Capítulo 1
Introdução

Um sistema computacional moderno consiste em um ou mais processadores, memória principal, discos, impressoras, teclado, mouse, monitor, interfaces de rede e outros dispositivos de entrada e saída. Enfim, é um sistema complexo. Se cada programador de aplicações tivesse de entender como tudo isso funciona em detalhes, nenhum código chegaria a ser escrito. Além disso, gerenciar todos esses componentes e usá-los de maneira otimizada é um trabalho extremamente difícil. Por isso, os computadores têm um dispositivo de software denominado **sistema operacional**, cujo trabalho é fornecer aos programas do usuário um modelo de computador melhor, mais simples e mais limpo e lidar com o gerenciamento de todos os recursos mencionados. Esses sistemas são o tema deste livro.

A maioria dos leitores deve ter alguma experiência com um sistema operacional como Windows, Linux, FreeBSD ou Mac OS X, mas as aparências podem enganar. O programa com o qual os usuários interagem, normalmente chamado de **shell** (ou interpretador de comandos) quando é baseado em texto e de **GUI** (*graphical user interface* — interface gráfica com o usuário) quando usa ícones, na realidade não é parte do sistema operacional embora o utilize para realizar seu trabalho.

Um panorama simples dos principais componentes em discussão aqui é oferecido na Figura 1.1. Na parte inferior vemos o hardware. Ele consiste em chips, placas, discos, teclado, monitor e objetos físicos semelhantes. Na parte superior do hardware está o software. A maioria dos computadores tem dois níveis de operação: modo núcleo e modo usuário. O sistema operacional é a peça mais básica de software e opera em **modo núcleo** (também chamado **modo supervisor**). Nesse modo ele tem acesso completo a todo o hardware e pode executar qualquer instrução que a máquina seja capaz de executar. O resto do software opera em **modo usuário**, no qual apenas um subconjunto de instruções da máquina está disponível. Em particular, aquelas instruções que afetam o controle da máquina ou realizam E/S (Entrada/Saída) são proibidas para programas de modo usuário. Retornaremos à diferença entre modo núcleo e modo usuário repetidamente ao longo deste livro.

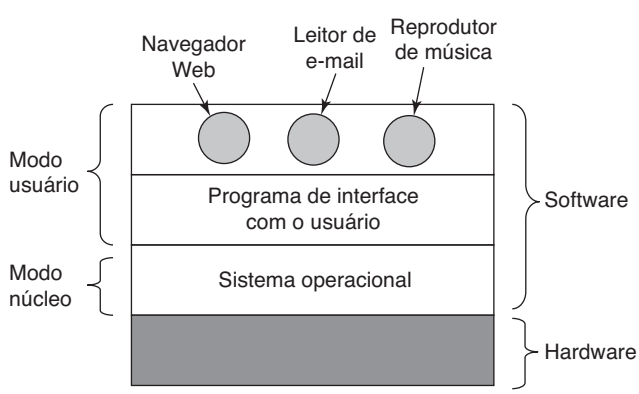

Figura 1.1 Onde o sistema operacional se encaixa.

O programa de interface com o usuário, shell ou GUI, é o nível mais inferior do software de modo usuário e permite que este inicie outros programas, como o navegador Web, leitor de e-mail ou reprodutor de música. Esses programas também usam muito o sistema operacional.

A área de atuação do sistema operacional é mostrada na Figura 1.1. Ele opera diretamente no hardware e fornece a base para todos os outros softwares.

Uma distinção importante entre o sistema operacional e o software normal (modo usuário) é que, se o usuário não gostar de um determinado leitor de e-mail, ele será livre para obter outro ou escrever seu próprio leitor de e-mail, se o quiser; mas não lhe é permitido escrever seu próprio manipulador de interrupção do relógio, que é parte do sistema operacional e normalmente está protegida pelo hardware contra tentativas de alterações pelo usuário.

Essa distinção, contudo, é às vezes confusa em sistemas embarcados (que podem não ter um modo núcleo) ou sistemas interpretados (como sistemas operacionais baseados em Java, que usam interpretação, e não hardware, para separar os componentes).

Em muitos sistemas, há programas executados em modo usuário, mas que auxiliam o sistema operacional ou realizam funções privilegiadas. Por exemplo, muitas vezes existe um programa que permite aos usuários mudarem suas senhas. Esse programa não faz parte do sistema operacional e não é executado em modo núcleo, mas realiza uma função claramente delicada e precisa ser protegido de maneira especial. Em alguns sistemas, essa ideia é levada ao extremo, e parte do que é tradicionalmente tido como sistema operacional (como o *filesystem* — sistema de arquivos) é executada em espaço do usuário. Nesses sistemas, é difícil definir um limite claro. Tudo o que é executado em modo núcleo sem dúvida constitui parte do sistema operacional, mas alguns programas executados fora dele também são

inquestionavelmente parte dele, ou pelo menos estão intimamente associados a ele.

Os sistemas operacionais diferem de programas de usuário (isto é, de aplicações) em outros aspectos além do lugar onde estão localizados. Em particular, eles são grandes, complexos e têm vida longa. O código-fonte de um sistema operacional como o Linux ou o Windows tem cerca de cinco milhões de linhas de código. Para se ter ideia do que isso significa, imagine imprimir cinco milhões de linhas em forma de livro, com 50 linhas por página e mil páginas por volume (maior que este livro). Seriam necessários cem volumes para enumerar um sistema operacional desse porte — basicamente uma caixa de livros inteira. Você consegue se imaginar começando em um emprego de manutenção de um sistema operacional e, no primeiro dia, ver seu chefe trazendo uma caixa de livros com o código e dizendo: "Aprenda-o"? E isso vale apenas para a parte que opera no núcleo. Programas de usuário como o GUI, bibliotecas e softwares de aplicação básica (como o Windows Explorer) podem atingir facilmente 10 ou 20 vezes esse valor.

Agora deve estar claro por que os sistemas operacionais têm vida longa — eles são muito difíceis de escrever e, uma vez tendo escrito um, o proprietário não se dispõe a descartá-lo e começar de novo. Em vez disso, eles evoluem por longos períodos de tempo. O Windows 95/98/Me era basicamente um sistema operacional e o Windows NT/2000/XP/Vista é um sistema diferente. Para os usuários, eles parecem semelhantes porque a Microsoft se assegurou de que a interface com o usuário do Windows 2000/XP fosse bastante parecida com o sistema que estava substituindo, principalmente o Windows 98. Entretanto, houve bons motivos para que a Microsoft eliminasse o Windows 98. Trataremos disso quando estudarmos o Windows em detalhes no Capítulo 11.

O outro exemplo principal que usaremos ao longo deste livro (além do Windows) é o UNIX e seus variantes e clones. Ele também evoluiu com o passar dos anos, com versões como System V, Solaris e FreeBSD derivadas do sistema original, ao passo que o Linux é uma base de código nova, embora tenha como modelo muito próximo o UNIX e seja altamente compatível com ele. Usaremos exemplos do UNIX ao longo deste livro e examinaremos o Linux em detalhes no Capítulo 10.

Neste capítulo, mencionaremos rapidamente vários aspectos importantes dos sistemas operacionais, inclusive o que são, sua história, os tipos existentes, alguns dos conceitos básicos e sua estrutura. Retornaremos a muitos desses importantes tópicos em mais detalhes em capítulos posteriores.

1.1 O que é um sistema operacional?

É difícil definir o que é um sistema operacional além de dizer que é o software que executa em modo núcleo — e mesmo isso nem sempre é verdade. Parte do problema ocorre porque os sistemas operacionais realizam basicamente duas funções não relacionadas: fornecer aos programadores de aplicativos (e aos programas aplicativos, naturalmente) um conjunto de recursos abstratos claros em vez de recursos confusos de hardware e gerenciar esses recursos de hardware. Dependendo do tipo de usuário, ele vai lidar mais com uma função ou com outra. Vejamos cada uma delas.

1.1.1 O sistema operacional como uma máquina estendida

A **arquitetura** (conjunto de instruções, organização de memória, E/S e estrutura de barramento) da maioria dos computadores em nível de linguagem de máquina é primitiva e de difícil programação, especialmente a entrada/saída. Para tornar isso mais concreto, examinemos rapidamente como é feita a E/S da unidade de discos flexíveis (disquetes) a partir de um chip controlador compatível com o NEC–PD765. Esse chip é usado na maioria dos computadores pessoais baseados em processadores Intel. Usamos discos flexíveis como exemplo porque, embora sejam obsoletos, são muito mais simples que os discos rígidos modernos. O PD765 tem 16 comandos, especificados pela carga de 1 a 9 bytes no registrador do dispositivo. Esses comandos são para leitura e escrita de dados, movimentação do braço do disco e formatação de trilhas. Servem também para inicialização, sinalização, reinicialização e recalibração do controlador e das unidades de disquetes.

Os comandos mais básicos são read e write; cada um deles requer 13 parâmetros agrupados em 9 bytes. Esses parâmetros especificam itens como o endereço do bloco de dados a ser lido, o número de setores por trilha, o modo de gravação usado no meio físico, o espaço livre entre setores e o que fazer com um marcador-de-endereço-de-dados-removidos. Se essas palavras não fizeram sentido para você, não se preocupe: é assim mesmo, um tanto místico. Quando a operação é completada, o chip controlador retorna 23 campos de status e de erros agrupados em 7 bytes. Como se isso não bastasse, o programador da unidade de discos flexíveis ainda deve saber se o motor está ligado ou não. Se estiver desligado, ele deverá ser ligado (com um longo atraso de inicialização) antes que os dados possam ser lidos ou escritos. O motor não pode permanecer ligado por muito tempo, senão o disco flexível poderá sofrer desgaste. O programador é, então, forçado a equilibrar dois fatores: longos atrasos de inicialização *versus* desgastes do disco flexível (e a perda dos dados nele gravados).

Sem entrar em detalhes *de fato*, é claro que um programador de nível médio provavelmente não se envolverá profundamente com os detalhes de programação das unidades de discos flexíveis (ou discos rígidos, que são piores). Em vez disso, o programador busca lidar com essas uni-

dades de um modo mais abstrato e simples. No caso dos discos, uma abstração típica seria aquela compreendida por um disco que contém uma coleção de arquivos com nomes. Cada arquivo pode ser aberto para leitura ou escrita e, então, ser lido ou escrito e, por fim, fechado. Detalhes como se a gravação deveria usar uma modulação por frequência modificada e qual seria o estado atual do motor não apareceriam na abstração apresentada ao programador da aplicação.

Abstração é o elemento-chave para gerenciar complexidade. Boas abstrações transformam uma tarefa quase impossível em duas manejáveis. A primeira delas é definir e implementar as abstrações. A segunda é usar essas abstrações para resolver o problema à mão. Uma abstração que quase todo usuário de computação entende é o arquivo. Ele é um fragmento de informação útil, como uma foto digital, uma mensagem de e-mail salva ou uma página da Web. Lidar com fotos, e-mails e páginas da Web é mais fácil do que manipular detalhes de discos, como o disco flexível descrito anteriormente. A tarefa do sistema operacional é criar boas abstrações e, em seguida, implementar e gerenciar os objetos abstratos criados. Neste livro, falaremos muito de abstrações. Elas são um dos elementos cruciais para compreender os sistemas operacionais.

Esse ponto é tão importante que convém repeti-lo em outras palavras. Com todo o respeito devido aos engenheiros industriais que projetaram o Macintosh, o hardware é feio. Processadores reais, memórias, discos e outros dispositivos são muito complicados e apresentam interfaces difíceis, desajeitadas, idiossincráticas e incoerentes para que as pessoas que precisam escrever softwares as utilizem. Algumas vezes isso se deve à necessidade de compatibilidade com a versão anterior do hardware, algumas vezes ao desejo de economizar dinheiro, mas algumas vezes os projetistas de hardware não percebem os problemas que estão causando ao software (ou não se importam com tais problemas). Pelo contrário, uma das principais tarefas do sistema operacional é ocultar o hardware e oferecer aos programas (e seus programadores) abstrações precisas, claras, elegantes e coerentes com as quais trabalhar. Os sistemas operacionais transformam o feio em bonito, como mostrado na Figura 1.2.

Deve-se observar que os clientes reais do sistema operacional são os programas aplicativos (via programadores de aplicativos, naturalmente). São eles que lidam diretamente com o sistema operacional e suas abstrações. Por outro lado, os usuários finais lidam com abstrações fornecidas pela interface do usuário, seja a linha de comandos shell ou uma interface gráfica. Embora as abstrações da interface com o usuário possam ser semelhantes às fornecidas pelo sistema operacional, nem sempre é o caso. Para esclarecer esse ponto, considere a área de trabalho normal do Windows e o prompt de comando, orientado a linhas de comando. Ambos são programas executados

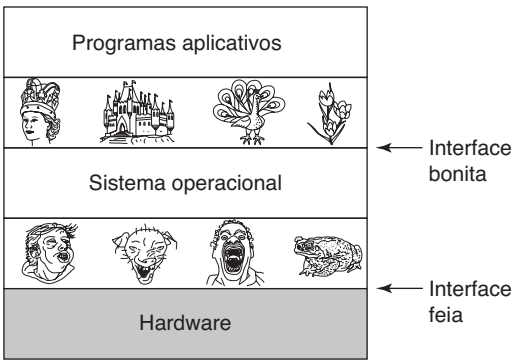

Figura 1.2 Sistemas operacionais transformam hardware feio em abstrações bonitas.

no sistema operacional e usam as abstrações que o Windows fornece, mas oferecem interfaces de usuário muito diferentes. De modo semelhante, um usuário Linux executando Gnome ou KDE vê uma interface muito diferente daquela vista por um usuário trabalhando diretamente sobre o X-Window System (orientado a texto) subjacente, mas as abstrações do sistema operacional subjacente são as mesmas em ambos os casos.

Neste livro, estudaremos as abstrações fornecidas pelos programas aplicativos em mais detalhes, mas falaremos muito menos sobre interfaces com o usuário. Esse é um tema amplo e importante, mas apenas perifericamente relacionado com sistemas operacionais.

1.1.2 O sistema operacional como um gerenciador de recursos

O conceito de um sistema operacional como provedor de uma interface conveniente a seus usuários é uma visão top-down (abstração do todo para as partes). Em uma visão alternativa, bottom-up (abstração das partes para o todo), o sistema operacional gerencia todas as partes de um sistema complexo. Computadores modernos são constituídos de processadores, memórias, temporizadores, discos, dispositivos apontadores tipo mouse, interfaces de rede, impressoras e uma ampla variedade de outros dispositivos. Segundo essa visão, o trabalho do sistema operacional é fornecer uma alocação ordenada e controlada de processadores, memórias e dispositivos de E/S entre vários programas que competem por eles.

Sistemas operacionais modernos permitem que múltiplos programas sejam executados ao mesmo tempo. Imagine o que aconteceria se três programas em execução em algum computador tentassem imprimir suas saídas simultaneamente na mesma impressora. As primeiras linhas poderiam ser do programa 1, as linhas seguintes seriam do programa 2, algumas outras do programa 3, e assim por diante. Resultado: uma confusão. O sistema operacional pode trazer ordem a essa confusão potencial, armazenando temporariamente no disco todas as saídas destinadas à impressora. Quando um programa é finalizado,

o sistema operacional poderia então enviar sua saída, que estaria no arquivo em disco, para a impressora. Ao mesmo tempo, o outro programa poderia continuar gerando mais saídas, que não estariam, obviamente, indo para a impressora (ainda).

Quando um computador (ou uma rede) tem múltiplos usuários, a necessidade de gerenciar e proteger a memória, dispositivos de E/S e outros recursos é muito maior, já que, de outra maneira, os usuários poderiam interferir uns nos outros. Além disso, os usuários muitas vezes precisam compartilhar não somente hardware, mas também informação (arquivos, bancos de dados etc.). Em resumo, essa visão do sistema operacional mostra que sua tarefa principal é manter o controle sobre quem está usando qual recurso, garantindo suas requisições de recursos, controlando as contas e mediando conflitos de requisições entre diferentes programas e usuários.

O gerenciamento de recursos realiza o **compartilhamento** (ou multiplexação) desses recursos de duas maneiras diferentes: no tempo e no espaço. Quando um recurso é compartilhado (multiplexado) no tempo, diferentes programas ou usuários aguardam sua vez de usá-lo. Primeiro, um deles obtém o uso do recurso, daí um outro, e assim por diante. Por exemplo, com somente uma CPU e múltiplos programas precisando ser executados nela, o sistema operacional aloca a CPU a um programa, e após este ser executado por tempo suficiente, outro programa obtém seu uso, então outro e, por fim, o primeiro programa novamente. Determinar como o recurso é compartilhado no tempo — quem vai depois de quem e por quanto tempo — é tarefa do sistema operacional. Outro exemplo de compartilhamento no tempo se dá com a impressora. Quando múltiplas saídas são enfileiradas para imprimir em uma única impressora, deve-se decidir sobre qual será a próxima saída a ser impressa.

O outro tipo de compartilhamento (multiplexação) é o de espaço. Em lugar de consumidores esperando sua vez, cada um ocupa uma parte do recurso. Por exemplo, a memória principal é normalmente dividida entre vários programas em execução. Assim, cada um pode residir ao mesmo tempo na memória (por exemplo, a fim de ocupar a CPU temporariamente). Existindo memória suficiente para abrigar múltiplos programas, é mais eficiente mantê-los nela em vez de destinar toda a memória a um só deles, especialmente se o programa precisar apenas de uma pequena fração do total. Naturalmente, isso levanta questões sobre justiça, proteção etc., e cabe ao sistema operacional resolvê-las. Outro recurso que é compartilhado no espaço é o disco (rígido). Em vários sistemas, um único disco pode conter arquivos de muitos usuários ao mesmo tempo. Alocar espaço em disco e manter o controle sobre quem está usando quais parcelas do disco é uma típica tarefa de gerenciamento de recursos do sistema operacional.

1.2 História dos sistemas operacionais

Os sistemas operacionais vêm passando por um processo gradual de evolução. Nas próximas seções veremos algumas das principais fases dessa evolução. Como a história dos sistemas operacionais é bastante ligada à arquitetura de computadores sobre a qual eles são executados, veremos as sucessivas gerações de computadores para entendermos as primeiras versões de sistemas operacionais. Esse mapeamento das gerações de sistemas operacionais em relação a gerações de computadores é um tanto vago, mas revela a existência de uma certa estrutura.

A sequência de eventos apresentada a seguir é em grande medida cronológica, mas foi um percurso acidentado. Os desenvolvimentos não esperaram que os anteriores terminassem antes de se iniciarem. Houve muitas sobreposições, para não falar de muitos falsos começos e becos sem saída. Considere-a como um guia, não como a palavra final.

O primeiro computador digital foi projetado pelo matemático inglês Charles Babbage (1792–1871). Embora Babbage tenha empregado a maior parte de sua vida e de sua fortuna para construir sua 'máquina analítica', ele nunca conseguiu vê-la funcionando de modo apropriado, pois era inteiramente mecânica e a tecnologia de sua época não poderia produzir as rodas, as engrenagens e as correias de alta precisão que eram necessárias. É óbvio que a máquina analítica não possuía um sistema operacional.

Outro aspecto histórico interessante foi que Babbage percebeu que seria preciso um software para sua máquina analítica. Para isso, ele contratou uma jovem chamada Ada Lovelace, filha do famoso poeta inglês Lord Byron, como a primeira programadora do mundo. A linguagem de programação Ada® foi assim denominada em sua homenagem.

1.2.1 A primeira geração (1945–1955) – válvulas

Depois dos infrutíferos esforços de Babbage, seguiram-se poucos progressos na construção de computadores digitais até a Segunda Guerra Mundial, que estimulou uma explosão de atividades. O professor John Atanasoff e seu então aluno de graduação Clifford Berry construíram o que consideramos o primeiro computador digital em funcionamento, na Universidade do Estado de Iowa. Ele usava 300 válvulas. Quase ao mesmo tempo, Konrad Zuse, em Berlim, construiu o computador Z3 de relés. Em 1944, o Colossus foi desenvolvido por um grupo em Bletchley Park, Inglaterra, o Mark foi construído por Howard Aiken em Harvard e o ENIAC foi construído por William Mauchley e seu aluno de graduação J. Presper Eckert na Universidade da Pensilvânia. Alguns eram binários, alguns usavam válvulas, alguns eram programáveis, mas todos eram muito primitivos e levavam segundos para executar até o cálculo mais simples.

Naquela época, um mesmo grupo de pessoas projetava, construía, programava, operava e realizava a manutenção de cada máquina. Toda a programação era feita em código de máquina absoluto e muitas vezes conectando plugs em painéis para controlar as funções básicas da máquina. Não havia linguagens de programação (nem mesmo a linguagem assembly existia). Os sistemas operacionais também ainda não haviam sido inventados. O modo normal de operação era o seguinte: o programador reservava antecipadamente tempo de máquina em uma planilha, ia para a sala da máquina, inseria seu painel de programação no computador e passava algumas horas torcendo para que nenhuma das 20 mil válvulas queimasse durante a execução. Praticamente todos os problemas eram cálculos numéricos diretos, como determinar tabelas de senos, cossenos e logaritmos.

No início da década de 1950, essa rotina havia sido aprimorada com a introdução das perfuradoras de cartões. Era possível, então, escrever programas em cartões e lê-los em lugar de painéis de programação; de outra maneira, o procedimento seria o mesmo.

1.2.2 | A segunda geração (1955–1965) – transistores e sistemas em lote (batch)

A introdução do transistor em meados da década de 1950 mudou o quadro radicalmente. Os computadores tornaram-se suficientemente confiáveis para que pudessem ser fabricados e comercializados com a expectativa de que continuariam a funcionar por tempo suficiente para executar algum trabalho útil. Pela primeira vez, havia uma clara separação entre projetistas, fabricantes, programadores e técnicos da manutenção.

Essas máquinas — então denominadas **computadores de grande porte** (*mainframes*) — ficavam isoladas em salas especiais com ar-condicionado, operadas por equipes profissionais. Somente grandes corporações, agências governamentais ou universidades podiam pagar vários milhões de dólares para tê-las. Para uma **tarefa** (isto é, um programa ou um conjunto de programas) ser executada, o programador primeiro escrevia o programa no papel (em Fortran ou em linguagem assembly) e depois o perfurava em cartões. Ele então levava o maço de cartões para a sala de entradas, entregava-o a um dos operadores e ia tomar um café até que a saída impressa estivesse pronta.

Ao fim da execução de uma tarefa pelo computador, um operador ia até a impressora, retirava sua saída e a levava para a sala de saídas, de modo que o programador pudesse retirá-la mais tarde. Ele então apanhava um dos maços de cartões que foram trazidos para a sala de entradas e o colocava na leitora de cartões. Se fosse necessário um compilador Fortran, o operador precisava retirar do armário o maço de cartões correspondente e lê-lo. Muito tempo de computador era desperdiçado enquanto os operadores andavam pela sala das máquinas.

Por causa do alto custo do equipamento, era natural que se começasse a buscar maneiras de reduzir o desperdício de tempo no uso da máquina. A solução geralmente adotada era a do **sistema em lote** (*batch*). A ideia era gravar várias tarefas em fita magnética usando um computador relativamente mais barato, como o IBM 1401, que era muito bom para ler cartões, copiar fitas e imprimir saídas, mas não tão eficiente em cálculos numéricos.

Outras máquinas mais caras, como a IBM 7094, eram usadas para a computação propriamente dita. Essa situação é mostrada na Figura 1.3.

Depois de aproximadamente uma hora acumulando um lote de tarefas, os cartões eram lidos em uma fita magnética, que era encaminhada para a sala das máquinas, onde era montada em uma unidade de fita. O operador, então, carregava um programa especial (o antecessor do sistema operacional de hoje), que lia a primeira tarefa da fita e executava-a. A saída não era impressa, mas gravada em uma segunda fita. Depois de cada tarefa terminada, o sistema operacional automaticamente lia a próxima tarefa da fita e começava a executá-la. Quando todo o lote era processado, o operador retirava as fitas de entrada e de saída, trocava a fita de entrada com a do próximo lote e levava a fita de saída para um computador 1401 imprimi-la **off-line**, isto é, não conectada ao computador principal.

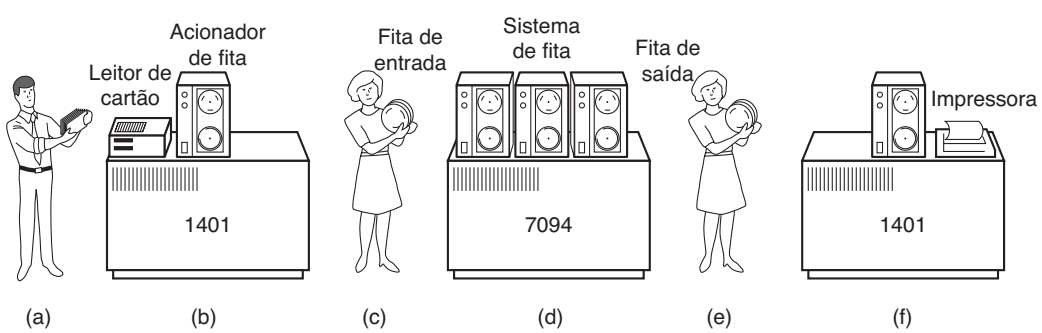

Figura 1.3 Um sistema em lote (*batch*) antigo. (a) Os programadores levavam os cartões para o 1401. (b) O 1401 gravava o lote de tarefas em fita. (c) O operador levava a fita de entrada para o 7094. (d) O 7094 executava o processamento. (e) O operador levava a fita de saída para o 1401. (f) O 1401 imprimia as saídas.

A estrutura de uma tarefa típica é mostrada na Figura 1.4. Ela começava com um cartão $JOB, que especificava o tempo máximo de processamento em minutos, o número da conta a ser debitada e o número do programador. Em seguida vinha um cartão $FORTRAN, que mandava o sistema operacional carregar o compilador Fortran a partir da fita de sistema. Depois vinham os cartões do programa a ser compilado e, então, um cartão $LOAD, que ordenava ao sistema operacional o carregamento do programa-objeto recém-compilado. (Os programas compilados muitas vezes eram gravados em fitas-rascunho e tinham de ser carregados explicitamente.) Era então a vez do cartão $RUN, que dizia para o sistema operacional executar o programa com o conjunto de dados constante nos cartões seguintes. Por fim, o cartão $END marcava a conclusão da tarefa. Esses cartões de controle foram os precursores das linguagens de controle de tarefas e dos interpretadores de comando atuais.

Os grandes computadores de segunda geração foram usados, em sua maioria, para cálculos científicos, como equações diferenciais parciais, muito frequentes na física e na engenharia. Eles eram preponderantemente programados em Fortran e em linguagem assembly. Os sistemas operacionais típicos eram o FMS (Fortran Monitor System) e o IBSYS, o sistema operacional da IBM para o 7094.

1.2.3 A terceira geração (1965–1980) – CIs e multiprogramação

No início na década de 1960, a maioria dos fabricantes de computador oferecia duas linhas de produtos distintas e totalmente incompatíveis. De um lado havia os computadores científicos de grande escala e orientados a palavras, como o 7094, usados para cálculos numéricos na ciência e na engenharia. De outro, existiam os computadores comerciais orientados a caracteres, como o 1401, amplamente usados por bancos e companhias de seguros para ordenação e impressão em fitas.

Desenvolver e manter duas linhas de produtos completamente diferentes demandava grande custo para os fabricantes. Além disso, muitos dos clientes precisavam inicialmente de uma pequena máquina, mas depois expandiam seus negócios e, com isso, passavam a demandar máquinas maiores que pudessem executar todos os seus programas antigos, porém mais rapidamente.

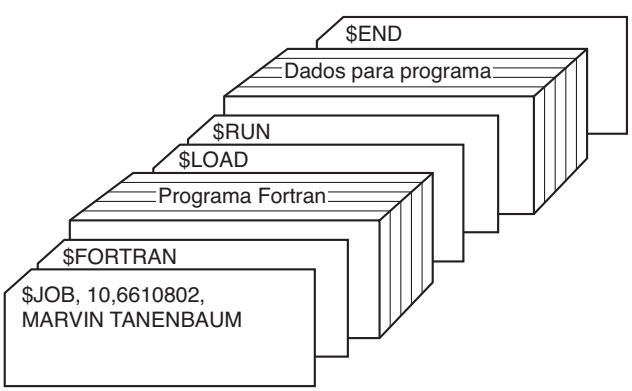

Figura 1.4 Estrutura de uma tarefa típica FMS.

A IBM tentou resolver esses problemas de uma única vez introduzindo o System/360. O 360 era uma série de máquinas — desde máquinas do porte do 1401 até as mais potentes que o 7094 — cujos softwares eram compatíveis. Os equipamentos divergiam apenas no preço e no desempenho (quantidade máxima de memória, velocidade do processador, número de dispositivos de E/S permitidos etc.). Como todas as máquinas tinham a mesma arquitetura e o mesmo conjunto de instruções, os programas escritos para uma máquina podiam ser executados em todas as outras, pelo menos teoricamente. Além disso, o 360 era voltado tanto para a computação científica (isto é, numérica) quanto para a comercial. Assim, uma única família de máquinas poderia satisfazer as necessidades de todos os clientes. Nos anos subsequentes, a IBM lançou sucessivas séries compatíveis com a linha 360, usando tecnologias mais modernas, conhecidas como séries 370, 4300, 3080 e 3090. O *Z series* é o mais novo descendente dessa linha, embora tenha se afastado consideravelmente do original.

O IBM 360 foi a primeira linha de computadores a usar **circuitos integrados** (CIs) em pequena escala, propiciando, assim, uma melhor relação custo–benefício em comparação à segunda geração de máquinas, construídas com transistores individuais. Foi um sucesso instantâneo, e a ideia de uma família de computadores compatíveis logo foi adotada por todos os outros fabricantes. Os descendentes dessas máquinas ainda estão em uso nos centros de computação. Atualmente são mais empregados para gerenciar imensos bancos de dados (por exemplo, para sistemas de reservas aéreas) ou como servidores para sites da Web, que precisam processar milhares de requisições por segundo.

O forte da ideia de 'família de máquinas' era simultaneamente sua maior fraqueza. A intenção era que qualquer software, inclusive o sistema operacional **OS/360**, pudesse ser executado em qualquer um dos modelos. O software precisava ser executado em sistemas pequenos — que muitas vezes apenas substituíam os 1401 na transferência de cartões perfurados para fita magnética — e em sistemas muito grandes, que frequentemente substituíam os 7094 na previsão do tempo e em outras operações pesadas. Tinha de ser eficiente tanto em sistemas com poucos periféricos como nos com muitos periféricos. Tinha de funcionar bem em ambientes comerciais e em ambientes científicos. E, acima de tudo, o sistema operacional precisava provar ser eficaz em todos esses diferentes usos.

Não havia como a IBM (ou qualquer outro fabricante) elaborar um software que resolvesse todos esses requisitos conflitantes. O resultado foi um sistema operacional enorme e extraordinariamente complexo, provavelmente duas ou três vezes maior que o FMS. Eram milhões de linhas

escritas em linguagem assembly por milhares de programadores, contendo milhares de erros, que precisavam de um fluxo contínuo de novas versões para tentar corrigi-los. Cada nova versão corrigia alguns erros, mas introduzia outros, fazendo com que, provavelmente, o número de erros permanecesse constante ao longo do tempo.

Um dos projetistas do OS/360, Fred Brooks, escreveu um livro genial e crítico (Brooks, 1996), descrevendo suas experiências nesse projeto. Embora seja impossível resumir o conteúdo do livro aqui, basta dizer que a capa mostra um rebanho de animais pré-históricos presos em um fosso. A capa do livro de Silberschatz et al. (2005) faz uma analogia parecida entre sistemas operacionais e dinossauros.

Apesar de seu enorme tamanho e de seus problemas, o OS/360 e os sistemas operacionais similares de terceira geração elaborados por outros fabricantes de computadores atendiam razoavelmente bem à maioria dos clientes. Também popularizavam várias técnicas fundamentais ausentes nos sistemas operacionais de segunda geração. Provavelmente a mais importante dessas técnicas foi a **multiprogramação**. No 7094, quando a tarefa atual parava para esperar por uma fita magnética terminar a transferência ou aguardava o término de outra operação de E/S, a CPU simplesmente permanecia ociosa até que a E/S terminasse. Para cálculos científicos com uso intenso do processador (*CPU-bound*), a E/S era pouco frequente, de modo que o tempo gasto com ela não era significativo. Para o processamento de dados comerciais, o tempo de espera pela E/S chegava a 80 ou 90 por cento do tempo total (*IO-bound*), de modo que algo precisava ser feito para evitar que a CPU ficasse ociosa todo esse tempo.

A solução a que se chegou foi dividir a memória em várias partes, com uma tarefa diferente em cada partição, conforme mostrado na Figura 1.5. Enquanto uma tarefa esperava que uma operação de E/S se completasse, outra poderia usar a CPU. Se um número suficiente de tarefas pudesse ser mantido na memória ao mesmo tempo, a CPU poderia permanecer ocupada por quase 100 por cento do tempo. Manter múltiplas tarefas de maneira segura na memória, por sua vez, requeria um hardware especial para proteger cada tarefa contra danos e transgressões causados por outras tarefas, mas o 360 e outros sistemas de terceira geração eram equipados com esse hardware.

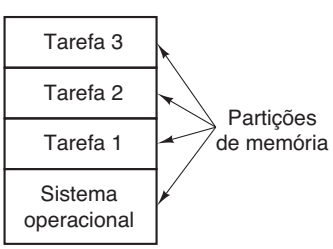

Figura 1.5 Um sistema de multiprogramação com três tarefas na memória.

Outro aspecto importante nos sistemas operacionais de terceira geração era a capacidade de transferir tarefas de cartões perfurados para discos magnéticos logo que esses chegassem à sala do computador. Dessa forma, assim que uma tarefa fosse completada, o sistema operacional poderia carregar uma nova tarefa a partir do disco na partição que acabou de ser liberada e, então, processá-lo. Essa técnica é denominada **spooling** (termo derivado da expressão *simultaneous peripheral operation online*) e também foi usada para arbitrar a saída. Com o spooling, os 1401 não eram mais necessários, e muito do leva e traz de fitas magnéticas desapareceu.

Embora os sistemas operacionais de terceira geração fossem adequados para grandes cálculos científicos e processamento maciço de dados comerciais, eram basicamente sistemas em lote (*batch systems*). Muitos programadores sentiam saudades da primeira geração, quando podiam dispor da máquina por algumas horas, podendo assim corrigir seus programas mais rapidamente. Com os sistemas de terceira geração, o intervalo de tempo entre submeter uma tarefa e obter uma saída normalmente era de muitas horas; assim, uma única vírgula errada poderia causar um erro de compilação, e o programador gastaria metade do dia para corrigi-lo.

O anseio por respostas mais rápidas abriu caminho para o tempo compartilhado ou **timesharing**, uma variante da multiprogramação na qual cada usuário se conectava por meio de um terminal on-line. Em um sistema de tempo compartilhado, se 20 usuários estivessem conectados e 17 deles estivessem pensando, falando ou tomando café, a CPU podia ser ciclicamente alocada a cada uma das três tarefas que estivessem requisitando a CPU. Como, ao depurar programas, emitem-se normalmente comandos curtos (por exemplo, compile uma rotina[1] de cinco páginas) em vez de comandos longos (por exemplo, ordene um arquivo de um milhão de registros), o computador era capaz de fornecer um serviço rápido e interativo a vários usuários e ainda processar grandes lotes de tarefas em background (segundo plano) nos instantes em que a CPU estivesse ociosa. O primeiro sistema importante de tempo compartilhado, o **CTSS** (*compatible time sharing system* — sistema compatível de tempo compartilhado, foi desenvolvido no MIT em um 7094 modificado (Corbató et al., 1962). Contudo, o tempo compartilhado só se popularizou durante a terceira geração, período em que a necessária proteção em hardware foi largamente empregada.

Depois do sucesso desse sistema (CTSS), o MIT, o Bell Labs e a General Electric (então um dos grandes fabricantes de computadores) decidiram desenvolver um 'computador utilitário', uma máquina que suportasse simultaneamente centenas de usuários compartilhando o tempo. Basearam-se no modelo do sistema de distribuição de eletricidade, ou seja, quando se precisa de energia elé-

1 Utilizaremos os termos 'rotina', 'procedimento', 'sub-rotina' e 'função' indistintamente ao longo deste livro.

trica, conecta-se o pino na tomada da parede e, não havendo nenhum problema, tem-se tanta energia quanto é necessário. Os projetistas desse sistema, conhecido como **MULTICS** (*Multiplexed information and computing service* — serviço de computação e de informação multiplexada), imaginaram uma enorme máquina fornecendo 'energia' computacional para toda a área de Boston. A ideia de que máquinas muito mais potentes que o computador de grande porte GE-645 fossem vendidas por mil dólares para milhões de pessoas, apenas 40 anos mais tarde, era ainda pura ficção científica. Seria como imaginar, hoje, trens submarinos transatlânticos e supersônicos.

O MULTICS foi projetado para suportar centenas de usuários em uma única máquina somente um pouco mais potente que um PC baseado no 386 da Intel, embora tendo muito mais capacidade de E/S. Isso não é tão absurdo quanto parece, já que, naqueles dias, sabia-se escrever programas pequenos e eficientes — uma habilidade que se vem perdendo a cada dia. Havia muitas razões para que o MULTICS não dominasse o mundo; uma delas era sua codificação em PL/I. O compilador PL/I chegou com anos de atraso e, quando isso aconteceu, dificilmente funcionava. Além disso, o MULTICS era muito ambicioso para seu tempo, tanto quanto a máquina analítica de Charles Babbage no século XIX. Apesar disso tudo, podemos dizer que a ideia lançada pelo MULTICS foi bem-sucedida.

Para resumir a história, o MULTICS introduziu muitas ideias seminais na literatura da computação, mas torná-lo um produto sério e um grande sucesso comercial era muito mais difícil do que se pensava. O Bell Labs retirou-se do projeto e a General Electric saiu do negócio de computadores. Contudo, o MIT persistiu e finalmente fez o MULTICS funcionar. Ele foi, então, vendido como produto comercial pela empresa que comprou o negócio de computadores da GE (a Honeywell) e instalado em cerca de 80 grandes empresas e universidades pelo mundo. Apesar de pouco numerosos, os usuários do MULTICS eram extremamente leais. A General Motors, a Ford e a Agência de Segurança Nacional dos Estados Unidos (U.S. National Security Agency), por exemplo, somente desligaram seus sistemas MULTICS no final dos anos 1990, três décadas depois de seu lançamento, após anos de tentativas para que a Honeywell atualizasse o hardware.

Por ora, o conceito de 'computador utilitário' permanece adormecido, mas pode muito bem ser trazido de volta na forma de poderosos servidores de Internet centralizados, nos quais máquinas usuárias mais simples são conectadas, com a maior parte do trabalho acontecendo nesses grandes servidores. A motivação para isso é que, possivelmente, a maioria das pessoas não pretende administrar um sistema cuja complexidade é crescente e cuja operação torna-se cada vez mais meticulosa, e, assim, será preferível que essa tarefa seja realizada por uma equipe de profissionais trabalhando para a empresa que opera o servidor. O comércio eletrônico (*e-commerce*) já está evoluindo nessa direção, com várias empresas no papel de centros comerciais eletrônicos (*e-malls*) em servidores multiprocessadores aos quais as máquinas dos clientes se conectam, no melhor espírito do projeto MULTICS.

Apesar da falta de sucesso comercial, o MULTICS exerceu uma enorme influência sobre os sistemas operacionais subsequentes. O MULTICS está descrito em Corbató et al. (1972), Corbató e Vyssotsky (1965), Daley e Dennis (1968), Organick (1972) e Saltzer (1974). Existe um site da Web, <www.multicians.org>, com muita informação disponível sobre o sistema, seus projetistas e seus usuários.

Outro grande desenvolvimento ocorrido durante a terceira geração foi o fenomenal crescimento dos minicomputadores, iniciado com o DEC PDP-1 em 1961. O PDP-1 tinha somente 4 K de palavras de 18 bits, mas cada máquina custava 120 mil dólares (menos de 5 por cento do preço de um 7094) e, mesmo assim, vendia como água. Para certos tipos de aplicações não numéricas, era tão rápido quanto o 7094 e deu origem a toda uma nova indústria. Rapidamente foi seguido por uma série de outros PDPs (diferentemente da família IBM, todos incompatíveis), culminando no PDP-11.

Ken Thompson, um dos cientistas da computação do Bell Labs que trabalharam no projeto MULTICS, achou um pequeno minicomputador PDP-7 que ninguém estava usando e aproveitou-o para escrever uma versão despojada e monousuário do MULTICS. Esse trabalho desenvolveu-se e deu origem ao sistema operacional **UNIX®**, que se tornou muito popular no mundo acadêmico, em agências governamentais e em muitas empresas.

A história do UNIX já foi contada em outros lugares (por exemplo, Salus, 1994). Parte dessa história será recontada no Capítulo 10. No momento, basta dizer que, em virtude de o código-fonte ter sido amplamente divulgado, várias organizações desenvolveram suas próprias (e incompatíveis) versões, o que levou ao caos. Duas das principais versões desenvolvidas, o **System V**, da AT&T, e o **BSD** (*Berkeley software distribution* — distribuição de software de Berkeley), da Universidade da Califórnia em Berkeley, também possuíam variações menores. Para tornar possível escrever programas que pudessem ser executados em qualquer sistema UNIX, o IEEE desenvolveu um padrão para o UNIX denominado **POSIX** (*portable operating system interface* — interface portátil para sistemas operacionais, ao qual a maioria das versões UNIX agora dá suporte. O POSIX define uma interface mínima de chamada de sistema a que os sistemas em conformidade com o UNIX devem dar suporte. Na verdade, alguns outros sistemas operacionais agora também dão suporte à interface POSIX.

Como comentário adicional, vale mencionar que, em 1987, o autor deste livro lançou um pequeno clone do UNIX, denominado **MINIX,** com objetivo educacional. Funcionalmente, o MINIX é muito similar ao UNIX, incluindo o suporte ao POSIX. Desde então, a versão original evoluiu

para MINIX 3, que é altamente modular e confiável. Ela tem a capacidade de detectar e substituir rapidamente módulos defeituosos ou mesmo danificados (como drivers de dispositivo de E/S) sem reinicializar e sem perturbar os programas em execução. Há um livro que descreve sua operação interna e que traz a listagem do código-fonte em seu apêndice (Tanenbaum e Woodhull, 1997). O sistema MINIX 3 está disponível gratuitamente (com o código-fonte) pela Internet em <www.minix3.org>.

O desejo de produzir uma versão gratuita do MINIX (diferente da educacional) levou um estudante finlandês, Linus Torvalds, a escrever o **Linux**. Esse sistema foi diretamente inspirado e desenvolvido a partir do MINIX e originalmente suportou vários de seus aspectos (por exemplo, o sistema de arquivos do MINIX). Ele tem sido estendido de várias maneiras, mas ainda mantém uma grande parte da estrutura comum ao MINIX e ao UNIX. Os leitores interessados em uma história detalhada do Linux e do movimento de fonte aberta podem querer ler o livro de Glyn Mood (2001). A maioria do que é dito sobre o UNIX nesse livro se aplica ao System V, ao BSD, ao MINIX, ao Linux e a outras versões e clones do UNIX também.

1.2.4 A quarta geração (1980–presente) – computadores pessoais

Com o desenvolvimento de circuitos integrados em larga escala (*large scale integration* — LSI), que são chips contendo milhares de transistores em um centímetro quadrado de silício, surgiu a era dos computadores pessoais. Em termos de arquitetura, os computadores pessoais (inicialmente denominados **microcomputadores**) não eram muito diferentes dos minicomputadores da classe PDP-11, mas no preço eram claramente diferentes. Se o minicomputador tornou possível para um departamento, uma empresa ou uma universidade terem seu próprio computador, o chip microprocessador tornou possível a um indivíduo qualquer ter seu próprio computador pessoal.

Em 1974, a Intel lançou o 8080, a primeira CPU de 8 bits de uso geral, e buscava um sistema operacional para o 8080, em parte para testá-lo. A Intel pediu a um de seus consultores, Gary Kildall, para escrevê-lo. Kildall e um amigo inicialmente construíram um controlador para a então recém-lançada unidade de discos flexíveis de 8 polegadas da Shugart Associates e a utilizaram com um 8080, produzindo, assim, o primeiro microcomputador com unidade de discos flexíveis. Kildall então escreveu para ele um sistema operacional baseado em disco denominado **CP/M** (*control program for microcomputers* — programa de controle para microcomputadores). Como a Intel não acreditava que microcomputadores baseados em disco tivessem muito futuro, Kildall requisitou os direitos sobre o CP/M e a Intel os cedeu. Kildall formou então uma empresa, a Digital Research, para aperfeiçoar e vender o CP/M.

Em 1977, a Digital Research reescreveu o CP/M para torná-lo adequado à execução em muitos microcomputadores utilizando 8080, Zilog Z80 e outros microprocessadores. Muitos programas aplicativos foram escritos para serem executados no CP/M, permitindo que ele dominasse completamente o mundo da microcomputação por cerca de cinco anos.

No início dos anos 1980, a IBM projetou o IBM PC e buscou um software para ser executado nele. O pessoal da IBM entrou em contato com Bill Gates para licenciar seu interpretador Basic. Também lhe foi indagado se ele conhecia algum sistema operacional que pudesse ser executado no PC. Gates sugeriu que a IBM contatasse a Digital Research, a empresa que dominava o mundo dos sistemas operacionais naquela época. Tomando seguramente a pior decisão de negócios registrada na história, Kildall recusou-se a se reunir com a IBM, enviando em seu lugar um subordinado. Para piorar, o advogado dele foi contra assinar um acordo de sigilo sobre o PC que ainda não havia sido divulgado. Consequentemente, a IBM voltou a Gates, perguntando-lhe se seria possível fornecer-lhes um sistema operacional.

Então Gates percebeu que uma fabricante local de computadores, a Seattle Computer Products, possuía um sistema operacional adequado, o **DOS** (*disk operating system* — sistema operacional de disco). Entrou em contato com essa empresa e disse que queria comprá-la (supostamente por 75 mil dólares), o que foi prontamente aceito. Gates ofereceu à IBM um pacote DOS/Basic, e ela aceitou. A IBM quis fazer algumas modificações, e para isso Gates contratou a pessoa que tinha escrito o DOS, Tim Paterson, como funcionário da empresa embrionária de Gates, a Microsoft. O sistema revisado teve seu nome mudado para **MS-DOS** (*MicroSoft disk operating system* — sistema operacional de disco da Microsoft) e rapidamente viria a dominar o mercado do IBM PC. Um fator decisivo para isso foi a decisão de Gates (agora, olhando o passado, extremamente sábia) de vender o MS-DOS para empresas de computadores acompanhando o hardware, em vez de tentar vender diretamente aos usuários finais (pelo menos inicialmente), como Kildall tentou fazer com o CP/M. Depois de todos esses acontecimentos, Kildall morreu repentina e inesperadamente de causas que não foram completamente reveladas.

Quando, em 1983, o sucessor do IBM PC, o IBM PC/AT, foi lançado utilizando a CPU Intel 80286, o MS-DOS avançava firmemente ao mesmo tempo que o CP/M definhava. O MS-DOS foi, mais tarde, também amplamente usado com o 80386 e o 80486. Mesmo com uma versão inicial bastante primitiva, as versões subsequentes do MS-DOS incluíram aspectos mais avançados, muitos deles derivados do UNIX. (A Microsoft conhecia bem o UNIX, pois, nos primeiros anos da empresa, vendeu uma versão para microcomputadores do UNIX, denominada XENIX.)

O CP/M, o MS-DOS e outros sistemas operacionais dos primeiros microcomputadores eram todos baseados na di-

gitação de comandos em um teclado, feita pelo usuário. Isso finalmente mudou graças a um trabalho de pesquisa de Doug Engelbart no Stanford Research Institute nos anos 1960. Engelbart inventou uma interface gráfica completa — voltada para o usuário — com janelas, ícones, menus e mouse, denominada **GUI** (*graphical user interface*). Essas ideias de Engelbart foram adotadas por pesquisadores do Xerox Parc e incorporadas às máquinas que eles projetaram.

Um dia, Steve Jobs, que coinventou o computador Apple na garagem de sua casa, visitou o Parc, viu uma interface gráfica GUI e instantaneamente percebeu seu potencial, algo que a gerência da Xerox reconhecidamente não tinha feito. Esse erro de proporção gigantesca levou à elaboração do livro *Fumbling the future* (Smith e Alexander, 1988). Jobs então iniciou a construção de um Apple dispondo de uma interface gráfica GUI. Esse projeto, denominado Lisa, foi muito dispendioso e falhou comercialmente. A segunda tentativa de Jobs, o Apple Macintosh, foi um enorme sucesso, não somente por seu custo muito menor que o do Lisa, mas também porque era mais **amigável ao usuário**, destinada a usuários que não só nada sabiam sobre computadores, mas também não tinham a menor intenção de um dia aprender sobre eles. No mundo criativo do design gráfico, da fotografia digital profissional e da produção profissional de vídeos digitais, os computadores Macintosh são amplamente empregados e os usuários são seus grandes entusiastas.

Quando a Microsoft decidiu elaborar um sucessor para o MS-DOS, estava fortemente influenciada pelo sucesso do Macintosh. Desenvolveu um sistema denominado Windows, baseado na interface gráfica GUI, que era executado originalmente em cima do MS-DOS (isto é, era como se fosse um interpretador de comandos — *shell* — em vez de um sistema operacional de verdade). Por aproximadamente dez anos, de 1985 a 1995, o Windows permaneceu apenas como um ambiente gráfico sobre o MS-DOS. Contudo, em 1995 lançou-se uma versão do Windows independente do MS-DOS, o Windows 95. Nessa versão, o Windows incorporou muitos aspectos de um sistema operacional, usando o MS-DOS apenas para ser carregado e executar programas antigos do MS-DOS. Em 1998, lançou-se uma versão levemente modificada desse sistema, chamada Windows 98. No entanto, ambos, o Windows 95 e o Windows 98, ainda continham uma grande quantidade de código em linguagem assembly de 16 bits da Intel.

Outro sistema operacional da Microsoft é o **Windows NT** (NT é uma sigla para *new technology*), que é compatível com o Windows 95 em um certo nível, mas reescrito internamente por completo. É um sistema de 32 bits completo. O líder do projeto do Windows NT foi David Cutler, que foi também um dos projetistas do sistema operacional VAX VMS, e, por isso, algumas ideias do VMS estão presentes no NT. Na realidade, havia tantas ideias do VMS no sistema que a proprietária do VMS, a DEC, processou a Microsoft. As partes entraram em acordo sobre o caso por uma enorme quantia de dinheiro. A Microsoft esperava que a primeira versão do NT 'aposentasse' o MS-DOS e todas as outras versões do Windows, já que o NT era muito superior, mas isso não aconteceu. Somente com a versão Windows NT 4.0 foi que ele finalmente deslanchou, especialmente em redes corporativas. A versão 5 do Windows NT foi renomeada Windows 2000 no início de 1999. Seu objetivo era suceder tanto o Windows 98 quanto o Windows NT 4.0.

Essa versão também não obteve êxito, e então a Microsoft lançou mais uma versão do Windows 98, denominada **Windows Me** (**Millennium edition**). Em 2001, uma versão ligeiramente atualizada do Windows 2000, chamada Windows XP, foi lançada. Essa versão teve duração muito maior (seis anos), substituindo basicamente todas as versões anteriores do Windows. Em janeiro de 2007, a Microsoft finalmente lançou o sucessor do Windows XP, chamado Vista. Ele apresentou uma nova interface gráfica, Aero, e muitos programas de usuário novos ou atualizados. A Microsoft espera que ele substitua o Windows XP completamente, mas esse processo pode levar a maior parte da década.

O outro grande competidor no mundo dos computadores pessoais é o UNIX (e seus vários derivados). O UNIX é o mais forte em servidores de rede e empresariais, mas está cada vez mais presente em desktops, especialmente em países em rápido desenvolvimento como Índia e China. Em computadores baseados em Pentium, o Linux está se tornando uma alternativa popular para estudantes e um crescente número de usuários corporativos. Como comentário adicional, por todo o livro usaremos o termo 'Pentium' para Pentium I, II, III e 4, bem como para seus sucessores, como o Core 2 Duo. O termo **x86** também é usado algumas vezes para indicar toda a série de CPUs Intel que remontam ao 8086, ao passo que 'Pentium' será utilizado para significar todas as CPUs do Pentium I em diante. É bem verdade que esse termo não é perfeito, mas não há outro melhor disponível. Deve-se tentar imaginar quem foi o gênio do marketing da Intel que descartou uma marca (Pentium) que metade do mundo conhecia bem e respeitava e a substituiu por termos como 'Core 2 duo', que poucas pessoas compreendem — pense rápido, o que significa o '2' e o que quer dizer 'duo'? Talvez 'Pentium 5' (ou 'Pentium 5 dual core' etc.) fosse difícil demais de lembrar. **FreeBSD** também é um derivado popular do UNIX, originado do projeto BSD em Berkeley. Todos os computadores Macintosh modernos executam uma versão modificada do FreeBSD. O UNIX também é padrão em estações de trabalho equipadas com chips RISC de alto desempenho, como os vendidos pela Hewlett-Packard e pela Sun Microsystems.

Muitos usuários do UNIX, especialmente programadores experientes, preferem uma interface baseada em comandos para uma GUI, de forma que quase todos os sistemas UNIX dão suporte a um sistema de janelas denominado **X** (**X Windows**, também conhecido como **X11**) produzido no MIT. Esse sistema trata o gerenciamento básico de janelas de modo a permitir que usuários criem, re-

movam, movam e redimensionem as janelas usando um mouse. Muitas vezes uma interface gráfica GUI completa, como o **Gnome** ou **KDE**, está disponível para ser executada em cima do sistema X Windows, dando ao UNIX a aparência do Macintosh ou do Microsoft Windows, para aqueles usuários UNIX que assim o desejarem.

Um fato interessante, que teve início em meados dos anos 1980, foi o desenvolvimento das redes de computadores pessoais executando **sistemas operacionais de rede** e **sistemas operacionais distribuídos** (Tanenbaum e Van Steen, 2002). Em um sistema operacional de redes, os usuários sabem da existência de múltiplos computadores e podem conectar-se a máquinas remotas e copiar arquivos de uma máquina para outra. Cada máquina executa seu próprio sistema operacional local e tem seu próprio usuário local (ou usuários locais).

Sistemas operacionais de rede não são fundamentalmente diferentes de sistemas operacionais voltados para um único processador: eles obviamente precisam de um controlador de interface de rede e de algum software de baixo nível para controlá-la, bem como de programas para conseguir sessões remotas e também ter acesso remoto a arquivos, mas esses acréscimos não alteram a estrutura essencial do sistema operacional.

Um sistema operacional distribuído, por outro lado, é aquele que parece aos olhos dos usuários um sistema operacional tradicional monoprocessador, mesmo que na realidade seja composto de múltiplos processadores. Os usuários não precisam saber onde seus programas estão sendo executados nem onde seus arquivos estão localizados, pois tudo é tratado automática e eficientemente pelo sistema operacional.

Os verdadeiros sistemas operacionais distribuídos requerem muito mais do que apenas adicionar algum código a um sistema operacional monoprocessador, pois os sistemas distribuídos e centralizados são muito diferentes em pontos fundamentais. Por exemplo, é comum que sistemas distribuídos permitam que aplicações sejam executadas em vários processadores ao mesmo tempo, o que exige algoritmos mais complexos de escalonamento de processadores para otimizar o paralelismo.

Atrasos de comunicação na rede muitas vezes significam que esses (e outros) algoritmos devam ser executados com informações incompletas, desatualizadas ou até mesmo incorretas. Essa situação é radicalmente diferente de um sistema monoprocessador, em que o sistema operacional tem toda a informação sobre o estado do sistema.

1.3 Revisão sobre hardware de computadores

Um sistema operacional está intimamente ligado ao hardware do computador no qual ele é executado. O sistema operacional estende o conjunto de instruções do computador e gerencia seus recursos. Para funcionar, ele deve ter um grande conhecimento sobre o hardware, pelo menos do ponto de vista do programador. Por isso, revisaremos brevemente o hardware tal como é encontrado nos computadores pessoais modernos. Em seguida, podemos entrar em detalhes sobre o que fazem os sistemas operacionais e como funcionam.

Conceitualmente, um computador pessoal simples pode ser abstraído para um modelo semelhante ao da Figura 1.6. A CPU, a memória e os dispositivos de E/S estão todos conectados por um barramento, que proporciona a comunicação de uns com os outros. Computadores pessoais modernos possuem uma estrutura mais complexa, que envolve múltiplos barramentos, os quais veremos depois. Por enquanto, o modelo apresentado é suficiente. Nas seções a seguir, revisaremos rapidamente cada um desses componentes e examinaremos alguns tópicos de hardware de interesse dos projetistas de sistemas operacionais. Não é preciso dizer que se trata de um resumo muito breve. Muitos livros sobre o tema hardware e organização de computadores foram escritos. Dois bastante conhecidos são Tanenbaum (2006) e Patterson e Hennessy (2004).

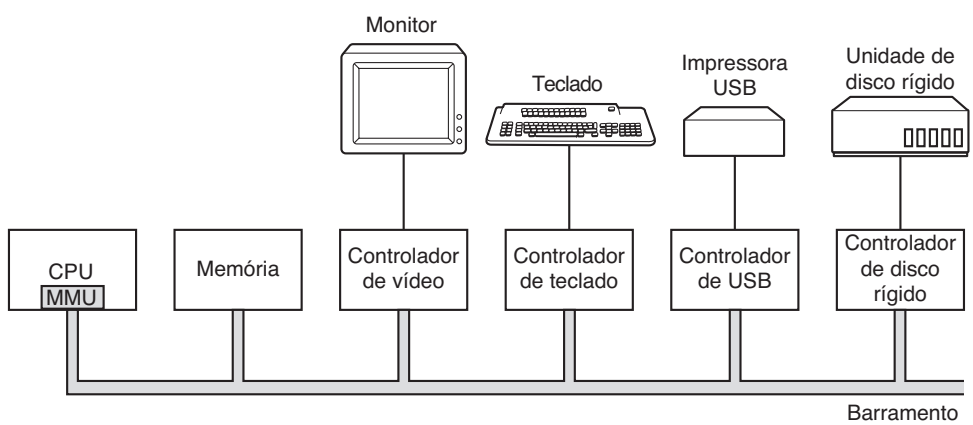

Figura 1.6 Alguns dos componentes de um computador pessoal simples.

1.3.1 Processadores

O 'cérebro' do computador é a CPU. Ela busca instruções na memória e as executa. O ciclo básico de execução de qualquer CPU é: buscar a primeira instrução da memória, decodificá-la para determinar seus operandos e qual operação executar com esses, executá-la e então buscar, decodificar e executar as instruções subsequentes. O ciclo é repetido até que o programa pare. É dessa maneira que os programas são executados.

Cada CPU tem um conjunto específico de instruções que ela pode executar. Assim, um Pentium não executa programas SPARC, nem uma SPARC consegue executar programas Pentium. Como o tempo de acesso à memória para buscar uma instrução ou operando é muito menor que o tempo para executá-la, todas as CPUs têm registradores internos para armazenamento de variáveis importantes e de resultados temporários. Por isso, o conjunto de instruções geralmente contém instruções para carregar uma palavra da memória em um registrador e armazenar uma palavra de um registrador na memória. Outras instruções combinam dois operandos provenientes de registradores, da memória ou de ambos, produzindo um resultado, como adicionar duas palavras e armazenar o resultado em um registrador ou na memória.

Além dos registradores de propósito geral, usados para conter variáveis e resultados temporários, a maioria dos computadores tem vários registradores especiais visíveis ao programador. Um deles é o **contador de programa**, que contém o endereço de memória da próxima instrução a ser buscada. Depois da busca de uma instrução, o contador de programa é atualizado para apontar a instrução seguinte.

Outro registrador especial é o **ponteiro de pilha**, que aponta para o topo da pilha atual na memória. A pilha contém uma estrutura para cada rotina chamada, mas que ainda não encerrou. Uma estrutura de pilha da rotina contém os parâmetros de entrada, as variáveis locais e as variáveis temporárias que não são mantidas nos registradores.

Outro registrador especial é a **PSW** (*program status word* — palavra de estado do programa). Esse registrador contém os bits do código de condições, os quais são alterados pelas instruções de comparação, pelo nível de prioridade da CPU, pelo modo de execução (usuário ou núcleo) e por vários outros bits de controle. Programas de usuários normalmente podem ler toda a PSW, mas em geral são capazes de alterar somente alguns de seus campos. A PSW desempenha um papel importante nas chamadas de sistema e em E/S.

O sistema operacional deve estar 'ciente' de todos os registradores. Quando o sistema operacional compartilha o tempo da CPU, ele muitas vezes interrompe a execução de um programa e (re)inicia outro. Toda vez que ele faz isso, o sistema operacional precisa salvar todos os registradores para que eles possam ser restaurados quando o programa for executado novamente.

Para melhorar o desempenho, os projetistas de CPU abandonaram o modelo simples de busca, decodificação e execução de uma instrução por vez. Muitas CPUs modernas têm recursos para executar mais de uma instrução ao mesmo tempo. Por exemplo, uma CPU pode ter unidades separadas de busca, decodificação e execução, de modo que, enquanto ela estiver executando a instrução n, ela também pode estar decodificando a instrução $n + 1$ e buscando a instrução $n + 2$. Essa organização é denominada **pipeline** e está ilustrada na Figura 1.7(a) para um pipeline com três estágios. Pipelines mais longos são comuns. Na maioria dos projetos de pipelines, uma vez que a instrução tenha sido trazida para o pipeline, ela deve ser executada, mesmo que a instrução precedente tenha constituído um desvio condicional satisfeito. Os pipelines causam grandes dores de cabeça aos projetistas de compiladores e de sistemas operacionais, pois expõem diretamente as complexidades subjacentes à máquina.

Ainda mais avançado que um processador pipeline é um processador **superescalar**, mostrado na Figura 1.7(b). Esse tipo de processador possui múltiplas unidades de execução, por exemplo, uma unidade para aritmética de números inteiros, uma unidade aritmética para ponto flutuante e outra unidade para operações booleanas. A cada vez, duas ou mais instruções são buscadas, decodificadas e armazenadas temporariamente em um buffer de instruções até que possam ser executadas. Tão logo uma unidade de execução esteja livre, o processador vai verificar se há alguma instrução que possa ser executada e, se for esse o caso, removerá a instrução do buffer de instruções e a executará. Uma implicação disso é que as instruções do programa muitas vezes são executadas fora de ordem. Normalmente, cabe ao hardware assegurar que o resultado produzido seja o mesmo de uma implementação sequencial, mas ainda permanece uma grande quantidade de problemas complexos a serem resolvidos pelo sistema operacional.

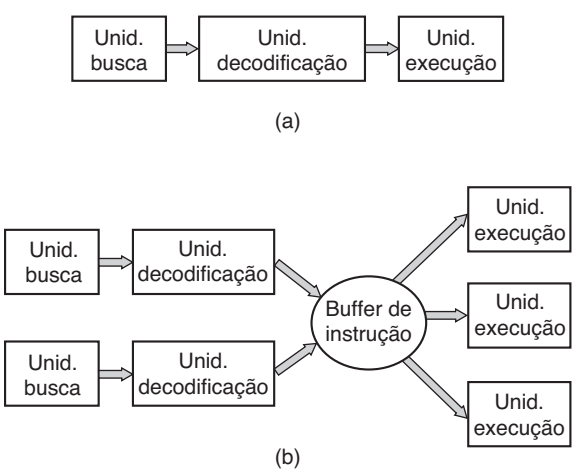

Figura 1.7 (a) Um processador com pipeline de três estágios. (b) Uma CPU superescalar.

A maioria das CPUs — exceto aquelas muito simples usadas em sistemas embarcados — apresenta dois modos de funcionamento: o modo núcleo e o modo usuário, conforme mencionado anteriormente. Em geral, o modo de funcionamento é controlado por um bit do registrador PSW. Executando em modo núcleo, a CPU pode executar qualquer instrução de seu conjunto de instruções e usar cada atributo de seu hardware. É o caso do sistema operacional: ele é executado em modo núcleo e tem acesso a todo o hardware.

Por outro lado, programas de usuários são executados em modo usuário, o que permite a execução de apenas um subconjunto das instruções e o acesso a apenas um subconjunto dos atributos. De modo geral, todas as instruções que envolvem E/S e proteção de memória são inacessíveis no modo usuário. Alterar o bit de modo no registrador PSW para modo núcleo é também, naturalmente, vedado.

Para obter serviços do sistema operacional, um programa de usuário deve fazer uma **chamada de sistema**, que, por meio de uma instrução TRAP, chaveia do modo usuário para o modo núcleo e passa o controle para o sistema operacional. Quando o trabalho do sistema operacional está terminado, o controle é retornado para o programa do usuário na instrução seguinte à da chamada de sistema. Adiante, neste mesmo capítulo, explicaremos os detalhes do processo de chamada de sistema, mas, por enquanto, pense nele como um tipo especial de procedimento de instrução de chamada que tem a propriedade adicional de chavear do modo usuário para o modo núcleo. Um aviso sobre a tipografia: usaremos letras minúsculas com fonte Helvética para indicar chamadas de sistema no decorrer do texto, como, por exemplo, read.

É bom observar que, para fazer uma chamada de sistema, há, além das instruções tipo TRAP, as armadilhas de hardware. Armadilhas de hardware advertem sobre uma situação excepcional, como a tentativa de dividir por 0 ou um underflow (incapacidade de representação de um número muito pequeno) em ponto flutuante. Em todos esses casos, o sistema operacional assume o controle e decide o que fazer. Algumas vezes o programa precisa ser fechado por causa de um erro. Outras vezes, o erro pode ser ignorado (a um número com underflow pode-se atribuir o valor 0). Por fim, quando o programa avisa previamente que quer tratar certos tipos de problemas, o controle pode ser passado de volta ao programa para deixá-lo tratar o problema.

Chips multithread e multinúcleo

A lei de Moore afirma que o número de transistores de um chip dobra a cada 18 meses. Essa 'lei' não é um tipo de lei da física, como a conservação do momento, mas é uma observação do cofundador da Intel, Gordon Moore, sobre a rapidez com que os engenheiros de processo das companhias de semicondutores são capazes de comprimir seus transistores. A lei de Moore foi válida por três décadas e espera-se que o seja por pelo menos mais uma década.

A abundância de transistores está levando a um problema: o que fazer com todos eles? Vimos uma abordagem anteriormente: arquiteturas superescalares, com unidades funcionais múltiplas. Mas, à medida que o número de transistores aumenta, há mais possibilidades ainda. Algo óbvio a fazer é colocar caches maiores no chip da CPU e, sem dúvida, isso está acontecendo mas, no fim, o ponto de rendimentos decrescentes será alcançado.

O próximo passo é replicar não apenas as unidades funcionais, mas também parte da lógica de controle. O Pentium 4 e alguns outros chips de CPU têm essa propriedade, chamada **multithreading** ou **hyperthreading** (o nome dado pela Intel). Para uma primeira aproximação, o que ela faz é permitir que a CPU mantenha o estado de dois threads diferentes e faça em seguida o chaveamento para trás e para adiante em uma escala de tempo de nanossegundos. (Um thread é um tipo de processo leve, que, por sua vez, é um programa de execução; nós o analisaremos em detalhes no Capítulo 2.) Por exemplo, se um dos processos precisa ler uma palavra a partir da memória (o que leva muitos ciclos de relógio), uma CPU multithread pode fazer o chaveamento para outro thread. O multithreading não oferece paralelismo real. Apenas um processo por vez é executado, mas o tempo de chaveamento é reduzido para a ordem de um nanossegundo.

O multithreading tem implicações para o sistema operacional porque cada thread aparece para o sistema operacional como uma CPU. Considere um sistema com duas CPUs efetivas, cada uma com dois threads. O sistema operacional verá quatro CPUs. Se houver trabalho suficiente para manter apenas duas CPUs ocupadas em dado momento, ele pode escalonar inadvertidamente dois threads na mesma CPU, deixando a outra completamente ociosa. Essa escolha é muito menos eficiente do que usar um thread em cada CPU. O sucessor do Pentium 4, a arquitetura Core (também o Core 2), não tem hyperthreading, mas a Intel anunciou que o sucessor do Core terá essa propriedade novamente.

Além do multithreading, temos chips de CPU com dois ou quatro ou mais processadores completos ou **núcleos**. Os chips multinúcleo da Figura 1.8 trazem, de fato, quatro

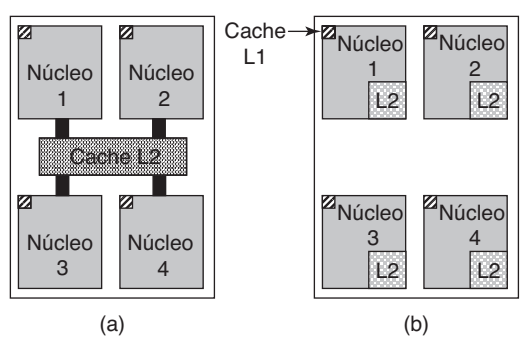

Figura 1.8 (a) Chip quad-core com uma cache L2 compartilhada. (b) Um chip quad-core com caches L2 separadas.

minichips, cada um com uma CPU independente. (As caches serão explicadas a seguir.) A utilização de tais chips multinúcleo requer um sistema operacional para multiprocessadores.

1.3.2 | Memória

O segundo principal componente em qualquer computador é a memória. Idealmente, uma memória deveria ser bastante rápida (mais veloz do que a execução de uma instrução, para que a CPU não fosse atrasada pela memória), além de muito grande e barata. Nenhuma tecnologia atual atinge todos esses objetivos e, assim, uma abordagem diferente tem sido adotada, ou seja, construir o sistema de memória como uma hierarquia de camadas, conforme mostra a Figura 1.9. A camada superior tem maior velocidade, menor capacidade e maior custo por bit que as camadas inferiores, frequentemente com uma diferença de um bilhão ou mais.

A camada superior consiste nos registradores internos à CPU. Eles são feitos com o mesmo material da CPU e são, portanto, tão rápidos quanto ela. Consequentemente, não há atraso em seu acesso. A capacidade de memória disponível neles em geral é de 32 × 32 bits para uma CPU de 32 bits e de 64 × 64 bits para uma CPU de 64 bits. Ou seja, menos de 1 KB em ambos os casos. Os programas devem gerenciar os registradores (isto é, decidir o que colocar neles) por si mesmos, no software.

Nessa hierarquia do sistema de memória, abaixo da camada de registradores, vem outra camada, denominada memória **cache**, que é controlada principalmente pelo hardware. A memória principal é dividida em **linhas de cache** (*cache lines*), normalmente com 64 bytes cada uma. A linha 0 consiste nos endereços de 0 a 63, a linha 1 consiste nos endereços entre 64 e 127, e assim por diante. As linhas da cache mais frequentemente usadas são mantidas em uma cache de alta velocidade, localizada dentro ou muito próxima à CPU. Quando o programa precisa ler uma palavra de memória, o hardware da memória cache verifica se a linha necessária está na cache. Se a linha requisitada estiver **presente na cache** (*cache hit*), a requisição será atendida pela cache e nenhuma requisição adicional é enviada à memória principal por meio do barramento. A busca na cache quando a linha solicitada está presente dura normalmente em torno de dois ciclos de CPU. Se a linha requisitada estiver **ausente da cache** (*cache miss*), uma requisição adicional será enviada à memória principal, com uma substancial penalidade de tempo. A memória cache tem tamanho limitado por causa de seu alto custo. Algumas máquinas têm dois ou até três níveis de cache, cada um mais lento e de maior capacidade que o anterior.

O conceito de caching desempenha um papel importante em muitas áreas da ciência da computação, não apenas em colocar linhas da RAM no cache. Sempre que houver um recurso grande que possa ser dividido em partes, alguns dos quais são muito mais utilizados que outros, caching é muitas vezes utilizado para aperfeiçoar o desempenho. Os sistemas operacionais o utilizam o tempo todo. Por exemplo, a maioria dos sistemas operacionais mantém (partes de) arquivos muito utilizados na memória principal para tentar evitar buscá-los no disco repetidamente. De modo semelhante, os resultados da conversão de nomes de rota longos como

/home/ast/projects/minix3/src/kernel/clock.c

no endereço de disco onde o arquivo está localizado podem ser registrados em cache para evitar repetir buscas. Por fim, quando um endereço de uma página da Web (URL) é convertido em um endereço de rede (endereço IP), o resultado pode ser armazenado para uso futuro. Há muitos outros usos.

Em qualquer sistema cache, muitas perguntas surgem rapidamente, incluindo:

1. Quando colocar um novo item no cache.
2. Em qual linha de cache colocar o novo item.
3. Que item remover da cache quando for preciso espaço.
4. Onde colocar um item recentemente desalojado na memória mais ampla.

Nem toda pergunta é relevante para cada situação de cache. Para linhas de cache da memória principal na cache da CPU, geralmente um novo item será inserido em cada ausência de cache. A linha de cache a ser usada em geral é calculada usando alguns dos bits de alta ordem do endereço de memória mencionado. Por exemplo, com 4.096 linhas de cache de 64 bytes e endereços 32 bits, os bits 6 a 17 podem

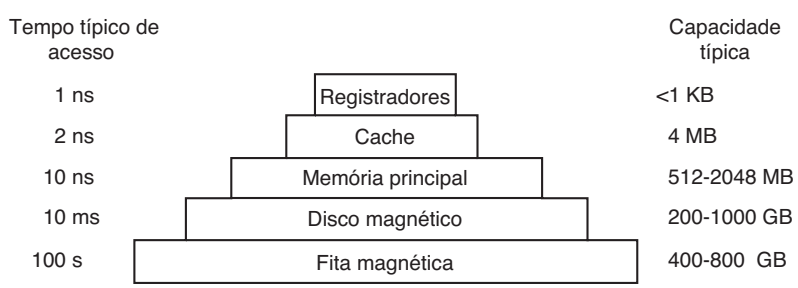

Figura 1.9 Hierarquia de memória típica. Os números são aproximações.

ser usados para especificar a linha de cache, com os bits de 0 a 5 especificando os bytes dentro da linha de cache. Nesse caso, o item a ser removido é o mesmo de quando novos dados são inseridos, mas em outros sistemas poderia ser diferente. Por fim, quando uma linha de cache é reescrita para a memória principal (se tiver sido modificada desde que foi colocada em cache), o lugar na memória para reescrevê-la é determinado exclusivamente pelo endereço em questão.

As caches são uma ideia tão boa que as CPUs modernas têm duas delas. O primeiro nível, ou **cache L1**, está sempre dentro da CPU e normalmente alimenta instruções decodificadas no mecanismo de execução da CPU. A maioria dos chips tem uma segunda cache L1 para palavras de dados muito utilizadas. As caches L1 geralmente têm 16 KB cada. Além disso, sempre há uma segunda cache, chamada **cache L2**, que armazena vários megabytes de palavras de memória usadas recentemente. A diferença entre caches L1 e L2 está na sincronização. O acesso à cache L1 é feito sem nenhum retardo, ao passo que o acesso à cache L2 envolve um retardo de um ou dois ciclos de relógio.

Tratando-se de chips multinúcleo, os projetistas têm de decidir onde colocar as caches. Na Figura 1.8(a), há uma única cache L2 compartilhada por todos os núcleos. Essa abordagem é usada em chips multinúcleo Intel. Em contraposição, na Figura 1.8(b), cada núcleo tem sua própria cache L2. Essa abordagem é usada pela AMD. Cada estratégia tem aspectos favoráveis e desfavoráveis. Por exemplo, a cache L2 compartilhada da Intel requer um controlador de cache mais complicado, mas o método da AMD dificulta manter a consistência entre as caches L2.

A memória principal é a camada seguinte na hierarquia da Figura 1.9. É a locomotiva do sistema de memória. A memória principal é muitas vezes chamada de **RAM** (*random access memory* — memória de acesso aleatório). Antigamente era chamada de **memória de núcleos** (*core memory*) porque a memória dos computadores dos anos 1950 e 1960 era constituída de pequenos núcleos de ferrite magnetizáveis. Hoje em dia, as memórias têm de centenas de megabytes a vários gigabytes e continuam crescendo rapidamente. Todas as requisições da CPU que não podem ser atendidas pela cache vão para a memória principal.

Além da memória principal, muitos computadores apresentam uma pequena memória de acesso aleatório não volátil. Ao contrário da RAM comum, a memória não volátil não perde seu conteúdo quando sua alimentação é desligada. Por exemplo, a **ROM** (*read only memory* — memória apenas de leitura) é programada na fábrica e não pode ser alterada. É rápida e barata. Em alguns computadores, o carregador (*bootstrap loader*), usado para inicializar o computador, está gravado em ROM. Algumas placas de E/S também vêm com programas em ROM para controle de dispositivos em baixo nível.

A **EEPROM** (*electrically erasable ROM* — ROM eletricamente apagável) e a **flash RAM** também são memórias de acesso aleatório não voláteis, mas diferentes da ROM, pois podem ser apagadas e reescritas. Contudo, escrever nelas leva várias vezes mais tempo que escrever em uma RAM. Desse modo, são usadas como as ROMs comuns, só que com a característica adicional de possibilitar correção de erros em programas por meio da regravação.

A memória flash também é normalmente usada como meio de armazenamento em dispositivos eletrônicos portáteis. Atua como o filme em câmeras digitais e como o disco em reprodutores de música portáteis. Essa memória tem velocidade intermediária entre as da memória RAM e de disco. Além do mais, diferentemente da memória de disco, se for apagada muitas vezes, ela se desgasta.

Existem ainda memórias voláteis em tecnologia CMOS. Muitos computadores usam memórias CMOS para manter data e hora atualizadas. A memória CMOS e o circuito de relógio que incrementa o tempo registrado nela são alimentados por uma pequena bateria, para que o tempo seja corretamente atualizado, mesmo que o computador seja desligado. A memória CMOS também pode conter os parâmetros de configuração, como de qual disco deve se inicializar a carga do sistema (*boot*). A tecnologia CMOS é usada porque consome bem menos energia, e, assim, as baterias originais instaladas na fábrica podem durar vários anos. Contudo, quando a bateria começa a falhar, o computador pode parecer um portador da doença de Alzheimer, esquecendo coisas que sabia havia anos, como, por exemplo, de qual disco rígido carregar o sistema operacional.

1.3.3 Discos

A camada seguinte nessa hierarquia é constituída pelo disco magnético (disco rígido). O armazenamento em disco é duas ordens de magnitude mais barato, por bit, que o da RAM e, muitas vezes, duas ordens de magnitude maior também. O único problema é que o tempo de acesso aleatório aos dados é cerca de três ordens de magnitude mais lento. Essa baixa velocidade é causada pelo fato de o disco ser um dispositivo mecânico, conforme ilustra a Figura 1.10.

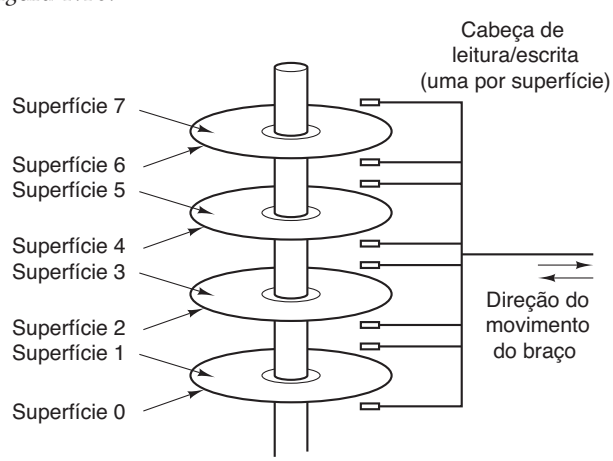

Figura 1.10 Estrutura de uma unidade de disco.

Um disco magnético consiste em um ou mais pratos metálicos que rodam a 5.400, 7.200 ou 10.800 rpm. Um braço mecânico move-se sobre esses pratos a partir da lateral, como um braço de toca-discos de um velho fonógrafo de 33 rpm tocando discos de vinil. A informação é escrita no disco em uma série de círculos concêntricos. Em qualquer posição do braço, cada cabeça pode ler uma região circular chamada de **trilha**. Juntas, todas as trilhas de uma dada posição do braço formam um **cilindro**.

Cada trilha é dividida em um certo número de setores. Cada setor tem normalmente 512 bytes. Nos discos atuais, os cilindros mais exteriores contêm mais setores que os mais interiores. Mover o braço de um cilindro para o próximo leva em torno de 1 ms. Movê-lo diretamente até um cilindro qualquer leva em geral de 5 a 10 ms, dependendo do dispositivo acionador. Uma vez com o braço na trilha correta, o acionador do disco deve esperar até que o setor desejado chegue abaixo da cabeça — um atraso adicional de 5 a 10 ms, dependendo da velocidade de rotação (rpm) do dispositivo acionador. Uma vez que o setor esteja sob a cabeça, a leitura ou a escrita ocorre a uma taxa de 50 MB/s em discos de baixo desempenho ou até 160 MB/s em discos mais rápidos.

Muitos computadores mantêm um esquema conhecido como **memória virtual**, que discutiremos em maiores detalhes no Capítulo 3. Esse esquema possibilita executar programas maiores que a memória física colocando-os em disco e usando a memória principal como um tipo de cache para as partes mais executadas. Esse esquema requer o mapeamento de endereços de memória rapidamente para converter o endereço que o programa gerou no endereço físico em RAM onde a palavra está localizada. Esse mapeamento é realizado por uma parte da CPU chamada **unidade de gerenciamento de memória** (*memory management unit* — MMU), como mostrado na Figura 1.6.

A presença da cache e da MMU pode ter um impacto importante sobre o desempenho. Em um sistema de multiprogramação, quando há o chaveamento entre programas, muitas vezes denominado **chaveamento de contexto**, pode ser necessário limpar da cache todos os blocos modificados e alterar os registros de mapeamento na MMU. Ambas são operações caras e os programadores tentam evitá-las a todo custo. Veremos algumas das implicações dessas táticas posteriormente.

1.3.4 Fitas

A última camada da hierarquia de memória é a fita magnética. Esse meio é muito utilizado como cópia de segurança (backup) do armazenamento em discos e para abrigar grandes quantidades de dados. Para ter acesso a uma fita, ela precisa ser colocada em uma unidade leitora de fitas, manualmente ou com a ajuda de um robô (manipuladores automáticos de fitas magnéticas são comuns em instalações com grandes bancos de dados). Então a fita terá de ser percorrida sequencialmente até chegar ao bloco requisitado. No mínimo, isso levaria alguns minutos. A grande vantagem da fita magnética é que ela tem um custo por bit muito baixo e é também removível — uma característica importante para fitas magnéticas utilizadas como cópias de segurança, as quais devem ser armazenadas distantes do local de processamento, para que estejam protegidas contra incêndios, inundações, terremotos etc.

A hierarquia de memória que temos discutido é o padrão mais comum, mas alguns sistemas não têm todas essas camadas ou algumas são diferentes delas (como discos ópticos). No entanto, em todas, conforme se desce na hierarquia, o tempo de acesso aleatório cresce muito, a capacidade, da mesma maneira, também aumenta bastante, e o custo por bit cai enormemente. Em vista disso, é bem provável que as hierarquias de memória ainda perdurem por vários anos.

1.3.5 Dispositivos de E/S

A CPU e a memória não são os únicos recursos que o sistema operacional tem de gerenciar. Os dispositivos de E/S também interagem intensivamente com o sistema operacional. Como se vê na Figura 1.6, os dispositivos de E/S são constituídos, geralmente, de duas partes: o controlador e o dispositivo propriamente dito. O controlador é um chip ou um conjunto de chips em uma placa que controla fisicamente o dispositivo. Ele recebe comandos do sistema operacional, por exemplo, para ler dados do dispositivo e para enviá-los.

Em muitos casos, o controle real do dispositivo é bastante complicado e cheio de detalhes. Desse modo, cabe ao controlador apresentar uma interface mais simples (mas ainda muito complexa) para o sistema operacional. Por exemplo, um controlador de disco, ao receber um comando para ler o setor 11.206 do disco 2, deve então converter esse número linear de setor em números de cilindro, setor e cabeça. Essa conversão pode ser muito complexa, já que os cilindros mais externos têm mais setores que os internos e que alguns setores danificados podem ter sido remapeados para outros. Então, o controlador precisa determinar sobre qual cilindro o braço do acionador está e emitir uma sequência de pulsos, correspondente à distância em número de cilindros. Ele deve aguardar até que o setor apropriado esteja sob a cabeça e, então, iniciar a leitura e o armazenamento de bits conforme vierem, removendo o cabeçalho e verificando a soma de verificação (*checksum*). Para realizar todo esse trabalho, em geral os controladores embutem pequenos computadores programados exclusivamente para isso.

A outra parte é o próprio dispositivo real. Os dispositivos possuem interfaces bastante simples porque não fazem nada muito diferente, e isso ajuda a torná-los padronizados. Padronizá-los é necessário para que, por exemplo, qualquer controlador de discos IDE possa controlar qual-

quer disco IDE. **IDE** é a sigla para *integrated drive electronics* e é o tipo padrão de discos de muitos computadores. Como a interface com o dispositivo real está oculta pelo controlador, tudo o que os sistemas operacionais veem é a interface do controlador, que pode ser muito diferente da interface para o dispositivo.

Uma vez que cada tipo de controlador é diferente, diferentes programas são necessários para controlá-los. O programa que se comunica com um controlador, emitindo comandos a ele e aceitando respostas, é denominado **driver de dispositivo**. Cada fabricante de controlador deve fornecer um driver específico para cada sistema operacional a que dá suporte. Assim, um digitalizador de imagens (scanner) pode vir com drivers para Windows 2000, Windows XP, Vista e Linux, por exemplo.

Para ser usado, o driver deve ser colocado dentro do sistema operacional para que possa ser executado em modo núcleo. Na realidade, os drivers podem ser executados fora do núcleo, mas poucos sistemas operacionais atuais são capacitados para essa atividade porque, para isso, é necessário permitir que um driver no espaço do usuário tenha acesso ao dispositivo de maneira controlada, algo praticamente inviável. Há três maneiras de colocar o driver dentro do núcleo. A primeira é religar o núcleo com o novo driver e, então, reinicializar o sistema. Muitos sistemas UNIX funcionam dessa maneira. A segunda maneira é adicionar uma entrada a um arquivo do sistema operacional informando que ele precisa do driver e, então, reiniciar o sistema. No momento da inicialização, o sistema operacional busca e encontra os drivers de que ele precisa e os carrega. O Windows, por exemplo, funciona assim. A terceira maneira é capacitar o sistema operacional a aceitar novos drivers enquanto estiver em execução e instalá-los sem a necessidade de reinicializar. Esse modo ainda é raro, mas está se tornando cada vez mais comum. Dispositivos acoplados a quente, como dispositivos USB e IEEE 1394 (que serão discutidos adiante), precisam sempre de drivers carregados dinamicamente.

Todo controlador tem um pequeno número de registradores usados na comunicação. Por exemplo, um controlador de discos deve ter, no mínimo, registradores para especificar endereços do disco e de memória, contador de setores e indicador de direção (leitura ou escrita). Para ativar o controlador, o driver recebe um comando do sistema operacional e o traduz em valores apropriados a serem escritos nos registradores dos dispositivos. O grupo de todos esses registradores de dispositivos forma o **espaço de porta de E/S**, um assunto ao qual retornaremos no Capítulo 5.

Em alguns computadores, os registradores dos dispositivos são mapeados no espaço de endereçamento do sistema operacional (os endereços que ele pode usar). Assim, eles podem ser lidos e escritos como se fossem palavras da memória principal. Para esses computadores, nenhuma instrução especial de E/S é necessária e os programas do usuário podem ser mantidos distantes do hardware deixando esses endereços de memória fora de seu alcance (por exemplo, usando-se registradores-base e limite). Em outros computadores, os registradores de dispositivo são colocados em um espaço especial de portas de E/S, e cada registrador tem seu endereço de porta. Nessas máquinas, instruções especiais IN e OUT são disponíveis em modo núcleo para que se permita que os drivers leiam e escrevam os registradores. O primeiro esquema elimina a necessidade de instruções especiais de E/S, mas consome parte do espaço de endereçamento. O último esquema não gasta o espaço de endereçamento, no entanto requer instruções especiais. Ambos são amplamente usados.

A entrada e a saída podem ser realizadas de três maneiras diferentes. No método mais simples, um programa de usuário emite uma chamada de sistema, a qual o núcleo traduz em uma chamada ao driver apropriado. O driver então inicia a E/S e fica em um laço perguntando continuamente se o dispositivo terminou a operação de E/S (em geral há um bit que indica se o dispositivo ainda está ocupado). Quando a operação de E/S termina, o driver põe os dados onde eles são necessários (se houver) e retorna. O sistema operacional então remete o controle para quem chamou. Esse método é chamado de **espera ocupada** e tem a desvantagem de manter a CPU ocupada interrogando o dispositivo até que a operação de E/S tenha terminado.

No segundo método, o driver inicia o dispositivo e pede a ele que o interrompa quando terminar. Dessa maneira, ele retorna o controle da CPU ao sistema operacional. O sistema operacional então bloqueia, se necessário, o programa que o chamou pedindo o serviço e procura outra tarefa para executar. Quando o controlador detecta o final da transferência, ele gera uma **interrupção** para sinalizar o término.

Interrupções são muito importantes para os sistemas operacionais; por isso, vamos examinar essa ideia mais de perto. Na Figura 1.11(a) vemos um processo de três passos para E/S. No passo 1, o driver informa ao controlador, escrevendo em seus registradores de dispositivo, o que deve ser feito. O controlador então inicia o dispositivo. Quando o controlador termina de transferir, ler ou escrever o número de bytes pedido, ele então sinaliza isso, no passo 2, ao chip controlador de interrupção por meio de certas linhas do barramento. Se o controlador de interrupção estiver preparado para aceitar essa interrupção (o que pode não ocorrer se ele estiver ocupado com outra interrupção de maior prioridade), ele sinaliza esse fato à CPU no passo 3. No passo 4, o controlador de interrupção põe o número do dispositivo no barramento para que a CPU o leia e saiba qual dispositivo acabou de terminar (muitos dispositivos podem estar executando ao mesmo tempo).

Uma vez que a CPU tenha decidido aceitar a interrupção, o contador de programa (PC) e a palavra de estado do programa (PSW) são então normalmente salvos na pilha atual e a CPU é chaveada para modo núcleo. O número do dispositivo pode ser usado como índice para uma parte

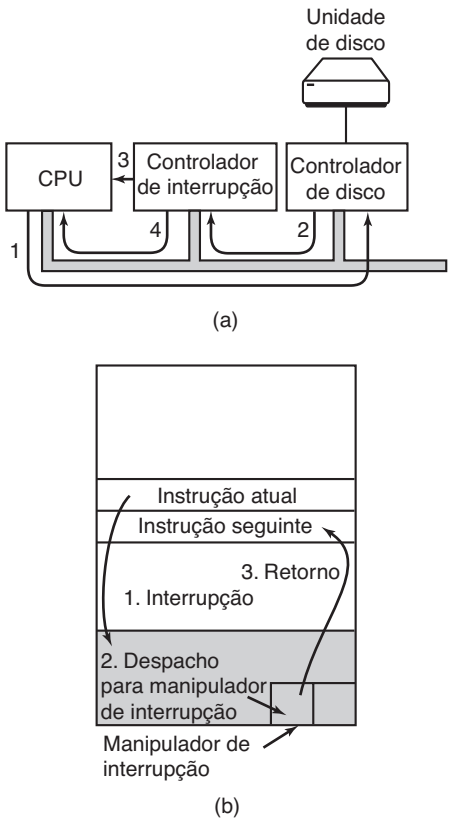

Figura 1.11 (a) Passos ao inicializar um dispositivo de E/S e obter uma interrupção. (b) O processamento da interrupção envolve fazer a interrupção, executar o manipulador de interrupção e retornar ao programa de usuário.

da memória denominada **vetor de interrupção**, que contém o endereço do manipulador de interrupção (*interrupt handler*) para esse dispositivo. Ao inicializar o 'manipulador de interrupção' (o código faz parte do driver do dispositivo que está interrompendo), ele remove o PC e a PSW da pilha e os salva. Então ele consulta o dispositivo para saber sua situação. Uma vez que o manipulador de interrupção tenha terminado, ele retorna para o programa do usuário que estava sendo executado — retorna para a primeira instrução que ainda não tenha sido executada. Esses passos são mostrados na Figura 1.11(b).

O terceiro método para implementar E/S utiliza um chip especial de acesso direto à memória (*direct memory access* — **DMA**), o qual controla o fluxo de bits entre a memória e algum controlador sem intervenção constante da CPU. A CPU configura o chip DMA, informando quantos bytes devem ser transferidos, os endereços do dispositivo e de memória envolvidos e a direção, e então o deixa executar. Quando o chip de DMA finalizar sua tarefa, causará uma interrupção, que é tratada conforme descrito anteriormente. Os hardwares de DMA e de E/S em geral serão discutidos em mais detalhes no Capítulo 5.

As interrupções muitas vezes podem acontecer em momentos totalmente inconvenientes, por exemplo, enquanto outro manipulador de interrupção estiver em execução. Por isso, a CPU tem uma maneira de desabilitar as interrupções e, então, reabilitá-las depois. Enquanto as interrupções estiverem desabilitadas, todos os dispositivos que terminem suas atividades continuam a emitir sinais de interrupção, mas a CPU não é interrompida até que as interrupções sejam habilitadas novamente. Se vários dispositivos finalizam enquanto as interrupções estiverem desabilitadas, o controlador de interrupção decide qual interrupção será acatada primeiro, com base, normalmente, em prioridades atribuídas estaticamente a cada dispositivo. O dispositivo de maior prioridade vence.

1.3.6 Barramentos

A organização da Figura 1.6 foi utilizada em minicomputadores durante muitos anos e também no IBM PC original. Contudo, à medida que os processadores e as memórias tornavam-se mais rápidos, a capacidade de um único barramento (e certamente do barramento IBM PC) tratar todo o tráfego foi chegando ao limite. Algo deveria ser feito. Como resultado, barramentos adicionais foram incluídos, tanto para dispositivos de E/S mais velozes quanto para o tráfego entre memória e CPU. Como consequência dessa evolução, um sistema Pentium avançado atualmente se parece com a Figura 1.12.

Esse sistema tem oito barramentos (cache, local, memória, PCI, SCSI, USB, IDE e ISA), cada um com diferentes funções e taxas de transferência. O sistema operacional deve conhecê-los bem para configurá-los e gerenciá-los. Os dois barramentos principais são o barramento original do IBM PC, o **ISA** (*industry standard architecture* — arquitetura--padrão industrial), e seu sucessor, o barramento **PCI** (*peripheral component interconnect* — interconexão de componentes periféricos). O barramento ISA, que foi originalmente o barramento do IBM PC/AT, funciona a 8,33 MHz e pode transferir 2 bytes de uma vez, com uma velocidade máxima de 16,67 MB/s. Ele é incluído para efeito de compatibilidade com as placas de E/S antigas e lentas. O barramento PCI foi inventado pela Intel para suceder o barramento ISA. Ele pode funcionar a 66 MHz e transferir 8 bytes por vez, resultando em uma taxa de 528 MB/s. A maioria dos dispositivos de E/S de alta velocidade atualmente usa o barramento PCI. Mesmo alguns computadores que não são da Intel utilizam o barramento PCI em virtude do grande número de placas de E/S disponíveis para ele. Os computadores novos estão sendo lançados com uma versão atualizada do barramento PCI chamada barramento **PCI Express**.

Nessa configuração, a CPU se comunica com um chip 'ponte' PCI por meio de um barramento local, e esse chip ponte PCI, por sua vez, comunica-se com a memória por intermédio de um barramento dedicado, frequentemente funcionando a 100 MHz. Os sistemas Pentium têm uma cache

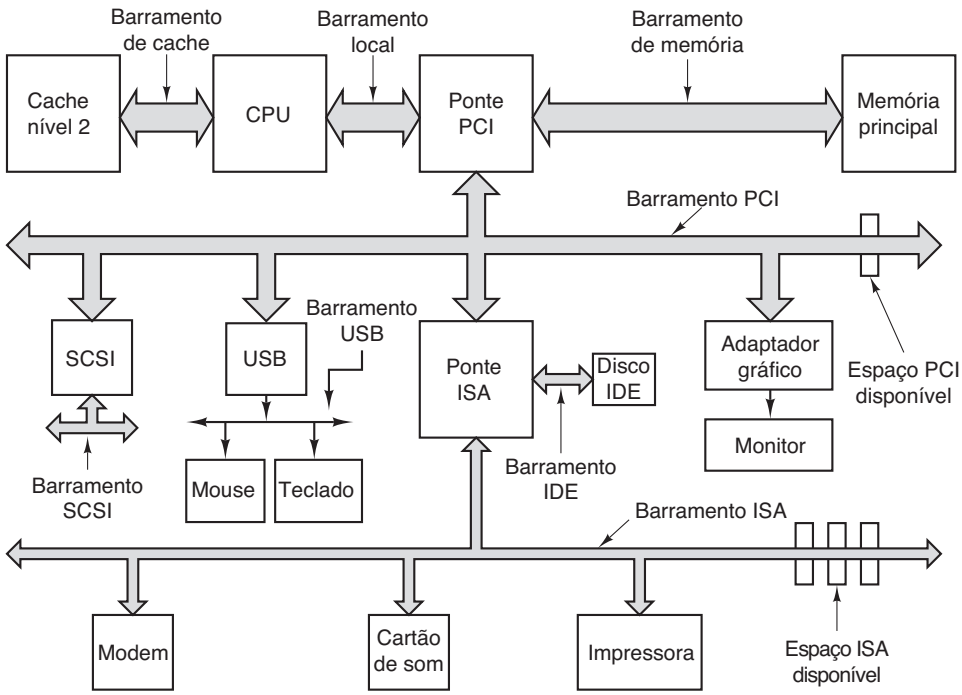

Figura 1.12 A estrutura de um sistema Pentium grande.

de nível 1 dentro do chip e uma cache de nível 2 muito maior fora do chip, conectada à CPU pelo barramento de cache.

Além disso, esse sistema contém três barramentos especializados: IDE, USB e SCSI. O barramento IDE serve para acoplar ao sistema dispositivos periféricos como discos e CD-ROMs. O barramento IDE é uma extensão da interface controladora de discos do PC/AT e atualmente constitui um padrão para disco rígido e muitas vezes também para CD-ROM em quase todos os sistemas baseados em Pentium.

O **USB** (*universal serial bus* — barramento serial universal) foi inventado para conectar ao computador todos os dispositivos lentos de E/S, como teclado e mouse. Ele usa um pequeno conector de quatro vias; duas delas fornecem alimentação aos dispositivos USB. O USB é um barramento centralizado no qual um dispositivo-raiz interroga os dispositivos de E/S a cada 1 ms para verificar se eles têm algo a ser transmitido. Ele pode tratar uma carga acumulada de 1,5 MB/s, mas o mais novo USB2.0 pode tratar 60 MB/s. Todos os dispositivos USB compartilham um único driver de dispositivo USB, tornando desnecessário instalar um novo driver para cada novo dispositivo USB. Consequentemente, os dispositivos USB podem ser adicionados ao computador sem precisar reiniciá-lo.

O barramento **SCSI** (*small computer system interface* — interface de pequeno sistema de computadores) é um barramento de alto desempenho destinado a discos rápidos, scanners e outros dispositivos que precisem de considerável largura de banda. Pode funcionar em até 160 MB/s. Está presente em sistemas Macintosh desde quando foram lançados e também é popular em sistemas UNIX e em alguns sistemas baseados na Intel.

Outro barramento (que não está ilustrado na Figura 1.12) é o **IEEE 1394**. Às vezes ele é chamado de *FireWire* — embora *FireWire* seja o nome que a Apple usa para sua implementação do 1394. Do mesmo modo que o USB, o IEEE 1394 é serial em bits, mas destinado à transferência de pacotes em velocidades de até 100 MB/s, tornando-o útil para conectar ao computador câmeras digitais e dispositivos multimídia similares. Ao contrário do USB, o IEEE 1394 não tem um controlador central.

Para funcionar em um ambiente como o da Figura 1.12, o sistema operacional deve saber o que há nele e configurá-lo. Esse requisito levou a Intel e a Microsoft a projetarem um sistema para o PC denominado **plug and play**, baseado em um conceito similar implementado pela primeira vez no Macintosh da Apple. Antes do plug and play, cada placa de E/S tinha um nível fixo de requisição de interrupção e endereços específicos para seus registradores de E/S. Por exemplo, o teclado era a interrupção 1 e usava os endereços de E/S entre 0×60 e 0×64; o controlador de disco flexível era a interrupção 6 e usava os endereços de E/S entre $0 \times 3F0$ e $0 \times 3F7$; a impressora era a interrupção 7 e usava os endereços de E/S entre 0×378 e $0 \times 37A$ etc.

Até aqui, tudo bem. O problema começava quando o usuário trazia uma placa de som e uma placa de modem e ocorria de ambas usarem, digamos, a interrupção 4. Elas conflitariam e não funcionariam juntas. A solução era incluir chaves DIP ou jumpers em todas as placas de E/S e instruir o usuário a configurá-las selecionando um nível de interrupção e endereços de dispositivos de E/S que não conflitassem com quaisquer outros no sistema do usuário.

Adolescentes que devotavam suas vidas às complexidades do hardware do PC podiam fazê-lo, às vezes sem cometer erros. Infelizmente, nem todos tinham esse know-how, o que levava ao caos.

O plug and play faz com que o sistema colete automaticamente informações sobre dispositivos de E/S, atribua de maneira centralizada os níveis de interrupção e os endereços de E/S e informe cada placa sobre quais são seus números. Esse trabalho está estreitamente relacionado à inicialização do computador, por isso vamos examiná-lo. Não se trata de algo completamente trivial.

1.3.7 Inicializando o computador

Muito resumidamente, o processo de iniciação no Pentium funciona da seguinte maneira: todo Pentium contém uma placa geral, denominada placa-mãe. Nela localiza-se um programa denominado **BIOS** (*basic input output system* — sistema básico de E/S). O BIOS conta com rotinas de E/S de baixo nível, para ler o teclado, escrever na tela, realizar a E/S no disco etc. Atualmente, ele fica em uma flash RAM, que é não volátil, mas que pode ser atualizada pelo sistema operacional se erros forem encontrados no BIOS.

Quando o computador é inicializado, o BIOS começa a executar. Ele primeiro verifica quanta memória RAM está instalada e se o teclado e outros dispositivos básicos estão instalados e respondendo corretamente. Ele segue varrendo os barramentos ISA e PCI para detectar todos os dispositivos conectados a eles. Alguns desses dispositivos são, em geral, **legados** (legacy, isto é, projetados antes que o plug and play tivesse sido inventado) e têm níveis de interrupção e endereços de E/S fixos (possivelmente configurados por chaves ou jumpers na placa de E/S, mas não modificáveis pelo sistema operacional). Esses dispositivos são gravados, assim como os dispositivos plug and play. Se os dispositivos presentes se mostrarem diferentes de quando o sistema foi inicializado pela última vez, esses novos dispositivos serão configurados.

O BIOS determina então o dispositivo de inicialização (*boot*) percorrendo uma lista de dispositivos armazenados na memória CMOS. O usuário pode alterar essa lista entrando em um programa de configuração do BIOS logo depois da inicialização. Normalmente, uma tentativa é feita para inicializar a partir do disco flexível. Se isso falha, é tentado o CD-ROM. Se nem o disco flexível nem o CD-ROM estiverem presentes, o sistema será inicializado a partir do disco rígido. O primeiro setor do dispositivo de inicialização é transferido para a memória e executado. Esse setor contém um programa que, em geral, examina a tabela de partições no final do setor de inicialização para determinar qual partição está ativa. Então, um carregador de inicialização secundário é lido daquela partição. Esse carregador lê o sistema operacional da partição ativa e, então, o inicia.

O sistema operacional consulta o BIOS para obter a informação de configuração. Para cada dispositivo, ele verifica se há o driver do dispositivo. Se não houver, ele pedirá para que o usuário insira um disco flexível ou um CD-ROM contendo o driver (fornecido pelo fabricante do dispositivo). Uma vez que todos os drivers dos dispositivos estejam disponíveis, o sistema operacional carrega-os dentro do núcleo. Então ele inicializa suas tabelas, cria processos em background — se forem necessários — e inicia um programa de identificação (*login*) ou uma interface gráfica GUI.

1.4 O zoológico de sistemas operacionais

Os sistemas operacionais existem há mais de 50 anos. Durante esse tempo, uma grande variedade deles foi desenvolvida, nem todos bem conhecidos. Nesta seção falaremos resumidamente sobre nove deles. Mais adiante, voltaremos a abordar alguns desses diferentes tipos de sistemas.

1.4.1 Sistemas operacionais de computadores de grande porte

No topo estão os sistemas operacionais para computadores de grande porte — aqueles que ocupam uma sala inteira, ainda encontrados em centros de dados de grandes corporações. Esses computadores distinguem-se dos computadores pessoais em termos de capacidade de E/S. Um computador de grande porte com mil discos e milhares de gigabytes de dados não é incomum; um computador pessoal com essas especificações seria algo singular. Os computadores de grande porte também estão ressurgindo como sofisticados servidores da Web, como servidores para sites de comércio eletrônico em larga escala e, ainda, como servidores para transações entre empresas (*business-to-business*).

Os sistemas operacionais para computadores de grande porte são sobretudo orientados para o processamento simultâneo de muitas tarefas, e a maioria deles precisa de quantidades prodigiosas de E/S. Esses sistemas operacionais normalmente oferecem três tipos de serviços: em lote (*batch*), processamento de transações e tempo compartilhado. Um sistema em lote processa tarefas de rotina sem a presença interativa do usuário. O processamento de apólices de uma companhia de seguros ou de relatórios de vendas de uma cadeia de lojas é, em geral, realizado em modo lote. Sistemas de processamento de transações administram grandes quantidades de pequenas requisições — por exemplo, processamento de verificações em um banco ou em reservas de passagens aéreas. Cada unidade de trabalho é pequena, mas o sistema precisa tratar centenas ou milhares delas por segundo. Sistemas de tempo compartilhado permitem que múltiplos usuários remotos executem suas tarefas simultaneamente no computador, como na realização de consultas a um grande banco de dados. Essas funções estão intimamente relacionadas; sistemas operacionais de computadores de grande porte muitas vezes realizam todas elas. Um exemplo de sistema operacional de computador

de grande porte é o OS/390, um descendente do OS/360. Entretanto, os sistemas operacionais de computadores de grande porte estão sendo gradualmente substituídos por variantes do UNIX, como o Linux.

1.4.2 Sistemas operacionais de servidores

Um nível abaixo estão os sistemas operacionais de servidores. Eles são executados em servidores, que são computadores pessoais muito grandes, em estações de trabalho ou até mesmo em computadores de grande porte. Eles servem múltiplos usuários de uma vez em uma rede e permitem-lhes compartilhar recursos de hardware e de software. Servidores podem fornecer serviços de impressão, de arquivo ou de Web. Provedores de acesso à Internet utilizam várias máquinas servidoras para dar suporte a seus clientes e sites da Web usam servidores para armazenar páginas e tratar requisições que chegam. Sistemas operacionais típicos de servidores são Solaris, FreeBSD, Linux e Windows Server 200x.

1.4.3 Sistemas operacionais de multiprocessadores

Um modo cada vez mais comum de obter potência computacional é conectar múltiplas CPUs em um único sistema. Dependendo precisamente de como elas estiverem conectadas e o que é compartilhado, esses sistemas são denominados computadores paralelos, multicomputadores ou multiprocessadores. Elas precisam de sistemas operacionais especiais, mas muitos deles são variações dos sistemas operacionais de servidores, com aspectos especiais de comunicação, conectividade e compatibilidade.

Com o advento recente de chips multinúcleo para computadores pessoais, até sistemas operacionais de computadores de mesa e de notebooks estão começando a lidar com, no mínimo, multiprocessadores de pequena escala e é provável que o número de núcleos cresça com o tempo. Felizmente, sabe-se bastante sobre sistemas operacionais de multiprocessadores como consequência de anos de pesquisas anteriores; desse modo, aplicar esse conhecimento a sistemas multinúcleo não deve ser difícil. A parte difícil seria fazer com que as aplicações usassem todo esse poder de computação. Muitos sistemas operacionais populares, inclusive Windows e Linux, são executados com multiprocessadores.

1.4.4 Sistemas operacionais de computadores pessoais

A categoria seguinte é o sistema operacional de computadores pessoais. Os computadores modernos dão suporte a multiprogramação, muitas vezes com dezenas de programas iniciados. Seu trabalho é oferecer uma boa interface para um único usuário. São amplamente usados para processadores de texto, planilhas e acesso à Internet. Exemplos comuns são Linux, FreeBSD, Windows Vista e o sistema operacional do Macintosh. Sistemas operacionais de computadores pessoais são tão amplamente conhecidos que é provável que precisem, aqui, de pouca introdução. Na verdade, muitas pessoas nem mesmo sabem da existência de outros tipos de sistemas operacionais.

1.4.5 Sistemas operacionais de computadores portáteis

Seguindo em direção a sistemas cada vez menores, chegamos aos computadores portáteis. Um computador portátil ou **assistente pessoal digital** (*personal digital assistant* — PDA) é um pequeno computador que cabe no bolso de uma camisa e executa um número pequeno de funções, como agenda de endereços e bloco de anotações. Além disso, muitos telefones celulares apresentam pequenas diferenças em relação aos PDAs, exceto pelo teclado e pela tela. De fato, PDAs e telefones celulares basicamente se fundiram, diferindo principalmente em tamanho, peso e interface com o usuário. Quase todos eles são baseados em CPUs de 32 bits com modo protegido e executam um sistema operacional sofisticado.

Os sistemas operacionais executados nesses computadores portáteis são cada vez mais sofisticados, com a capacidade de manipular telefonia, fotografia digital e outras funções. Muitos deles também executam aplicações de terceiras partes. De fato, alguns deles estão começando a se parecer com sistemas operacionais de computadores pessoais de uma década atrás. Uma diferença importante entre portáteis e PCs é que os primeiros não têm discos rígidos multigigabyte, o que faz muita diferença. Dois dos sistemas operacionais para portáteis mais populares são Symbian OS e Palm OS.

1.4.6 Sistemas operacionais embarcados

Sistemas embarcados são executados em computadores que controlam dispositivos que geralmente não são considerados computadores e que não aceitam softwares instalados por usuários. Exemplos típicos são fornos de micro-ondas, aparelhos de TV, carros, aparelhos de DVD, telefones celulares e reprodutores de MP3. A propriedade principal que distingue os sistemas embarcados dos portáteis é a certeza de que nenhum software não confiável jamais será executado nele. Você não pode baixar novas aplicações para seu forno de micro-ondas — todo software está no ROM. Isso significa que não há necessidade de proteção entre as aplicações, levando a algumas simplificações. Sistemas como QNX e VxWorks são populares nesse domínio.

1.4.7 Sistemas operacionais de nós sensores (*sensor node*)

Redes de nós sensores minúsculos estão sendo empregadas com inúmeras finalidades. Esses nós são computadores minúsculos que se comunicam entre si e com uma

estação-base usando comunicação sem fio. Essas redes de sensores são utilizadas para proteger os perímetros de prédios, guardar fronteiras nacionais, detectar incêndios em florestas, medir temperatura e níveis de precipitação para previsão do tempo, colher informações sobre movimentos dos inimigos em campos de batalha e muito mais.

Os sensores são computadores pequenos movidos a bateria com rádios integrados. Eles têm energia limitada e devem funcionar por longos períodos de tempo, sozinhos ao ar livre, frequentemente em condições ambientais severas. A rede deve ser robusta o suficiente para tolerar falhas de nós individuais, o que acontece com frequência cada vez maior à medida que as baterias começam a se esgotar.

Cada nó sensor é um verdadeiro computador, com CPU, RAM, ROM e um ou mais sensores ambientais. Executa um pequeno sistema operacional próprio, normalmente dirigido por eventos, reagindo a eventos externos ou obtendo medidas periodicamente com base em um relógio interno. O sistema operacional tem de ser pequeno e simples porque os nós têm RAM pequena e a duração da bateria é uma questão importante. Além disso, tal como nos sistemas embarcados, todos os programas são carregados antecipadamente; os usuários não iniciam repentinamente os programas que baixaram da Internet, o que torna o projeto muito mais simples. O TinyOS é um sistema operacional muito conhecido para nós sensores.

1.4.8 Sistemas operacionais de tempo real

Outro tipo de sistema operacional é o de tempo real. Esses sistemas são caracterizados por terem o tempo como um parâmetro fundamental. Por exemplo, em sistemas de controle de processos industriais, computadores de tempo real devem coletar dados sobre o processo de produção e usá-los para controlar as máquinas na fábrica. É bastante comum a existência de prazos rígidos para a execução de determinadas tarefas. Por exemplo, se um carro está se movendo por uma linha de montagem, certas ações devem ser realizadas em momentos específicos. Se um robô soldador realizar seu trabalho — soldar — muito cedo ou muito tarde, o carro estará perdido. Se as ações *precisam* necessariamente ocorrer em determinados instantes (ou em um determinado intervalo de tempo), tem-se então um **sistema de tempo real crítico**. Muitos deles são encontrados no controle de processos industriais, aviônica, exército e áreas de aplicação semelhantes. Esses sistemas devem fornecer garantia absoluta de que determinada ação ocorrerá em determinado momento.

Outro tipo de sistema de tempo real é o **sistema de tempo real não crítico**, no qual o descumprimento ocasional de um prazo, embora não desejável, é aceitável e não causa nenhum dano permanente. Sistemas de áudio digital ou multimídia pertencem a essa categoria. Telefones digitais também são sistemas de tempo real não críticos.

Uma vez que cumprir prazos rigorosos é crucial em sistemas de tempo real, algumas vezes o sistema operacional é simplesmente uma biblioteca conectada com os programas aplicativos, em que tudo está rigorosamente acoplado e não há proteção entre as partes do sistema. Um exemplo desse tipo de sistema em tempo real é e-Cos.

As categorias de sistemas portáteis, embarcados e de tempo real se sobrepõem de modo considerável. Quase todas elas têm pelo menos alguns aspectos de tempo real. Os sistemas embarcado e de tempo real executam apenas softwares colocados pelos projetistas do sistema; os usuários não podem acrescentar seus próprios softwares, o que facilita a proteção. Os sistemas portáteis e embarcados são planejados para consumidores, ao passo que sistemas de tempo real são mais direcionados ao uso industrial. Entretanto, eles têm algumas coisas em comum.

1.4.9 Sistemas operacionais de cartões inteligentes (*smart cards*)

Os menores sistemas operacionais são executados em cartões inteligentes — dispositivos do tamanho de cartões de crédito que contêm um chip de CPU. Possuem grandes restrições de consumo de energia e de memória.Alguns deles obtêm energia por contatos com o leitor em que estão inseridos, porém, os cartões inteligentes sem contato obtêm energia por indução, o que limita muito aquilo que podem realizar. Alguns deles podem realizar apenas uma única função, como pagamentos eletrônicos, mas outros podem gerenciar múltiplas funções no mesmo cartão inteligente. São comumente sistemas proprietários.

Alguns cartões inteligentes são orientados a Java. Isso significa que a ROM no cartão inteligente contém um interpretador para a máquina virtual Java (*Java virtual machine* — JVM). As pequenas aplicações Java (applets) são carregadas no cartão e interpretadas pela JVM. Alguns desses cartões podem tratar múltiplas applets Java ao mesmo tempo, acarretando multiprogramação e a consequente necessidade de escalonamento. O gerenciamento de recursos e a proteção também são um problema quando duas ou mais applets estão presentes simultaneamente. Esses problemas devem ser tratados pelo sistema operacional (normalmente muito primitivo) contido no cartão.

1.5 Conceitos sobre sistemas operacionais

A maioria dos sistemas operacionais fornece certos conceitos e abstrações básicos, como processos, espaços de endereçamento e arquivos, que são fundamentais para entendê-los. Nas próximas seções, veremos alguns desses conceitos básicos de modo bastante breve, como uma introdução. Voltaremos a cada um deles detalhando-os neste livro. Para ilustrar esses conceitos, de vez em quando usaremos exemplos, geralmente tirados do UNIX. Contudo, exem-

plos similares normalmente existem para outros sistemas. Estudaremos o Windows Vista em detalhes no Capítulo 11.

1.5.1 | Processos

Um conceito fundamental para todos os sistemas operacionais é o de **processo**. Um processo é basicamente um programa em execução. Associado a cada processo está o seu **espaço de endereçamento**, uma lista de posições de memória, que vai de 0 até um máximo, que esse processo pode ler e escrever. O espaço de endereçamento contém o programa executável, os dados do programa e sua pilha. Também associado a cada processo está um conjunto de recursos, normalmente incluindo registradores (que incluem o contador de programa e o ponteiro para a pilha), uma lista dos arquivos abertos, alarmes pendentes, listas de processos relacionados e todas as demais informações necessárias para executar um programa. Um processo é fundamentalmente um contêiner que armazena todas as informações necessárias para executar um programa.

Retomaremos, no Capítulo 2, o conceito de processo com muito mais detalhes; por ora, o modo mais fácil de intuitivamente entender um processo é pensar em sistemas de multiprogramação. O usuário pode ter iniciado um programa de edição de vídeo e o instruído para converter um vídeo de uma hora para um determinado formato (algo que pode levar horas) e, a seguir, começar a navegar na Web. Enquanto isso, a execução de um processo de fundo subordinado que desperta periodicamente para verificar os e-mails que chegam pode ter sido iniciada. Desse modo, temos (pelo menos) três processos ativos: o editor de vídeo, o navegador da Web e o receptor de e-mail. Periodicamente, o sistema operacional decide parar de executar um processo e iniciar a execução de outro porque, por exemplo, o primeiro havia excedido seu tempo de compartilhamento da CPU.

Quando um processo é suspenso temporariamente dessa maneira, ele deverá ser reiniciado mais tarde, exatamente do mesmo ponto em que estava quando foi interrompido. Isso significa que todas as informações relativas ao processo devem estar explicitamente salvas em algum lugar durante a suspensão. Por exemplo, um processo pode ter, ao mesmo tempo, vários arquivos abertos para leitura. Existe um ponteiro, associado a cada um desses arquivos, que indica a posição atual (isto é, o número do próximo byte ou registro a ser lido). Quando um processo é suspenso temporariamente, todos esses ponteiros devem ser salvos de forma que uma chamada read, executada após o reinício desse processo, possa ler os dados corretamente. Em muitos sistemas operacionais, todas as informações relativas a um processo — que não sejam o conteúdo de seu próprio espaço de endereçamento — são armazenadas em uma tabela do sistema operacional denominada **tabela de processos**, que é um arranjo (ou uma lista encadeada) de estruturas, uma para cada processo existente.

Assim, um processo (suspenso) é constituído de seu espaço de endereçamento, normalmente chamado de **imagem do núcleo** (em homenagem às memórias de núcleo magnético usadas antigamente), e de sua entrada na tabela de processos, a qual armazena o conteúdo de seus registradores, entre outros itens necessários para reiniciar o processo mais tarde.

As principais chamadas de sistema de gerenciamento de processos são aquelas que lidam com a criação e o término de processos. Considere um exemplo típico: um processo denominado **interpretador de comandos** ou **shell** lê os comandos de um terminal. O usuário acaba de digitar um comando pedindo que um programa seja compilado. O shell deve então criar um novo processo, que executará o compilador. Assim que esse processo tiver terminado a compilação, ele executa uma chamada de sistema para se autofinalizar.

Se um processo pode criar um ou mais processos (chamados **processos filhos**), e se esses processos, por sua vez, puderem criar outros processos filhos, rapidamente chegaremos a uma estrutura de árvores como a da Figura 1.13. Processos relacionados que estiverem cooperando para executar alguma tarefa precisam frequentemente se comunicar um com o outro e sincronizar suas atividades. Essa comunicação é chamada de **comunicação entre processos** e será analisada no Capítulo 2.

Outras chamadas de sistema permitem requisitar mais memória (ou liberar memória sem uso), esperar que um processo filho termine e sobrepor (*overlay*) seu programa por outro diferente.

Ocasionalmente, há a necessidade de levar uma informação para um processo em execução que não esteja esperando por essa informação. Por exemplo, um processo que está se comunicando com outro processo em outro computador envia mensagens para o processo remoto por intermédio de uma rede de computadores. Para se prevenir contra a possibilidade de que uma mensagem ou sua resposta se perca, o processo emissor pode requisitar que seu próprio sistema operacional notifique-o após um determinado número de segundos, para que possa retransmitir a mensagem se nenhuma confirmação (*acknowledgement*) enviada pelo processo receptor tiver sido recebida. Depois de ligar esse temporizador, o programa pode ser retomado e realizar outra tarefa.

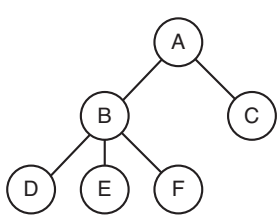

Figura 1.13 Uma árvore de processo. O processo *A* criou dois processos filhos, *B* e *C*. O processo *B* criou três processos filhos, *D*, *E* e *F*.

Decorrido o número especificado de segundos, o sistema operacional avisa o processo por meio de um **sinal de alarme**. Esse sinal faz com que o processo suspenda temporariamente o que estiver fazendo, salve seus registradores na pilha e inicie a execução de uma rotina especial para tratamento desse sinal — por exemplo, para retransmitir uma mensagem presumivelmente perdida. Quando a rotina de tratamento desse sinal termina, o processo em execução é reiniciado a partir do mesmo ponto em que estava logo antes de ocorrer o sinal. Sinais são os análogos em software das interrupções em hardware e podem ser gerados por diversas causas além da expiração de um temporizador. Muitas armadilhas detectadas por hardware — como tentar executar uma instrução ilegal ou usar um endereço inválido — também são convertidas em sinais para o processo causador.

A cada pessoa autorizada a usar um sistema é atribuída uma **UID** (*user identification* — identificação do usuário) pelo administrador do sistema. Todo processo iniciado tem a UID de quem o iniciou. Um processo filho tem a mesma UID de seu processo pai. Os usuários podem ser membros de grupos, cada qual com uma **GID** (*group identification* — identificação do grupo).

Uma UID, denominada **superusuário** (em UNIX), tem poderes especiais e pode violar muitas das regras de proteção. Em grandes instalações, somente o administrador do sistema sabe a senha necessária para se tornar um superusuário, mas muitos usuários comuns (especialmente estudantes) fazem de tudo para encontrar falhas no sistema que lhes permitam tornarem-se superusuários sem a senha.

Estudaremos processos, comunicações entre processos e assuntos relacionados no Capítulo 2.

1.5.2 Espaços de endereçamento

Todo computador tem alguma memória principal que utiliza para armazenar programas em execução. Em um sistema operacional muito simples, apenas um programa por vez está na memória. Para executar um segundo programa, o primeiro tem de ser removido e o segundo, colocado na memória.

Sistemas operacionais mais sofisticados permitem que múltiplos programas estejam na memória ao mesmo tempo. Para impedi-los de interferir na atividade uns dos outros (e com o sistema operacional), algum tipo de mecanismo de proteção é necessário. Embora esse mecanismo deva estar no hardware, é controlado pelo sistema operacional.

O ponto de vista apresentado anteriormente diz respeito ao gerenciamento e à proteção da memória principal do computador. Um tema diferente relacionado à memória, mas igualmente importante, é o gerenciamento do espaço de endereçamento dos processos. Normalmente, cada processo tem um conjunto de endereços que pode utilizar, geralmente indo de 0 até alguma quantidade máxima. No caso mais simples, a quantidade máxima de espaço de endereçamento que um processo tem é menor que a memória principal. Desse modo, um processo pode preencher todo seu espaço de endereçamento e haverá espaço suficiente na memória para armazená-lo completamente.

Contudo, em muitos computadores os endereços são de 32 ou 64 bits, dando um espaço de endereçamento de 2^{32} ou 2^{64} bytes, respectivamente. O que acontece se um processo tiver maior espaço de endereçamento que a memória principal do computador e o processo quiser utilizá-lo completamente? Nos primeiros computadores, esse processo não tinha sorte. Atualmente, existe uma técnica chamada memória virtual, como mencionado anteriormente, na qual o sistema operacional mantém parte do espaço de endereçamento na memória principal e parte no disco, trocando os pedaços entre eles conforme a necessidade. Em essência, o sistema operacional cria a abstração de um espaço de endereçamento como o conjunto de endereços ao qual um processo pode se referir. O processo de endereçamento é desacoplado da memória física da máquina e pode ser maior ou menor que a memória física. O gerenciamento de espaços de endereçamento e da memória física forma uma parte importante das atividades do sistema operacional; por isso, o Capítulo 3 é dedicado a esse tópico.

1.5.3 Arquivos

Outro conceito fundamental que compõe praticamente todos os sistemas operacionais é o sistema de arquivos. Como se observou anteriormente, uma das principais funções do sistema operacional é ocultar as peculiaridades dos discos e de outros dispositivos de E/S, fornecendo ao programador um modelo de arquivos agradável e claro, independentemente de dispositivos. Chamadas de sistema são obviamente necessárias para criar, remover, ler e escrever arquivos. Antes que possa ser lido, um arquivo deve ser localizado no disco, aberto e, depois de lido, ser fechado. Desse modo, chamadas de sistema são fornecidas para fazer essas tarefas.

Para ter um local para guardar os arquivos, a maioria dos sistemas operacionais fornece o conceito de **diretório** como um modo de agrupar arquivos. Um estudante, por exemplo, pode ter um diretório para cada curso que ele estiver fazendo (para os programas necessários àquele curso), outro diretório para seu correio eletrônico e ainda outro para sua página da Web. São necessárias chamadas de sistema para criar e remover diretórios. Também são fornecidas chamadas para colocar um arquivo em um diretório e removê-lo de lá. Entradas de diretório podem ser arquivos ou outros diretórios. Esse modelo dá origem também a outra hierarquia — o sistema de arquivos — conforme mostra a Figura 1.14.

Hierarquias de processos e de arquivos são organizadas como árvores, mas a semelhança acaba aí. Hierarquias de processos normalmente não são muito profundas (mais de três níveis não é comum); já hierarquias de arquivos compõem-se, em geral, de quatro, cinco ou mais níveis de profundidade. Hierarquias de processos costumam ter pou-

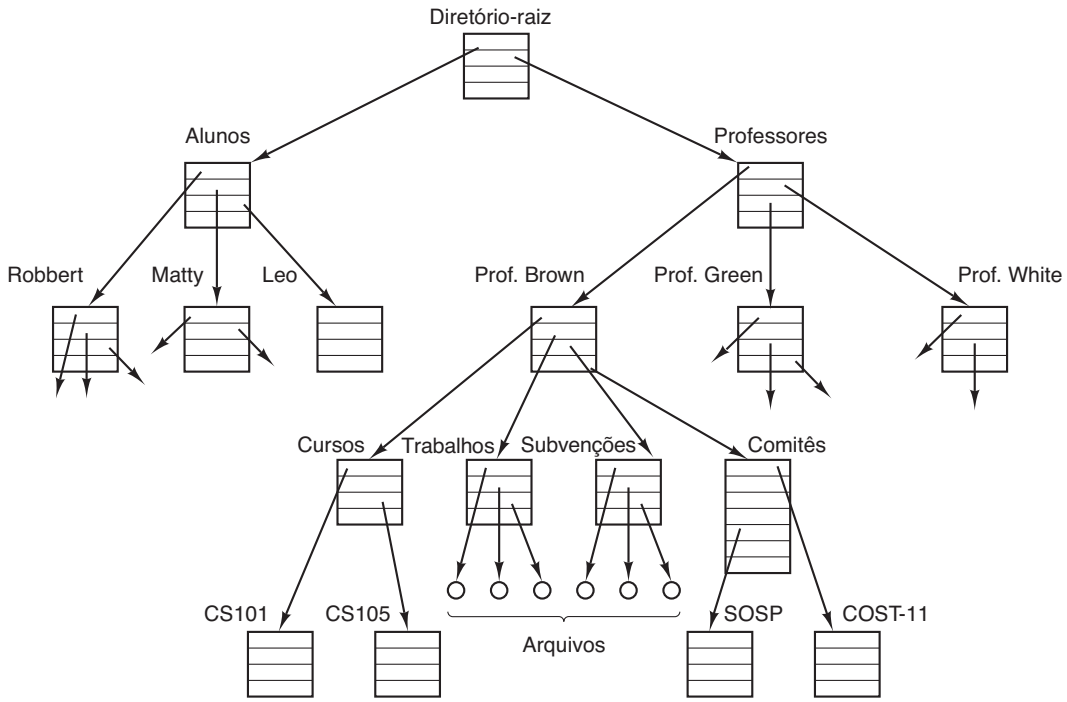

Figura 1.14 Sistema de arquivos para um departamento universitário.

co tempo de vida, no máximo alguns minutos, enquanto hierarquias de diretórios podem existir por anos. Propriedade e proteção também diferem entre processos e arquivos. Normalmente, apenas um processo pai pode controlar ou acessar um processo filho, mas quase sempre existem mecanismos para que arquivos e diretórios sejam lidos por um grupo mais amplo que apenas pelo proprietário.

Cada arquivo dentro da hierarquia de diretórios pode ser especificado fornecendo-se o **caminho** (*path name*) a partir do topo da hierarquia de diretórios, o **diretório-raiz**. Esses caminhos absolutos formam uma lista de diretórios que deve ser percorrida a partir do diretório-raiz para chegar até o arquivo, com barras separando os componentes. Na Figura 1.14, o caminho para o arquivo *CS101* é */Professores/Prof.Brown/Cursos/CS101*. A primeira barra indica que o caminho é absoluto, isto é, parte-se do diretório-raiz. Uma observação: no MS-DOS e no Windows, o caractere barra invertida (\) é usado como separador, em vez do caractere barra (/); assim, o caminho do arquivo dado acima seria escrito como *\Professores\Prof.Brown\Cursos\CS101*. Neste livro usaremos geralmente a convenção UNIX para caminhos.

A todo momento, cada processo tem um **diretório de trabalho** atual, no qual são buscados nomes de caminhos que não se iniciam com uma barra. Por exemplo, na Figura 1.14, se */Professores/Prof.Brown* for o diretório de trabalho, então o uso do caminho *Cursos/CS101* resultará no mesmo arquivo do caminho absoluto dado anteriormente. Os processos podem mudar seu diretório atual emitindo, para isso, uma chamada de sistema especificando o novo diretório de trabalho.

Antes que possa ser lido ou escrito, um arquivo precisa ser aberto e, nesse momento, as permissões são verificadas. Se o acesso for permitido, o sistema retorna um pequeno valor inteiro, chamado **descritor de arquivo**, para usá-lo em operações subsequentes. Se o acesso for proibido, um código de erro será retornado.

Outro conceito importante em UNIX é o de montagem do sistema de arquivos. Quase todos os computadores pessoais têm uma ou mais unidades de discos flexíveis nos quais CD-ROMs e DVDs podem ser inseridos e removidos. Eles quase sempre têm portas USB, nas quais dispositivos de memória USB (na verdade, unidades de disco de estado sólido) podem ser conectados, e alguns computadores possuem discos flexíveis ou discos rígidos externos. Para fornecer um modo melhor de tratar com meios removíveis, o UNIX permite que o sistema de arquivos em um CD-ROM ou DVD seja agregado à árvore principal. Considere a situação da Figura 1.15(a). Antes de uma chamada mount, o **sistema de arquivos-raiz** no disco rígido e um segundo sistema de arquivos em um CD-ROM estão separados e não estão relacionados.

Contudo, o sistema de arquivos em um CD-ROM não pode ser usado, pois não há um modo de especificar caminhos (*path names*) nele. O UNIX não permite que caminhos sejam prefixados por um nome ou número de dispositivo acionador. Esse seria exatamente o tipo de dependência ao dispositivo que os sistemas operacionais precisam eliminar. Em vez disso, a chamada de sistema mount permite que o sistema de arquivos em CD-ROM seja agregado ao sistema de arquivos-raiz sempre que seja solicitado pelo programa. Na Figura 1.15(b), o sistema de arquivos em CD-ROM deve

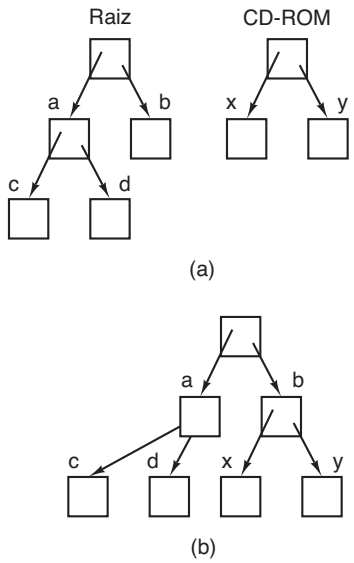

Figura 1.15 (a) Antes da montagem, os arquivos no CD-ROM não estão acessíveis. (b) Após a montagem, tornam-se parte da hierarquia de arquivos.

ser montado no diretório *b*, permitindo, com isso, o acesso aos arquivos */b/x* e */b/y*. Se o diretório *b* contivesse algum arquivo, esse arquivo não estaria acessível enquanto o CD-ROM estivesse montado, já que */b* se referiria ao diretório-raiz do CD-ROM. (A impossibilidade de acesso a esses arquivos não é um problema tão sério quanto parece: sistema de arquivos são quase sempre montados em diretórios vazios.) Se um sistema contiver múltiplos discos rígidos, eles poderão ser montados também em uma única árvore.

Outro conceito importante em UNIX é o de **arquivo especial**. Os arquivos especiais permitem que dispositivos de E/S pareçam-se com arquivos. Desse modo, eles podem ser lidos e escritos com as mesmas chamadas de sistema usadas para ler e escrever arquivos. Existem dois tipos de arquivos especiais: **arquivos especiais de bloco** e **arquivos especiais de caracteres**. Os arquivos especiais de bloco são usados para modelar dispositivos que formam uma coleção de blocos aleatoriamente endereçáveis, como discos. Abrindo um arquivo especial de blocos e lendo o bloco 4, um programa pode ter acesso direto ao quarto bloco do dispositivo, sem se preocupar com a estrutura do sistema de arquivos contido nele. Da mesma maneira, os arquivos especiais de caracteres são usados para modelar impressoras, modems e outros dispositivos que recebem ou enviam caracteres serialmente. Por convenção, os arquivos especiais são mantidos no diretório */dev*. Por exemplo, */dev/lp* pode ser uma impressora (antes chamada de impressora de linha).

O último aspecto que discutiremos aqui é o que relaciona processos e arquivos: os pipes. Um **pipe** é um tipo de pseudoarquivo que pode ser usado para conectar dois processos, conforme ilustra a Figura 1.16. Se os processos *A* e *B* quiserem se comunicar usando um pipe, eles deverão

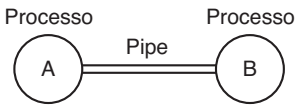

Figura 1.16 Dois processos conectados por um pipe.

ser configurados antecipadamente. Se um processo *A* pretende enviar dados para o processo *B*, o processo *A* escreve no pipe como se ele fosse um arquivo de saída. De fato, a implementação de um pipe é muito semelhante à de um arquivo. O processo *B* pode ler os dados lendo-os do pipe como se esse fosse um arquivo de entrada. Assim, a comunicação entre os processos em UNIX assemelha-se muito com leituras e escritas de arquivos comuns. A única maneira de um processo 'ficar sabendo' se o arquivo de saída em que está escrevendo não é realmente um arquivo, mas na verdade um pipe, é fazendo uma chamada de sistema especial. Sistemas de arquivos são muito importantes. Teremos muito mais a dizer sobre eles nos capítulos 4, 10 e 11.

1.5.4 Entrada e saída

Todos os computadores têm dispositivos físicos para entrada e saída. Afinal, para que serviria um computador se os usuários não pudessem dizer o que deve ser feito e não conseguissem verificar os resultados depois do trabalho solicitado? Existem vários tipos de dispositivos de entrada e saída, como teclados, monitores e impressoras. Cabe ao sistema operacional gerenciar esses dispositivos.

Consequentemente, todo sistema operacional possui um subsistema de E/S para gerenciar seus dispositivos de E/S. Alguns dos programas de E/S são independentes de dispositivo, isto é, aplicam-se igualmente bem a muitos ou a todos os dispositivos. Outras partes dele, como os drivers de dispositivo, são específicas a cada dispositivo de E/S. No Capítulo 5 estudaremos a programação de E/S.

1.5.5 Segurança

Computadores contêm muitas informações que os usuários, muitas vezes, querem manter confidenciais. Essas informações podem ser mensagens de correio eletrônico, planos de negócios, impostos devidos etc. Cabe ao sistema operacional gerenciar o sistema de segurança para que os arquivos, por exemplo, sejam acessíveis apenas por usuários autorizados.

Como um exemplo simples, apenas para termos uma ideia de como a segurança pode funcionar, considere o UNIX. Arquivos em UNIX são protegidos atribuindo-se a cada um deles um código de proteção de 9 bits. O código de proteção consiste em campos de 3 bits, um para o proprietário, um para outros membros do grupo do proprietário (os usuários são divididos em grupos pelo administrador do sistema) e outro para qualquer usuário. Cada campo tem

um bit de permissão de leitura, um bit de permissão de escrita e outro bit de permissão de execução. Esses 3 bits são conhecidos como **bits rwx**. Por exemplo, o código de proteção *rwxr-x--x* significa que o proprietário pode ler (***r**ead*), escrever (***w**rite*) ou executar (*e**x**ecute*) o arquivo, que outros membros do grupo podem ler ou executar (mas não escrever) o arquivo, e qualquer um pode executar (mas não ler ou escrever) o arquivo. Para um diretório, *x* indica a permissão de busca. Um traço significa ausência de permissão.

Além da proteção ao arquivo, há muitos outros tópicos sobre segurança. Proteger o sistema contra intrusos indesejáveis, humanos ou não (por exemplo, vírus), é um deles. Estudaremos vários desses assuntos de segurança no Capítulo 9.

1.5.6 O interpretador de comandos (shell)

O sistema operacional é o código que executa as chamadas de sistema. Editores, compiladores, montadores, ligadores (*linkers*) e interpretadores de comandos não fazem, com certeza, parte do sistema operacional, mesmo sendo importantes e úteis. Correndo o risco de confundir um pouco as coisas, nesta seção estudaremos resumidamente o interpretador de comandos do UNIX, chamado **shell**. Embora não seja parte do sistema operacional, o shell faz uso intensivo de muitos aspectos do sistema operacional e serve, assim, como um bom exemplo sobre como as chamadas de sistema podem ser usadas. Ele é também a interface principal entre o usuário à frente de seu terminal e o sistema operacional, a menos que o usuário esteja usando uma interface gráfica de usuário. Existem muitos shells, dentre eles o *sh*, o *csh*, o *ksh* e o *bash*. Todos eles dão suporte à funcionalidade descrita a seguir, que deriva do shell original (*sh*).

Quando um usuário se conecta, um shell é iniciado. Este tem o terminal como entrada-padrão e saída-padrão. Ele inicia emitindo um caractere de **prompt** (*prontidão*) — por exemplo, o cifrão —, que diz ao usuário que o shell está esperando receber um comando. Se o usuário então digitar

```
date
```

por exemplo, o shell cria um processo filho e executa o programa *date* utilizando a estrutura de dados desse processo filho. Enquanto o processo filho estiver em execução, o shell permanece aguardando-o terminar. Quando o processo filho é finalizado, o shell emite o sinal de *prompt* novamente e tenta ler a próxima linha de entrada.

O usuário pode especificar que a saída-padrão seja redirecionada para um arquivo, por exemplo,

```
date >arq
```

Do mesmo modo, a entrada-padrão pode ser redirecionada, como em

```
sort <arq1>arq2
```

que invoca o programa *sort* com a entrada vindo de *arq1* e a saída enviada para *arq2*.

A saída de um programa pode ser usada como a entrada para outro programa conectando-os por meio de um pipe. Assim,

```
cat arq1 arq2 arq3 | sort >/dev/lp
```

invoca o programa *cat* para concatenar três arquivos e enviar a saída para que o *sort* organize todas as linhas em ordem alfabética. A saída de *sort* é redirecionada ao arquivo */dev/lp*, que normalmente é a impressora.

Se um usuário colocar o caractere & após um comando, o shell não vai esperar que ele termine e, assim, envia imediatamente o caractere de prompt. Consequentemente,

```
cat arq1 arq2 arq3 | sort >/dev/lp &
```

inicia o *sort* como uma tarefa em background, permitindo que o usuário continue trabalhando normalmente enquanto a ordenação prossegue. O shell tem vários outros aspectos interessantes, que não temos espaço para discutir aqui. A maioria dos livros sobre UNIX aborda detidamente o shell (por exemplo, Kernighan e Pike, 1984; Kochan e Wood, 1990; Medinets, 1999; Newham e Rosenblatt, 1998; Robbins, 1999).

Atualmente, muitos computadores pessoais usam uma interface gráfica GUI. De fato, a interface GUI é apenas um programa sendo executado na camada superior do sistema operacional, como um shell. Nos sistemas Linux, esse fato se torna óbvio porque o usuário tem uma escolha de (pelo menos) duas interfaces GUIs: Gnome e KDE ou nenhuma (usando uma janela do terminal no X11). No Windows, também é possível substituir a área de trabalho com interface GUI padrão (Windows Explorer) com um programa diferente alterando alguns valores no registro, embora poucas pessoas o façam.

1.5.7 Ontogenia recapitula a filogenia

Depois que o livro *A origem das espécies*, de Charles Darwin, foi publicado, o zoólogo alemão Ernst Haeckel afirmou que a "ontogenia recapitula a filogenia". Com isso ele queria dizer que o desenvolvimento de um embrião (ontogenia) repete (isto é, relembra) a evolução das espécies (filogenia). Em outras palavras, depois da fertilização, um embrião humano passa por estágios de um peixe, de um leitão e assim por diante, até se tornar um bebê humano. Biólogos modernos consideram essa afirmação uma simplificação grosseira, mas no fundo há ainda alguma verdade nela.

Algo ligeiramente análogo tem acontecido na indústria da computação. Cada nova espécie (computador de grande porte, minicomputador, computador pessoal, computador embarcado, cartões inteligentes etc.) parece passar pelo mesmo desenvolvimento de seus ancestrais, tanto no que se refere ao hardware como ao software. Esquecemo-nos frequentemente de que muito do que acontece na indústria da computação e em muitos outros campos é orientado pela tecnologia. A razão pela qual os romanos não tinham

carros não é porque eles gostavam muito de caminhar. É porque eles não sabiam como construí-los. Computadores pessoais *não* existem porque milhões de pessoas têm um desejo contido por muitos séculos de ter um computador, mas porque agora é possível fabricá-los de modo mais barato. Muitas vezes nos esquecemos de como a tecnologia afeta nossa visão dos sistemas e de que convém refletir sobre o assunto de vez em quando.

Em particular, frequentemente uma alteração tecnológica torna alguma ideia obsoleta e ela desaparece rapidamente. Entretanto, outra mudança tecnológica poderia reavivá-la. Isso é especialmente verdadeiro quando a alteração tem a ver com o desempenho relativo de partes diferentes do sistema. Por exemplo, quando as CPUs se tornam muito mais velozes que as memórias, as caches se tornam importantes para acelerar a memória 'lenta'. Se a nova tecnologia de memória algum dia tornar as memórias muito mais velozes que a CPU, as caches desaparecerão. E, se uma nova tecnologia de CPU torná-las mais velozes que as memórias novamente, as caches reaparecerão. Em biologia, a extinção é definitiva, mas em ciência da computação ela às vezes ocorre por apenas alguns anos.

Como consequência dessa impermanência, neste livro examinaremos conceitos 'obsoletos' de vez em quando, isto é, ideias que não são as mais adequadas à tecnologia atual. Entretanto, mudanças tecnológicas podem trazer de volta alguns dos chamados 'conceitos obsoletos'. Por isso, é importante compreender por que um conceito é obsoleto e quais mudanças no ambiente podem trazê-lo de volta.

Para esclarecer esse ponto, consideremos um exemplo simples. Os primeiros computadores tinham conjuntos de instruções implementadas no hardware. As instruções eram executadas diretamente pelo hardware e não podiam ser alteradas. Mais tarde veio a microprogramação (introduzida pela primeira vez em grande escala com o IBM 360), na qual um interpretador subjacente executava as 'instruções do hardware' no software. A execução física se tornou obsoleta. Não era suficientemente flexível. Em seguida, os computadores RISC foram inventados e a microprogramação (isto é, execução interpretada) se tornou obsoleta porque a execução direta era mais veloz. Agora estamos vendo o ressurgimento da reinterpretação na forma de applets Java que são enviados pela Internet e interpretados após a chegada. A velocidade de execução nem sempre é crucial, porque retardos na rede são tão grandes que eles tendem a predominar. Dessa forma, o pêndulo já oscilou entre diferentes ciclos de execução direta e interpretação e pode oscilar novamente no futuro.

Memórias grandes

Examinemos agora alguns desenvolvimentos históricos de hardware e o modo como afetaram os softwares repetidamente. Os primeiros computadores de grande porte tinham memória limitada. Um IBM 7090 ou 7094 completamente carregados, que tinham supremacia do final de 1959 até 1964, tinham apenas 128 KB de memória. Eles eram, em sua maior parte, programados em linguagem assembly e seus sistemas operacionais eram escritos em linguagem assembly para economizar memória preciosa.

Com o passar do tempo, compiladores para linguagens como Fortran e Cobol se desenvolveram o suficiente para que a linguagem assembly fosse considerada morta. Mas quando o primeiro minicomputador comercial (o PDP-1) foi lançado, tinha apenas 4.096 palavras de memória de 18 bits e a linguagem assembly teve uma recuperação surpreendente. No fim, os microcomputadores adquiriram mais memória e as linguagens de alto nível se tornaram predominantes.

Quando os microcomputadores se tornaram um sucesso, no início da década de 1980, os primeiros tinham memórias de 4 KB e a programação assembly ressurgiu dos mortos. Computadores embarcados muitas vezes usavam os mesmos chips de CPU que os microcomputadores (8080s, Z80s e, mais tarde, 8086s) e também eram inicialmente programados em linguagem assembly. Agora seus descendentes, os computadores pessoais, têm muita memória e são programados em C, C++, Java e outras linguagens de alto nível. Os cartões inteligentes estão passando por desenvolvimentos semelhantes, embora sempre tenham, além de certas extensões, um interpretador Java e executem programas Java de modo interpretativo, em vez de compilarem o Java para a linguagem de máquina do cartão inteligente.

Hardware de proteção

Os primeiros computadores de grande porte, como o IBM 7090/7094, não possuíam hardware de proteção, por isso executavam apenas um programa por vez. Um programa defeituoso poderia apagar o sistema operacional e quebrar a máquina com facilidade. Com a introdução do IBM 360, uma forma primitiva de hardware de proteção foi disponibilizada e essas máquinas puderam armazenar vários programas na memória simultaneamente e permitir que estas se alternassem na execução (multiprogramação). A monoprogramação foi declarada obsoleta.

Pelo menos até o surgimento do primeiro minicomputador — sem hardware de proteção — a multiprogramação não era possível. O PDP-1 e o PDP-8 não tinham nenhum hardware de proteção, mas o PDP-11 possuía, e essa característica levou à multiprogramação e, no fim, ao UNIX.

Quando os primeiros microcomputadores foram construídos, usavam o chip de CPU Intel 8080, que não tinha nenhuma proteção de hardware; assim, voltamos à monoprogramação. Foi somente com o surgimento do Intel 80286 que essa proteção de hardware foi acrescentada e a multiprogramação se tornou possível. Até hoje, muitos sistemas embarcados não têm hardware de proteção e executam apenas um programa.

Agora observemos os sistemas operacionais. Os primeiros computadores de grande porte não possuíam hardware

de proteção nem davam suporte a multiprogramação. Desse modo, neles eram executados sistemas operacionais simples que tratavam apenas um programa por vez, carregado manualmente. Mais tarde eles adquiriram o suporte de hardware de proteção e do sistema operacional para lidar com vários programas de uma vez e, posteriormente, com a completa capacidade de suporte ao uso com compartilhamento de tempo.

Quando os minicomputadores surgiram, também não tinham proteção de hardware, e os programas eram executados um a um, carregados manualmente, embora a multiprogramação já estivesse bem estabelecida no mundo dos computadores de grande porte. Aos poucos, adquiriram a proteção de hardware e a capacidade de executar dois ou mais programas simultaneamente. Os primeiros microcomputadores eram capazes, ainda, de executar somente um programa por vez, mas depois passaram a contar com a capacidade de multiprogramação. Os computadores portáteis e os cartões inteligentes seguiram pelo mesmo caminho.

Em todos os casos, o desenvolvimento do software foi ditado pela tecnologia. Os primeiros microcomputadores, por exemplo, tinham algo como somente 4 KB de memória e nenhum hardware de proteção. Linguagens de alto nível e multiprogramação eram simplesmente grandes demais para serem tratadas em sistemas tão pequenos. À medida que os microcomputadores evoluíram, tornando-se modernos computadores pessoais, adquiriram o hardware e o software necessários para tratar aspectos mais avançados. Provavelmente esse desenvolvimento continuará. Outros campos também parecem ter esse ciclo de reencarnação evolutiva, mas, na indústria dos computadores, ele parece girar mais rápido.

Discos

Os primeiros computadores de grande porte eram em grande medida baseados em fita magnética. Eles costumavam ler um programa a partir da fita, compilá-lo, executá-lo e escrever os resultados de volta em outra fita. Não havia discos e nenhum conceito de um sistema de arquivos. Isso começou a mudar quando a IBM introduziu o primeiro disco rígido — o RAMAC (*RAndoM ACcess*, acesso aleatório) em 1956. Ele ocupava cerca de 4 metros quadrados de espaço e podia armazenar cinco milhões de caracteres de 7 bits, o suficiente para uma foto digital de resolução média. Mas com o valor do aluguel anual de $35.000, montar a quantidade suficiente deles para armazenar o equivalente a um rolo de filme muito rapidamente tornou-se caro. Mas, no fim, os preços caíram e sistemas de arquivos primitivos foram desenvolvidos.

O CDC 6600 foi um desses desenvolvimentos típicos, introduzido em 1964 e, durante muitos anos, o computador mais rápido do mundo. Os usuários podiam criar os chamados 'arquivos permanentes' dando-lhes nomes e esperando que nenhum outro usuário também tivesse decidido que, por exemplo, 'dados' fosse um nome adequado para um arquivo. Tratava-se de um diretório de um nível. No fim, os computadores de grande porte desenvolveram sistemas de arquivos hierárquicos complexos, que por acaso culminaram no sistema de arquivos MULTICS.

Quando os minicomputadores começaram a ser usados, eles também tinham discos rígidos. O disco-padrão no PDP-11, quando foi introduzido em 1970, era o disco RK05, com uma capacidade de 2,5 MB, cerca de metade do RAMAC IBM, mas tinha apenas cerca de 40 cm de diâmetro e 5 cm de altura. Mas ele também tinha um diretório de um nível inicialmente. Quando os microcomputadores foram lançados, o CP/M foi, a princípio, o sistema operacional predominante e também dava suporte a apenas um diretório no disco (flexível).

Memória virtual

A memória virtual (discutida no Capítulo 3) confere a capacidade de executar programas maiores que a memória física da máquina movendo peças entre a memória RAM e o disco. Ela passou por um desenvolvimento semelhante, aparecendo primeiro em computadores de grande porte, depois em mini e microcomputadores. A memória virtual também ativou a capacidade de conectar de modo dinâmico um programa a uma biblioteca no momento de execução em vez de compilá-lo. O MULTICS foi o primeiro sistema a permitir isso. No fim, a ideia se propagou ao longo da linha e agora é amplamente usada na maioria dos sistemas UNIX e Windows.

Em todos esses desenvolvimentos, vemos ideias que são inventadas em um contexto, são descartadas mais tarde quando o contexto muda (programação de linguagem assembly, monoprogramação, diretórios de um nível etc.) e reaparecem em um contexto diferente, muitas vezes uma década depois. Por essa razão, neste livro algumas vezes examinaremos ideias e algoritmos que podem parecer obsoletos nos PCs de gigabytes de hoje, mas que podem retornar em computadores embarcados e cartões inteligentes.

1.6 Chamadas de sistema (system calls)

Vimos que os sistemas operacionais têm duas funções principais: fornecer abstrações aos programas de usuários e administrar os recursos do computador. Em sua maior parte, a interação entre programas de usuário e o sistema operacional lida com a primeira; por exemplo, criar, escrever, ler e excluir arquivos. A parte de gerenciamento de recursos é, em grande medida, transparente para os usuários e feita automaticamente. Desse modo, a interface entre o sistema operacional e os programas de usuários trata primeiramente sobre como lidar com as abstrações. Para entender o que os sistemas operacionais fazem realmente, devemos observar essa interface mais de perto. As chamadas de sistema disponíveis na interface variam de um sistema operacional para outro (embora os conceitos básicos tendam a ser parecidos).

Somos, assim, forçados a escolher entre (1) generalidades vagas ("os sistemas operacionais possuem chamadas de sistema para leitura de arquivos") e (2) algum sistema específico ("UNIX possui uma chamada de sistema read com três parâmetros: um para especificar o arquivo, um para informar onde os dados deverão ser colocados e um para indicar quantos bytes deverão ser lidos").

Escolhemos a última abordagem. É mais trabalhosa, mas permite entender melhor o que os sistemas operacionais realmente fazem. Embora essa discussão refira-se especificamente ao POSIX (Padrão Internacional 9945-1) — consequentemente também ao UNIX, System V, BSD, Linux, MINIX 3 etc. —, a maioria dos outros sistemas operacionais modernos tem chamadas de sistema que desempenham as mesmas funções, diferindo apenas nos detalhes. Como os mecanismos reais para emissão de uma chamada de sistema são altamente dependentes da máquina e muitas vezes devem ser expressos em código assembly, é disponibilizada uma biblioteca de rotinas para tornar possível realizar chamadas de sistema a partir de programas em C e também, muitas vezes, a partir de programas em outras linguagens.

Convém ter em mente o seguinte: qualquer computador com uma única CPU pode executar somente uma instrução por vez. Se um processo estiver executando um programa de usuário em modo usuário e precisar de um serviço do sistema, como ler dados de um arquivo, terá de executar uma instrução TRAP para transferir o controle ao sistema operacional. Este verifica os parâmetros para, então, descobrir o que quer o processo que está chamando. Em seguida, ele executa uma chamada de sistema e retorna o controle para a instrução seguinte à chamada. Em certo sentido, fazer uma chamada de sistema é como realizar um tipo especial de chamada de rotina, só que as chamadas de sistema fazem entrar em modo núcleo e as chamadas de rotina, não.

Para esclarecer melhor o mecanismo de chamada de sistema, vamos dar uma rápida olhada na chamada de sistema read. Conforme já mencionado, ela tem três parâmetros: o primeiro especifica o arquivo, o segundo é um ponteiro para o buffer e o terceiro dá o número de bytes que deverão ser lidos. Como em quase todas as chamadas de sistema, ela é chamada a partir de programas em C chamando uma rotina de biblioteca com o mesmo nome da chamada de sistema: *read*. Uma chamada a partir de um programa em C pode ter o seguinte formato:

```
contador = read(arq, buffer, nbytes);
```

A chamada de sistema (assim como a rotina de biblioteca) retorna o número de bytes realmente lidos em *contador*. Esse valor é normalmente o mesmo de *nbytes*, mas pode ser menor se, por exemplo, o caractere fim-de-arquivo for encontrado durante a leitura.

Se a chamada de sistema não puder ser realizada, tanto por causa de um parâmetro inválido como por um erro de disco, o *contador* passará a valer –1 e o número do erro será colocado em uma variável global, *errno*. Os programas devem verificar sempre os resultados de uma chamada de sistema para saber se ocorreu um erro.

As chamadas de sistema são realizadas em uma série de passos. Para melhor esclarecer esse conceito, examinemos a chamada read discutida anteriormente. Antes da chamada da rotina *read* de biblioteca, que é na verdade quem faz a chamada de sistema read, o programa que chama read, antes de tudo, armazena os parâmetros na pilha, conforme mostram os passos 1 a 3 na Figura 1.17.

Compiladores C e C++ armazenam os parâmetros na pilha em ordem inversa por razões históricas (isso tem de ver com fazer o primeiro parâmetro de *printf*, a cadeia de caracteres do formato, aparecer no topo da pilha). O primeiro e o terceiro parâmetros são chamados por valor, mas o segundo parâmetro é por referência — isso quer dizer que é passado o endereço do buffer (indicado por &), e não seu conteúdo. Daí vem a chamada real à rotina de biblioteca (passo 4). Essa instrução é a chamada normal de rotina, usada para chamar todas as rotinas.

A rotina de biblioteca, possivelmente escrita em linguagem assembly, em geral coloca o número da chamada de sistema em um local esperado pelo sistema operacional — por exemplo, em um registrador (passo 5). Então, ele executa uma instrução TRAP para passar do modo usuário para o modo núcleo e iniciar a execução em um determinado endereço dentro do núcleo (passo 6). A instrução TRAP é, na verdade, bastante semelhante à chamada de rotina no sentido de que a instrução seguinte é recebida de um local distante e o endereço de retorno é salvo na pilha para uso posterior.

Entretanto, a instrução TRAP também difere da instrução call (chamada de rotina) de dois modos fundamentais. Primeiro, como efeito colateral, ela desvia para o modo núcleo. A instrução de chamada de rotina não altera o modo. Segundo, em vez de fornecer um endereço absoluto ou relativo onde o procedimento esteja localizado, a instrução TRAP não pode transferir para um endereço arbitrário. Dependendo da arquitetura, ela transfere para um local fixo — há um campo de 8 bits na instrução, fornecendo o índice em uma tabela na memória que contém endereços de transferência — ou para o equivalente a isso.

O código do núcleo que se inicia após a instrução TRAP verifica o número da chamada de sistema e, então, o despacha para a rotina correta de tratamento dessa chamada, geralmente por meio de uma tabela de ponteiros que indicam as rotinas de tratamento de chamadas de sistema indexadas pelo número da chamada (passo 7). Nesse ponto, é executada a rotina de tratamento da chamada de sistema (passo 8). Uma vez que a rotina de tratamento tenha terminado seu trabalho, o controle pode retornar para a rotina de biblioteca no espaço do usuário, na instrução seguinte à instrução TRAP (passo 9). Normalmente essa rotina retorna ao programa do usuário da mesma maneira que fazem as chamadas de rotina (passo 10).

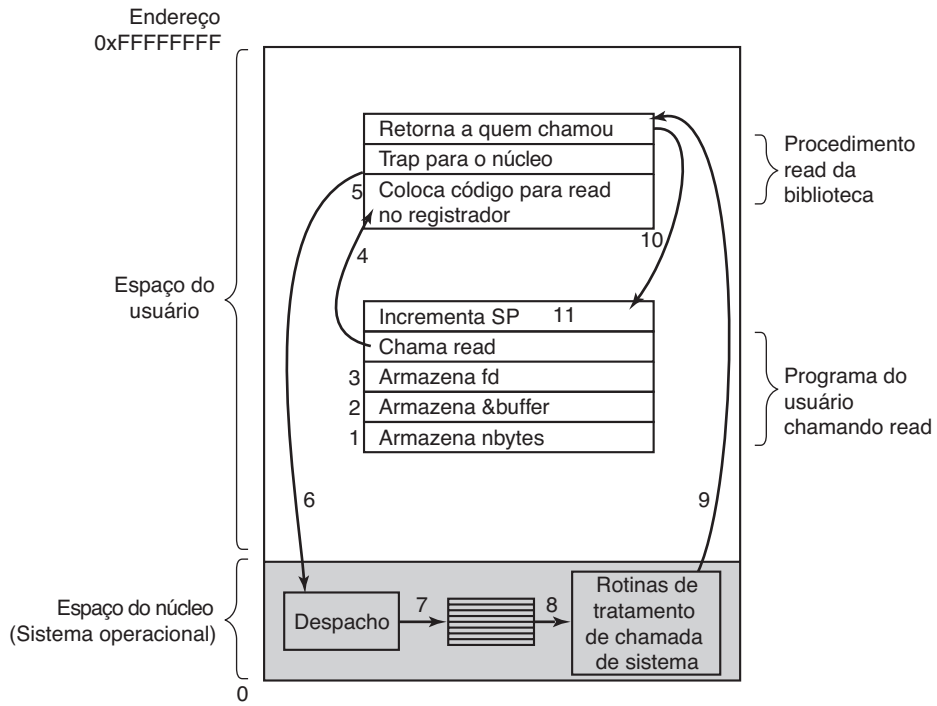

Figura 1.17 Os 11 passos na realização da chamada de sistema read (fd, buffer, nbytes).

Para finalizar a tarefa, o programa do usuário deve limpar a pilha, como se faz depois de qualquer chamada de rotina (passo 11). Presumindo que a pilha cresce para baixo, como muitas vezes ocorre, o código compilado incrementa o ponteiro da pilha exatamente o necessário para remover os parâmetros armazenados antes da chamada *read*. O programa agora está liberado para fazer o que quiser.

No passo 9, foi por uma boa razão que dissemos "pode retornar para a rotina da biblioteca no espaço do usuário": a chamada de sistema pode bloquear quem a chamou, impedindo-o de continuar. Por exemplo, se quem chamou estiver tentando ler do teclado e nada foi digitado ainda, ele será bloqueado. Nesse caso, o sistema operacional verificará se algum outro processo pode ser executado. Depois, quando a entrada desejada estiver disponível, esse processo terá a atenção do sistema e os passos 9 a 11 serão executados.

Nas próximas seções estudaremos algumas das chamadas de sistema em POSIX mais usadas ou, mais especificamente, as rotinas de biblioteca que realizam essas chamadas de sistema. O POSIX tem cerca de cem chamadas de rotina. Algumas das mais importantes estão listadas na Tabela 1.1, agrupadas, por conveniência, em quatro categorias. No texto examinaremos resumidamente cada chamada para entender o que cada uma delas faz.

Os serviços oferecidos por essas chamadas determinam a maior parte do que o sistema operacional deve realizar, já que o gerenciamento de recursos em computadores pessoais é mínimo (pelo menos, se comparado a grandes máquinas com vários usuários). Os serviços englobam coisas como criar e finalizar processos, criar, excluir, ler e escrever arquivos, gerenciar diretórios e realizar entrada e saída.

Convém observar que em POSIX o mapeamento de chamadas de rotina em chamadas de sistema não é de uma para uma. O POSIX especifica várias rotinas que um sistema em conformidade com esse padrão deve disponibilizar, mas não especifica se se trata de chamadas de sistema, chamadas de biblioteca ou qualquer outra coisa. Se uma rotina pode ser executada sem invocar uma chamada de sistema (isto é, sem desviar para o núcleo), ele é executado geralmente em modo usuário, por razões de desempenho. Contudo, o que a maioria das rotinas POSIX invoca são chamadas de sistema, em geral com uma rotina mapeando diretamente uma chamada de sistema. Em poucos casos — em especial naqueles em que diversas rotinas são somente pequenas variações umas das outras —, uma chamada de sistema é invocada por mais de uma chamada de biblioteca.

1.6.1 Chamadas de sistema para gerenciamento de processos

O primeiro grupo de chamadas na Tabela 1.1 lida com gerenciamento de processos. A chamada fork é um bom ponto de partida para a discussão, sendo o único modo de criar um novo processo em UNIX. Ela gera uma cópia exata do processo original, incluindo todos os descritores de arquivo, registradores — tudo. Depois de a chamada fork acontecer, o processo original e sua cópia (o processo pai e o processo filho) seguem caminhos separados. Todas as

Gerenciamento de processos

Chamada	Descrição
pid = fork( )	Cria um processo filho idêntico ao pai
pid = waitpid(pid, &statloc, options)	Espera que um processo filho seja concluído
s = execve(name, argv, environp)	Substitui a imagem do núcleo de um processo
exit(status)	Conclui a execução do processo e devolve status

Gerenciamento de arquivos

Chamada	Descrição
Fd = open(file, how, ...)	Abre um arquivo para leitura, escrita ou ambos
s = close(fd)	Fecha um arquivo aberto
n = read(fd, buffer, nbytes)	Lê dados a partir de um arquivo em um buffer
n = write(fd, buffer, nbytes)	Escreve dados a partir de um buffer em um arquivo
position = lseek(fd, offset, whence)	Move o ponteiro do arquivo
s = stat(name, &buf)	Obtém informações sobre um arquivo

Gerenciamento do sistema de diretório e arquivo

Chamada	Descrição
s = mkdir(name, mode)	Cria um novo diretório
s = rmdir(name)	Remove um diretório vazio
s = link(name1, name2)	Cria uma nova entrada, name2, apontando para name1
s = unlink(name)	Remove uma entrada de diretório
s = mount(special, name, flag)	Monta um sistema de arquivos
s = umount(special)	Desmonta um sistema de arquivos

Diversas

Chamada	Descrição
s = chdir(dirname)	Altera o diretório de trabalho
s = chmod(name, mode)	Altera os bits de proteção de um arquivo
s = kill(pid, signal)	Envia um sinal para um processo
seconds = time(&seconds)	Obtém o tempo decorrido desde 1º de janeiro de 1970

Tabela 1.1 Algumas das principais chamadas de sistema do POSIX. O código de retorno *s* é –1 se um erro tiver ocorrido. Os códigos de retorno são os seguintes: *pid* é um processo id, *fd* é um descritor de arquivo, *n* é um contador de bytes, *position* é uma compensação no interior do arquivo e *seconds* é o tempo decorrido. Os parâmetros são explicados no texto.

variáveis têm valores idênticos no momento da chamada fork, mas, como os dados do processo pai são copiados para criar o processo filho, mudanças subsequentes em um deles não afetam o outro. (O texto do programa, que é inalterável, é compartilhado entre processo pai e processo filho.) A chamada fork retorna um valor, que é zero no processo filho e igual ao identificador de processo (**PID**) do processo filho no processo pai. Usando o PID retornado, os dois processos podem verificar que um é o processo pai e que o outro é o processo filho.

Na maioria dos casos, depois de uma chamada fork, o processo filho precisará executar um código diferente daquele do processo pai. Considere o caso do shell. Ele lê um comando do terminal, cria um processo filho, espera que o processo filho execute o comando e, então, lê o próximo comando quando o processo filho termina. Para esperar a finalização do processo filho, o processo pai executa uma chamada de sistema waitpid, que somente aguarda até que o processo filho termine (qualquer processo filho, se existir mais de um). A chamada waitpid pode esperar por um processo filho específico ou por um processo filho qualquer atribuindo-se –1 ao primeiro parâmetro. Quando a chamada waitpid termina, o endereço apontado pelo segundo parâmetro, *statloc*, será atribuído como estado de saída do processo filho (término normal ou anormal e valor de saída). Várias opções também são oferecidas e especificadas pelo terceiro parâmetro.

Agora, considere como a chamada fork é usada pelo shell. Quando um comando é digitado, o shell cria um novo processo. Esse processo filho deve executar o comando do usuário. Ele faz isso usando a chamada de sistema execve, que faz com que toda a sua imagem do núcleo seja substituída pelo arquivo cujo nome está em seu primeiro parâmetro. (De fato, a chamada de sistema em si é exec, mas várias rotinas de biblioteca diferentes a chamam com diferentes parâmetros e nomes um pouco diferentes. Aqui, as trataremos como chamadas de sistema.) Um shell muito simplificado que ilustra o uso das chamadas fork, waitpid e execve é mostrado na Figura 1.18.

No caso mais geral, a chamada execve possui três parâmetros: o nome do arquivo a ser executado, um ponteiro para o arranjo de argumentos e um ponteiro para o arranjo do ambiente. Esses parâmetros serão descritos resumidamente. Várias rotinas de biblioteca — inclusive a *execl*, a *execv*, a *execle* e a *execve* — são fornecidas para que seja possível omitir parâmetros ou especificá-los de várias maneiras. Ao longo de todo este livro, usaremos o nome exec para representar a chamada de sistema invocada por todas essas rotinas.

Consideremos o caso de um comando como

```
cp arq1 arq2
```

usado para copiar *arq1* para *arq2*. Depois que o shell criou o processo filho, este localiza e executa o arquivo *cp* e passa para ele os nomes dos arquivos de origem e de destino.

O programa principal de *cp* (e o programa principal da maioria dos outros programas em C) contém a declaração

```
main(argc, argv, envp)
```

onde *argc* é um contador do número de itens na linha de comando, incluindo o nome do programa. Para esse exemplo, *argc* é 3.

O segundo parâmetro, *argv*, é um ponteiro para um arranjo. O elemento *i* desse arranjo é um ponteiro para a *i*-ésima cadeia de caracteres na linha de comando. Em nosso exemplo, *argv*[0] apontaria para a cadeia de caracteres 'cp', *argv*[1] apontaria a cadeia de caracteres 'arq1' e *argv*[2] apontaria para a cadeia de caracteres 'arq2'.

O terceiro parâmetro do *main*, *envp*, é um ponteiro para o ambiente, um arranjo de cadeias de caracteres que contém atribuições da forma *nome* = *valor*, usadas para passar para um programa informações como o tipo de terminal e o nome do diretório *home*. Há procedimentos de biblioteca que podem ser chamados por programas para se obter variáveis de ambiente, que nomalmente são usados para customizar como um usuário quer executar certas tarefas (por exemplo, a impressora-padrão a ser usada). Na Figura 1.18, nenhum ambiente é passado para o processo filho; assim, o terceiro parâmetro de *execve* é um zero.

```
#define TRUE 1

while (TRUE) {                                          /* repita para sempre   */
      type _prompt( );                                  /* mostra prompt na tela   */
      read _command(command,    parameters);            /* lê entrada do terminal   */

      if (fork( ) != 0) {                               /* cria processo filho   */
           /* Código do processo pai.   */
           waitpid( –1, &status, 0);                    /* aguarda o processo filho acabar   */
      } else {
           /* Código do processo filho.   */
           execve(command, parameters, 0);              /* executa o comando   */
      }
}
```

Figura 1.18 Um interpretador de comandos simplificado. Neste livro, presume-se *TRUE* como 1.

Se a chamada exec parecer-lhe complicada, não se desespere: ela é (semanticamente) a mais complexa de todas as chamadas de sistema em POSIX. Todas as outras são muito mais simples. Como um exemplo das simples, considere a chamada exit, que os processos devem usar para terminar sua execução. Ela possui um parâmetro, o estado da saída (0 a 255), que é retornado ao processo pai via *statloc* na chamada de sistema waitpid.

Os processos em UNIX têm suas memórias divididas em até três segmentos: o **segmento de texto** (isto é, o código do programa), o **segmento de dados** (as variáveis) e o **segmento de pilha**. O segmento de dados cresce para cima e a pilha cresce para baixo, conforme mostra a Figura 1.19. Entre eles há uma lacuna de espaço de endereçamento não usado. A pilha cresce automaticamente para dentro da lacuna, conforme se fizer necessário, mas a expansão do segmento de dados é feita mediante o uso explícito de uma chamada de sistema brk, que especifica o novo endereço no qual o segmento de dados termina. Contudo, essa chamada não é definida pelo padrão POSIX, já que os programadores são incentivados a usar a rotina de biblioteca *malloc* para alocar memória dinamicamente, e a implementação subjacente do *malloc* não foi planejada para que fosse uma candidata adequada à padronização, pois poucos programadores usam-na diretamente e é questionável se algum deles já percebeu que brk não está no POSIX.

1.6.2 Chamadas de sistema para gerenciamento de arquivos

Muitas chamadas de sistema estão relacionadas com o sistema de arquivos. Nesta seção estudaremos as chamadas que operam sobre arquivos individuais; na próxima, serão abordadas as que envolvem diretórios ou o sistema de arquivos como um todo.

Para ler ou escrever um arquivo, deve-se primeiro abri-lo usando open. Essa chamada especifica o nome do arquivo a ser aberto, como um caminho (*path name*) absoluto ou relativo ao diretório de trabalho e um código de *O_RDONLY*, *O_WRONLY* ou *O_RDWR*, significando abri-lo para leitura, escrita ou ambas. Para criar um novo arquivo, é usado o parâmetro *O_CREAT*. O descritor de arquivos retornado pode, então, ser usado para ler ou escrever. Logo em seguida, o arquivo pode ser fechado por close, que torna o descritor de arquivos disponível para reutilização em um open subsequente.

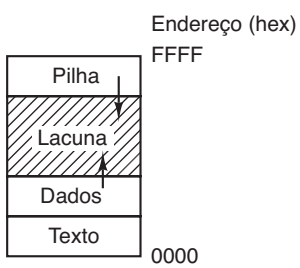

Figura 1.19 Os processos têm três segmentos: texto, dados e pilha.

As chamadas mais intensivamente utilizadas são, sem sombra de dúvida, read e write. Vimos read anteriormente. Write possui os mesmos parâmetros.

Embora a maioria dos programas leia e escreva arquivos sequencialmente, alguns programas aplicativos precisam ter disponibilidade de acesso aleatório a qualquer parte de um arquivo. Associado a cada arquivo existe um ponteiro que indica a posição atual no arquivo. Ao ler (escrever) sequencialmente, aponta-se geralmente para o próximo byte a ser lido (escrito). A chamada lseek altera o valor do ponteiro de posição, para que chamadas subsequentes de read ou write possam começar em qualquer ponto do arquivo.

A chamada lseek tem três parâmetros: o primeiro é o descritor de arquivo; o segundo é uma posição no arquivo e o terceiro informa se a posição no arquivo é relativa ao início, à posição atual ou ao final do arquivo. O valor retornado pela chamada lseek é a posição absoluta no arquivo (em bytes) depois de alterar o ponteiro.

Para cada arquivo, o UNIX registra o tipo do arquivo (arquivo regular, arquivo especial, diretório etc.), o tamanho e o momento da última modificação, entre outras informações. Os programas podem pedir para ver essa informação por meio da chamada de sistema stat. O primeiro parâmetro dessa chamada especifica o arquivo a ser inspecionado; o segundo é um ponteiro para uma estrutura na qual a informação deverá ser colocada. As chamadas fstat fazem a mesma coisa por um arquivo aberto.

1.6.3 Chamadas de sistema para gerenciamento de diretórios

Nesta seção veremos algumas chamadas de sistema mais relacionadas com diretórios ou com o sistema de arquivos como um todo, do que com um arquivo específico, como na seção anterior. As duas primeiras chamadas, mkdir e rmdir, respectivamente, criam e apagam diretórios vazios. A próxima chamada é a link. Seu intuito é permitir que um mesmo arquivo apareça com dois ou mais nomes, inclusive em diretórios diferentes. Um uso típico da chamada link é permitir que vários membros da mesma equipe de programação compartilhem um arquivo comum, cada um deles tendo o arquivo aparecendo em seu próprio diretório, possivelmente com diferentes nomes. Compartilhar um arquivo não é o mesmo que dar a cada membro da equipe uma cópia privada, mas significa que as mudanças feitas por qualquer membro dessa equipe ficam instantaneamente visíveis aos outros membros — há somente um arquivo. Quando são feitas cópias de um arquivo compartilhado, as mudanças subsequentes para uma cópia específica não afetam as outras.

Para vermos como a chamada link funciona, considere a situação da Figura 1.20(a). Ali estão dois usuários, *ast* e *jim*; cada um possui seus próprios diretórios com alguns arquivos. Então, se *ast* executar um programa que contenha a chamada de sistema

```
link("/usr/jim/memo", "/usr/ast/note");
```

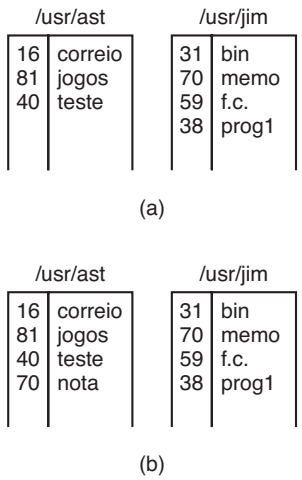

Figura 1.20 (a) Dois diretórios antes do link de */usr/jim/memo* ao diretório *ast*. (b) Os mesmos diretórios depois desse link.

o arquivo *memo* no diretório de *jim* estará agora aparecendo no diretório de *ast* com o nome *note*. A partir de então, */usr/jim/memo* e */usr/ast/note* referem-se ao mesmo arquivo. Vale notar que manter os diretórios de usuário em */usr*, */user*, */home* ou em algum outro lugar é simplesmente uma decisão tomada pelo administrador local do sistema.

Entender como a chamada link funciona provavelmente esclarecerá o que ela faz. Todo arquivo em UNIX tem um número único, seu i-número, que o identifica. Esse i-número é um índice em uma tabela de **i-nodes**, um por arquivo, informando quem possui o arquivo, onde seus blocos de disco estão, e assim por diante. Um diretório é simplesmente um arquivo contendo um conjunto de pares (i-número, nome em ASCII). Nas primeiras versões do UNIX, cada entrada de diretório era de 16 bytes — 2 bytes para o i-número e 14 bytes para o nome. Agora, é necessária uma estrutura mais complexa para dar suporte a nomes longos de arquivos, mas, conceitualmente, um diretório ainda é um conjunto de pares (i-número, nome em ASCII). Na Figura 1.20, correio tem o i-número 16, e assim por diante. O que a chamada link faz é simplesmente criar uma nova entrada de diretório com um nome (possivelmente novo), usando o i-número de um arquivo existente. Na Figura 1.20(b), duas entradas têm o mesmo i-número (70) e, portanto, fazem referência ao mesmo arquivo. Se uma das duas é removida posteriormente, usando-se uma chamada de sistema unlink, o outro arquivo permanece. Se ambos forem removidos, o UNIX perceberá que não há entradas para o arquivo (um campo no i-node registra o número de entradas de diretório apontando para o arquivo), e então o arquivo é removido do disco.

Conforme mencionado anteriormente, a chamada de sistema mount permite que dois sistemas de arquivo sejam unificados. Uma situação comum é ter em disco rígido o sistema de arquivos-raiz contendo as versões binárias (executáveis) dos comandos comuns e outros arquivos intensivamente usados. O usuário pode, então, inserir um disquete com arquivos a serem lidos na unidade de CD-ROM.

Executando a chamada de sistema mount, o sistema de arquivos do disco flexível pode ser anexado ao sistema de arquivos-raiz, conforme mostra a Figura 1.21. Um comando típico em C para realizar essa montagem é

```
mount("/dev/fd0", "/mnt", 0);
```

no qual o primeiro parâmetro é o nome de um arquivo especial de blocos para a unidade de disco 0, o segundo parâmetro é o local na árvore onde ele será montado e o terceiro parâmetro informa se o sistema de arquivos deve ser montado como leitura e escrita ou apenas para leitura.

Depois da chamada mount, pode-se ter acesso a um arquivo na unidade de disco 0 apenas usando seu caminho a partir do diretório-raiz ou do diretório de trabalho, sem se preocupar com qual unidade de disco isso será feito. Na verdade, a segunda, a terceira e a quarta unidades de disco também podem ser montadas em qualquer lugar na árvore. A chamada mount torna possível integrar meios removíveis a uma única hierarquia integrada de arquivos, sem a necessidade de saber em qual unidade se encontra um arquivo. Embora esse exemplo trate especificamente de unidades de CD-ROM, podemos montar também porções de discos rígidos (muitas vezes chamadas de **partições** ou **dispositivos secundários**) desse modo, assim como com discos rígidos externos e dispositivos stick USB. Quando não for mais necessário, um sistema de arquivos poderá ser desmontado com a chamada de sistema umount.

1.6.4 Outras chamadas de sistema

Existe uma variedade de outras chamadas de sistema. Estudaremos apenas quatro delas aqui. A chamada chdir altera o diretório atual de trabalho. Depois da chamada

```
chdir("/usr/ast/test");
```

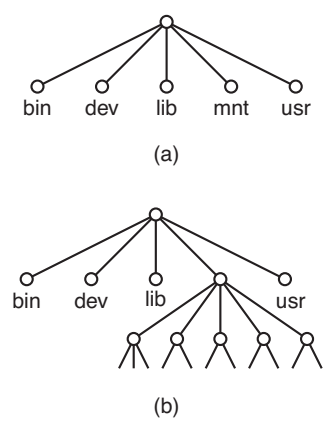

Figura 1.21 (a) O sistema de arquivos antes da montagem. (b) O sistema de arquivos depois da montagem.

uma abertura do arquivo *xyz* abrirá */usr/ast/test/xyz*. O conceito de diretório de trabalho elimina a necessidade de digitar (longos) nomes de caminhos absolutos a todo momento.

Em UNIX, todo arquivo tem um modo para proteção. Esse modo inclui os bits leitura-escrita-execução para o proprietário, para o grupo e para os outros. A chamada de sistema chmod possibilita a alteração do modo de um arquivo. Por exemplo, para definir um arquivo como somente leitura para todos, exceto seu proprietário, poderia ser executado

```
chmod("arq", 0644);
```

A chamada de sistema kill é a maneira que os usuários e seus processos têm para enviar sinais. Se um processo está preparado para capturar um sinal em particular, então uma rotina de tratamento desse sinal é executada quando ele chega. Se o processo não está preparado para tratar um sinal, então sua chegada 'mata' o processo (e, por conseguinte, o nome da chamada).

O POSIX define várias rotinas para lidar com o tempo. Por exemplo, a chamada time retorna o tempo atual em segundos, com 0 correspondendo à meia-noite do dia 1º de janeiro de 1970 (como se nesse instante o dia estivesse começando, e não terminando). Em computadores com palavras de 32 bits, o valor máximo que a chamada time pode retornar é $2^{32} - 1$ segundos (supondo que esteja usando inteiros sem sinal). Esse valor corresponde a pouco mais de 136 anos. Assim, no ano 2106, os sistemas UNIX de 32 bits irão entrar em pane, como o famoso bug do milênio em 2000, que foi evitado graças aos grandes esforços da indústria de TI para corrigir esse problema. Se você atualmente possui um sistema UNIX de 32 bits, aconselho-o a trocá-lo por um de 64 bits em algum momento antes de 2106.

1.6.5 | A API Win32 do Windows

Até aqui temos nos concentrado principalmente no UNIX. Agora é a vez de estudarmos resumidamente o Windows. O Windows e o UNIX diferem de uma maneira fundamental em seus respectivos modelos de programação. Um programa UNIX consiste em um código que faz uma coisa ou outra, executando chamadas de sistema para rea-lizar certos serviços. Por outro lado, um programa Windows é normalmente dirigido por eventos — o programa principal espera acontecer algum evento e então chama uma rotina para tratá-lo. Eventos típicos são teclas sendo pressionadas, a movimentação do mouse, um botão de mouse sendo pressionado ou um disco flexível sendo inserido. Os manipuladores (*handlers*) são, então, chamados para processar o evento, atualizar a tela e o estado interno do programa. De modo geral, isso leva a um estilo de programação diferente daquele do UNIX, mas, como o foco deste livro é a função e a estrutura do sistema operacional, esses diferentes modelos de programação não nos interessarão muito.

É claro que o Windows também tem chamadas de sistema. Em UNIX, há quase um relacionamento de 1-para-1 entre as chamadas de sistema (por exemplo, read) e as rotinas de biblioteca (por exemplo, *read*) usadas para invocar as chamadas de sistema. Em outras palavras, para cada chamada de sistema há, *grosso modo*, uma rotina de biblioteca que é chamada para invocá-la, conforme indica a Figura 1.17. Além disso, POSIX possui somente cerca de cem chamadas de rotina.

Com o Windows, a situação é radicalmente diferente. Para começar, as chamadas de biblioteca e as chamadas reais ao sistema são bastante desacopladas. A Microsoft definiu um conjunto de rotinas, denominado **API Win32** (*application program interface* — interface do programa de aplicativo), para que os programadores tivessem acesso aos serviços do sistema operacional. Essa interface tem sido (parcialmente) suportada em todas as versões do Windows desde o Windows 95. Desacoplando-se a interface das chamadas reais ao sistema, a Microsoft detém a capacidade de mudar as chamadas reais ao sistema quando bem entender (até de versão para versão) sem invalidar os programas existentes. O que realmente constitui o subsistema Win32 é um pouco ambíguo, já que o Windows 2000, o Windows XP e o Windows Vista têm muitas chamadas novas que não estavam anteriormente disponíveis. Nesta seção, Win32 significa a interface suportada por todas as versões do Windows.

O número de chamadas da API Win32 é extremamente grande, chegando a milhares. Além disso, enquanto muitas delas invocam chamadas de sistema, uma quantidade substancial é executada totalmente no espaço de usuário. Como consequência disso, no Windows é impossível ver o que é uma chamada de sistema (isto é, realizada pelo núcleo) e o que constitui simplesmente uma chamada de biblioteca do espaço de usuário. Na verdade, o que é uma chamada de sistema em uma versão do Windows pode ser executado no espaço de usuário em uma versão diferente e vice-versa. Quando discutirmos as chamadas de sistema do Windows neste livro, usaremos as rotinas Win32 (onde for apropriado), já que a Microsoft garante que essas rotinas permanecerão estáveis com o passar do tempo. Mas é bom lembrar que nem todas elas são verdadeiras chamadas de sistema (isto é, desviam o controle para o núcleo).

A API Win32 tem um número imenso de chamadas para gerenciar janelas, figuras geométricas, textos, fontes de caracteres, barras de rolagem, caixas de diálogos, menus e outros aspectos da interface gráfica GUI. Com o intuito de estender o subsistema gráfico para executar em modo núcleo (o que é válido para algumas versões do Windows, mas não para todas), a interface gráfica GUI é composta de chamadas de sistema; do contrário, elas conteriam apenas chamadas de biblioteca. Deveríamos discutir essas chamadas neste livro ou não? Como elas não são realmente relacionadas com a função de sistema operacional, decidimos que não, mesmo sabendo que elas podem ser executadas pelo núcleo. Leitores interessados na API Win32 podem

consultar um dos muitos livros sobre o assunto, como, por exemplo, Hart (1997), Rector e Newcomer (1997) e Simon (1997).

Já que introduzir todas as chamadas da interface API Win32 está fora de questão, ficaremos limitados às chamadas que correspondem, grosso modo, à funcionalidade das chamadas UNIX relacionadas na Tabela 1.1. Elas estão enumeradas na Tabela 1.2.

Vamos agora percorrer rapidamente a lista da Tabela 1.2. CreateProcess cria um novo processo; funciona como uma combinação de fork e de execve em UNIX. Possui muitos parâmetros que especificam as propriedades do processo recentemente criado. O Windows não tem uma hierarquia de processos como o UNIX; portanto, não há o conceito de processo pai e processo filho. Depois que um processo foi criado, o criador e a criatura são iguais. WaitForSingleObject é usada para esperar por um evento. É possível esperar muitos eventos com essa chamada. Se o parâmetro especificar um processo, então quem chamou esperará o processo especificado sair, o que é feito usando ExitProcess.

UNIX	Win32	Descrição
fork	CreateProcess	Cria um novo processo
waitpid	WaitForSingleObject	Pode esperar que um processo saia
execve	(nenhuma)	CreateProcess = fork + execve
exit	ExitProcess	Conclui a execução
open	CreateFile	Cria um arquivo ou abre um arquivo existente
close	CloseHandle	Fecha um arquivo
read	ReadFile	Lê dados a partir de um arquivo
write	WriteFile	Escreve dados em um arquivo
lseek	SetFilePointer	Move o ponteiro do arquivo
stat	GetFileAttributesEx	Obtém vários atributos do arquivo
mkdir	CreateDirectory	Cria um novo diretório
rmdir	RemoveDirectory	Remove um diretório vazio
link	(nenhuma)	Win32 não dá suporte a links
unlink	DeleteFile	Destrói um arquivo existente
mount	(nenhuma)	Win32 não dá suporte a mount
umount	(nenhuma)	Win32 não dá suporte a mount
chdir	SetCurrentDirectory	Altera o diretório de trabalho atual
chmod	(nenhuma)	Win32 não dá suporte a segurança (embora o NT suporte)
kill	(nenhuma)	Win32 não dá suporte a sinais
time	GetLocalTime	Obtém o tempo atual

Tabela 1.2 As chamadas da API Win32 que correspondem aproximadamente às chamadas do UNIX da Tabela 1.1.

As próximas seis chamadas operam sobre arquivos e são funcionalmente similares a suas correspondentes do UNIX, embora sejam diferentes quanto aos parâmetros e alguns detalhes. Além disso, os arquivos podem ser abertos, fechados, lidos e escritos de um modo muito similar ao do UNIX. As chamadas SetFilePointer e GetFileAttributesEx alteram a posição no arquivo e obtêm alguns atributos de arquivo.

O Windows possui diretórios que são criados e removidos com CreateDirectory e RemoveDirectory, respectivamente. Há também a noção de diretório atual, determinada por SetCurrentDirectory. O tempo atual é obtido por GetLocalTime.

A interface Win32 não tem links para arquivos, nem sistemas de arquivos montados, nem segurança ou sinais. Portanto, essas chamadas, correspondentes às chamadas em UNIX, não existem. É claro que o subsistema Win32 possui uma grande quantidade de chamadas que o UNIX não tem, especialmente para gerenciar a interface gráfica GUI, e o Windows Vista tem um elaborado sistema de segurança e também dá suporte a links (ligações de arquivos).

Por fim, talvez seja melhor fazer uma observação sobre o subsistema Win32: ele não é uma interface totalmente uniforme e consistente. A principal acusação contra ele é a necessidade de compatibilidade retroativa com as interfaces de 16 bits antigamente usadas no Windows 3.x.

1.7 Estrutura de sistemas operacionais

Agora que tivemos uma visão externa de um sistema operacional — isto é, da interface dele com o programador —, é o momento de olharmos para sua estrutura interna. Nas próximas seções, vamos examinar cinco diferentes estruturas de sistemas operacionais, para termos uma ideia do espectro de possibilidades. Isso não significa que esgotaremos o assunto, mas que daremos uma ideia sobre alguns projetos que têm sido usados na prática. Os seis grupos abordados serão os seguintes: sistemas monolíticos, sistemas de camadas, micronúcleo, sistemas cliente–servidor, máquinas virtuais e exonúcleo.

1.7.1 Sistemas monolíticos

A organização monolítica é de longe a mais comum; nesta abordagem, o sistema operacional inteiro é executado como um único programa no modo núcleo. O sistema

operacional é escrito como uma coleção de rotinas, ligadas a um único grande programa binário executável. Nessa abordagem, cada rotina do sistema tem uma interface bem definida quanto a parâmetros e resultados e cada uma delas é livre para chamar qualquer outra, se esta oferecer alguma computação útil de que a primeira necessite. A existência de milhares de rotinas que podem chamar umas às outras sem restrição muitas vezes leva a dificuldades de compreensão do sistema.

Para construir o programa-objeto real do sistema operacional usando essa abordagem, primeiro compilam-se todas as rotinas individualmente (ou os arquivos que contêm as rotinas). Então, juntam-se todas em um único arquivo-objeto usando o ligador (*linker*) do sistema. Não existe essencialmente ocultação de informação; todas as rotinas são visíveis umas às outras (o oposto de uma estrutura de módulos ou pacotes, na qual muito da informação é ocultado dentro de módulos e somente os pontos de entrada designados podem ser chamados do lado de fora do módulo).

Contudo, mesmo em sistemas monolíticos, é possível ter um mínimo de estrutura. Os serviços (chamadas de sistema) providos pelo sistema operacional são requisitados colocando-se os parâmetros em um local bem definido (na pilha, por exemplo) e, então, executando uma instrução de desvio de controle (*trap*). Essa instrução chaveia a máquina do modo usuário para o modo núcleo e transfere o controle para o sistema operacional, mostrado como passo 6 na Figura 1.17. O sistema operacional busca então os parâmetros e determina qual chamada de sistema será executada. Depois disso, ele indexa uma tabela que contém na linha *k* um ponteiro para a rotina que executa a chamada de sistema *k* (passo 7 na Figura 1.17).

Essa organização sugere uma estrutura básica para o sistema operacional:

1. Um programa principal que invoca a rotina do serviço requisitado.
2. Um conjunto de rotinas de serviço que executam as chamadas de sistema.
3. Um conjunto de rotinas utilitárias que auxiliam as rotinas de serviço.

Segundo esse modelo, para cada chamada de sistema há uma rotina de serviço que se encarrega dela. As rotinas utilitárias realizam tarefas necessárias para as várias rotinas de serviço, como buscar dados dos programas dos usuários. Essa divisão de rotinas em três camadas é mostrada na Figura 1.22.

Além do sistema operacional principal que é carregado quando um computador é iniciado, muitos sistemas operacionais dão suporte a extensões carregáveis, como drivers de E/S e sistemas de arquivos. Esses componentes são carregados conforme a demanda.

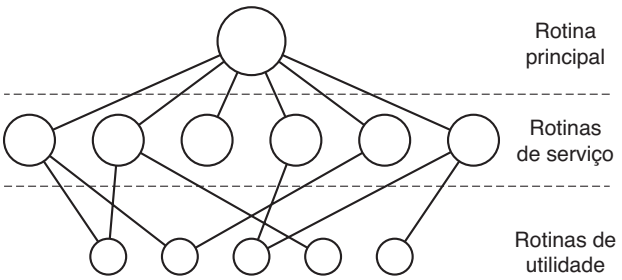

Figura 1.22 Um modelo de estruturação simples para um sistema monolítico.

1.7.2 Sistemas de camadas

Uma generalização da abordagem da Figura 1.22 é a organização do sistema operacional como uma hierarquia de camadas, cada uma delas construída sobre a camada imediatamente inferior. O primeiro sistema construído dessa maneira foi o THE, cuja sigla deriva do Technische Hogeschool Eindhoven, na Holanda, onde foi implementado por E.W. Dijkstra (1968) e seus alunos. O sistema THE era um sistema em lote (*batch*) simples para um computador holandês, o Electrologica X8, que tinha 32 K de palavras de 27 bits (os bits eram caros naquela época).

O sistema possuía seis camadas, conforme mostra a Tabela 1.3. A camada 0 tratava da alocação do processador, realizando chaveamento de processos quando ocorriam as interrupções ou quando os temporizadores expiravam. Acima da camada 0, o sistema era formado por processos sequenciais; cada um deles podia ser programado sem a preocupação com o fato de múltiplos processos estarem executando em um único processador. Em outras palavras, a camada 0 fornecia a multiprogramação básica da CPU.

A camada 1 encarregava-se do gerenciamento de memória. Ela alocava espaço para processos na memória principal e em um tambor magnético de 512 K palavras, que armazenava as partes de processos (páginas) para os quais não havia espaço na memória principal. Acima da camada 1, os processos não precisavam prestar atenção a se estavam na memória principal ou no tambor magnético; a camada 1 cuidava disso, assegurando que as páginas eram trazidas para a memória principal quando necessário.

Camada	Função
5	O operador
4	Programas de usuário
3	Gerenciamento de entrada/saída
2	Comunicação operador–processo
1	Memória e gerenciamento de tambor
0	Alocação do processador e multiprogramação

Tabela 1.3 Estrutura do sistema operacional THE.

A camada 2 encarregava-se da comunicação entre cada processo e o console de operação (isto é, o usuário). Acima dessa camada, cada processo tinha efetivamente seu próprio console de operação. A camada 3 encarregava-se do gerenciamento dos dispositivos de E/S e armazenava temporariamente os fluxos de informação que iam para esses dispositivos ou que vinham deles. Acima da camada 3, cada processo podia lidar com dispositivos abstratos de E/S mais amigáveis, em vez de dispositivos reais cheios de peculiaridades. Na camada 4 encontravam-se os programas de usuário. Eles não tinham de se preocupar com o gerenciamento de processo, de memória, console ou E/S. O processo operador do sistema estava localizado na camada 5.

Outra generalização do conceito de camadas estava presente no sistema MULTICS. Em vez de camadas, o MULTICS era descrito como uma série de anéis concêntricos, sendo que cada anel interno era mais privilegiado que os externos. Quando uma rotina em um anel externo queria chamar uma rotina no anel interno, ela deveria fazer o equivalente a uma chamada de sistema, isto é, uma instrução de desvio, TRAP, e a validade dos parâmetros era cuidadosamente verificada antes de permitir que a chamada continuasse. Embora no MULTICS todo o sistema operacional fosse parte do espaço de endereçamento de cada processo de usuário, o hardware possibilitava ao sistema designar rotinas individuais (na verdade, segmentos de memória) como protegidas contra leitura, escrita ou execução.

O esquema de camadas do sistema THE era somente um suporte ao projeto, pois todas as partes do sistema eram, ao final, agrupadas em um único programa-objeto. Já no MULTICS, o mecanismo de anéis estava muito mais presente em tempo de execução e reforçado pelo hardware. Esse mecanismo de anéis era vantajoso porque podia facilmente ser estendido para estruturar subsistemas de usuário. Por exemplo, um professor podia escrever um programa para testar e atribuir notas a programas de alunos executando-o no anel *n*, enquanto os programas dos alunos seriam executados no anel *n* + 1, a fim de que nenhum deles pudesse alterar suas notas.

1.7.3 | Micronúcleo

Com a abordagem do sistema de camadas, os projetistas podem escolher onde traçar a fronteira núcleo–usuário. Tradicionalmente, todas as camadas entram no núcleo, mas isso não é necessário. Na realidade, apresentam-se fortes argumentos para colocação do mínimo possível em modo núcleo porque erros no núcleo podem derrubar o sistema instantaneamente. Por outro lado, processos de usuário podem ser configurados com menos potência de modo que um erro não seja fatal.

Vários pesquisadores têm estudado o número de erros por mil linhas de código (por exemplo, Basilli e Perricone, 1984; Ostrand e Weyuker, 2002). A densidade de erros depende do tamanho do módulo, da idade do módulo etc., mas uma cifra aproximada para sistemas industriais sérios é de dez erros por mil linhas de código. Isso significa que é provável que um sistema operacional monolítico de cinco milhões de linhas de código contenha algo como 50 mil erros no núcleo. É claro que nem todos são fatais, visto que alguns erros podem ser coisas como emitir uma mensagem de erro incorreta em uma situação que raramente ocorre. Contudo, os sistemas operacionais são suficientemente sujeitos a erro e, por isso, os fabricantes de computadores inserem botões de reinicialização neles (frequentemente no painel frontal), algo que fabricantes de aparelhos de TV, rádios e carros não fazem, apesar da grande quantidade de softwares nesses dispositivos.

A ideia básica por trás do projeto do micronúcleo é alcançar alta confiabilidade por meio da divisão do sistema operacional em módulos pequenos, bem definidos, e apenas um desses módulos — o micronúcleo — é executado no modo núcleo e o restante é executado como processos de usuário comuns relativamente sem potência. Em particular, quando há a execução de cada driver de dispositivo e de cada sistema de arquivos como um processo de usuário separado, um erro em um deles pode quebrar aquele componente, mas não pode quebrar o sistema inteiro. Desse modo, um erro na unidade de áudio fará com que o som seja adulterado ou interrompido, mas não travará o computador. Por outro lado, em um sistema monolítico, com todas as unidades no núcleo, uma unidade de áudio defeituosa pode facilmente dar como referência um endereço de memória inválido e parar o sistema instantaneamente.

Muitos micronúcleo foram implementados e utilizados (Accetta et al., 1986; Haertig et al., 1997; Heiser et al., 2006; Herder et al., 2006; Hildebrand, 1992; Kirsch et al., 2005; Liedtke, 1993, 1995, 1996; Pike et al., 1992; Zuberi et al., 1999). Eles são especialmente comuns em aplicações de tempo real, industriais, de aviônica e militares, que são cruciais e têm requisitos de confiabilidade muito altos. Alguns dos micronúcleos mais conhecidos são Integrity, K42, L4, PikeOS, QNX, Symbian e MINIX 3. Faremos agora um breve resumo do MINIX 3, que levou a ideia de modularidade ao limite, decompondo a maior parte do sistema operacional em vários processos independentes no modo usuário. O MINIX 3 é um sistema de código aberto disponível gratuitamente em <www.minix3.org> e compatível com o POSIX (Herder et al., 2006a; Herder et al., 2006b).

O micronúcleo do MINIX 3 tem apenas cerca de 3.200 linhas de C e 800 linhas de assembler para funções de nível muito baixo, como contenção de interrupções e processos de chaveamento. O código C gerencia e escalona processos, controla a comunicação entre processos (trocando mensagens entre eles) e oferece um conjunto de cerca de 35 chamadas ao núcleo para permitir que o resto do sistema operacional faça seu trabalho. Essas chamadas executam funções como associar os manipuladores (*handlers*) às interrupções, transferir dados entre espaços de endereçamento e instalar novos mapas de memória para proces-

sos recém-criados. A estrutura de processo do MINIX 3 é mostrada na Figura 1.23 e os manipuladores de chamada de núcleo (*kernel call handlers*) são rotulados como *Sys*. O driver de dispositivo para o relógio também está no núcleo porque o escalonador interage proximamente com ele. Todos os outros drivers de dispositivo operam separadamente como processos de usuário.

Fora do núcleo, o sistema é estruturado em três camadas de processos, todas executando em modo usuário. A camada inferior contém os drivers de dispositivos. Visto que eles executam em modo usuário, não têm acesso físico ao espaço de porta de E/S e não podem emitir comandos de E/S diretamente. Em vez disso, para programar um dispositivo de E/S, o driver constrói uma estrutura dizendo que valores escrever em quais portas de E/S e gera uma chamada ao núcleo para realizar a escrita. Essa abordagem permite que o núcleo verifique o que o driver está escrevendo (ou lendo) a partir da E/S que está autorizada a utilizar. Consequentemente (e diferentemente do projeto monolítico), uma unidade de áudio defeituosa não pode escrever acidentalmente no disco.

Acima dos drivers está outra camada no modo usuário que contém os servidores, que fazem a maior parte do trabalho do sistema operacional. Um ou mais servidores de arquivos gerenciam o(s) sistema(s) de arquivo, o gerenciador de processos cria, destrói e gerencia processos etc. Os programas de usuários obtêm serviços do sistema operacional enviando mensagens curtas aos servidores solicitando chamadas de sistema POSIX. Por exemplo, um processo que precise fazer um read envia uma mensagem a um dos servidores de arquivo dizendo-lhe o que ler.

Um servidor interessante é o **servidor de reencarnação**, cujo trabalho é verificar se os outros servidores e drivers estão funcionando corretamente. Nos casos em que se detecta uma unidade defeituosa, ela é automaticamente substituída sem nenhuma intervenção do usuário. Desse modo, o sistema é capaz de autorrecuperação e pode atingir grande confiabilidade.

Esse sistema tem muitas restrições que limitam a potência de cada processo. Como mencionado anteriormente, os drivers podem tocar apenas portas de E/S autorizadas, mas o acesso a chamadas ao núcleo também é controlado por processo, assim como a capacidade de enviar mensagens a outros processos. Os processos também podem conceder autorização limitada a outros para que o núcleo acesse seus espaços de endereçamento. Por exemplo, um sistema de arquivos pode permitir que a unidade de disco deixe o núcleo colocar uma leitura recente de um bloco do disco em um endereço específico no espaço de endereço do sistema de arquivos. A soma de todas essas restrições é que cada driver e servidor tem exatamente a potência para fazer seu trabalho e nada mais, o que limita enormemente os danos que poderiam ser causados por um componente defeituoso.

Uma ideia relacionada ao núcleo mínimo é colocar o **mecanismo** para fazer algo no núcleo e não a **política**. Para compreendermos melhor esse ponto, consideremos o escalonamento de processos. Um algoritmo de escalonamento relativamente simples é atribuir uma prioridade a cada processo e, em seguida, fazer com que o núcleo execute o processo executável de maior prioridade. O mecanismo — no núcleo — é procurar o processo de maior prioridade e executá-lo. A política — atribuir prioridades aos processos — pode ser feita por processos de modo usuário. Desse modo, política e mecanismo podem ser desacoplados e o núcleo pode ser reduzido.

1.7.4 | O modelo cliente–servidor

Uma ligeira variação da ideia do micronúcleo é distinguir entre duas classes de processos, os **servidores**, que prestam algum serviço, e os **clientes**, que usam esses serviços. Esse modelo é conhecido como modelo do **cliente–servidor**. Frequentemente a camada inferior é o micronúcleo, mas ele não é obrigatório. A essência é a presença de processos clientes e servidores.

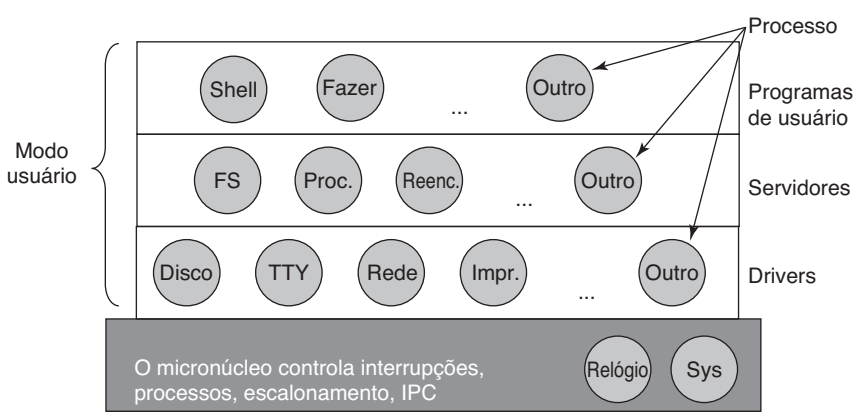

Figura 1.23 Estrutura do sistema MINIX 3.

A comunicação entre clientes e servidores é muitas vezes realizada por meio de troca de mensagens. Para obter um serviço, um processo cliente constrói uma mensagem dizendo o que deseja e a envia ao serviço apropriado. Este faz o trabalho e envia a resposta de volta. Se o cliente e o servidor forem executados na mesma máquina, certas otimizações são possíveis, mas, conceitualmente, estamos falando de troca de mensagens neste caso.

Uma generalização óbvia dessa ideia é executar clientes e servidores em computadores diferentes, conectados por uma rede local ou de grande área, conforme a ilustração da Figura 1.24. Uma vez que os clientes se comunicam com os servidores enviando mensagens, eles não precisam saber se as mensagens são entregues localmente em suas próprias máquinas ou se são enviadas através de uma rede a servidores em uma máquina remota. No que se refere aos clientes, o mesmo ocorre em ambos os casos: solicitações são enviadas e as respostas, devolvidas. Dessa forma, o modelo cliente–servidor é uma abstração que pode ser usada para uma única máquina ou para uma rede de máquinas.

Cada vez mais vários sistemas envolvem usuários em seus computadores pessoais, como clientes e máquinas grandes em outros lugares, como servidores. De fato, grande parte da Web opera dessa forma. Um PC envia uma solicitação de página da Web ao servidor e a página da Web retorna. Esse é um uso típico do modelo cliente–servidor em uma rede.

1.7.5 | Máquinas virtuais

As versões iniciais do sistema operacional OS/360 eram estritamente em lote (*batch*). Porém, muitos usuários do IBM 360 desejavam compartilhamento de tempo. Então, vários grupos, de dentro e de fora da IBM, decidiram escrever sistemas de tempo compartilhado para o IBM 360. O sistema de tempo compartilhado oficial da IBM, o TSS/360, foi lançado muito tarde e, quando finalmente se tornou mais popular, estava tão grande e lento que poucos clientes converteram suas aplicações. Ele foi finalmente abandonado depois de já ter consumido cerca de 50 milhões de dólares em seu desenvolvimento (Graham, 1970). Mas um grupo do Centro Científico da IBM em Cambridge, Massachusetts, produziu um outro sistema, radicalmente diferente, que a IBM finalmente adotou. Um descendente linear desse sistema, chamado **z/VM**, agora é amplamente usado nos computadores de grande porte atuais da IBM, a série z, que são muito utilizados em grandes centros de dados corporativos, por exemplo, como servidores de comércio eletrônico que controlam centenas de milhares de transações por segundo e usam bases de dados cujo tamanho pode chegar a milhões de gigabytes.

VM/370

Esse sistema, originalmente denominado CP/CMS e depois renomeado VM/370 (Seawright e MacKinnon, 1979), foi baseado em uma observação perspicaz: um sistema de tempo compartilhado fornece (1) multiprogramação e (2) uma máquina estendida com uma interface mais conveniente do que a que o hardware oferece. A essência do VM/370 é a separação completa dessas duas funções.

O coração do sistema, conhecido como **monitor de máquina virtual**, é executado diretamente sobre o hardware e implementa a multiprogramação, provendo assim não uma, mas várias máquinas virtuais para a próxima camada situada acima, conforme mostra a Figura 1.25. Contudo, ao contrário dos demais sistemas operacionais, essas máquinas virtuais não são máquinas estendidas, com arquivos e outras características convenientes. Na verdade, são cópias *exatas* do hardware, inclusive com modos núcleo/usuário, E/S, interrupções e tudo o que uma máquina real tem.

Como cada máquina virtual é uma cópia exata do hardware, cada uma delas pode executar qualquer sistema operacional capaz de ser executado diretamente sobre o hardware. Diferentes máquinas virtuais podem — e isso ocorre com frequência — executar diferentes sistemas operacionais. Em algumas dessas máquinas virtuais é executado um dos sistemas operacionais descendentes do OS/360 para processamento em lote (*batch*) ou de transações, enquanto se executa em outras um sistema operacional monousuário interativo, denominado **CMS** (*conversational monitor system* — sistema monitor conversacional), dedicado a usuários interativos em tempo compartilhado, o qual era popular entre os programadores.

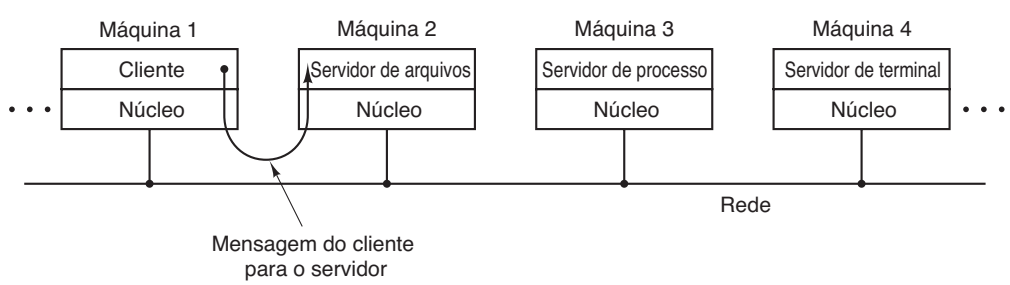

Figura 1.24 O modelo cliente–servidor em uma rede.

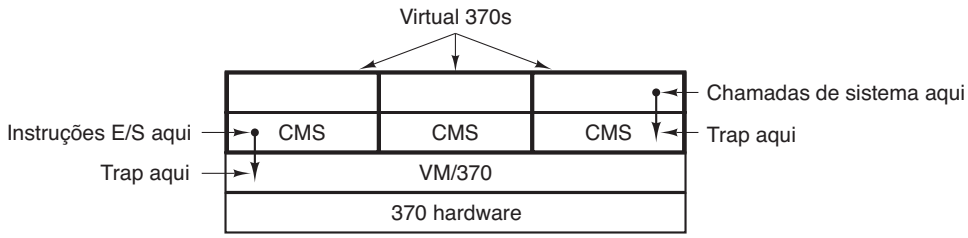

Figura 1.25 Estrutura do VM/370 com CMS.

Quando um programa CMS executa uma chamada de sistema, ela é desviada para o sistema operacional, que executa em sua própria máquina virtual, e não para o VM/370, como se estivesse executando sobre uma máquina real e não sobre uma máquina virtual. Então, o sistema operacional CMS emite as instruções normais de hardware para E/S a fim de, por exemplo, ler seu disco virtual ou executar outro serviço qualquer pedido pela chamada. Essas instruções de E/S são, por sua vez, desviadas pelo VM/370, que as executa como parte de sua simulação do hardware real. A partir da separação completa das funções de multiprogramação e da provisão de uma máquina estendida, pode-se ter partes muito mais simples, flexíveis e fáceis de serem mantidas.

Em sua encarnação moderna, o z/VM normalmente é utilizado para executar sistemas operacionais completos múltiplos em vez de sistemas de usuário único desmontados como o CMS. Por exemplo, a série z é capaz de executar uma ou mais máquinas virtuais Linux com sistemas operacionais IBM tradicionais.

Máquinas virtuais redescobertas

Embora a IBM tenha um produto de máquina virtual disponível há quatro décadas e algumas outras companhias, inclusive a Sun Microsystems e a Hewlett-Packard, tenham acrescentado recentemente um suporte de máquina virtual a seus servidores empresariais de alto desempenho, a ideia de virtualização foi em grande medida ignorada na indústria da computação até pouco tempo atrás. Mas, nos últimos anos, uma combinação de novas necessidades, novos softwares e novas tecnologias tornou essa ideia um tópico de interesse.

Primeiro as necessidades. Muitas companhias tradicionalmente executavam seus servidores de correio, servidores da Web, servidores FTP e outros em computadores separados, algumas vezes com sistemas operacionais diferentes. Elas percebem a virtualização como um modo de executar todos eles na mesma máquina sem que uma falha em um servidor afete o resto.

A virtualização também é popular na indústria de hospedagem de páginas da Web. Sem a virtualização, os clientes da hospedagem na Web são forçados a escolher entre **hospedagem compartilhada** (que lhes dá apenas uma conta de acesso a um servidor da Web, mas não lhes permite controlar o software do servidor) e hospedagem dedicada (que lhes oferece uma máquina própria, que é muito flexível mas pouco econômica para sites da Web de pequeno a médio porte). Quando uma companhia de hospedagem na Web aluga máquinas virtuais, uma única máquina física pode executar muitas máquinas virtuais e cada uma delas parece ser uma máquina completa. Os clientes que alugam uma máquina virtual podem executar quaisquer sistemas operacionais e softwares que desejem, mas por uma fração do custo de um servidor dedicado (porque a mesma máquina física dá suporte a muitas máquinas virtuais ao mesmo tempo).

A virtualização também é utilizada por usuários finais que querem executar dois ou mais sistemas operacionais ao mesmo tempo, por exemplo Windows e Linux, porque alguns de seus pacotes de aplicações favoritos são executados em um dos sistemas e outros pacotes em outro sistema. Essa situação é ilustrada na Figura 1.26(a), na qual o termo 'monitor de máquina virtual' foi alterado para **hipervisor** tipo 1 nos últimos anos.

Agora o software. Embora ninguém discuta a atratividade das máquinas virtuais, o problema foi a implementação. Para executar o software de máquina virtual em

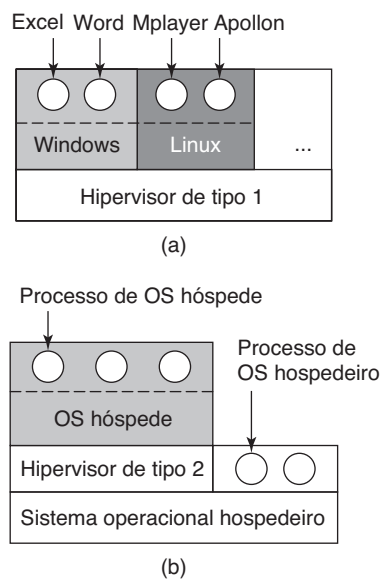

Figura 1.26 (a) Hipervisor de tipo 1. (b) Hipervisor de tipo 2.

um computador, sua CPU deve ser virtualizável (Popek e Goldberg, 1974). Em poucas palavras, há um problema neste caso. Quando um sistema operacional sendo executado em uma máquina virtual (em modo usuário) executa uma instrução privilegiada, como modificar a PSW ou fazer E/S, é essencial que o hardware crie um dispositivo para o monitor da máquina virtual, de modo que a instrução possa ser emulada em software. Em algumas CPUs — principalmente Pentium, seus predecessores e seus clones — tentativas de executar instruções não privilegiadas no modo usuário são simplesmente ignoradas. Essa propriedade impossibilitou a existência de máquinas virtuais nesse hardware, o que explica a falta de interesse na indústria da computação. É claro que havia interpretadores para o Pentium que eram executados nele mas, com uma perda de desempenho de 5-10x em geral, eles não eram úteis para trabalhos importantes.

Essa situação mudou como resultado de vários projetos de pesquisa acadêmica na década de 1990, particularmente o Disco em Stanford (Bugnion et al., 1997), que conduziu a produtos comerciais (por exemplo, VMware Workstation) e a uma retomada do interesse em máquinas virtuais. O VMware Worsktation é um hipervisor de tipo 2, mostrado na Figura 1.26(b). Ao contrário dos hipervisores de tipo 1, que são executados diretamente no hardware, os hipervisores de tipo 2 são executados como programas aplicativos na camada superior do Windows, Linux ou algum outro sistema operacional, conhecido como **sistema operacional hospedeiro**. Depois de ser iniciado, um hipervisor de tipo 2 lê o CD-ROM de instalação para o **sistema operacional hóspede** escolhido e instala um disco virtual, que é só um arquivo grande no sistema de arquivos do sistema operacional hospedeiro.

Quando o sistema operacional hóspede é inicializado, faz o mesmo que no verdadeiro hardware, normalmente iniciando algum processo subordinado e, em seguida, uma interface gráfica GUI. Alguns hipervisores traduzem os programas binários do sistema operacional convidado bloco a bloco, substituindo determinadas instruções de controle por chamadas ao hipervisor. Os blocos traduzidos são executados e armazenados para uso posterior.

Uma abordagem diferente para o gerenciamento de instruções de controle é modificar o sistema operacional para removê-las. Essa abordagem não é uma virtualização autêntica, e sim uma **paravirtualização**. Discutiremos a virtualização em maiores detalhes no Capítulo 8.

A máquina virtual Java

Outra área na qual máquinas virtuais são usadas, mas de maneira um pouco diferente, é na execução de programas Java. Quando a Sun Microsystems inventou a linguagem de programação Java, inventou também uma máquina virtual (isto é, uma arquitetura de computador) denominada **JVM** (*Java virtual machine* — máquina virtual Java). O compilador Java produz código para JVM, que então normalmente é executada por um programa interpretador da JVM. A vantagem desse sistema é que o código JVM pode ser enviado pela Internet a qualquer computador que tenha um interpretador JVM e ser executado lá. Se o compilador produzisse, por exemplo, código binário para a SPARC ou para o Pentium, esses códigos não poderiam ser tão facilmente levados de um lugar para outro. (É claro que a Sun poderia ter produzido um compilador que gerasse código binário para a SPARC e então ter distribuído um interpretador SPARC, mas a JVM é uma arquitetura muito mais simples de interpretar.) Outra vantagem do uso da JVM é a seguinte: se o interpretador for implementado adequadamente — o que não é muito comum —, os programas JVM que chegam podem ser verificados, por segurança, e então executados em um ambiente protegido, de modo que não possam roubar dados ou causar quaisquer danos.

1.7.6 Exonúcleo

Em vez de clonar a máquina real, como é feito no caso das máquinas virtuais, outra estratégia é dividi-la ou, em outras palavras, dar a cada usuário um subconjunto de recursos. Assim, uma máquina virtual pode obter os blocos 0 a 1.023 do disco, uma outra os blocos 1.024 a 2.047 e assim por diante.

Na camada mais inferior, executando em modo núcleo, há um programa denominado **exonúcleo**. Sua tarefa é alocar recursos às máquinas virtuais e então verificar as tentativas de usá-los para assegurar-se de que nenhuma máquina esteja tentando usar recursos de outra. Cada máquina virtual, em nível de usuário, pode executar seu próprio sistema operacional, como no VM/370 e na máquina virtual 8086 do Pentium, exceto que cada uma está restrita a usar somente os recursos que pediu e que foram alocados.

A vantagem do esquema exonúcleo é que ele poupa uma camada de mapeamento. Nos outros projetos, cada máquina virtual pensa que tem seu próprio disco, com blocos indo de 0 a um valor máximo, de modo que o monitor de máquina virtual deve manter tabelas para remapear os endereços de disco (e todos os outros recursos). Com o exonúcleo, esse mapeamento deixa de ser necessário. Ele precisa somente manter o registro de para qual máquina virtual foi atribuído qual recurso. Esse método ainda tem a vantagem adicional de separar, com menor custo, a multiprogramação (no exonúcleo) do código do sistema operacional do usuário (no espaço do usuário), já que tudo que o exonúcleo tem de fazer é manter as máquinas virtuais umas fora do alcance das outras.

1.8 O mundo de acordo com a linguagem C

Os sistemas operacionais normalmente são grandes programas C (ou algumas vezes C++), que consistem de muitos fragmentos escritos por muitos programadores. O

ambiente usado para desenvolver sistemas operacionais é muito diferente daquele a que estão acostumados os indivíduos (como os estudantes) quando escrevem pequenos programas Java. Esta seção é uma tentativa de fazer uma breve introdução ao mundo da escrita de sistemas operacionais para programadores de Java modestos.

1.8.1 | A linguagem C

Este não é um guia para a linguagem C, mas um breve resumo das diferenças entre C e Java. A linguagem Java é baseada em C, por isso há muitas semelhanças entre as duas. Ambas são linguagens imperativas com tipos de dados, variáveis e comandos de controle, por exemplo. Os tipos de dados primitivos em C são números inteiros (inclusive curtos e longos), caracteres e números de ponto flutuante. Tipos de dados compostos podem ser construídos usando arranjos (*arrays*), estruturas (*structures*) e uniões (*unions*). Os comandos de controle em C são semelhantes aos de Java, inclusive os comandos if, switch, for e while. Funções e parâmetros são quase os mesmos em ambas as linguagens.

Uma característica de C que Java não tem são os ponteiros explícitos. Um **ponteiro** é uma variável que aponta para uma variável ou estrutura de dados (isto é, contém o endereço dela). Considere as linhas

```
char c1, c2, *p;
c1 = 'c';
p = &c1;
c2 = *p;
```

que declaram que *c1* e *c2* são variáveis do tipo caracter e que *p* é uma variável que aponta para um caractere (isto é, contém o endereço dele). A primeira atribuição armazena o código ASCII para o caractere 'c' na variável *c1*. A segunda atribui o endereço de *c1* à variável ponteiro *p*. A terceira atribui os conteúdos da variável apontada por *p* à variável *c2*; desse modo, depois que esses comandos são executados, *c2* também contém o código ASCII para 'c'. Na teoria, os ponteiros são tipificados; assim, programadores não deveriam atribuir o endereço de um número em ponto flutuante a um ponteiro de caractere, mas, na prática, os compiladores aceitam essas atribuições, embora algumas vezes com uma advertência. Os ponteiros são uma construção muito eficaz, mas também uma grande fonte de erros quando usados sem cuidado.

C não tem, entre outras coisas, cadeia de caracteres incorporadas, threads, pacotes, classes, objetos, segurança de tipos (*type safety*) e coletor de lixo. O último é um defeito fatal para sistemas operacionais. Todo o armazenamento em C é estático ou explicitamente alocado e liberado pelo programador, normalmente com a função biblioteca *malloc* e *free*. É a última propriedade — controle total do programador sobre a memória — com ponteiros explícitos que torna a linguagem C atraente para a escrita de sistemas operacionais. Os sistemas operacionais são, até certo ponto, sistemas em tempo real, mesmo no caso de sistemas de propósito geral. Quando ocorre uma interrupção, o sistema operacional pode ter apenas alguns microssegundos para executar alguma ação ou perder informações críticas. A entrada em operação do coletor de lixo em um momento arbitrário é intolerável.

1.8.2 | Arquivos de cabeçalho

Um projeto de sistema operacional geralmente consiste em alguns diretórios, cada um contendo muitos arquivos *.c*, que contêm o código para alguma parte do sistema, com alguns arquivos de cabeçalho (*header*) *.h* que contêm declarações e definições usadas por um ou mais arquivos de código. Arquivos de cabeçalho também podem incluir **macros**, como

```
#define BUFFER_SIZE 4096
```

que permitem que o programador nomeie constantes, de modo que, quando o *BUFFER_SIZE* for usado no código, seja substituído durante a compilação pelo número 4096. Uma boa prática de programação em C é nomear todas as constantes exceto 0, 1 e –1 e, algumas vezes, nomear até mesmo essas. As macros podem ter parâmetros, como

```
#define max(a, b) (a > b ? a : b)
```

que permitem que o programador escreva

```
i = max(j, k+1)
```

e obtenha

```
i = (j > k+1 ? j: k+1)
```

para armazenar a maior parte de *j* e *k+1* em *i*. Os cabeçalhos também podem conter compilação condicional, por exemplo

```
#ifdef PENTIUM
intel_int_ack();
#endif
```

que compila uma chamada à função *intel_int_ack* apenas se a macro *PENTIUM* for definida. A compilação condicional é muito utilizada para isolar códigos dependentes de arquitetura, de modo que certo código seja inserido apenas quando o sistema for compilado no Pentium, outro código seja inserido apenas quando o sistema for compilado no SPARC, e assim por diante. Um arquivo *.c* pode incluir conjuntamente zero ou mais arquivos de cabeçalho usando a instrução *#include*. Há também muitos arquivos de cabeçalho comuns a quase todo *.c* e esses são armazenados em um diretório central.

1.8.3 | Grandes projetos de programação

Para construir um sistema operacional, cada *.c* é compilado em um **arquivo-objeto** pelo compilador C. Arquivos-objeto, que têm o sufixo *.o*, contêm instruções binárias para a máquina-destino. Eles serão executados posteriormente

pela CPU. Não há nada semelhante ao Java byte code e, o código Java compilado para a JVM, na linguagem C.

O primeiro passo do compilador C é chamado **pré-processador C.** Quando lê cada arquivo *.c*, toda vez que atinge uma instrução *#include*, ele captura o arquivo de cabeçalho nomeado e o processa, expandindo macros, controlando a compilação adicional (e outras coisas) e transferindo os resultados ao próximo passo do compilador como se eles estivessem fisicamente incluídos.

Uma vez que os sistemas operacionais são muito grandes (cinco milhões de linhas de código não são incomuns), compilar tudo novamente a cada vez que um arquivo fosse alterado seria inaceitável. Por outro lado, alterar um arquivo de cabeçalho importante que esteja incluído em milhares de outros arquivos requer nova compilação desses arquivos. Acompanhar quais arquivos-objeto dependem de quais arquivos de cabeçalho é totalmente impraticável sem uma ferramenta de auxílio.

Felizmente, os computadores são muito bons em fazer exatamente esse tipo de coisa. Nos sistemas UNIX, há um programa chamado *make* (com numerosas variantes, como *gmake, pmake* etc.) que lê o *Makefile*, que lhe diz quais arquivos são dependentes de quais outros arquivos. O que o *make* faz é identificar quais os arquivos-objeto requeridos imediatamente para construir o binário do sistema operacional e, para cada um, verificar se algum dos arquivos dos quais depende (o código e os cabeçalhos) foi modificado após a última criação do arquivo-objeto. Em caso afirmativo, esse arquivo-objeto deve ser recompilado. Quando *make* tiver determinado que arquivos *.c* devem ser recompilados, invoca um compilador C para compilá-los novamente, reduzindo assim o número de compilações ao mínimo. Em grandes projetos, a criação do *Makefile* é propensa a erro e, por isso, há ferramentas que o fazem automaticamente.

Uma vez que todos os arquivos *.o* estejam prontos, são transferidos a um programa chamado **linker** (ligador) para combinar todos eles em um único arquivo binário executável. Quaisquer funções de biblioteca chamadas também são incluídas nesse ponto, referências entre funções são resolvidas e endereços de máquinas são relocados conforme a necessidade. Quando a ligação é concluída, o resultado é um programa executável, tradicionalmente chamado *a.out* em sistemas UNIX. Os vários componentes desse processo são ilustrados na Figura 1.27 para um programa com três arquivos C e dois arquivos de cabeçalho. Embora nossa discussão seja sobre o desenvolvimento de sistemas operacionais, tudo isso se aplica ao desenvolvimento de qualquer programa de grande porte.

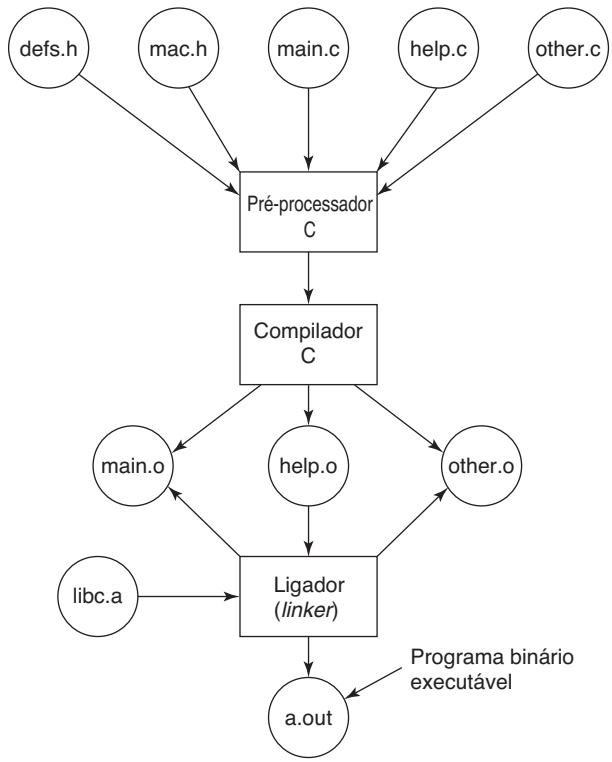

Figura 1.27 O processo de compilação C e arquivos de cabeçalho para criar um arquivo executável.

1.8.4 | O modelo de execução

Uma vez que o sistema operacional binário tenha sido ligado, o computador pode ser reiniciado e o novo sistema operacional carregado. Ao ser executado, pode carregar de modo dinâmico pedaços que não foram estaticamente incluídos no sistema binário, como drivers de dispositivo e sistemas de arquivo. No tempo de execução, o sistema operacional pode consistir de segmentos múltiplos, para o texto (o código do programa), os dados e a pilha. O segmento de texto é normalmente imutável, não mudando durante a execução. O segmento de dados começa com determinado tamanho e é inicializado com determinados valores, mas pode mudar e crescer conforme a necessidade. A pilha está inicialmente vazia, mas cresce e encolhe à medida que as funções são chamadas e retornam. Muitas vezes o segmento de texto é colocado próximo à parte inferior da memória, os segmentos de dados logo acima, com a capacidade de crescer para cima, e o segmento de pilha em um endereço virtual alto, com a capacidade de crescer para baixo (em direção ao endereço zero de memória), mas sistemas diferentes funcionam de modos diferentes.

Em todos os casos, o código do sistema operacional é executado diretamente pelo hardware, sem interpretadores e sem compilação just-in-time, como é normal no caso da linguagem Java.

1.9 Pesquisas em sistemas operacionais

A ciência da computação avança muito rapidamente e é difícil dizer para onde se dirige. Pesquisadores em universidades e em laboratórios industriais estão sempre desenvolvendo novas ideias; algumas delas não levam a nada, porém outras tornam-se a base para futuros produtos

e causam altos impactos à indústria e aos consumidores. Fazer uma retrospectiva de como as coisas evoluíram é mais fácil do que predizer como evoluirão. Separar o joio do trigo é muito difícil, porque muitas vezes uma ideia leva de 20 a 30 anos para causar algum impacto.

Por exemplo, quando o presidente norte-americano Eisenhower criou a ARPA (Advanced Research Projects Agency, a agência de pesquisas em projetos avançados do Departamento de Defesa), em 1958, ele estava tentando resolver o problema da influência avassaladora que o Exército detinha sobre o orçamento de pesquisas do Pentágono em detrimento da Marinha e da Força Aérea. Ele não estava tentando inventar a Internet. Mas uma das coisas que a ARPA fez foi financiar algumas pesquisas em universidades sobre o então obscuro conceito de comutação de pacotes, que rapidamente levou à primeira rede experimental de comutação de pacotes, a ARPANET. Essa rede nasceu em 1969. Antes, porém, outras redes de pesquisa financiadas pela ARPA foram conectadas à ARPANET, e assim nasceu a Internet. A Internet foi usada durante 20 anos por pesquisadores acadêmicos para trocar mensagens eletrônicas. No início da década de 1990, Tim Berners-Lee concebeu a World Wide Web em seu laboratório de pesquisas no CERN em Genebra, e Marc Andreesen projetou um visualizador (browser) gráfico para essa rede mundial na Universidade de Illinois. De um momento para outro, a Internet estava repleta de adolescentes batendo papo. O presidente Eisenhower está, provavelmente, rolando em sua sepultura.

As pesquisas em sistemas operacionais também têm levado a mudanças dramáticas nos sistemas práticos. Conforme discutido anteriormente, os primeiros computadores comerciais eram todos sistemas em lote (*batch*), até que o MIT inventou o tempo compartilhado interativo no início dos anos 1960. Os computadores eram todos baseados em texto, até que Doug Engelbart inventou o mouse e a interface gráfica com o usuário (GUI) no Stanford Institute of Research no final da década de 1960. Quem de vocês sabe o que veio depois?

Nesta seção e em outras afins, por todo este livro, conheceremos algumas das pesquisas em sistemas operacionais dos últimos cinco ou dez anos, apenas para termos uma ideia do que pode surgir no horizonte. Esta introdução certamente não é abrangente e baseia-se amplamente em artigos publicados nos melhores periódicos e seminários, ideias que, pelo menos, sobreviveram a um rigoroso processo de avaliação antes de serem publicadas. A maioria dos artigos citados nas seções de pesquisa foi publicada pela ACM, pela IEEE Computer Society ou pela USENIX e está disponível na Internet aos membros (estudantes) dessas organizações. Para mais informações sobre essas organizações e suas bibliotecas digitais, consulte os sites da Web a seguir:

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

Quase todos os pesquisadores da área sabem que os sistemas operacionais atuais são maciços, rígidos, não confiáveis, inseguros e cheios de erros, alguns mais que outros (os *nomes não são citados aqui para proteger os culpados*). Consequentemente, há muitas pesquisas sobre como construir sistemas operacionais melhores. Recentemente foram publicados trabalhos sobre novos sistemas operacionais (Krieger et al., 2006), estrutura de sistemas operacionais (Fassino et al., 2002), precisão de sistemas operacionais (Elphinstone et al., 2007; Kumar e Li, 2002; Yang et al., 2006), confiabilidade de sistemas operacionais (Swift et al., 2006; LeVasseur et al., 2004), máquinas virtuais (Barham et al., 2003; Garfinkel et al., 2003; King et al., 2003; Whitaker et al., 2002), vírus e vermes (*worms*) (Costa et al., 2005; Portokalidis et al., 2006; Tucek et al., 2007; Vrable et al., 2005), erros e depuração (Chou et al., 2001; King et al., 2005), hyper threading e multithreading (Fedorova, 2005; Bulpin e Pratt, 2005) e comportamento do usuário (Yu et al., 2006), entre muitos outros tópicos.

1.10 Delineamento do restante deste livro

Acabamos de dar uma panorâmica nos sistemas operacionais. É o momento, então, de entrarmos nos detalhes. Conforme mencionamos anteriormente, do ponto de vista do programador, a principal finalidade de um sistema operacional é fornecer algumas abstrações fundamentais, das quais as mais importantes são processos e threads, espaços de endereçamento e arquivos. Portanto, os três capítulos seguintes são dedicados a esses tópicos cruciais.

O Capítulo 2 trata de processos e threads. Nele são discutidas suas propriedades e como eles se comunicam entre si. São dados também vários exemplos detalhados sobre como funciona a comunicação entre processos e como evitar algumas ciladas.

No Capítulo 3, estudaremos em detalhes os espaços de endereçamento e seus auxiliares, o gerenciamento de memória. O importante tópico da memória virtual será examinado, com conceitos estreitamente relacionados, como paginação e segmentação.

Em seguida, no Capítulo 4, tratamos do tema importantíssimo de sistemas de arquivos. Em grande medida, o que o usuário vê é, em sua maior parte, o sistema de arquivos. Examinaremos tanto a interface como a implementação do sistema de arquivos.

A entrada/saída é abordada no Capítulo 5. Os conceitos de independência e dependência ao dispositivo são estudados. Vários dispositivos importantes — incluindo discos, teclados e monitores — são usados como exemplos.

O Capítulo 6 é sobre impasses (*deadlocks*). Descrevemos brevemente o que são impasses, mas há muito mais a dizer sobre eles. São discutidas também soluções preventivas.

Nesse ponto termina nosso estudo sobre os princípios básicos de sistemas operacionais com uma única CPU. Con-

tudo, há mais a dizer, especialmente sobre tópicos avançados. No Capítulo 7, então, nosso estudo avança, tratando de sistemas multimídia, que têm várias propriedades e diversos requisitos que diferem dos sistemas operacionais convencionais. Entre outros itens, o escalonamento e o sistema de arquivos são afetados pela natureza da multimídia. Outro tópico avançado são os sistemas com múltiplos processadores, incluindo multiprocessadores, computadores paralelos e sistemas distribuídos. Esses assuntos são analisados no Capítulo 8.

Um tema importantíssimo é a segurança do sistema operacional, que é vista no Capítulo 9. Entre os tópicos discutidos nesse capítulo, estão as ameaças (por exemplo, de vírus e de vermes). Também são abordados mecanismos de proteção e modelos de segurança.

Em seguida, estudamos alguns sistemas operacionais reais. São eles: Linux (Capítulo 10), Windows Vista (Capítulo 11) e Symbian (Capítulo 12). O livro termina com algumas reflexões sobre projeto de sistemas operacionais no Capítulo 13.

1.11 Unidades métricas

Para evitar qualquer confusão, é melhor dizer claramente que, neste livro, como na ciência da computação em geral, são usadas unidades métricas, e não as tradicionais unidades inglesas (sistema *furlong-stone-fortnight*). Os principais prefixos métricos são relacionados na Tabela 1.4. Normalmente esses prefixos são abreviados por suas primeiras letras, com as unidades maiores que 1 em letras maiúsculas. Assim, um banco de dados de 1 TB ocupa 10^{12} bytes de memória e um tique de relógio de 100 pseg (ou 100 ps) ocorre a cada 10^{-10} segundos. Como ambos os prefixos, mili e micro, começam com a letra 'm', foi necessário fazer uma escolha. Normalmente, 'm' é para mili e 'μ' (a letra grega *mu*) é para micro.

Convém também observar que, para medir tamanhos de memória, as unidades têm significados um pouco diferentes. O quilo corresponde a 2^{10} (1.024), e não a 10^3 (1.000), pois as memórias são sempre expressas em potências de 2. Assim, uma memória de 1 KB contém 1.024 bytes, e não 1.000 bytes. De maneira similar, uma memória de 1 MB contém 2^{20} (1.048.576) bytes, e uma memória de 1 GB contém 2^{30} (1.073.741.824) bytes. Contudo, uma linha de comunicação de 1 Kbps transmite a 1.000 bits por segundo, e uma rede local (LAN) de 10 Mbps transmite a 10.000.000 bits por segundo, pois essas velocidades não são potências de 2. Infelizmente, muitas pessoas tendem a misturar esses dois sistemas, especialmente em tamanhos de disco. Para evitar ambiguidade, neste livro usaremos os símbolos KB, MB e GB para 2^{10}, 2^{20} e 2^{30} bytes, respectivamente, e os símbolos Kbps, Mbps e Gbps para 10^3, 10^6, 10^9 bits/segundo, respectivamente.

1.12 Resumo

Os sistemas operacionais podem ser analisados de dois pontos de vista: como gerenciadores de recursos e como máquinas estendidas. Como gerenciador de recursos, o trabalho dos sistemas operacionais é gerenciar eficientemente as diferentes partes do sistema. Sob o ponto de vista da máquina estendida, sua tarefa é oferecer aos usuários abstrações que sejam mais convenientes ao uso do que a máquina real. Elas incluem processos, espaços de endereçamento e arquivos.

Os sistemas operacionais têm uma longa história, que começou quando eles substituíram o operador e vai até os sistemas modernos de multiprogramação, com destaques para os sistemas em lote (*batch*), sistemas de multiprogramação e sistemas de computadores pessoais.

Como os sistemas operacionais interagem intimamente com o hardware, algum conhecimento sobre o hardware

Exp.	Explícito	Prefixo	Exp.	Explícito	Prefixo
10^{-3}	0,001	mili	10^{3}	1.000	quilo
10^{-6}	0,000001	micro	10^{6}	1.000.000	mega
10^{-9}	0,000000001	nano	10^{9}	1.000.000.000	giga
10^{-12}	0,000000000001	pico	10^{12}	1.000.000.000.000	tera
10^{-15}	0,000000000000001	femto	10^{15}	1.000.000.000.000.000	peta
10^{-18}	0,000000000000000001	atto	10^{18}	1.000.000.000.000.000.000	exa
10^{-21}	0,000000000000000000001	zepto	10^{21}	1.000.000.000.000.000.000.000	zetta
10^{-24}	0,000000000000000000000001	yocto	10^{24}	1.000.000.000.000.000.000.000.000	yotta

Tabela 1.4 Os principais prefixos métricos.

de computadores é útil para entendê-los. Os computadores são constituídos de processadores, memórias e dispositivos de E/S. Essas partes são conectadas por barramentos.

Os conceitos básicos sobre os quais todos os sistemas operacionais são construídos são: processos, gerenciamento de memória, gerenciamento de E/S, sistema de arquivos e segurança. Trata-se de cada um desses conceitos em um capítulo subsequente.

O coração de qualquer sistema operacional é o conjunto de chamadas de sistema com o qual ele pode lidar. Essas chamadas dizem o que o sistema operacional realmente faz. Para o UNIX, estudamos quatro grupos de chamadas de sistema. O primeiro relaciona-se com a criação e a finalização de processos. O segundo grupo é para leitura e escrita em arquivos. O terceiro é voltado ao gerenciamento de diretórios. O quarto grupo contém chamadas diversas.

Os sistemas operacionais podem ser estruturados de várias maneiras. As mais comuns são as seguintes: como sistemas monolíticos, como uma hierarquia de camadas, como um micronúcleo, como um sistema de máquina virtual, como um exonúcleo ou por meio do modelo cliente–servidor.

Problemas

1. O que é multiprogramação?
2. O que é a técnica de spooling? Você acha que computadores pessoais avançados terão o spooling como uma característica-padrão no futuro?
3. Nos primeiros computadores, todo byte de dados lido ou escrito era tratado pela CPU (isto é, não havia DMA). Quais as implicações disso para a multiprogramação?
4. A ideia de família de computadores foi introduzida nos anos 1960 com os computadores de grande porte IBM System/360. Essa ideia está morta e sepultada ou ainda vive?
5. Uma razão para a demora da adoção das interfaces gráficas GUI era o custo do hardware necessário para dar suporte a elas. De quanta RAM de vídeo se precisa para dar suporte a uma tela de texto monocromática com 25 linhas × 80 colunas de caracteres? Quanto é necessário para dar suporte a um mapa de bits com 1.024 × 768 pixels de 24 bits? Qual é o custo dessa RAM em preços de 1980 (5 dólares/KB)? Quanto custa agora?
6. Há várias metas de projeto na construção de um sistema operacional; por exemplo, utilização de recursos, oportunidade, robustez etc. Dê um exemplo de duas metas de projeto que possam ser contraditórias.
7. Das instruções a seguir, quais só podem ser executadas em modo núcleo?
   (a) Desabilite todas as interrupções.
   (b) Leia o horário do relógio.
   (c) Altere o horário do relógio.
   (d) Altere o mapa de memória.
8. Considere um sistema que tem duas CPUs e cada CPU tem dois threads (hyperthreading). Suponha que três programas, *P0, P1* e *P2*, sejam iniciados com tempos de execução de 5, 10 e 20 ms, respectivamente. Quanto tempo seria necessário para concluir a execução desses programas? Suponha que todos os três programas sejam 100% CPU bound (limitados pela CPU, ou seja, que não fazem E/S), não bloqueiem durante a execução e não mudem de CPUs uma vez realizada a atribuição.
9. Um computador tem um pipeline de quatro estágios. Cada estágio leva o mesmo tempo para fazer seu trabalho — digamos, 1 ns. Quantas instruções por segundo essa máquina pode executar?
10. Considere um sistema de computador que tem memória cache, memória principal (RAM) e disco. O sistema operacional usa memória virtual. São necessários 2 ns para acessar uma palavra a partir da cache, 10 ns para acessar uma palavra a partir da RAM e 10 ms para acessar uma palavra a partir do disco. Se a taxa de acerto da cache é de 95% e a da memória principal (após uma falta de cache) é de 99%, qual é o tempo médio de acesso a uma palavra?
11. Um revisor alerta sobre um erro de ortografia no original de um livro-texto sobre sistemas operacionais que está para ser impresso. O livro tem aproximadamente 700 páginas, cada uma com 50 linhas de 80 caracteres. Quanto tempo será preciso para percorrer eletronicamente o texto no caso de a cópia estar em cada um dos níveis de memória da Figura 1.9? Para métodos de armazenamento interno, considere que o tempo de acesso é dado por caractere; para discos, considere que o tempo é por bloco de 1.024 caracteres; e para fitas, que o tempo dado é a partir do início dos dados com acesso subsequente na mesma velocidade que o acesso a disco.
12. Quando um programa de usuário faz uma chamada de sistema para ler ou escrever um arquivo em disco, ele fornece uma indicação de qual arquivo ele quer, um ponteiro para o buffer de dados e um contador. O controle é, então, transferido ao sistema operacional, que chama o driver apropriado. Suponha que o driver inicie o disco, termine e só volte quando uma interrupção ocorrer. No caso da leitura do disco, obviamente quem chama deverá ser bloqueado (pois não há dados para ele). E no caso da escrita no disco? Quem chama precisa ser bloqueado aguardando o final da transferência do disco?
13. O que é uma instrução trap? Explique seu uso em sistemas operacionais.
14. Qual é a diferença fundamental entre um trap e uma interrupção?
15. Por que é necessária uma tabela de processos em sistemas de compartilhamento de tempo? Essa tabela também é essencial em sistemas de computador pessoal (PC), nos quais existe apenas um processo, que detém o comando de toda a máquina até que ele termine?
16. Há alguma razão para se querer montar um sistema de arquivos em um diretório não vazio? Se há, qual é?