que as de grafos acíclicos.

11.4 • Proteção

Quando as informações são mantidas em um sistema de computação, uma importante preocupação é a confiabilidade, ou ficar livre de danos físicos. Outra é a proteção, ou não permitir acesso indevido.

A confiabilidade geralmente é obtida por cópias duplicadas dos arquivos. Muitos computadores têm programas de sistemas que copiam automaticamente (ou por meio da intervenção do operador do computador) os arquivos do disco para fita em intervalos regulares (uma vez por dia, semana ou mês) para manter uma cópia em caso de destruição acidental ou maliciosa do sistema de arquivos. Os sistemas de arquivos podem ser danificados por problemas de hardware (tais como erros de leitura e escrita), surtos ou falhas de energia, choques da cabeça de leitura, sujeira, temperaturas extremas ou vandalismo. Os arquivos podem ser excluídos acidentalmente. Os bugs no software do sistema de arquivos também podem fazer com que haja perda do conteúdo de arquivos. A confiabilidade é tratada no Capítulo 13.

A proteção pode ser provida de muitas formas. Para um sistema pequeno, monousuário, podemos remover os disquetes fisicamente e colocá-los em um gabinete de arquivos ou gaveta. Em um sistema multiusuário, outros mecanismos são necessários.

11.4.1 Tipos de acesso

A necessidade de proteger arquivos é resultado direto da capacidade de acessar arquivos. Nos sistemas que não permitem acesso aos arquivos de outros usuários, a proteção não é necessária. Assim, um extremo seria fornecer proteção completa proibindo o acesso. O outro extremo é fornecer acesso livre sem proteção. As duas abordagens são extremas demais para uso geral. O que é necessário é o acesso controlado.

Os mecanismos de proteção fornecem acesso controlado limitando os tipos de acesso a arquivo que podem ser feitos. O acesso é permitido ou negado dependendo de vários fatores, um dos quais é o tipo de acesso solicitado. Vários tipos distintos de operações podem ser controladas:

- *Ler:* Ler um arquivo.
- *Escrever:* Escrever ou reescrever um arquivo.
- *Executar:* Carregar o arquivo na memória e executá-lo.
- *Anexar:* Escrever novas informações no final do arquivo.
- *Excluir:* Excluir o arquivo e liberar seu espaço para possível reutilização.

- *Listar:* Listar o nome e os atributos do arquivo.

Outras operações - tais como renomear, copiar ou editar o arquivo - também podem ser controladas. Para muitos sistemas, no entanto, essas funções de mais alto nível (tais como copiar) podem ser implementadas por um programa de sistema que faz chamadas de baixo nível ao sistema. A proteção é fornecida apenas no nível inferior. Por exemplo, copiar um arquivo pode ser implementado simplesmente por uma sequência de pedidos de leitura. Nesse caso, um usuário com acesso de leitura também poderá fazer o arquivo ser copiado, impresso etc.

Muitos mecanismos diferentes de proteção têm sido propostos. Cada esquema tem suas vantagens e desvantagens, por isso você deve selecionar aquele apropriado para o uso pretendido. Um sistema de computação pequeno que seja usado apenas por poucos membros de um grupo de pesquisa talvez não precise dos mesmos tipos de proteção que um computador de uma grande corporação utilizado para realizar operações de pesquisa, finanças e de pessoal. Nesta seção, discutimos a proteção e como ela se relaciona com o sistema de arquivos. No Capítulo 18, o tratamento completo do problema de proteção é apresentado.

11.4.2 Listas de acesso e grupos

A abordagem mais comum ao problema de proteção é tornar o acesso dependente da identidade do usuário. Vários usuários podem precisar de diferentes tipos de acesso a um arquivo ou diretório. O esquema mais geral para implementar o acesso dependente da identidade é associar a cada arquivo e diretório uma lista de acesso, especificando para cada nome de usuário listado os tipos de acesso permitidos. Quando um usuário solicita acesso a determinado arquivo, o sistema operacional verifica a lista de acesso associada àquele arquivo. Se o usuário estiver listado para o acesso solicitado, o acesso será permitido. Caso contrário, ocorrerá uma violação de proteção, e o job do usuário não receberá acesso ao arquivo. ^

O principal problema com as listas de acesso é o seu tamanho. Se queremos que todos leiam um arquivo, é preciso listar todos os usuários e dar a eles acesso de leitura. Essa técnica tem duas consequências indesejáveis:

1. Construir uma lista desse tipo pode ser uma tarefa entediante e pouco compensadora, especialmente se não soubermos de" antemão qual é a lista de usuários no sistema.
2. A entrada de diretório que anteriormente tinha tamanho fixo agora precisa ser de tamanho variável, resultando em uma gerência de espaço mais complicada.

Podemos resolver esses problemas usando uma versão condensada da lista de acesso.

Para condensar o tamanho da lista de acesso, muitos sistemas reconhecem três classificações de usuários em relação a cada arquivo:

- Proprietário: O usuário que criou o arquivo é o proprietário.
- Grupo: Um conjunto de usuários que compartilha o arquivo e precisa de acesso semelhante é um grupo, ou grupo de trabalho.
- Universo: Todos os outros usuários no sistema constituem o universo.

Como exemplo, considere uma pessoa, Sara, que está escrevendo um livro novo. Ela contratou três alunos de pós-graduação (Jim, Dawn e Jill) para ajudá-la no projeto. O texto do livro é mantido em um arquivo chamado *book.* A proteção associada a esse arquivo é a seguinte:

- Sara deve ser capaz de invocar todas as operações sobre o arquivo.
- Jim, Dawn e Jill só devem ser capazes de ler e gravar o arquivo; não devem ter permissão para excluir o arquivo.
- Todos os outros usuários devem ter permissão para ler o arquivo, mas não para gravá-lo ou excluí-lo. (Sara está interessada em deixar o maior número possível de pessoas lerem o texto de modo que ela possa obter o retorno apropriado.)

Para chegar a essa proteção, é preciso criar um novo grupo, como text, com os membros Jim, Dawn e Jill. O nome do grupo text deve ser então associado ao arquivo *book,* e o direito de acesso deve ser definido de acordo com a política que descrevemos.

Observe que, para esse esquema funcionar adequadamente, os membros do grupo devem ser muito bem controlados. Esse controle pode ser alcançado de várias formas. Por exemplo, no sistema UNIX, os grupos podem ser criados e modificados apenas pelo gerente da instalação (ou qualquer superusuário). Assim, esse controle é obtido através da interação humana. O sistema VMS utiliza listas de acesso.

Com essa classificação de proteção mais limitada, só precisamos de três campos para definir a proteção. Cada campo é geralmente uma coleção de bits, cada qual permite ou impede o acesso associado a ele. Por exemplo, o sistema UNIX define três campos de 3 bits cada: rwx, onde r controla o acesso de leitura *(read),* w controla o acesso de escrita *(write) ex* controla a execução *(execution).* Campos separados são mantidos para o proprietário do arquivo, para o grupo do arquivo e para todos os outros usuários. Nesse esquema, 9 bits por arquivo são necessários para registrar as informações de proteção. Assim, no nosso exemplo, os campos de proteção para o arquivo book são os seguintes: para o proprietário Sara, todos os 3 bits estão ativos; para o grupo text, os bits r e w estão ativos e para o universo, apenas o bit r está ativo.

Observe, no entanto, que esse esquema não é tão geral quanto o esquema de lista de acesso. Para ilustrar nossa posição, vamos voltar ao exemplo do livro. Vamos supor que Sara tenha uma discussão séria com Jason e agora queira excluí-lo da lista de pessoas que podem ler o texto. Ela não pode fazer isso usando o esquema de proteção básico definido.

11.4.3 Outras abordagens de proteção

Existem outras abordagens ao problema de proteção. Uma delas é associar uma senha com cada arquivo. Assim como o acesso a um sistema de computador por si só é geralmente controlado por uma senha, o acesso a cada arquivo pode ser controlado por uma senha. Se as senhas forem escolhidas aleatoriamente e alteradas com frequência, esse esquema pode ser eficaz na limitação do acesso a um arquivo apenas aos usuários que conheçam a senha. Existem, no entanto, várias desvantagens nesse esquema. Em primeiro lugar, se associarmos uma senha separada a cada arquivo, o número de senhas que um usuário deve se lembrar pode aumentar muito, tornando o esquema impraticável. Se apenas uma senha for usada para todos os arquivos, então, assim que ela for descoberta, todos os arquivos serão acessíveis. Alguns sistemas (por exemplo, o TOPS-20) permitem que um usuário associe uma senha com um subdiretório, em vez de com urruarquivo individual, para lidar com esse problema. O sistema operacional VM/CMS da IBM permite três senhas para um minidisco: uma para acesso de leitura, outra para escrita e outra para multiescrita. Em segundo lugar, geralmente apenas uma senha está associada a cada arquivo. Assim, a proteção funciona com base em tudo ou nada. Para fornecer proteção em um nível mais detalhado, devemos usar múltiplas senhas.

A proteção de arquivo limitada também está disponível em sistemas monousuário, como o MS-DOS e o sistema operacional do Macintosh. Esses sistemas operacionais, quando foram projetados, basicamente ignoravam o problema de proteção. Agora, no entanto, eles estão sendo colocados em redes nas quais o compartilhamento de arquivos e a comunicação são necessários, de modo que mecanismos de proteção estão sendo preparados para eles. E quase sempre mais fácil incluir um recurso no projeto original de um sistema operacional do que adicionar um recurso a um sistema existente. Tais atualizações geralmente são pouco eficazes e não são transparentes.

Em uma estrutura de diretório de multinível, precisamos não só proteger os arquivos individuais, mas também proteger coleções de arquivos contidos em um subdiretório; ou seja, precisamos fornecer um mecanismo para a proteção de diretório. As operações de diretório que precisam ser protegidas são ligeiramente diferentes das operações de arquivo. Devemos controlar a criação e exclusão de arquivos em um diretório. Além disso, provavelmente vamos querer controlar se um usuário pode ou não determinar a existência de um arquivo no diretório. Às vezes, o conhecimento da existência e do nome de um arquivo pode ser significativo por si só. Assim, listar o conteúdo de um diretório deve ser uma operação protegida. Portanto, se um nome de caminho fizer referência a um arquivo em um diretório, o usuário deverá ter acesso ao diretório e ao arquivo. Nos sistemas em que os arquivos podem ter vários nomes de caminho (como grafos acíclicos ou genéricos), os usuários podem ter diferentes direitos de acesso a um arquivo, dependendo dos nomes de caminho utilizados.

11.4.4 Um exemplo: UNIX

No sistema UNIX, a proteção de diretório é tratada como proteção de arquivo. Ou seja, associados com cada subdiretório estão três campos - proprietário, grupo e universo - cada qual consistindo nos 3 bits rwx. Assim, um usuário só pode listar o conteúdo de um subdiretório se o bit r estiver ativo no campo apropriado. Da mesma forma, um usuário poderá mudar seu diretório corrente para outro diretório (digamos *foo)* somente se o bit x associado com o subdiretório *foo* estiver ativo no campo apropriado.

Um exemplo de listagem de diretório de um ambiente UNIX está apresentado na Figura 11.12. O primeiro campo descreve a proteção de arquivo ou diretório. Um d como primeiro caractere indica um subdiretório. Também indicado está o número de links para o arquivo, o nome do proprietário, o nome do grupo, o tamanho do arquivo em unidades de bytes, a data de criação e finalmente o nome do arquivo (com extensão opcional).

-rw-rw-r-	1pbg	staff	31200	Sep3 08:30	intro.ps
drwx———	5pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2pbg	staff	512	Jul 8 09:35	doe/
drwxrwx—	2pbg	student	512	Aug3 14:13	student-proj/
-rw-r-r-	1pbg	staff	9423	Feb 24 1998	program.c
-rwxr-xr-x	1pbg	staff	20471	Feb 24 1998	program
drw-x-x	4pbg	faculty	512	Jul 31 10:31	lib/
drwx———	3pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3 pbg	staff	512	Jul 8 09:35	test/

Figura 11.12 Um exemplo de listagem de diretórios.

.5 • Estrutura do sistema de arquivos

Os discos são a principal parte do armazenamento secundário no qual um sistema de arquivos é mantido. Para melhorar a eficiência de I/O, realizamos transferências entre a memória e o disco em unidades de blocos. Cada bloco tem um ou mais setores. Os setores em diferentes unidades de disco variam de 32 bytes a 4.096 bytes; geralmente, eles têm 512 bytes. Os discos têm duas características importantes que os tornam um meio conveniente para armazenar múltiplos arquivos:

1. Eles podem ser regravados; é possível ler um bloco do disco, modificar o bloco e gravá-lo de volta na mesma posição.
2. Podemos acessar diretamente qualquer bloco de informações no disco. Assim, é simples acessar qualquer arquivo quer sequencial ou aleatoriamente, e alternar de um arquivo a outro requer apenas mover as cabeças de leitura e escrita e esperar que o disco gire.

O Capítulo 13 discute a estrutura do disco.

11.5.1. Organização do sistema de arquivos

Para fornecer um acesso eficiente e conveniente ao disco, o sistema operacional impõe um sistema de arquivos para que os dados sejam facilmente armazenados, localizados e recuperados. Desenvolver um sistema de arquivos gera dois problemas de projeto diferentes. O primeiro problema é definir como o sistema de arquivos deve se apresentar ao usuário. Essa tarefa envolve a definição de um arquivo e seus atributos, das operações permitidas em um arquivo e da estrutura de diretório para organizar os arquivos. Em seguida, os algoritmos e as estruturas de dados devem ser criados para mapear o sistema de arquivos lógico nos dispositivos físicos de armazenamento secundário.

O sistema de arquivos por si só geralmente é composto por muitos níveis diferentes. A estrutura apresentada na Figura 11.13 é um exemplo de um projeto em camadas. Cada nível no projeto utiliza os recursos dos níveis inferiores para criar novos recursos a serem utilizados pelos níveis superiores.

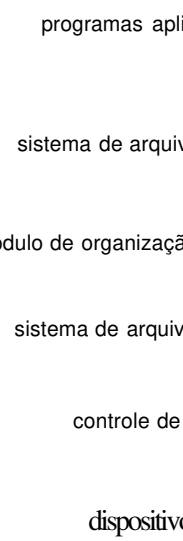


Figura 11.13 Sistema de arquivos em camadas.

O nível mais baixo, o **controle de I/O,** consiste em **drivers de dispositivo** e rotinas de tratamento de interrupções que transferem informações entre a memória e o sistema de disco. Um driver de dispositivo pode ser visto como um tradutor. Sua entrada consiste em comandos de alto nível, tais como "retrieve block 123". Sua saída consiste em instruções específicas de hardware, de baixo nível, que são utilizadas pela controladora do hardware, que faz a interface do dispositivo de entrada/saída com o resto do sistema. O driver de dispositivo geralmente escreve padrões específicos de bits em posições especiais na memória da controladora de I/O para informá-la sobre a posição do dispositivo onde atuar e que ações devem ser tomadas. Os drivers de dispositivo **e** a infra-estrutura de I/O são discutidos no Capítulo 12.

O sistema de arquivos básico só precisa emitir comandos genéricos ao driver de dispositivo apropriado para ler e gravar blocos físicos no disco. Cada bloco físico é identificado pelo seu endereço numérico de disco (por exemplo, unidade 1, cilindro 73, trilha 2, setor 10).

O **módulo de organização de arquivos** conhece os arquivos e seus blocos lógicos, assim como os blocos físicos. Ao conhecer o tipo de alocação de arquivo usada e a posição do arquivo, o módulo de organização de arquivos pode traduzir os endereços de bloco lógico em endereços de bloco físico para que o sistema de arquivos básico faça a transferência. Cada bloco lógico de arquivo é numerado de 0 (ou 1) a N, enquanto os blocos físicos contendo os dados geralmente não correspondem aos números lógicos, por isso há necessidade de tradução para localizar cada bloco. O módulo de organização de arquivos também inclui um gerenciador de espaço livre, que rastreia os blocos não-alocados e fornece esses blocos ao módulo de organização de arquivos quando necessário.

Finalmente, o **sistema de arquivos lógico** utiliza a estrutura de diretório para fornecer ao módulo de organização de arquivos as informações necessárias, considerando um nome de arquivo simbólico. O sistema de arquivos lógico também é responsável pela proteção e segurança, conforme discutido na Seção 11.4 e que será abordado em maiores detalhes no Capítulo 18.

Para criar um novo arquivo, um programa aplicativo chama o sistema de arquivos lógico. Este, por sua vez, conhece o formato das estruturas de diretório. Para criar um novo arquivo, ele lê o diretório apropriado na memória, atualiza-o com a nova entrada e o grava de volta no disco. Alguns sistemas operacionais, incluindo o UNIX, tratam um diretório exatamente como um arquivo - com um campo de tipo indicando que é um diretório. Outros sistemas operacionais, incluindo o Windows NT, implementam chamadas ao sistema separadas para arquivos e diretórios e tratam os diretórios como entidades separadas dos arquivos. Quando um diretório é tratado como um arquivo especial, o sistema de arquivos lógico pode chamar o módulo de organização de arquivos para mapear as operações de I/O do diretório em números de bloco de disco, que são passados para o sistema de arquivos básico e o sistema de controle de I/O.

Agora que o arquivo foi criado, ele pode ser usado para I/O. Para cada operação de I/O, a estrutura de diretório poderia ser pesquisada para encontrar o arquivo, seus parâmetros poderiam ser verificados, seus blocos de dados examinados e, finalmente, a operação nesses blocos de dados realizada. Cada operação envolve um

alto custo. Em vez disso, antes que o arquivo seja utilizado para procedimentos de I/O, ele precisa ser **aberto.** Quando um arquivo é aberto, a estrutura de diretório é pesquisada para encontrar a entrada de arquivo desejada. Partes da estrutura de diretório são armazenadas em cache para acelerar as operações de diretório. Assim que o arquivo é encontrado, as informações a ele associadas - como tamanho, proprietário, permissões de acesso e posições dos blocos de dados - são copiadas para uma tabela na memória. Essa tabela de **arquivos abertos** contém informações sobre todos os arquivos abertos no momento (Figura 11.14).

A primeira referência a um arquivo (normalmente open) faz com que a estrutura do diretório seja pesquisada e que a entrada de diretório para esse arquivo seja copiada para a tabela de arquivos abertos. O índice nessa tabela é devolvido ao programa de usuário, e todas as referências futuras são feitas através do índice em vez do nome simbólico. O nome dado ao índice varia. Os sistemas UNIX referem-se a ele como **descritor de arquivo** *(file descriptor),* o Windows NT como **handle de arquivo** e os outros sistemas como **bloco de controle de arquivo.** Consequentemente, desde que o arquivo não esteja fechado, todas as operações de arquivo são feitas na tabela de arquivos abertos. Quando o arquivo é fechado por todos os usuários que o abriram, as informações do arquivo atualizado são copiadas para a estrutura de diretório baseada no disco.

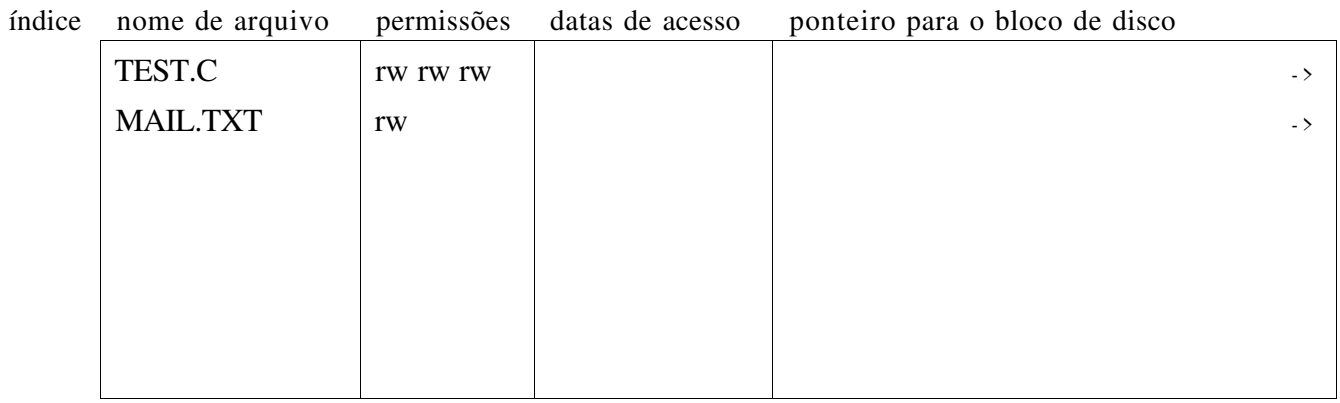


Figura 11.14 Uma tabela de arquivos abertos típica.

Alguns sistemas complicam esse esquema ainda mais usando multinível de tabelas na memória. Por exemplo, no sistema de arquivos do UNIX BSD, cada processo tem uma tabela de arquivos abertos que armazena uma lista de ponteiros, indexados por descritor. Os ponteiros levam a uma tabela de arquivos abertos em todo o sistema. Essa tabela contém informações sobre a entidade subjacente que está aberta. Para arquivos, ela aponta para uma tabela de inodes ativos. Para outras entidades, como conexões de rede e dispositivos, ela aponta para informações de acesso semelhantes. A **tabela de inodes ativos** é um cache em memória dos inodes em uso no momento, e inclui os campos de índice de inode que apontam para os blocos de dados no disco. Assim que um arquivo é aberto, tudo está na memória para acesso rápido por qualquer processo que acessa o arquivo, menos os blocos de dados reais. Na verdade, open primeiro pesquisa a tabela de arquivos abertos para ver se o arquivo já está em uso por outro processo. Se o arquivo estiver em uso, uma entrada na tabela local de arquivos abertos é criada apontando para a tabela global de arquivos abertos. Se o arquivo não estiver em uso, o inode é copiado para a tabela de inodes ativos e são criadas uma nova entrada global e uma nova entrada local.

O sistema UNIX BSD é típico na sua forma de utilizar os caches sempre que operações de I/O em disco puderem ser economizadas. Sua taxa média de acerto de cache de 85% mostra que essas técnicas são de implementação válida. O sistema UNIX BSD está descrito no Capítulo 20. A tabela de arquivos abertos está detalhada na Seção 11.1.2.

11.5.2 Montagem do sistema de arquivos

Assim como um arquivo deve estar aberto antes de ser usado, um sistema de arquivos deve ser **montado** antes de poder estar disponível aos processos no sistema. O procedimento de montagem simples. O sistema operacional recebe o nome do dispositivo e a posição na estrutura de arquivos na qual anexar o sistema de arquivos

(chamado ponto de montagem). Por exemplo, em um sistema UNIX, o sistema de arquivos que contém os diretórios de trabalho dos usuários pode ser montado como */home;* em seguida, para acessar a estrutura de diretório naquele sistema de arquivos, precedemos os nomes de diretórios com */home,* como em */home/jane.* Montar esse sistema de arquivos em *lusers* permitiria ao usuário utilizar o nome de caminho */users/jane* para alcançar o mesmo diretório.

Em seguida, o sistema operacional verifica se o dispositivo contém um sistema de arquivos válido. Isso é feito pedindo ao driver de dispositivo para ler o diretório do dispositivo e verificar se o diretório tem o formato esperado. Finalmente, o sistema operacional registra na sua estrutura de diretório que um sistema de arquivos está montado no ponto de montagem especificado. Esse esquema permite que o sistema operacional percorra sua estrutura de diretório, alternando entre os sistemas de arquivos, conforme apropriado.

Considere as ações do Sistema Operacional do Macintosh. Sempre que o sistema encontra um disco pela primeira vez (os discos rígidos são encontrados no momento de inicialização, os disquetes são vistos quando são inseridos na unidade), o Sistema Operacional do Macintosh pesquisa o sistema de arquivos no dispositivo. Se encontrar um, ele automaticamente montará o sistema de arquivos no nível da raiz, adicionando um ícone de pasta na tela identificada com o nome do sistema de arquivos (conforme armazenado no diretório do dispositivo). O usuário pode então clicar no ícone e assim exibir o sistema de arquivos recém-montado.

A montagem de sistemas de arquivos é discutida em maiores detalhes nas Seções 17.6 e 20.7.5.

11.6 • Métodos de alocação

A natureza de acesso direto dos discos permite flexibilidade na implementação de arquivos. Em quase todos os casos, muitos arquivos serão armazenados no mesmo disco. O principal problema é como alocar espaço a esses arquivos de modo que o espaço em disco seja utilizado com eficácia e os arquivos sejam acessados rapidamente. Existem três métodos principais muito utilizados para alocar espaço em disco: *contíguo, encadeado* e *indexado.* Cada método tem suas vantagens e desvantagens. Da mesma forma, alguns sistemas (tais como RDOS da Data General para sua linha de computadores Nova) suportam todos os três métodos. Mais comumente, um sistema usará um método particular para todos os arquivos.

11.6.1 Alocação contígua

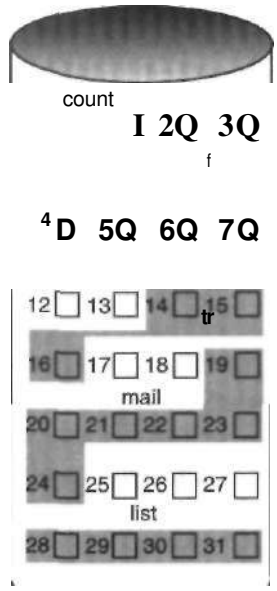
O método de alocação contígua requer que cada arquivo ocupe um conjunto de blocos contíguos no disco. Os endereços de disco definem uma ordenação linear no disco. Observe que, com essa ordenação, considerando que apenas um job está acessando o disco, um acesso ao bloco $b + 1$ depois do bloco b normalmente não exige movimento da cabeça do disco. Quando movimento é necessário (do último setor de um cilindro para o primeiro setor do próximo cilindro), é só de uma trilha. Assim, o número de buscas de disco necessárias para acessar arquivos alocados contiguamente é mínimo, assim como o tempo de busca quando uma busca finalmente é necessária. O sistema operacional VM/CMS da IBM utiliza alocação contígua porque essa abordagem fornece um bom desempenho.

A alocação contígua de um arquivo é definida pelo endereço de disco (do primeiro bloco) e tamanho (em unidades de blocos). Se o arquivo começar na posição b e tiver n blocos de comprimento, ele ocupará os blocos fr, $b + l,b + 2,...,b +$ « - 1 . A entrada de diretório para cada arquivo indica o endereço do bloco de início e o tamanho da área alocada para esse arquivo (Figura 11.15).

Acessar um arquivo que tenha sido alocado de forma contígua é fácil. Para acesso sequencial, o sistema de arquivos lembra do endereço de disco do último bloco referenciado e, quando necessário, lê o próximo bloco. Para acesso direto ao bloco / de um arquivo que começa no bloco *b,* podemos acessar imediatamente o bloco $b + /'$. Portanto, tanto o acesso sequencial quanto o direto podem ser suportados pela alocação contígua.

Uma dificuldade com a alocação contígua é encontrar espaço para um novo arquivo. A implementação do sistema de gerência de espaço livre, discutida na Seção 11.7, determina como essa tarefa é realizada. Qualquer sistema de gerência pode ser usado, mas alguns são mais lentos do que outros.

O problema de alocação contígua de espaço de disco pode ser considerado uma aplicação específica do problema de alocação dinâmica de memória genérica discutido na Seção 9.3, que consiste em como atender um pedido de tamanho n de uma lista de blocos de memória livres. As estratégias de *first fit* e *best fit* são as



diretório

arquivo	início	tamanho
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Figura 11.15 Alocação contígua de espaço em disco.

mais comuns que podem ser usadas para selecionar um bloco de memória livre do conjunto de blocos de memória disponíveis. As simulações indicam que tanto uma quanto a outra são mais eficientes do que a estratégia de *worst fit* em termos de utilização de tempo e memória. *First fit* e *best fit* têm desempenho semelhante em termos de utilização de memória, mas *first fit* geralmente é mais rápido.

Esses algoritmos sofrem do problema de **fragmentação externa.** A medida que os arquivos são alocados e excluídos, o espaço livre em disco é dividido em pequenas partes. A fragmentação externa existe sempre que o espaço livre é dividido em pedaços. Torna-se um problema quando o maior pedaço contíguo é insuficiente para um pedido; o armazenamento é fragmentado em uma série de blocos livres, nenhum dos quais é grande o suficiente para armazenar dados. Dependendo do tamanho total de armazenamento em disco e do tamanho médio dos arquivos, a fragmentação externa pode ser um problema mais ou menos grave.

Alguns sistemas de microcomputador mais antigos usavam a alocação contígua em disquetes. Para evitar a perda de quantidades significativas de espaço em disco devido à fragmentação externa, o usuário tinha de executar uma rotina de recompactação que copiava todo o sistema de arquivos para outro disquete ou para uma fita. O disquete de origem era então completamente liberado, criando um grande espaço livre contíguo. A rotina então copiava os arquivos de volta para o disquete, alocando espaço contíguo desse grande bloco de armazenamento. Esse esquema efetivamente **compacta** todo o espaço livre em um espaço contíguo, resolvendo o problema de fragmentação. O custo dessa compactação é o tempo. O custo de tempo é particularmente grave para grandes discos rígidos que utilizam alocação contígua, onde compactar todo o espaço pode levar horas e talvez seja necessário semanalmente. Durante esse **tempo indisponível,** a operação normal do sistema normalmente não é permitida, de modo que essa compactação é evitada a todo custo em máquinas de produção.

Existem outros problemas com a alocação contígua. Um problema importante é determinar quanto espaço é necessário para um arquivo. Quando o arquivo é criado, o total de espaço que será necessário precisará ser encontrado e alocado. Como o criador (programa ou pessoa) sabe o tamanho do arquivo a ser criado? Em alguns casos, essa determinação pode ser razoavelmente simples (copiar um arquivo existente, por exemplo); em geral, no entanto, o tamanho de um arquivo de saída pode ser difícil de estimar.

Se alocarmos pouco espaço para um arquivo, talvez esse arquivo não possa ser estendido. Especialmente com a estratégia de alocação *best-fit,* o espaço nos dois lados do arquivo pode estar em uso. Portanto, não é possível tornar o arquivo maior. Existem então duas possibilidades. Primeiro, o programa de usuário pode ser encerrado, com uma mensagem de erro apropriada. O usuário deverá então alocar mais espaço e executar o programa novamente. Essas execuções repetidas podem ser caras. Para evitá-las, o usuário normalmente superestimará a quantidade de espaço necessária, resultando em considerável espaço desperdiçado.

A outra possibilidade é encontrar um bloco livre maior, para copiar o conteúdo do arquivo para o novo espaço e liberar o espaço anterior. Essa série de ações pode ser repetida enquanto houver espaço, embora também seja muito demorada. Observe, no entanto, que nesse caso, o usuário não precisa ser informado explicitamente sobre o que está acontecendo; o sistema continua apesar do problema, embora cada vez mais lento.

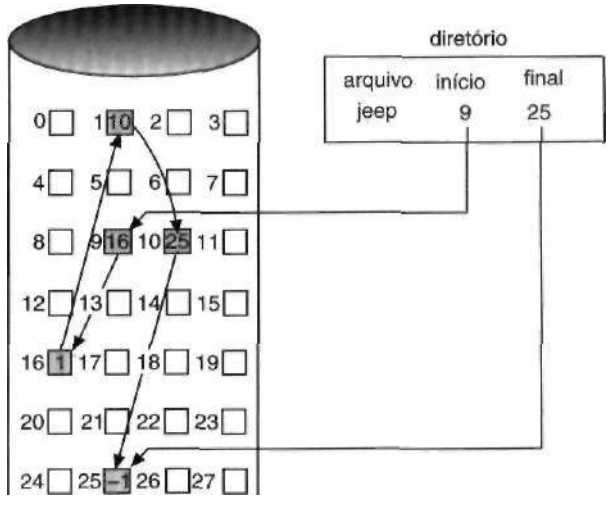
Mesmo que a quantidade de espaço total necessária para um arquivo seja conhecida de antemão, a pré-alocação pode ser ineficiente. Um arquivo que cresce lentamente em um longo período (meses ou anos) deve receber espaço suficiente para seu tamanho final, embora boa parte desse espaço fique sem uso durante um longo período. O arquivo, portanto, tem uma grande quantidade de *fragmentação interna.*

Para evitar várias dessas desvantagens, alguns sistemas operacionais utilizam um esquema de alocação contígua modificado, no qual um pedaço de espaço contíguo é alocado inicialmente e, em seguida, quando essa quantidade não é grande o suficiente, outro pedaço de espaço contíguo, uma extensão, é adicionada à alocação inicial. A posição dos blocos de um arquivo é registrada como uma posição e um contador de blocos, mais uma ligação ao primeiro bloco da próxima extensão. Em alguns sistemas, o proprietário do arquivo pode definir o tamanho da zona de extensão, mas essa definição resultará em ineficiências caso o proprietário esteja incorreto. A fragmentação interna ainda poderá ser um problema se as extensões forem muito grandes, e a fragmentação externa poderá ser um problema à medida que as extensões de tamanho variável são alocadas e desalocadas.

11.6.2 Alocação encadeada

A alocação encadeada resolve todos os problemas de alocação contígua. Com a alocação encadeada, cada arquivo é uma lista encadeada de blocos de disco; os blocos de disco podem estar dispersos em qualquer parte do disco. O diretório contém um ponteiro ao primeiro e último blocos do arquivo. Por exemplo, um arquivo de cinco blocos pode começar no bloco 9, continuar no bloco 16, depois bloco 1, bloco 10 e finalmente bloco 25 (Figura 11.16). Cada bloco contém um ponteiro ao próximo bloco. Esses ponteiros não são disponibilizados para o usuário. Assim, se cada bloco tiver 512 bytes, e um endereço de disco (o ponteiro) requer 4 bytes, então o usuário verá blocos de 508 bytes.

Para criar um novo arquivo, simplesmente criamos uma nova entrada no diretório. Com a alocação encadeada, cada entrada de diretório tem um ponteiro ao primeiro bloco de disco do arquivo. Esse ponteiro é inicializado para *nil* (o valor de fim de lista) para significar um arquivo vazio. O campo de tamanho também é ajustado para 0. Uma escrita no arquivo faz com que um bloco livre seja encontrado através do sistema de gerência de espaço livre, e esse novo bloco é então gravado e encadeado no final do arquivo. Para ler um arquivo, simplesmente fazemos a leitura dos blocos seguindo os ponteiros de bloco a bloco.



28n29D30D3ID

Figura 11.16 Alocação encadeada de espaço em disco.

Não existe fragmentação externa com alocação encadeada, e podemos usar qualquer bloco livre na lista de espaço livre para atender a um pedido. Observe também que não há necessidade de declarar o tamanho de um arquivo quando esse arquivo é criado. Um arquivo pode continuar a crescer desde que hajam blocos livres. Consequentemente, nunca será necessário compactar espaço em disco.

Entretanto, a alocação encadeada possui desvantagens também. O principal problema é que ela pode ser usada efetivamente apenas para arquivos de acesso sequencial. Para encontrar o bloco *i* de um arquivo, é preciso começar no início desse arquivo e seguir os ponteiros até chegar ao bloco *i.* Cada acesso a um ponteiro requer uma leitura de disco e, às vezes, uma busca no disco. Consequentemente, é ineficiente suportar uma capacidade de acesso direto para arquivos com alocação encadeada.

Outra desvantagem da alocação encadeada é o espaço necessário para os ponteiros. Se um ponteiro precisar de 4 bytes de um bloco de 512 bytes, então 0,78% do disco estará sendo usado para ponteiros em vez de para informações. Cada arquivo requer um pouco mais de espaço do que seria necessário de outro modo.

A solução normal para esse problema seria agrupar blocos em múltiplos, chamados **clusters,** e alocar clusters em vez de blocos. Por exemplo, o sistema de arquivos pode definir um cluster como 4 blocos, e operar no disco apenas em unidades de cluster. Os ponteiros então utilizam uma percentagem bem menor do espaço em disco do arquivo. Esse método permite que o mapeamento de bloco lógico em físico permaneça simples, mas melhora o throughput de disco (menos buscas da cabeça do disco) e diminui o espaço necessário para a alocação de blocos e a gerência da lista de espaço livre. O custo dessa abordagem é um aumento na fragmentação interna, porque mais espaço será desperdiçado se um cluster estiver parcialmente cheio do que quando um bloco estiver parcialmente cheio. Os clusters podem melhorar o tempo de acesso ao disco para muitos outros algoritmos, por isso são usados na maioria dos sistemas operacionais.

Outro problema é a confiabilidade. Como os arquivos são mantidos encadeados por ponteiros dispersos em todo o disco, considere o que aconteceria se um ponteiro fosse perdido ou danificado. Um bug no software do sistema operacional ou uma falha no hardware de disco poderia resultar na escolha do ponteiro errado. Esse erro resultaria em um encadeamento na lista de espaço livre ou em outro arquivo. As soluções parciais são listas de encadeamento duplo ou armazenar o nome de arquivo e o número de bloco relativo em cada bloco; no entanto, esses esquemas requerem ainda mais custo para cada arquivo.

Uma variação importante no método de alocação encadeada é o uso de uma **tabela** de alocação de arquivos **(File Allocation Table - FAT).** Esse método simples mas eficiente de alocação de espaço em disco é usado pelos sistemas operacionais MS-DOS e OS/2. Uma seção de disco no início de cada partição é reservada para conter a tabela. A tabela tem uma entrada para cada bloco de disco, e é indexada pelo número de bloco. A FAT é usada como uma lista encadeada. A entrada de diretório contém o número de bloco do primeiro bloco do arquivo. A entrada da tabela indexada pelo número de bloco contém o número do próximo bloco no arquivo. Essa cadeia continua até o último bloco, que tem um valor especial "fim-de-arquivo" como entrada na tabela. Blocos não usados são indicados por um valor 0. Alocar um novo bloco a um arquivo é uma simples questão de encontrar a primeira entrada na tabela com valor 0, e substituir o valor fim-de-arquivo anterior pelo endereço do novo bloco. O 0 é então substituído pelo valor fim-de-arquivo. Um exemplo ilustrativo é a estrutura FAT da Figura 11.17 para um arquivo que consiste nos blocos de disco 217, 618 e 339.

Observe que o esquema de alocação FAT pode resultar em um número significativo de buscas da cabeça de disco, a menos que haja cache da FAT. A cabeça do disco deve mover-se para o início da partição para ler a FAT e encontrar o local do bloco em questão, depois mover-se para a posição do bloco propriamente dito. Na pior das hipóteses, as duas movimentações ocorrem para cada um dos blocos. Uma vantagem é que o tempo de acesso aleatório é melhorado, porque a cabeça do disco pode encontrar a posição de qualquer bloco lendo as informações na FAT.

11.6.3 Alocação indexada

A alocação encadeada resolve os problemas de fragmentação externa e de declaração de tamanho da alocação contígua.

No entanto, na ausência de FAT, a alocação encadeada não pode suportar acesso direto eficiente, já que os ponteiros aos blocos estão dispersos nos próprios blocos em todo o disco e precisam ser recuperados em ordem. A **alocação indexada** resolve esse problema reunindo todos os ponteiros em um só local: o **bloco** de **índice.**

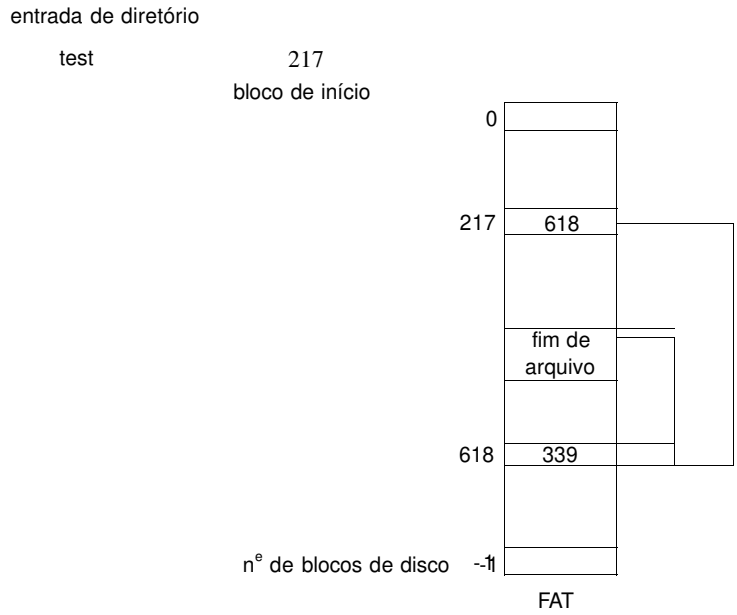


Figura 11.17 Tabela de alocação de arquivos.

Cada arquivo tem seu próprio bloco de índice, que é um vetor de endereços de bloco de disco. A entrada *i* no bloco de índice aponta para o bloco *i* do arquivo. O diretório contém o endereço do bloco de índice (Figura 11.18). Para ler o bloco i, usamos o ponteiro na entrada *i* do bloco de índice para encontrar e ler o bloco desejado. Esse esquema é semelhante ao esquema de paginação descrito no Capítulo 9.

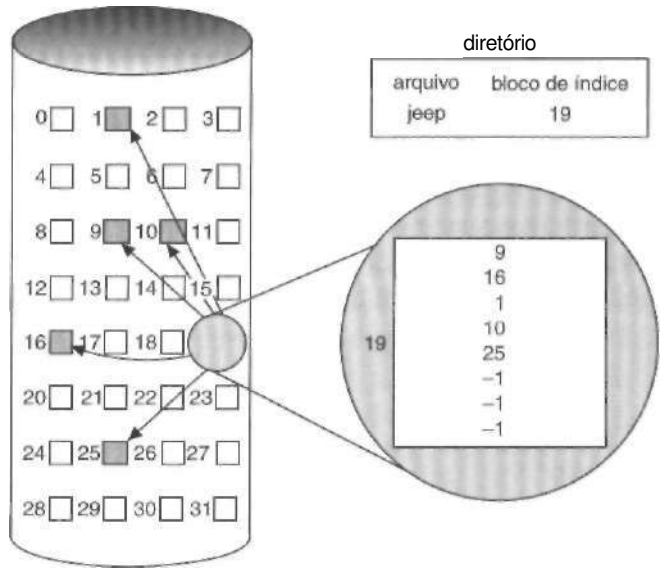


Figura 11.18 Alocação indexada de espaço de disco.

Quando o arquivo é criado, todos os ponteiros no bloco de índice são definidos como *nil.* Quando o bloco *i* é gravado pela primeira vez, um bloco é obtido do gerenciador de espaço livre e o seu endereço é colocado na entrada *i* do bloco de índice.

A alocação indexada suporta o acesso direto, sem sofrer de fragmentação externa, porque qualquer bloco livre no disco pode satisfazer um pedido por mais espaço.

A alocação indexada sofre de desperdício de espaço. O custo dos ponteiros do bloco de índice é geralmente maior do que o custo dos ponteiros da alocação encadeada. Considere um caso comum no qual temos um arquivo com apenas um ou dois blocos. Com a alocação encadeada, perdemos o espaço de apenas um ponteiro por bloco (um ou dois ponteiros). Com a alocação indexada, um bloco de índice inteiro deverá ser alocado, mesmo se apenas um ou dois ponteiros serão *não-nil.*

Este ponto levanta a questão de qual deve ser o tamanho de um bloco de índice. Todo arquivo deve ter um bloco de índice, por isso esse bloco deve ser o menor possível. Entretanto, se o bloco de índice for pequeno demais, ele não poderá conter ponteiros suficientes para um arquivo grande, e um mecanismo deverá estar disponível para tratar essa questão:

- *Esquema encadeado:* Um bloco de índice geralmente ocupa um bloco de disco. Assim, ele pode ser lido e gravado diretamente por si só. Para permitir a existência de arquivos grandes, vários blocos de índice podem ser ligados entre si. Por exemplo, um bloco de índice pode conter um pequeno cabeçalho com o nome do arquivo e um conjunto dos primeiros 100 endereços de bloco de disco. O próximo endereço (a última palavra no bloco de índice) é *nil* (para um arquivo pequeno) ou um ponteiro para outro bloco de índice (para um arquivo grande).
- *índice de multinível:* Uma variante da representação encadeada é usar um bloco de índice de primeiro nível para apontar para um conjunto de blocos de índice de segundo nível, que, por sua vez, aponta para os blocos de arquivo. Para acessar um bloco, o sistema operacional utiliza o índice de primeiro nível para localizar um bloco de índice de segundo nível, e esse bloco para localizar o bloco de dados desejado. Essa abordagem pode ainda ser estendida a um terceiro ou quarto níveis, dependendo do tamanho máximo de arquivo desejado. Com blocos de 4.096 bytes, podemos armazenar 1.024 ponteiros de 4 bytes em um bloco de índice. Dois níveis de índice permitem 1.048.576 blocos de dados, possibilitando um arquivo de até 4 gigabytes.
- *Esquema combinado:* Outra alternativa, usada no sistema UNIX BSD, é manter os primeiros 15 ponteiros, por exemplo, do bloco de índice no bloco de índice do arquivo (ou inode). (A entrada de diretório aponta para o inode, como discutido na Seção 20.7.) Os primeiros 12 ponteiros apontam para **blocos diretos;** ou seja, contêm endereços de blocos que contêm dados do arquivo. Assim, os dados para arquivos pequenos (não mais do que 12 blocos) não precisam de um bloco de índice separado. Se o tamanho do bloco for 4K, até 48K de dados podem ser acessados diretamente. Os três ponteiros seguintes apontam para **blocos indiretos.** O primeiro ponteiro de bloco indireto é o endereço de um **bloco** indireto **simples.** O bloco indireto simples é um bloco de índice que não contém dados, mas endereços dos blocos que contêm dados. Em seguida, existe um ponteiro de **bloco indireto duplo,** que contém o endereço de um bloco que contém os endereços dos blocos que contêm ponteiros aos blocos de dados reais. O último ponteiro conteria o endereço de um **bloco indireto triplo.** Usando esse método, o número de blocos que podem ser alocados a um arquivo excede a quantidade de espaço endereçável pelos ponteiros de arquivo de 4 bytes usados por muitos sistemas operacionais. Um ponteiro de arquivo de 32 bits atinge apenas 2^{32} bytes, ou 4 gigabytes. Um inode aparece na Figura **11.19.**

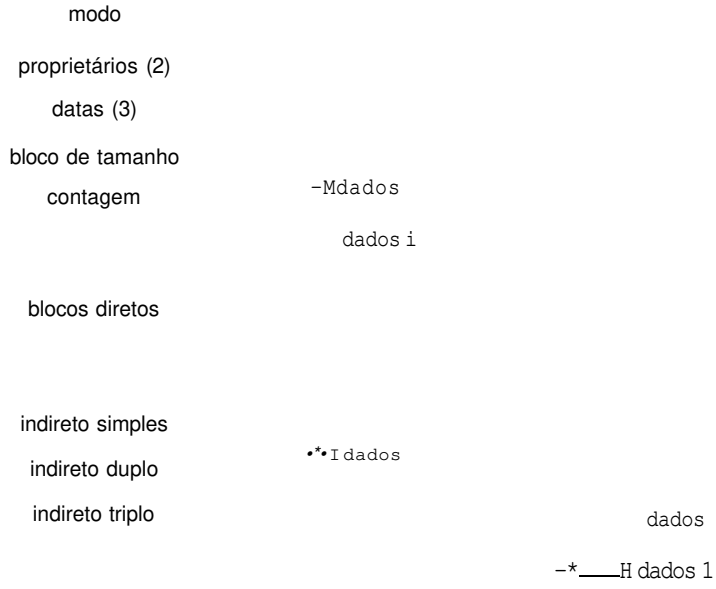


Figura 11.19 O inode UNIX.

Observe que a alocação indexada sofre de alguns dos mesmos problemas de desempenho existentes na alocação encadeada. Especificamente, os blocos de índice podem ser armazenados em cache na memória, mas os blocos de dados podem estar dispersos em toda a partição.

11.6.4 Desempenho

Os métodos de alocação que discutimos variam em termos de eficiência de armazenamento e tempos de acesso aos blocos de dados. Ambos são critérios importantes na Seleção do(s) método(s) adequado(s) para implementação por um sistema operacional.

Uma dificuldade na comparação do desempenho dos vários sistemas é determinar como os sistemas serão usados. Um sistema onde prevalece o acesso sequencial deve usar um método diferente de um sistema com acesso basicamente aleatório. Para qualquer tipo de acesso, a alocação contígua requer apenas um acesso para obter um bloco de disco. Como podemos facilmente manter o endereço inicial do arquivo na memória, podemos calcular imediatamente o endereço de disco do bloco *i* (ou o bloco seguinte) e lê-lo diretamente.

Para a alocação encadeada, também podemos manter o endereço do próximo bloco na memória e fazer a leitura diretamente. Esse método funciona para acesso sequencial, para o acesso direto, no entanto, um acesso ao bloco *i* pode exigir *i* leituras de disco. Esse problema indica porque a alocação encadeada não deve ser usada para uma aplicação que exija acesso direto.

Como resultado, alguns sistemas suportam arquivos de acesso direto usando alocação contígua e acesso sequencial por alocação encadeada. Para esses sistemas, o tipo de acesso a ser feito deve ser declarado quando o arquivo é criado. Um arquivo criado para acesso sequencial será encadeado e não poderá ser usado para acesso direto. Um arquivo criado para acesso direto será contíguo e poderá suportar acesso direto e sequencial, mas seu tamanho máximo deverá ser declarado quando ele for criado. Observe que, nesse caso, o sistema operacional deve ter estruturas de dados e algoritmos apropriados para suportar os *dois* métodos de alocação. Os arquivos podem ser convertidos de um tipo para outro pela criação de um novo arquivo do tipo desejado, no qual o conteúdo do arquivo antigo é copiado. O arquivo anterior pode então ser excluído, e o novo arquivo renomeado.

A alocação indexada é mais complexa. Se o bloco de índice já estiver na memória, o acesso poderá ser feito diretamente. No entanto, manter o bloco de índice na memória requer espaço considerável. Se não houver espaço disponível na memória, talvez seja necessário ler primeiro o bloco de índice e, em seguida, o bloco de dados desejado. Para um índice de dois níveis, duas leituras de bloco de índice podem ser necessárias. Para um arquivo extremamente grande, acessar um bloco próximo ao fim do arquivo exigiria ler todos os blocos de índice para seguir a cadeia de ponteiros antes que o bloco de dados finalmente pudesse ser lido. Assim, o desempenho da alocação indexada depende da estrutura do índice, do tamanho do arquivo e da posição do bloco desejado.

Alguns sistemas combinam a alocação contígua com a indexada usando a alocação contígua para arquivos pequenos (até três ou quatro blocos), e alternando automaticamente para a alocação indexada se o arquivo ficar grande. Como a maioria dos arquivos é pequena, e a alocação contígua é eficiente para arquivos pequenos, o desempenho médio pode ser bom.

Por exemplo, em 1991, a Sun Microsystems mudou sua versão do sistema operacional UNIX para melhorar o desempenho do algoritmo de alocação do sistema de arquivos. As medidas de desempenho indicaram que o throughput máximo de disco em uma estação de trabalho típica (SparcStation 1 com 12 MIPS) ocupava 50% da CPU e produzia uma largura de banda de disco de apenas 1,5 megabytes por segundo. Para melhorar o desempenho, a Sun implementou mudanças para alocar espaço em clusters de 56K de tamanho, sempre que possível. Essa alocação reduziu a fragmentação externa e os tempos de busca e latência. Além disso, as rotinas de leitura de disco foram otimizadas para fazer a leitura nesses grandes clusters. A estrutura de inode ficou inalterada. Essas mudanças, juntamente com o uso das técnicas de *read-ahead* e *free-bebind* (discutidas na Seção 11.9.2), resultaram em 25% menos uso de CPU para um throughput substancialmente maior.

Muitas outras otimizações estão em uso. Considerando a disparidade entre a velocidade da CPU e do disco, não é absurdo adicionar milhares de instruções extras ao sistema operacional apenas para economizar alguns movimentos da cabeça do disco. Além disso, essa disparidade está aumentando com o tempo, a um ponto em que centenas de milhares de instruções poderiam ser usadas razoavelmente para otimizar os movimentos da cabeça de leitura.

11.7 • Gerência de espaço livre

Como só existe uma quantidade limitada de espaço em disco, é necessário reutilizar o espaço de arquivos excluídos para novos arquivos, se possível. (Os discos óticos somente de leitura permitem apenas a escrita em determinado setor, sendo fisicamente impossível a sua reutilização.) Para controlar a quantidade de espaço livre em disco, o sistema mantém uma lista de espaço **livre.** Essa lista registra todos os blocos de disco que estão **livres,** ou seja, os que não estão alocados a algum arquivo ou diretório. Para criar um arquivo, pesquisamos a lista de espaço livre para encontrar a quantidade de espaço desejado, e alocamos esse espaço ao novo arquivo. Esse espaço é então removido da lista de espaço livre. Quando um arquivo é excluído, seu espaço em disco é adicionado à lista de espaço livre. A lista, apesar do nome, talvez não seja implementada como uma lista, conforme veremos em nossa discussão a seguir.

11.7.1 Vetor de bits

Frequentemente, a lista de espaço livre é implementada como um **mapa de bits** ou um **vetor de bits.** Cada bloco é representado por 1 bit. Se o bloco estiver livre, o bit será 1; se o bloco estiver alocado, o bit será 0.

Por exemplo, considere um disco no qual os blocos 2,3,4,5,8,9,10,11,12,13,17,18,25,26 e 27 estão livres, e o resto dos blocos está alocado. O mapa de bits do espaço livre seria:

001111001111110001100000011100000...

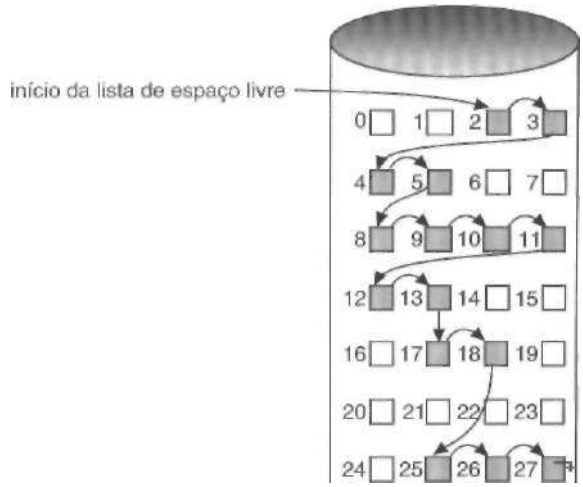
A principal vantagem dessa abordagem é que é relativamente simples e eficiente encontrar o primeiro bloco livre, ou *n* blocos livres consecutivos no disco. Na verdade, muitos computadores fornecem instruções de manipulação de bits que podem ser usadas com eficácia para esse propósito. Por exemplo, a família Intel a partir do 80386 e a família Motorola a partir do 68020 (processadores dos PCs e Macintosh, respectivamente) têm instruções que retornam o deslocamento do primeiro bit com o valor 1 em uma palavra. Na verdade, o Sistema Operacional do Apple Macintosh utiliza o método de vetor de bits para alocar espaço em disco. Para encontrar o primeiro bloco livre, o Sistema Operacional do Macintosh verifica cada palavra sequencialmente no mapa de bits para ver se esse valor é ou não zero, já que uma palavra de valor zero tem todos os bits 0 e representa um conjunto de blocos alocados. A primeira palavra não-zero é analisada para encontrar o primeiro bit 1, que é a posição do primeiro bloco livre. O cálculo do número do bloco é

(número de bits por palavra) x (número de palavras em zero) +
deslocamento do primeiro bit 1

Mais uma vez, vemos os recursos de hardware orientando a funcionalidade do software. Infelizmente, os vetores de bits são ineficientes, a menos que todo vetor seja mantido na memória principal (e seja gravado no disco ocasionalmente para fins de recuperação). Mantê-lo na memória principal é possível para discos menores, tais como em microcomputadores, mas não para os maiores. Um disco de 1.3 gigabytes com blocos de 512 bytes precisaria de um mapa de bits de mais de 332K para rastrear seus blocos livres. O clustering dos blocos em grupos de quatro reduz esse número para 83K por disco.

11.7.2 Lista encadeada

Outra abordagem é encadear todos os blocos de disco livres, mantendo um ponteiro ao primeiro bloco livre em uma posição especial no disco e armazenando-o em cache na memória. Esse primeiro bloco contém um ponteiro ao próximo bloco livre de disco, e assim por diante. Em nosso exemplo (Seção 11.7.1), manteríamos um ponteiro ao bloco 2, como o primeiro bloco livre. O bloco 2 conteria um ponteiro para o bloco 3, que apontaria para o bloco 4, que apontaria para o bloco 5, que apontaria para o bloco 8, e assim por diante (Figura 11.20). No entanto, esse esquema não é eficiente; para percorrer a lista, precisamos ler cada bloco, o que requer tempo substancial de I/O. Felizmente, percorrer a lista livre não é uma ação frequente. Geralmente, o sistema operacional simplesmente precisa de um bloco livre para que possa alocar esse bloco a um arquivo, assim o primeiro bloco na lista é usado. Observe que o método FAT incorpora a contabilização de blocos livres na estrutura de dados de alocação. Nenhum método separado é necessário.



S»[]29P30n3in

Figura 11.20 Lista encadeada de espaço livre no disco.

11.7.3 Agrupamento

Uma modificação da abordagem de lista livre é armazenar os endereços de *n* blocos livres no primeiro bloco livre. Os primeiros *n* - 1 desses blocos estão realmente livres. O bloco final contém os endereços de outros *n* blocos livres, e assim por diante. A importância dessa implementação é que os endereços de um grande número de blocos livres podem ser rapidamente encontrados, diferentemente da abordagem da lista encadeada padrão.

11.7.4 Contadores

Outra abordagem é aproveitar o fato de que, em geral, vários blocos contíguos podem ser alocados ou liberados simultaneamente, sobretudo quando o espaço é alocado com o algoritmo de alocação contígua ou através do clustering. Portanto, em vez de manter uma lista de *n* endereços de disco livres, podemos manter o endereço do primeiro bloco livre e o número *n* de blocos contíguos livres que seguem esse primeiro bloco. Cada entrada na lista de espaço livre consiste então em um endereço de disco e um contador. Embora cada entrada exija mais espaço do que um endereço de disco simples precisaria, a lista global seria mais curta, desde que o contador fosse geralmente maior do que 1.

11.8 • Implementação de diretórios

A Seleção dos algoritmos de alocação e gerência de diretórios tem um grande efeito na eficiência, desempenho e confiabilidade do sistema de arquivos. Portanto, é importante entender o que está envolvido nesses algoritmos.

11.8.1 Lista linear

O método mais simples de implementar um diretório é usar uma lista linear de nomes de arquivos com ponteiros aos blocos de dados. Uma lista linear de entradas de diretório requer uma pesquisa linear para encontrar uma determinada entrada. Esse método é simples de programar, mas demorado de executar. Para criar um novo arquivo, devemos primeiro pesquisar o diretório para ter certeza de que nenhum arquivo existente tem o mesmo nome. Em seguida, adicionamos uma nova entrada ao fim do diretório. Para excluir um arquivo, pesquisamos o diretório procurando o arquivo em questão, em seguida, liberamos o espaço alocado a ele. Para reutilizar a entrada de diretório, existem várias opções. Podemos marcar a entrada como não-usada (atribuindo a ela um nome especial, como um nome em branco ou um bit usado/não-usado em cada entrada),

ou podemos anexá-la a uma lista de entradas de diretório livres. Uma terceira alternativa é copiar a entrada final para a posição liberada no diretório e diminuir o tamanho do diretório. Além disso, podemos usar uma lista encadeada para diminuir o tempo de exclusão de um arquivo.

A verdadeira desvantagem de uma lista linear de entradas de diretório é a pesquisa linear para encontrar um arquivo. As informações de diretório são usadas frequentemente, e uma implementação lenta de acesso a elas seria percebida pelos usuários. Na verdade, muitos sistemas operacionais implementam um cache de software para armazenar as informações de diretório usadas mais recentemente. A busca no cache evita ter de ler constantemente as informações do disco. Uma lista ordenada permite a pesquisa binária **e** diminui o tempo médio de pesquisa. No entanto, a exigência de que a lista seja mantida ordenada pode complicar a criação e exclusão de arquivos, já que talvez seja necessário mover grandes quantidades de informações do diretório para mantê-lo ordenado. Uma estrutura de dados em árvore mais sofisticada, como uma árvore B *(B-tree),* pode ajudar aqui. Uma vantagem da lista ordenada é que uma listagem de diretório ordenada pode ser gerada sem uma etapa de ordenação separada.

11.8.2 Tabelas de hash

Outra estrutura de dados que tem sido utilizada para diretórios de arquivos é uma **tabela de hash.** Nesse método, uma lista linear armazena as entradas de diretório, mas uma estrutura de dados com hash também é usada. A **tabela de hash** pega um valor calculado a partir do nome do arquivo e retorna um ponteiro ao nome de arquivo na lista linear. Portanto, pode diminuir em muito o tempo de pesquisa no diretório. A inserção e a exclusão também são relativamente fáceis, embora seja preciso prever as colisões - situações em que dois nomes de arquivo são mapeados na mesma posição. As principais dificuldades com uma tabela de hash são o tamanho fixo da tabela e a dependência da função de hash do tamanho da tabela.

Por exemplo, vamos supor que tenhamos criado uma tabela de hash de busca linear com 64 entradas. A função de hash converte os nomes de arquivos em inteiros de 0 a 63, por exemplo, usando o resto da divisão por 64. Se mais tarde tentarmos criar o arquivo número *65,* devemos aumentar a **tabela de hash** do diretório, digamos, para 128 entradas. Como resultado, precisamos de uma nova função de hash, que deverá mapear os nomes de arquivo na faixa de 0 a 127, e devemos reorganizar as entradas de diretório existentes para refletir seus novos valores de hash. Como alternativa, uma tabela de hash com encadeamento pode ser usada. Cada entrada de hash pode ser uma lista encadeada em vez de um valor individual, e podemos resolver as colisões adicionando uma nova entrada à lista encadeada. As pesquisas podem ficar mais lentas, porque a pesquisa por um nome pode exigir percorrer uma lista encadeada de entradas de tabela em colisão, mas essa operação é provavelmente muito mais rápida do que uma pesquisa linear em todo o diretório.

11.9 • Eficiência e desempenho

Agora que já discutimos as opções de alocação de bloco e gerência de diretório, podemos considerar o seu efeito no desempenho e uso eficiente do disco. Os discos tendem a ser um importante gargalo no desempenho do sistema, já que são o componente principal mais lento do computador. Nesta seção, discutimos técnicas que melhoram a eficiência e o desempenho do armazenamento secundário.

11.9.1 Eficiência

O uso eficiente do espaço em disco depende muito dos algoritmos de alocação de disco e diretório em uso. Por exemplo, os inodes do UNIX são pré-alocados em uma partição. Mesmo um disco "vazio" tem uma percentagem do seu espaço perdida para os inodes. No entanto, ao pré-alocar os inodes e dispersá-los em toda a partição, nós melhoramos o desempenho do sistema de arquivos. Esse desempenho melhorado é resultado dos algoritmos de espaço livre e alocação do UNIX, que tentam manter os blocos de dados de um arquivo próximos ao bloco de inode do arquivo, para reduzir o tempo de busca.

Como outro exemplo, vamos reconsiderar o esquema de clustering discutido na Seção 11.6, que ajuda o desempenho de busca de arquivo e transferência de arquivos ao custo da fragmentação interna. Para reduzir essa fragmentação, o UNIX BSD varia o tamanho do cluster à medida que o arquivo cresce. Clusters grandes

são usados onde podem ser preenchidos, e clusters pequenos são usados para arquivos pequenos e o último cluster de um arquivo. Esse sistema será descrito no Capítulo 20.

Também exigem consideração os tipos de dados normalmente mantidos em uma entrada de diretório (ou inode) de arquivo. Geralmente, a "data de última escrita" é registrada para fornecer informações ao usuário e determinar se o arquivo precisa de backup. Alguns sistemas também mantêm um registro da "data de último acesso", para que o usuário possa determinar quando o arquivo foi lido pela última vez. O resultado de manter essas informações é que, sempre que um arquivo é lido, um campo na estrutura de diretório precisa ser gravado. Essa alteração requer que o bloco seja lido na memória, que uma seção seja alterada e que o bloco seja gravado de volta para o disco, porque as operações nos discos ocorrem apenas em partes do bloco (ou clusters). Portanto, sempre que um arquivo é aberto para leitura, sua entrada de diretório deve ser lida e gravada também. Essa exigência causa a operação ineficiente para arquivos acessados com frequência, por isso devemos pesar seus benefícios em relação ao custo de desempenho quando estivermos projetando um sistema de arquivos. Em geral, *todo* item de dados associado a um arquivo precisa ser considerado para avaliar seu efeito na eficiência e desempenho.

Como exemplo, considere como a eficiência é afetada pelo tamanho dos ponteiros usados para acessar dados. A maioria dos sistemas utiliza ponteiros de 16 ou 32 bits em todo o sistema operacional. Esses tamanhos de ponteiro limitam o tamanho de um arquivo para 2^{16} (64K) ou 2^{32} bytes (4 gigabytes). Alguns sistemas implementam ponteiros de 64 bits para aumentar esse limite para 2^{64} bytes, o que é um número muito grande. No entanto, os ponteiros de 64 bits ocupam mais espaço de armazenamento e, por sua vez, fazem com que os métodos de alocação e gerência de espaço livre (listas encadeadas, índices, e assim por diante) utilizem mais espaço em disco.

Uma das dificuldades de escolher um tamanho de ponteiro, ou na verdade qualquer tamanho de alocação fixo em um sistema operacional, é planejar os efeitos de mudanças na tecnologia. Considere que o IBM PC XT tinha um disco rígido de 10 megabytes, e um sistema de arquivos do MS-DOS que podia suportar apenas 32 megabytes. (Cada entrada na FAT tinha 12 bits, apontando para um cluster de 8K.) A medida que as capacidades de disco aumentaram, discos maiores tiveram de ser divididos em partições de 32 megabytes, porque o sistema de arquivos não poderia controlar blocos além de 32 megabytes. Assim que os discos rígidos de mais de 100 megabytes de capacidade passaram a ser comuns, as estruturas de dados de disco e os algoritmos no MS-DOS tiveram de ser modificados para permitir sistemas de arquivos maiores. (Cada entrada na FAT foi expandida para 16 bits, e mais tarde para 32 bits.) As decisões iniciais relativas aos sistemas de arquivos foram tomadas para fins de eficiência; no entanto, com o advento do MS-DOS Versão 4, milhões de usuários de computador ficaram incomodados quando tiveram de mudar para o novo e maior sistema de arquivos.

Como outro exemplo, considere a evolução do sistema operacional Solaris da Sun. Originalmente, muitas estruturas de dados tinham tamanho fixo, alocado na inicialização do sistema. Essas estruturas incluíam a tabela de processos e a tabela de arquivos abertos. Quando a tabela de processos ficava cheia, mais nenhum processo poderia ser criado. Quando a tabela de arquivos ficava cheia, mais nenhum arquivo podia ser aberto. O sistema não conseguia fornecer serviço aos usuários. Esses tamanhos de tabela poderiam ser aumentados somente pela recompilação do kernel e nova inicialização do sistema. Desde o lançamento do Solaris 2, quase todas as estruturas do kernel são alocadas dinamicamente, eliminando esses limites artificiais no desempenho do sistema. É claro que os algoritmos que manipulam essas tabelas são mais complicados, e o sistema operacional é um pouco mais lento porque ele deve alocar e desalocar dinamicamente as entradas nas tabelas, mas esse preço é comum para obter mais generalidade funcional.

11.9.2 Desempenho

Assim que os métodos de disco básicos forem selecionados, ainda existem várias formas de melhorar o desempenho. Como observado no Capítulo 2, a maioria das controladoras de disco incluem memória local para formar um cache embutido suficientemente grande para armazenar trilhas inteiras de uma vez. Assim que uma busca é executada, a trilha é lida no cache de disco começando no setor sob a cabeça de disco (aliviando o tempo de latência). A controladora de disco transfere então qualquer pedido de setor para o sistema operacional. Assim que os blocos passarem da controladora de disco para a memória principal, o sistema operacional poderá armazená-los em cache. Alguns sistemas mantêm uma seção separada de memória principal

para um **cache de disco,** no qual os blocos são mantidos considerando que serão usados novamente em breve. O LRU é um algoritmo de uso geral razoável para a substituição de bloco. Outros sistemas (como a versão do UNIX da Sun) tratam toda memória física não-utilizada como um pool de buffers que é compartilhado pelo sistema de paginação e o sistema de cache de blocos de disco. Um sistema que realiza muitas operações de I/O usará boa parte da sua memória como um cache de blocos, enquanto um sistema que executa muitos programas usará mais memória como espaço de paginação.

Alguns sistemas otimizam seu cache de disco usando diferentes algoritmos de substituição, dependendo do tipo de acesso do arquivo. Um arquivo sendo lido ou gravado sequencialmente não deve ter seus blocos substituídos na ordem LRU, porque o bloco usado mais recentemente será usado por último, ou talvez nunca mais. Em vez disso, o acesso sequencial pode ser otimizado por técnicas conhecidas como *free-behind* e *read-ahead.* A técnica ***free** bebind* remove um bloco do buffer assim que o próximo bloco é solicitado. Os blocos anteriores provavelmente não serão usados novamente e desperdiçam espaço de buffer. Com a técnica *read ahead,* um bloco solicitado e vários blocos subsequentes são lidos e armazenados em cache. É provável que esses blocos sejam solicitados depois que o bloco atual for processado. Recuperar esses blocos do disco em uma transferência e armazená-los em cache economiza muito tempo. Um cache de trilha na controladora não elimina a necessidade da técnica *read-ahead* em um sistema multiprogramado, por causa da alta latência e custo de muitas transferências pequenas do cache de trilha para a memória principal.

Outro método de usar a memória principal para melhorar o desempenho é comum em computadores pessoais. Uma seção de memória é reservada e tratada como um **disco virtual,** ou **disco de RAM.** Nesse caso, um driver de dispositivo de disco de RAM aceita todas as operações de disco padrão, mas realiza essas operações naquela seção da memória, em vez de no disco. Todas as operações de disco podem ser executadas nesse disco de RAM e, exceto pela altíssima velocidade, os usuários não perceberão a diferença. Infelizmente, os discos de RAM são úteis apenas para armazenamento temporário, já que uma falha de energia ou reinicialização do sistema geralmente os apagarão. Em geral, arquivos temporários, como arquivos intermediários de compilação, são armazenados nesses discos.

A diferença entre um disco de RAM e um cache de disco é que o conteúdo do disco de RAM é totalmente controlado pelo usuário, enquanto o conteúdo do cache de disco está sob controle do sistema operacional. Por exemplo, um disco de RAM ficará vazio até que o usuário (ou programas sob a direção do usuário) crie arquivos ali. A Figura 11.21 mostra os possíveis locais de cache em um sistema.

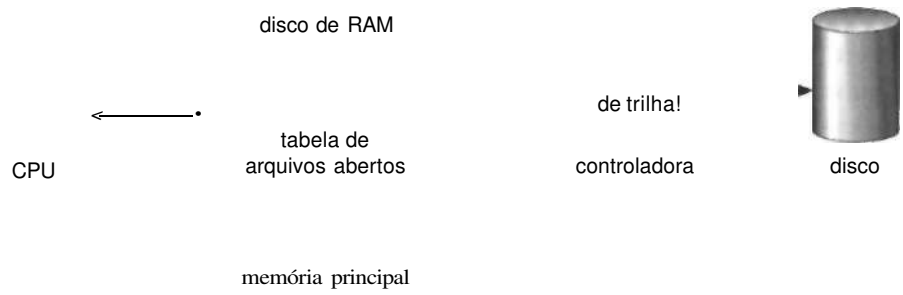


Figura 11.21 Várias posições de cache de disco.

11.10 • Recuperação

Como os arquivos e diretórios são mantidos na memória principal e no disco, devemos cuidar para garantir que uma falha no sistema não resulte em perda ou inconsistência de dados.

11.10.1 Verificação de consistência

Como discutido na Seção 11.8, parte das informações do diretório são mantidas na memória principal (no cache) para acelerar o acesso. As informações do diretório na memória principal são geralmente mais atualizadas do que as informações correspondentes no disco, porque a escrita de informações de diretório em cache para o disco não ocorre necessariamente assim que ocorre a atualização.

Considere o possível efeito de uma falha do computador. A tabela de arquivos abertos geralmente se perde e, com ela, quaisquer alterações nos diretórios dos arquivos abertos. Esse evento pode deixar o sistema de arquivos em estado inconsistente. O estado real de alguns arquivos não é o que aparece na estrutura de diretório. Com frequência, um programa especial é executado no momento da reinicialização para verificar e corrigir inconsistências de disco.

A verificação de consistência compara os dados na estrutura de diretório com os blocos de dados no disco e tenta corrigir quaisquer inconsistências encontradas. Os algoritmos de alocaçâo e gerência de espaço livre determinam os tipos de problemas que o verificador poderá encontrar e qual será a taxa de sucesso na correção desses problemas. Por exemplo, se a alocaçâo encadeada for usada e houver uma ligação de cada bloco para seu próximo bloco, então o arquivo inteiro pode ser reconstruído a partir dos blocos de dados e a estrutura de diretório pode ser recriada. A perda de uma entrada de diretório no sistema de alocaçâo indexada pode ser desastrosa, porque os blocos de dados não têm conhecimento uns dos outros. Por esse motivo, o UNIX armazena em cache as entradas de diretório para as operações de leitura, mas qualquer escrita de dados que resulte em alocaçâo de espaço geralmente faz com que o bloco de inode seja gravado para disco antes que os blocos de dados correspondentes o sejam.

11.10,2 Backup e restauração

Como os discos magnéticos às vezes falham, devemos ter certeza que não haja perda de dados. Para isso, podemos usar os programas do sistema para fazer **backup** dos dados do disco para outro dispositivo de armazenamento, como um disquete, fita magnética ou disco ótico. A recuperação da perda de um arquivo individual, ou de um disco inteiro, pode então ser uma questão de **restaurar** os dados do backup.

Para minimizar a quantidade de cópia necessária, podemos usar informações da entrada de diretório de cada arquivo. Por exemplo, se o programa de backup souber quando o último backup de um arquivo foi feito, e a data da última escrita no diretório indicar que o arquivo não sofreu alterações desde então, o arquivo não precisará ser copiado novamente. Um escalonamento de backup típico pode ser:

- *Dia 1*: Copiar para uma mídia de backup todos os arquivos do disco - chamado **backup completo.**
- *Dia 2*: Copiar para outra mídia todos os arquivos alterados desde o dia **1** - um **backup incremental.**
- *Dia 3:* Copiar para outra mídia todos os arquivos alterados desde o dia 2.

- *Dia* N: Copiar para outra mídia todos os arquivos alterados desde o dia N - 1 . Em seguida, voltar para o dia 1.

O novo ciclo pode ter o seu backup gravado sobre o conjunto anterior ou em um novo conjunto de mídias de backup. Desse modo, podemos restaurar um disco inteiro começando com restaurações a partir do backup completo, e continuando através de cada backup incremental. Evidentemente, quanto maior N, mais fitas ou discos precisam ser lidos para uma restauração completa. Uma vantagem adicional desse ciclo de backup é que é possível restaurar qualquer arquivo excluído por acidente durante o ciclo recuperando o arquivo excluído do backup do dia anterior. O tamanho do ciclo é determinado pelo compromisso entre a quantidade de mídia física de backup necessária e o número de dias para trás a partir do qual a restauração pode ser feita.

Pode acontecer que determinado usuário observe que um arquivo está faltando ou corrompido muito depois que ocorreu o problema. Para proteger contra essa situação, é comum fazer um backup completo de tempos em tempos que será mantido "para sempre", em vez da mídia de backup ser reutilizada. Também é uma boa ideia armazenar esses backups permanentes em um local longe dos backups regulares, para proteger contra perigos como incêndios que destroem o computador e todos os seus backups também.

Se o ciclo de backup reutilizar mídias físicas, devemos ter cuidado para não fazer isso muitas vezes - se houver desgaste das mídias, talvez não seja possível restaurar dados.

11.11 • Resumo

Um arquivo é um tipo abstrato de dados definido e implementado pelo sistema operacional. É uma sequência de registros lógicos. Um registro lógico pode ser um byte, uma linha (de tamanho fixo ou variável) ou um item de dados mais complexo. O sistema operacional pode suportar especificamente vários tipos de registro ou pode deixar o suporte ao programa aplicativo.

A principal tarefa do sistema operacional é mapear o conceito de arquivo lógico em dispositivos de armazenamento físico tais como fita ou disco magnético. Como o tamanho do registro físico do dispositivo talvez não seja igual ao tamanho do registro lógico, pode ser necessário encaixar registros lógicos em registros físicos. Mais uma vez, essa tarefa pode ser suportada pelo sistema operacional ou deixada para o programa aplicativo.

Os sistemas de arquivos baseados em fita são limitados; a maioria dos sistemas de arquivos são baseados em disco. As fitas são comumente usadas para transporte de dados entre máquinas, ou para armazenamento de backup ou arquivamento.

Cada dispositivo em um sistema de arquivos mantêm um índice de volume ou diretório de dispositivo listando a posição dos arquivos no dispositivo. Além disso, é útil criar diretórios para permitir a organização dos arquivos. Um diretório de nível único em um sistema multiusuário causa problemas de nomeação, já que cada arquivo deve ter um nome exclusivo. Um diretório de dois níveis resolve esse problema criando um diretório separado para cada usuário. Cada usuário tem seu próprio diretório, que contém seus próprios arquivos.

O diretório lista os arquivos por nome, e inclui informações como a posição do arquivo no disco, seu tamanho, tipo, proprietário, hora da criação, hora da última utilização etc.

A generalização natural de um diretório de dois níveis é um diretório estruturado em árvore. Um diretório em árvore permite que o usuário crie subdiretórios para organizar seus arquivos. Estruturas de diretórios de grafos acíclicos permitem o compartilhamento de arquivos e diretórios, mas complicam a pesquisa e exclusão. Uma estrutura de grafo genérico permite flexibilidade total no compartilhamento de arquivos e diretórios, mas às vezes requer o uso da coleta de lixo para recuperar espaço em disco não utilizado.

Como os arquivos são o principal mecanismo de armazenamento de informações na maioria dos sistemas de computação, a proteção de arquivo é necessária. O acesso aos arquivos pode ser controlado de forma separada para cada tipo de acesso: ler, gravar, executar, anexar, listar diretório e assim por diante. A proteção de arquivo pode ser fornecida por senhas, listas de acesso, ou por técnicas especiais *ad hoc.*

O sistema de arquivos reside permanentemente no *armazenamento secundário,* que tem como exigência principal o fato de poder armazenar grandes quantidades de dados de forma permanente. O meio de armazenamento secundário mais comum é o disco.

Os sistemas de arquivos são muitas vezes implementados em uma estrutura em camadas ou modular. Os níveis inferiores tratam das propriedades físicas dos dispositivos de armazenamento. Os níveis superiores lidam com nomes de arquivo simbólicos e as propriedades lógicas dos arquivos. Os níveis intermediários mapeiam os conceitos de arquivo lógico em propriedades de dispositivos físicos.

Os vários arquivos podem ser alocados no disco de três formas: através de alocação contígua, encadeada ou indexada. A alocação contígua pode sofrer de fragmentação externa. O acesso direto é muito ineficiente com a alocação encadeada. A alocação indexada pode exigir custo substancial para seu bloco de índice. Existem muitas formas nas quais esses algoritmos podem ser otimizados. O espaço contíguo pode ser ampliado por meio de extensões para aumentar a flexibilidade e diminuir a fragmentação externa. A alocação indexada pode ser feita em clusters de múltiplos blocos para aumentar o throughput e reduzir o número de entradas de índice necessárias. A indexação em clusters grandes é semelhante à alocação contígua com extensões.

Os métodos de alocação de espaço livre também influenciam a eficiência de uso do espaço em disco, o desempenho do sistema de arquivos, e a confiabilidade do armazenamento secundário. Os métodos usados incluem vetores de bits e listas encadeadas. As otimizações incluem agrupamento, contadores e a FAT, que coloca a lista encadeada em uma área contígua.

As rotinas de gerência de diretórios devem considerar os aspectos de eficiência, desempenho e confiabilidade. Uma tabela de hash é o método mais frequentemente usado; é rápido e eficiente. Infelizmente, danos à tabela ou uma falha no sistema podem fazer com que as informações do diretório não correspondam ao conteúdo do disco. Um *verificador de consistência* - um programa de sistema como f sck no UNIX, ou chkdsk no MS-DOS - pode ser usado para reparar o dano.

• Exercícios

11.1 Considere um sistema de arquivos no qual um arquivo possa ser excluído e seu espaço de disco reutilizado enquanto ainda houver links para aquele arquivo. Que problemas poderão ocorrer se um novo arquivo for criado na mesma área de armazenamento ou com o mesmo nome de caminho absoluto? Como esses problemas podem ser evitados?

11.2 Alguns sistemas excluem automaticamente todos os arquivos de usuário quando um usuário efetua logoff ou um job termina, a menos que o usuário solicite explicitamente que os arquivos sejam mantidos; outros sistemas mantêm todos os arquivos a menos que o usuário explicitamente os exclua. Discuta os méritos relativos de cada abordagem.

11.3 Por que alguns sistemas controlam o tipo de um arquivo, enquanto outros deixam isso nas mãos do usuário ou simplesmente não implementam múltiplos tipos de arquivo? Que sistema é "melhor"? Justifique a sua resposta.

11.4 Da mesma forma, alguns sistemas suportam muitos tipos de estruturas para os dados de um arquivo, enquanto outros simplesmente suportam um fluxo de bytes. Quais são as vantagens e desvantagens de cada um?

11.5 Quais as vantagens e desvantagens de registrar o nome do programa criador com os atributos do arquivo (como é feito no sistema operacional do Macintosh)?

11.6 Você poderia simular uma estrutura de diretório de multinível com uma estrutura de diretório de nível único na qual nomes arbitrariamente longos podem ser usados? Se a sua resposta for afirmativa, explique como você faria isso, e compare esse esquema com um esquema de diretório de multinível. Se a sua resposta for negativa, explique o que impede o sucesso dessa simulação. A sua resposta mudaria se os nomes de arquivo fossem limitados a sete caracteres? Explique sua resposta.

11.7 Explique o objetivo das operações de open e close.

11.8 Alguns sistemas abrem automaticamente um arquivo quando ele é referenciado pela primeira vez, e o fecham quando o job termina. Discuta as vantagens e desvantagens desse esquema. Compare-o com o mais tradicional, no qual o usuário tem de abrir e fechar o arquivo explicitamente.

11.9 Dê um exemplo de uma aplicação na qual os dados em um arquivo devem ser acessados na seguinte ordem:

a. Sequencialmente
b. Aleatoriamente

11.10 Alguns sistemas fornecem compartilhamento mantendo uma única cópia de um arquivo; outros sistemas mantêm várias cópias, uma para cada um dos usuários compartilhando o arquivo. Discuta os méritos relativos de cada abordagem.

11.11 Em alguns sistemas, um subdiretório pode ser lido e gravado por um usuário autorizado, da mesma forma como os arquivos comuns.

a. Descreva dois problemas de proteção que poderiam surgir.
b. Sugira um esquema para tratar cada um dos problemas de proteção indicados no item a.

11.12 Considere um sistema que suporte 5.000 usuários. Vamos supor que você deseja permitir que 4.990 desses usuários sejam capazes de acessar um arquivo.

a. Como você especificaria esse esquema de proteção no UNIX?
b. Sugira um esquema de proteção que seja mais eficiente do que o esquema fornecido pelo **UNIX.**

11.13 Pesquisadores sugeriram que, em vez de ter uma lista de acesso associada a cada arquivo (especificando que usuários podem acessar o arquivo e como), devemos ter uma *lista de controle de usuário* associada a cada usuário (especificando que arquivos um usuário pode acessar e como). Discuta os méritos relativos desses dois esquemas.

11.14 Considere um arquivo que no momento consiste em 100 blocos. Considere que o bloco de controle do arquivo (e o bloco de índice, no caso de alocação indexada) já está na memória. Calcule quantas operações de I/O de disco são necessárias para as estratégias de alocação contígua, encadeada e inde-

xada (de nível único) se, para um bloco, as seguintes condições forem verdadeiras. No caso da alocação contígua, suponha que não há espaço para crescer no início, mas sim no final do arquivo. Considere que as informações do bloco a ser adicionado estão armazenadas na memória.

a. O bloco é adicionado no início.
b. O bloco é adicionado no meio.
c. O bloco é adicionado no fim.
d. O bloco é removido do início.
e. O bloco é removido do meio.
f. O bloco é removido do fim.

11.15 Considere um sistema no qual o espaço livre é mantido em uma lista de espaço livre.

a. Suponha que o ponteiro para a lista de espaço livre tenha sido perdido. O sistema pode reconstruir a lista de espaço livre? Explique sua resposta.
b. Sugira um esquema para garantir que o ponteiro nunca seja perdido devido a uma falha de memória.

11.16 Que problemas poderiam ocorrer se um sistema permitisse que um sistema de arquivos fosse montado simultaneamente em mais de um local?

11.17 Por que o mapa de bits para a alocação de arquivos deve ser mantido no armazenamento de massa, em vez de na memória principal?

11.18 Considere um sistema que suporta as estratégias de alocação contígua, encadeada e indexada. Que critérios devem ser utilizados para decidir qual a melhor estratégia a ser utilizada para um arquivo em particular?

11.19 Considere um sistema de arquivos em um disco que tem tamanhos de bloco físico e lógico de 512 bytes. Suponha que a informação sobre cada arquivo já está na memória. Para cada uma das três estratégias de alocação (contígua, encadeada e indexada), responda as seguintes questões:

a. Como o mapeamento de endereço lógico para físico é obtido nesse sistema? (Para a alocação indexada, suponha que um arquivo é sempre menor que 512 blocos.)
b. Se você estiver no bloco lógico 10 (o último bloco acessado foi o bloco 10) e quiser acessar o bloco lógico 4, quantos blocos físicos devem ser lidos do disco?

11.20 Um problema com a alocação contígua é que o usuário deve pré-alocar espaço suficiente para cada arquivo. Se o arquivo ficar maior do que o espaço alocado para ele, ações especiais devem ser tomadas. Uma solução para esse problema é definir uma estrutura de arquivos consistindo em uma área contígua inicial (de um tamanho especificado). Se essa área for preenchida, o sistema operacional definirá automaticamente uma área de overflow que é ligada à área contígua inicial. Se a área de overflow ficar cheia, outra área será alocada. Compare essa implementação de um arquivo com as implementações contíguas e encadeadas padrão.

11.21 A fragmentação em um dispositivo de armazenamento pode ser eliminada pela recompactação das informações. Dispositivos de disco típicos não têm registradores de base ou de relocação (tais como aqueles utilizados quando a memória é compactada), então como é possível relocar arquivos? Apresente três motivos pelos quais a recompactação e a relocação de arquivos são em geral evitadas.

11.22 Como os caches ajudam a melhorar o desempenho? Por que os sistemas não usam mais caches ou caches maiores?

11.23 Em que situações usar uma memória como disco de RAM seria mais útil do que usá-la como um cache de disco?

11.24 Por que é vantajoso para o usuário que o sistema operacional aloque suas tabelas internas dinamicamente? Quais são as penalidades para o sistema operacional por fazê-lo?

11.25 Considere os seguinte esquema de backup:

- Dia 1: Copiar para uma mídia de backup todos os arquivos do disco.
- Dia 2: Copiar para outra mídia todos os arquivos alterados desde o dia 1.
- Dia 3: Copiar para outra mídia todos os arquivos alterados desde o dia 1.

Esse programa difere daquele na Seção 11.10.2, pois todos os backups subsequentes copiam todos os arquivos modificados desde o primeiro backup completo. Quais os benefícios desse sistema em relação ao apresentado na Seção 11.10.2? Quais as desvantagens? As operações de restauração ficam mais fáceis ou mais difíceis? Explique sua resposta.

Notas bibliográficas

Discussões gerais relativas aos sistemas de arquivos foram apresentadas por Grosshans [1986]. Os sistemas de bancos de dados e suas estruturas de arquivos foram descritos em detalhes em Silberschatz e colegas [1997].

Uma estrutura de diretório multinível foi implementada pela primeira vez no sistema MULTICS [Organick 1972]. A maioria dos sistemas operacionais agora implementam estruturas de diretório multinível, incluindo o UNIX, o sistema operacional do Apple Macintosh [Apple 1991] e o MS-DOS.

O Network File System (NFS) foi projetado pela Sun Microsystems e permite que as estruturas de diretórios fiquem dispersas em sistemas de computador em rede. Discussões relativas ao NFS foram apresentadas em Sandberg e colegas [1985], Sandberg [1987] e na publicação da Sun Microsystems [1990].

A publicação da Apple [1991] descreve o esquema de gerência de espaço em disco do Apple Macintosh. O sistema FAT do MS-DOS foi explicando em Norton e Wilton [1988]; a descrição do OS/2 está em Iacobucci [1988]. Os métodos de alocação da IBM foram descritos por Deitei [1990]. Os aspectos internos do sistema UNIX BSD foram discutidos em detalhes por Leffler e colegas [1989]. McVoy e Kleiman [1991] apresentaram otimizações desses métodos feitas no SunOS.