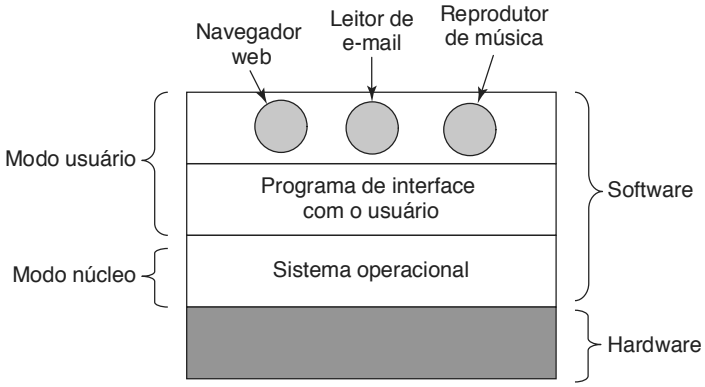
CAPÍTULO 1 INTRODUÇÃO

Um computador moderno consiste em um ou mais processadores, alguma memória principal, discos, impressoras, um teclado, um mouse, um monitor, interfaces de rede e vários outros dispositivos de entrada e saída. Como um todo, trata-se de um sistema complexo. Se todo programador de aplicativos tivesse de compreender como todas essas partes funcionam em detalhe, nenhum código jamais seria escrito. Além disso, gerenciar todos esses componentes e usá-los de maneira otimizada é um trabalho extremamente desafiador. Por essa razão, computadores são equipados com um dispositivo de software chamado de **sistema operacional**, cuja função é fornecer aos programas do usuário um modelo do computador melhor, mais simples e mais limpo, assim como lidar com o gerenciamento de todos os recursos mencionados. Sistemas operacionais é o assunto deste livro.

A maioria dos leitores já deve ter tido alguma experiência com um sistema operacional como Windows, Linux, FreeBSD, ou OS X, mas as aparências podem ser enganadoras. O programa com o qual os usuários interagem, normalmente chamado de **shell** (ou interpretador de comandos) quando ele é baseado em texto e de **GUI (Graphical User Interface)** quando ele usa ícones, na realidade não é parte do sistema operacional, embora use esse sistema para realizar o seu trabalho.

Uma visão geral simplificada dos principais componentes em discussão aqui é dada na Figura 1.1, em que vemos o hardware na parte inferior. Ele consiste em chips, placas, discos, um teclado, um monitor e objetos físicos similares. Em cima do hardware está o software. A maioria dos computadores tem dois modos de operação: modo núcleo e modo usuário. O sistema operacional, a peça mais fundamental de software, 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 for

FIGURA 1.1 Onde o sistema operacional se encaixa.



capaz de executar. O resto do software opera em **modo usuário**, no qual apenas um subconjunto das 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 neste livro. Ela exerce um papel crucial no modo como os sistemas operacionais funcionam.

O programa de interface com o usuário, shell ou GUI, é o nível mais inferior de software de modo usuário, e permite que ele inicie outros programas, como um navegador web, leitor de e-mail, ou reprodutor de música. Esses programas, também, utilizam bastante o sistema operacional.

O posicionamento do sistema operacional é mostrado na Figura 1.1. Ele opera diretamente sobre o hardware e proporciona a base para todos os outros softwares.

Uma distinção importante entre o sistema operacional e o software normal (modo usuário) é que se um usuário não gosta de um leitor de e-mail em particular, ele é livre para conseguir um leitor diferente ou escrever o seu próprio, se assim quiser; ele não é livre para escrever seu próprio tratador de interrupção de relógio, o qual faz parte do sistema operacional e é protegido por hardware contra tentativas dos usuários de modificá-lo.

Essa distinção, no entanto, às vezes é confusa em sistemas embarcados (que podem não ter o modo núcleo) ou interpretados (como os baseados em Java que usam interpretação, não hardware, para separar os componentes).

Também, em muitos sistemas há programas que operam em modo usuário, mas ajudam o sistema operacional ou realizam funções privilegiadas. Por exemplo, muitas vezes há um programa que permite aos usuários que troquem suas senhas. Não faz parte do sistema operacional e não opera em modo núcleo, mas claramente realiza uma função sensível e precisa ser protegido de uma maneira especial. Em alguns sistemas, essa ideia é levada ao extremo, e partes do que é tradicionalmente entendido como sendo o sistema operacional (como o sistema de arquivos) é executado em espaço do usuário. Em tais sistemas, é difícil traçar um limite claro. Tudo o que está sendo executado em modo núcleo faz claramente parte do sistema operacional, mas alguns programas executados fora dele também podem ser considerados uma parte dele, ou pelo menos estão associados a ele de modo próximo.

Os sistemas operacionais diferem de programas de usuário (isto é, de aplicativos) de outras maneiras além de onde estão localizados. Em particular, eles são enormes, complexos e têm vida longa. O código-fonte do coração de um sistema operacional como Linux ou Windows tem cerca de cinco milhões de linhas. Para entender o que isso significa, considere como seria imprimir cinco milhões de linhas em forma de livro, com 50 linhas por página e 1.000 páginas por volume. Seriam necessários 100 volumes para listar um sistema operacional desse tamanho — em essência, uma estante de livros inteira. Imagine-se conseguindo um trabalho de manutenção de um sistema operacional e no primeiro dia seu chefe o leva até uma estante de livros com o código e diz: "Você precisa aprender isso". E isso é apenas para a parte que opera no núcleo. Quando bibliotecas compartilhadas essenciais são incluídas, o Windows tem bem mais de 70 milhões de linhas de código ou 10 a 20 estantes de livros. E isso exclui softwares de aplicação básicos (do tipo Windows Explorer, Windows Media Player e outros).

Deve estar claro agora por que sistemas operacionais têm uma longa vida — eles são dificílimos de escrever, e tendo escrito um, o proprietário reluta em jogá-lo fora e começar de novo. Em vez disso, esses sistemas evoluem por longos períodos de tempo. O Windows 95/98/Me era basicamente um sistema operacional e o Windows NT/2000/XP/Vista/Windows 7 é outro. Eles são parecidos para os usuários porque a Microsoft tomou todo o cuidado para que a interface com o usuário do Windows 2000/XP/Vista/Windows 7 fosse bastante parecida com a do sistema que ele estava substituindo, majoritariamente o Windows 98. Mesmo assim, havia razões muito boas para a Microsoft livrar-se do Windows 98. Chegaremos a elas quando estudarmos o Windows em detalhe no Capítulo 11.

Além do Windows, o outro exemplo fundamental que usaremos ao longo deste livro é o UNIX e suas variáveis e clones. Ele também evoluiu com os anos, com versões como System V, Solaris e FreeBSD sendo derivadas do sistema original, enquanto o Linux possui um código base novo, embora muito proximamente modelado no UNIX e muito compatível com ele. Usaremos exemplos do UNIX neste livro e examinaremos o Linux em detalhes no Capítulo 10.

Neste capítulo abordaremos brevemente uma série de aspectos fundamentais dos sistemas operacionais, incluindo o que eles são, sua história, que tipos há por aí, alguns dos conceitos básicos e sua estrutura. Voltaremos

mais detalhadamente a muitos desses tópicos importantes em capítulos posteriores.

1.1 O que é um sistema operacional?

É difícil dizer com absoluta precisão o que é um sistema operacional, além de ele ser o software que opera em modo núcleo — e mesmo isso nem sempre é verdade. Parte do problema é que os sistemas operacionais realizam duas funções essencialmente não relacionadas: fornecer a programadores de aplicativos (e programas aplicativos, claro) um conjunto de recursos abstratos limpo em vez de recursos confusos de hardware, e gerenciar esses recursos de hardware. Dependendo de quem fala, você poderá ouvir mais a respeito de uma função do que de outra. Examinemos as duas então.

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 complicada de programar, especialmente para entrada/saída. Para deixar esse ponto mais claro, considere os discos rígidos modernos **SATA (Serial ATA)** usados na maioria dos computadores. Um livro (ANDERSON, 2007) descrevendo uma versão inicial da interface do disco — o que um programador deveria saber para usar o disco —, tinha mais de 450 páginas. Desde então, a interface foi revista múltiplas vezes e é mais complicada do que em 2007. É claro que nenhum programador são iria querer lidar com esse disco em nível de hardware. Em vez disso, um software, chamado **driver de disco**, lida com o hardware e fornece uma interface para ler e escrever blocos de dados, sem entrar nos detalhes. Sistemas operacionais contêm muitos drivers para controlar dispositivos de E/S.

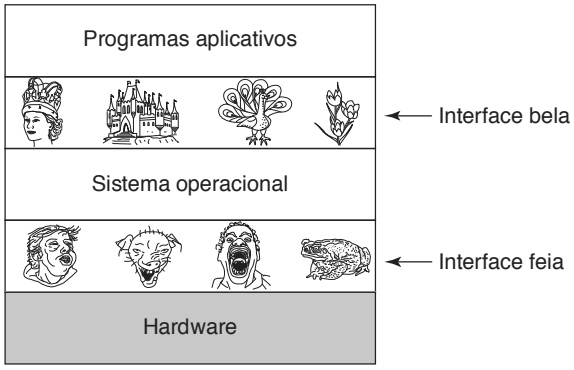
Mas mesmo esse nível é baixo demais para a maioria dos aplicativos. Por essa razão, todos os sistemas operacionais fornecem mais um nível de abstração para se utilizarem discos: arquivos. Usando essa abstração, os programas podem criar, escrever e ler arquivos, sem ter de lidar com os detalhes complexos de como o hardware realmente funciona.

Essa abstração é a chave para gerenciar toda essa complexidade. Boas abstrações transformam uma tarefa praticamente impossível em duas tarefas gerenciáveis. A primeira é definir e implementar as abstrações. A segunda é utilizá-las para solucionar o problema à mão. Uma abstração que quase todo usuário de computadores compreende é o arquivo, como mencionado anteriormente. Trata-se de um fragmento de informação útil, como uma foto digital, uma mensagem de e-mail, música ou página da web salvas. É muito mais fácil lidar com fotos, e-mails, músicas e páginas da web do que com detalhes de discos SATA (ou outros). A função dos sistemas operacionais é criar boas abstrações e então implementar e gerenciar os objetos abstratos criados desse modo. Neste livro, falaremos muito sobre abstrações. Elas são uma das chaves para compreendermos os sistemas operacionais.

Esse ponto é tão importante que vale a pena repeti-lo em outras palavras. Com todo o devido respeito aos engenheiros industriais que projetaram com tanto cuidado o Macintosh, o hardware é feio. Processadores reais, memórias, discos e outros dispositivos são muito complicados e apresentam interfaces difíceis, desajeitadas, idiossincráticas e inconsistentes para as pessoas que têm de escrever softwares para elas utilizarem. Às vezes isso decorre da necessidade de haver compatibilidade com a versão anterior do hardware, ou, então, é uma tentativa de poupar dinheiro. Muitas vezes, no entanto, os projetistas de hardware não percebem (ou não se importam) os problemas que estão causando ao software. Uma das principais tarefas dos sistemas operacionais é esconder o hardware e em vez disso apresentar programas (e seus programadores) com abstrações de qualidade, limpas, elegantes e consistentes com as quais trabalhar. Sistemas operacionais transformam o feio em belo, como mostrado na Figura 1.2.

Deve ser observado que os clientes reais dos sistemas operacionais são os programas aplicativos (via programadores de aplicativos, é claro). São eles que lidam diretamente com as abstrações fornecidas pela interface do usuário, seja uma linha de comandos (shell) ou

FIGURA 1.2 Sistemas operacionais transformam hardwares feios em belas abstrações.



uma interface gráfica. Embora as abstrações na interface com o usuário possam ser similares às abstrações fornecidas pelo sistema operacional, nem sempre esse é o caso. Para esclarecer esse ponto, considere a área de trabalho normal do Windows e o prompt de comando orientado a linhas. Ambos são programas executados no sistema operacional Windows e usam as abstrações que o Windows fornece, mas eles oferecem interfaces de usuário muito diferentes. De modo similar, um usuário de Linux executando Gnome ou KDE vê uma interface muito diferente daquela vista por um usuário Linux trabalhando diretamente sobre o X Window System, mas as abstrações do sistema operacional subjacente são as mesmas em ambos os casos.

Neste livro, esmiuçaremos o estudo das abstrações fornecidas aos programas aplicativos, mas falaremos bem menos sobre interfaces com o usuário. Esse é um assunto grande e importante, mas apenas perifericamente relacionado aos sistemas operacionais.

1.1.2 O sistema operacional como um gerenciador de recursos

O conceito de um sistema operacional como fundamentalmente fornecendo abstrações para programas aplicativos é uma visão top-down (abstração de cima para baixo). Uma visão alternativa, bottom-up (abstração de baixo para cima), sustenta que o sistema operacional está ali para gerenciar todas as partes de um sistema complexo. Computadores modernos consistem de processadores, memórias, temporizadores, discos, dispositivos apontadores do tipo mouse, interfaces de rede, impressoras e uma ampla gama de outros dispositivos. Na visão bottom-up, a função do sistema operacional é fornecer uma alocação ordenada e controlada dos processadores, memórias e dispositivos de E/S entre os vários programas competindo por eles.

Sistemas operacionais modernos permitem que múltiplos programas estejam na memória e sejam executados ao mesmo tempo. Imagine o que aconteceria se três programas executados em um determinado computador tentassem todos imprimir sua saída simultaneamente na mesma impressora. As primeiras linhas de impressão poderiam ser do programa 1, as seguintes do programa 2, então algumas do programa 3 e assim por diante. O resultado seria o caos absoluto. O sistema operacional pode trazer ordem para o caos em potencial armazenando temporariamente toda a saída destinada para a impressora no disco. Quando um programa é finalizado, o sistema operacional pode então copiar a sua saída do arquivo de disco onde ele foi armazenado para a impressora, enquanto ao mesmo tempo o outro programa pode continuar a gerar mais saída, alheio ao fato de que a saída não está realmente indo para a impressora (ainda).

Quando um computador (ou uma rede) tem mais de um usuário, a necessidade de gerenciar e proteger a memória, dispositivos de E/S e outros recursos é ainda maior, tendo em vista que os usuários poderiam interferir um com o outro de outra maneira. Além disso, usuários muitas vezes precisam compartilhar não apenas o hardware, mas a informação (arquivos, bancos de dados etc.) também. Resumindo, essa visão do sistema operacional sustenta que a sua principal função é manter um controle sobre quais programas estão usando qual recurso, conceder recursos requisitados, contabilizar o seu uso, assim como mediar requisições conflitantes de diferentes programas e usuários.

O gerenciamento de recursos inclui a **multiplexação** (compartilhamento) de recursos de duas maneiras diferentes: no tempo e no espaço. Quando um recurso é multiplexado no tempo, diferentes programas ou usuários se revezam usando-o. Primeiro, um deles usa o recurso, então outro e assim por diante. Por exemplo, com apenas uma CPU e múltiplos programas querendo ser executados nela, o sistema operacional primeiro aloca a CPU para um programa, então, após ele ter sido executado por tempo suficiente, outro programa passa a fazer uso da CPU, então outro, e finalmente o primeiro de novo. Determinar como o recurso é multiplexado no tempo — quem vai em seguida e por quanto tempo — é a tarefa do sistema operacional. Outro exemplo da multiplexação no tempo é o compartilhamento da impressora. Quando múltiplas saídas de impressão estão na fila para serem impressas em uma única impressora, uma decisão tem de ser tomada sobre qual deve ser impressa em seguida.

O outro tipo é a multiplexação de espaço. Em vez de os clientes se revezarem, cada um tem direito a uma parte do recurso. Por exemplo, a memória principal é normalmente dividida entre vários programas sendo executados, de modo que cada um pode ser residente ao mesmo tempo (por exemplo, a fim de se revezar usando a CPU). Presumindo que há memória suficiente para manter múltiplos programas, é mais eficiente manter vários programas na memória ao mesmo tempo do que dar a um deles toda ela, especialmente se o programa precisa apenas de uma pequena fração do total. É claro, isso gera questões de justiça, proteção e assim por diante, e cabe ao sistema operacional solucioná-las. Outro recurso que é multiplexado no espaço é o disco. Em muitos sistemas um único disco pode conter arquivos de

muitos usuários ao mesmo tempo. Alocar espaço de disco e controlar quem está usando quais blocos do disco é uma tarefa típica do sistema operacional.

1.2 História dos sistemas operacionais

Sistemas operacionais têm evoluído ao longo dos anos. Nas seções a seguir examinaremos brevemente alguns dos destaques dessa evolução. Tendo em vista que os sistemas operacionais estiveram historicamente muito vinculados à arquitetura dos computadores na qual eles são executados, examinaremos sucessivas gerações de computadores para ver como eram seus sistemas operacionais. Esse mapeamento de gerações de sistemas operacionais em relação às gerações de computadores é impreciso, mas proporciona alguma estrutura onde de outra maneira não haveria nenhuma.

A progressão apresentada a seguir é em grande parte cronológica, embora atribulada. Novos desenvolvimentos não esperaram que os anteriores tivessem terminado adequadamente antes de começarem. Houve muita sobreposição, sem mencionar muitas largadas falsas e becos sem saída. Tome-a como um guia, não como a palavra final.

O primeiro computador verdadeiramente digital foi projetado pelo matemático inglês Charles Babbage (1792–1871). Embora Babbage tenha gasto a maior parte de sua vida e fortuna tentando construir a "máquina analítica", nunca conseguiu colocá-la para funcionar para valer porque ela era puramente mecânica, e a tecnologia da época não conseguia produzir as rodas, acessórios e engrenagens de alta precisão de que ele precisava. Desnecessário dizer que a máquina analítica não tinha um sistema operacional.

Como um dado histórico interessante, Babbage percebeu que ele precisaria de um software para sua máquina analítica, então ele contratou uma jovem chamada Ada Lovelace, que era a filha do famoso poeta inglês Lord Byron, como a primeira programadora do mundo. A linguagem de programação Ada® é uma homenagem a ela.

1.2.1 A primeira geração (1945-1955): válvulas

Após os esforços malsucedidos de Babbage, pouco progresso foi feito na construção de computadores digitais até o período da Segunda Guerra Mundial, que estimulou uma explosão de atividade. O professor John Atanasoff e seu aluno de graduação Clifford Berry construíram o que hoje em dia é considerado o primeiro computador digital funcional na Universidade do Estado de Iowa. Ele usava 300 válvulas. Mais ou menos na mesma época, Konrad Zuse em Berlim construiu o computador Z3 a partir de relés eletromagnéticos. Em 1944, o Colossus foi construído e programado por um grupo de cientistas (incluindo Alan Turing) em Bletchley Park, Inglaterra, o Mark I 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, outros usavam válvulas e ainda outros eram programáveis, mas todos eram muito primitivos e levavam segundos para realizar mesmo o cálculo mais simples.

No início, um único grupo de pessoas (normalmente engenheiros) projetava, construía, programava, operava e mantinha cada máquina. Toda a programação era feita em código de máquina absoluto, ou, pior ainda, ligando circuitos elétricos através da conexão de milhares de cabos a painéis de ligações para controlar as funções básicas da máquina. Linguagens de programação eram desconhecidas (mesmo a linguagem de montagem era desconhecida). Ninguém tinha ouvido falar ainda de sistemas operacionais. O modo usual de operação consistia na reserva pelo programador de um bloco de tempo na ficha de registro na parede, então ele descer até a sala de máquinas, inserir seu painel de programação no computador e passar as horas seguintes torcendo para que nenhuma das cerca de 20.000 válvulas queimasse durante a operação. Virtualmente todos os problemas eram cálculos numéricos e matemáticos diretos e simples, como determinar tabelas de senos, cossenos e logaritmos, ou calcular trajetórias de artilharia.

No início da década de 1950, a rotina havia melhorado de certa maneira com a introdução dos cartões perfurados. Era possível agora escrever programas em cartões e lê-los em vez de se usarem painéis de programação; de resto, o procedimento era o mesmo.

1.2.2 A segunda geração (1955-1965): transistores e sistemas em lote (batch)

A introdução do transistor em meados dos anos 1950 mudou o quadro radicalmente. Os computadores tornaram-se de tal maneira confiáveis que podiam ser fabricados e vendidos para clientes dispostos a pagar por eles com a expectativa de que continuariam a funcionar por tempo suficiente para realizar algum trabalho útil. Pela primeira vez, havia uma clara separação entre projetistas, construtores, operadores, programadores e pessoal de manutenção.

Essas máquinas — então chamadas de **computadores de grande porte** (*mainframes*) —, ficavam isoladas

em salas grandes e climatizadas, especialmente designadas para esse fim, com equipes de operadores profissionais para operá-las. Apenas grandes corporações ou importantes agências do governo ou universidades conseguiam pagar o alto valor para tê-las. Para executar uma **tarefa** [isto é, um programa ou conjunto de programas], um programador primeiro escrevia o programa no papel [em FORTRAN ou em linguagem de montagem (assembly)], então o perfurava nos cartões. Ele levava então o maço de cartões até a sala de entradas e o passava a um dos operadores e ia tomar um café até que a saída estivesse pronta.

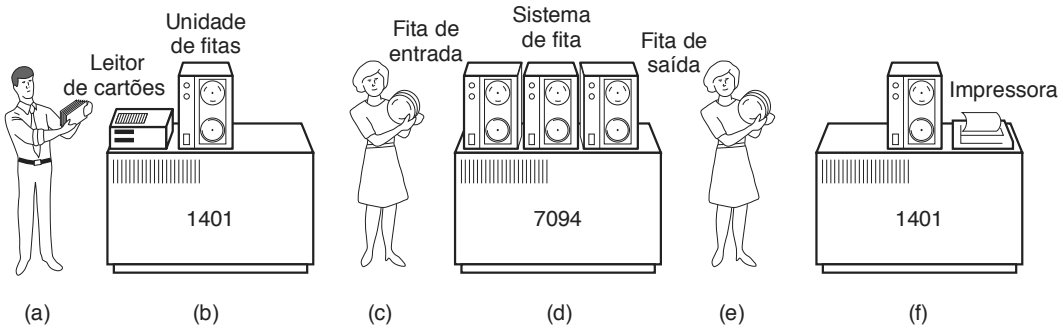
Quando o computador terminava qualquer tarefa que ele estivesse executando no momento, um operador ia até a impressora, pegava a sua saída e a levava até a sala de saídas a fim de que o programador pudesse buscá-la mais tarde. Então ele pegava um dos maços de cartões que haviam sido levados da sala de entradas e o colocava para a leitura. Se o compilador FORTRAN fosse necessário, o operador teria de tirá-lo de um porta-arquivos e fazer a leitura. Muito tempo do computador era desperdiçado enquanto os operadores caminhavam em torno da sala de máquinas.

Dado o alto custo do equipamento, não causa surpresa que as pessoas logo procuraram maneiras de reduzir o tempo desperdiçado. A solução geralmente adotada era o **sistema em lote** (*batch*). A ideia por trás disso era reunir um lote de tarefas na sala de entradas e então passá-lo para uma fita magnética usando um computador pequeno e (relativamente) barato, como um IBM 1401, que era muito bom na leitura de cartões, cópia de fitas e impressão de saídas, mas ruim em cálculos numéricos. Outras máquinas mais caras, como o IBM 7094, eram usadas para a computação real. Essa situação é mostrada na Figura 1.3.

Após cerca de uma hora coletando um lote de tarefas, os cartões eram lidos para uma fita magnética, que era levada até a sala de 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 então a executava. A saída era escrita em uma segunda fita, em vez de ser impressa. Após cada tarefa ter sido concluída, o sistema operacional automaticamente lia a tarefa seguinte da fita e começava a executá-la. Quando o lote inteiro estava pronto, o operador removia as fitas de entrada e saída, substituía a fita de entrada com o próximo lote e trazia a fita de saída para um 1401 para impressão **off-line** (isto é, não conectada ao computador principal).

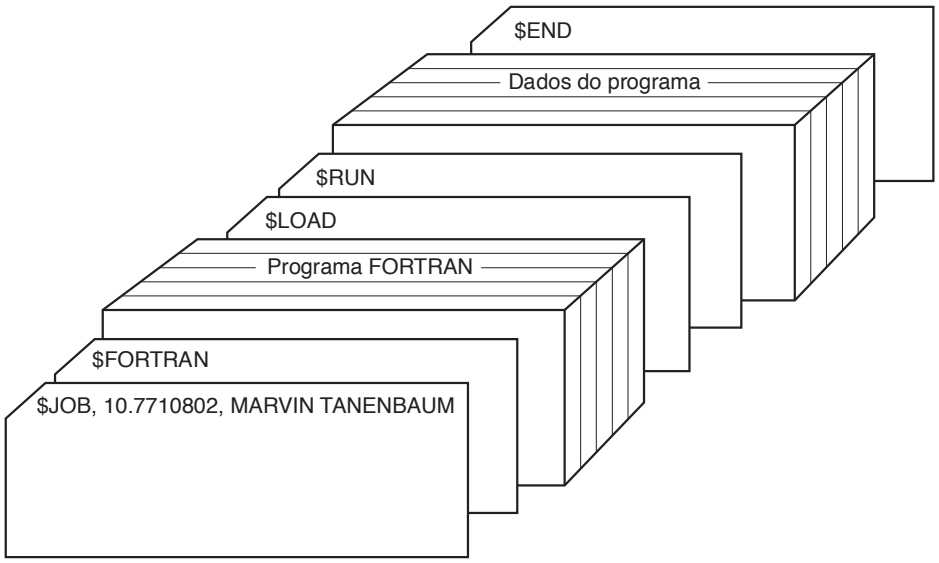
A estrutura de uma tarefa de entrada típica é mostrada na Figura 1.4. Ela começava com um cartão $JOB, especificando um tempo máximo de processamento em minutos, o número da conta a ser debitada e o nome do programador. Então vinha um cartão $FORTRAN, dizendo ao sistema operacional para carregar o compilador FORTRAN da fita do sistema. Ele era diretamente seguido pelo programa a ser compilado, e então um cartão $LOAD, direcionando o sistema operacional a carregar o programa-objeto recém-compilado. (Programas compilados eram muitas vezes escritos em fitas-rascunho e tinham de ser carregados explicitamente.) Em seguida vinha o cartão $RUN, dizendo ao sistema operacional para executar o programa com os dados em seguida. Por fim, o cartão $END marcava o término da tarefa. Esses cartões de controle primitivos foram os

FIGURA 1.3 Um sistema em lote (*batch*) antigo.



(a) Programadores levavam cartões para o 1401.
(b) O 1401 lia o lote de tarefas em uma 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.

FIGURA 1.4 Estrutura de uma tarefa FMS típica.



precursores das linguagens de controle de tarefas e interpretadores de comando modernos.

Os grandes computadores de segunda geração eram usados na maior parte para cálculos científicos e de engenharia, como solucionar as equações diferenciais parciais que muitas vezes ocorrem na física e na engenharia. Eles eram em grande parte programados em FORTRAN e linguagem de montagem. Sistemas operacionais típicos eram o FMS (o 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 da década de 1960, a maioria dos fabricantes de computadores tinha duas linhas de produto distintas e incompatíveis. Por um lado, havia os computadores científicos de grande escala, orientados por palavras, como o 7094, usados para cálculos numéricos complexos na ciência e engenharia. De outro, os computadores comerciais, orientados por caracteres, como o 1401, que eram amplamente usados para ordenação e impressão de fitas por bancos e companhias de seguro.

Desenvolver e manter duas linhas de produtos completamente diferentes era uma proposição cara para os fabricantes. Além disso, muitos clientes novos de computadores inicialmente precisavam de uma máquina pequena, no entanto mais tarde a sobreutilizavam e queriam uma máquina maior que executasse todos os seus programas antigos, porém mais rápido.

A IBM tentou solucionar ambos os problemas com uma única tacada introduzindo o System/360. O 360 era uma série de máquinas com softwares compatíveis, desde modelos do porte do 1401 a modelos muito maiores, mais potentes que o poderoso 7094. As máquinas diferiam apenas em preço e desempenho (memória máxima, velocidade do processador, número de dispositivos de E/S permitidos e assim por diante). Tendo em vista que todos tinham a mesma arquitetura e conjunto de instruções, programas escritos para uma máquina podiam operar em todas as outras — pelo menos na teoria. (Mas como Yogi Berra[1] teria dito: "Na teoria, a teoria e a prática são a mesma coisa; na prática, elas não são".) Tendo em vista que o 360 foi projetado para executar tanto computação científica (isto é, numérica) como comercial, uma única família de máquinas poderia satisfazer necessidades de todos os clientes. Nos anos seguintes, a IBM apresentou sucessores compatíveis com a linha 360, usando tecnologias mais modernas, conhecidas como as séries 370, 4300, 3080 e 3090. A zSeries é a descendente mais recente dessa linha, embora ela tenha divergido consideravelmente do original.

O IBM 360 foi a primeira linha importante de computadores a usar **CIs (circuitos integrados)** de pequena escala, proporcionando desse modo uma vantagem significativa na relação preço/desempenho sobre as máquinas de segunda geração, que foram construídas sobre transistores individuais. Foi um sucesso imediato, e a ideia de uma família de computadores compatíveis

[1] Ex-jogador de beisebol norte-americano conhecido por suas frases espirituosas. (N. T.)

foi logo adotada por todos os principais fabricantes. Os descendentes dessas máquinas ainda estão em uso nos centros de computadores atuais. Nos dias de hoje, eles são muitas vezes usados para gerenciar enormes bancos de dados (para sistemas de reservas de companhias aéreas, por exemplo) ou como servidores para sites da web que têm de processar milhares de requisições por segundo.

O forte da ideia da "família única" foi ao mesmo tempo seu maior ponto fraco. A intenção original era de que todo software, incluindo o sistema operacional, **OS/360**, funcionasse em todos os modelos. Ele tinha de funcionar em sistemas pequenos — que muitas vezes apenas substituíam os 1401 na cópia de cartões para fitas —, e em sistemas muito grandes, que muitas vezes substituíam os 7094 para realizar previsões do tempo e outras tarefas de computação pesadas. Ele tinha de funcionar bem em sistemas com poucos periféricos e naqueles com muitos periféricos, além de ambientes comerciais e ambientes científicos. Acima de tudo, ele tinha de ser eficiente para todos esses diferentes usos.

Não havia como a IBM (ou qualquer outra empresa) criar um software que atendesse a todas essas exigências conflitantes. O resultado foi um sistema operacional enorme e extraordinariamente complexo, talvez duas a três vezes maior do que o FMS. Ele consistia em milhões de linhas de linguagem de montagem escritas por milhares de programadores e continha dezenas de milhares de erros (bugs), que necessitavam de um fluxo contínuo de novas versões em uma tentativa de corrigi-los. Cada nova versão corrigia alguns erros e introduzia novos, de maneira que o número de erros provavelmente seguiu constante através do tempo.

Um dos projetistas do OS/360, Fred Brooks, subsequentemente escreveu um livro incisivo e bem-humorado (BROOKS, 1995) descrevendo as suas experiências com o OS/360. Embora seja impossível resumi-lo aqui, basta dizer que a capa mostra um rebanho de feras pré-históricas atoladas em um poço de piche. A capa de Silberschatz et al. (2012) faz uma analogia entre os sistemas operacionais e os dinossauros.

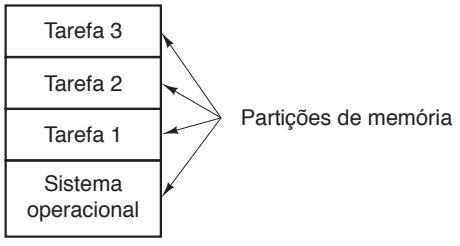
Apesar do tamanho enorme e dos problemas, o OS/360 e os sistemas operacionais de terceira geração similares produzidos por outros fabricantes de computadores na realidade proporcionaram um grau de satisfação relativamente bom para a maioria de seus clientes. Eles também popularizaram várias técnicas-chave ausentes nos sistemas operacionais de segunda geração. Talvez a mais importante dessas técnicas tenha sido a **multiprogramação.** No 7094, quando a tarefa atual fazia uma pausa para esperar por uma fita ou outra operação de E/S terminar, a CPU simplesmente ficava ociosa até o término da E/S. Para cálculos científicos com uso intenso da CPU, a E/S é esporádica, de maneira que o tempo ocioso não é significativo. Para o processamento de dados comercial, o tempo de espera de E/S pode muitas vezes representar de 80 a 90% do tempo total, de maneira que algo tem de ser feito para evitar que a CPU (cara) fique ociosa tanto tempo.

A solução encontrada foi dividir a memória em várias partes, com uma tarefa diferente em cada partição, como mostrado na Figura 1.5. Enquanto uma tarefa ficava esperando pelo término da E/S, outra podia usar a CPU. Se um número suficiente de tarefas pudesse ser armazenado na memória principal ao mesmo tempo, a CPU podia se manter ocupada quase 100% do tempo. Ter múltiplas tarefas na memória ao mesmo tempo de modo seguro exige um hardware especial para proteger cada uma contra interferências e transgressões por parte das outras, mas o 360 e outros sistemas de terceira geração eram equipados com esse hardware.

Outro aspecto importante presente nos sistemas operacionais de terceira geração foi a capacidade de transferir tarefas de cartões para o disco tão logo eles eram trazidos para a sala do computador. Então, sempre que uma tarefa sendo executada terminava, o sistema operacional podia carregar uma nova tarefa do disco para a partição agora vazia e executá-la. Essa técnica é chamada de **spooling** (da expressão **Simultaneous Peripheral Operation On Line**) e também foi usada para saídas. Com spooling, os 1401 não eram mais necessários, e muito do leva e traz de fitas desapareceu.

Embora sistemas operacionais de terceira geração fossem bastante adequados para grandes cálculos científicos e operações maciças de processamento de dados comerciais, eles ainda eram basicamente sistemas em lote. Muitos programadores sentiam saudades dos tempos de computadores de primeira geração quando eles tinham a máquina só para si por algumas horas e assim podiam corrigir eventuais erros em seus programas rapidamente. Com sistemas de terceira geração, o tempo

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



entre submeter uma tarefa e receber de volta a saída era muitas vezes de várias horas, então uma única vírgula colocada fora do lugar podia provocar a falha de uma compilação, e o desperdício de metade do dia do programador. Programadores não gostavam muito disso.

Esse desejo por um tempo de resposta rápido abriu o caminho para o **timesharing** (compartilhamento de tempo), uma variante da multiprogramação, na qual cada usuário tem um terminal on-line. Em um sistema de timesharing, se 20 usuários estão conectados e 17 deles estão pensando, falando ou tomando café, a CPU pode ser alocada por sua vez para as três tarefas que demandam serviço. Já que ao depurar programas as pessoas em geral emitem comandos curtos (por exemplo, compile um procedimento de cinco páginas)[2] em vez de comandos longos (por exemplo, ordene um arquivo de um milhão de registros), o computador pode proporcionar um serviço interativo rápido para uma série de usuários e talvez também executar tarefas de lote grandes em segundo plano quando a CPU estiver ociosa. O primeiro sistema de compartilhamento de tempo para fins diversos, o **CTSS (Compatible Time Sharing System** — Sistema compatível de tempo compartilhado), foi desenvolvido no M.I.T. em um 7094 especialmente modificado (CORBATÓ et al., 1962). No entanto, o timesharing não se tornou popular de fato até que o hardware de proteção necessário passou a ser utilizado amplamente durante a terceira geração.

Após o sucesso do sistema CTSS, o M.I.T., a Bell Labs e a General Electric (à época uma grande fabricante de computadores) decidiram embarcar no desenvolvimento de um "computador utilitário", isto é, uma máquina que daria suporte a algumas centenas de usuários simultâneos com compartilhamento de tempo. O modelo era o sistema de eletricidade — quando você precisa de energia elétrica, simplesmente conecta um pino na tomada da parede e, dentro do razoável, terá tanta energia quanto necessário. Os projetistas desse sistema, conhecido como **MULTICS** (**MULTiplexed Information and Computing Service** — Serviço de Computação e Informação Multiplexada), previram uma máquina enorme fornecendo energia computacional para todas as pessoas na área de Boston. A ideia de que máquinas 10.000 vezes mais rápidas do que o computador de grande porte GE-645 seriam vendidas (por bem menos de US$ 1.000) aos milhões apenas 40 anos mais tarde era pura ficção científica. Mais ou menos como a ideia de trens transatlânticos supersônicos submarinos hoje em dia.

O MULTICS foi um sucesso relativo. Ele foi projetado para suportar centenas de usuários em uma máquina apenas um pouco mais poderosa do que um PC baseado no 386 da Intel, embora ele tivesse muito mais capacidade de E/S. A ideia não é tão maluca como parece, tendo em vista que à época as pessoas sabiam como escrever programas pequenos e eficientes, uma habilidade que depois foi completamente perdida. Havia muitas razões para que o MULTICS não tomasse conta do mundo, dentre elas, e não menos importante, o fato de que ele era escrito na linguagem de programação PL/I, e o compilador PL/I estava anos atrasado e funcionava de modo precário quando enfim chegou. Além disso, o MULTICS era muito ambicioso para sua época, de certa maneira muito parecido com a máquina analítica de Charles Babbage no século XIX.

Resumindo, o MULTICS introduziu muitas ideias seminais na literatura da computação, mas transformá-lo em um produto sério e um grande sucesso comercial foi muito mais difícil do que qualquer um havia esperado. A Bell Labs abandonou o projeto, e a General Electric abandonou completamente o negócio dos computadores. Entretanto, o M.I.T. persistiu e finalmente colocou o MULTICS para funcionar. Em última análise ele foi vendido como um produto comercial pela empresa (Honeywell) que comprou o negócio de computadores da GE, e foi instalado por mais ou menos 80 empresas e universidades importantes mundo afora. Embora seus números fossem pequenos, os usuários do MULTICS eram muito leais. A General Motors, a Ford e a Agência de Segurança Nacional Norte-Americana, por exemplo, abandonaram os seus sistemas MULTICS apenas no fim da década de 1990, trinta anos depois de o MULTICS ter sido lançado e após anos de tentativas tentando fazer com que a Honeywell atualizasse o hardware.

No fim do século XX, o conceito de um computador utilitário havia perdido força, mas ele pode voltar para valer na forma da **computação na nuvem** (*cloud computing*), na qual computadores relativamente pequenos (incluindo smartphones, tablets e assim por diante) estejam conectados a servidores em vastos e distantes centros de processamento de dados onde toda a computação é feita com o computador local apenas executando a interface com o usuário. A motivação aqui é que a maioria das pessoas não quer administrar um sistema computacional cada dia mais complexo e detalhista, e preferem que esse trabalho seja realizado por uma equipe de profissionais, por exemplo, pessoas trabalhando para a empresa que opera o centro de processamento

[2] Usaremos os termos "procedimento", "sub-rotina" e "função" indistintamente ao longo deste livro. (N. A.)

de dados. O comércio eletrônico (*e-commerce*) já está evoluindo nessa direção, com várias empresas operando e-mails em servidores com múltiplos processadores aos quais as máquinas simples dos clientes se conectam de maneira bem similar à do projeto MULTICS.

Apesar da falta de sucesso comercial, o MULTICS teve uma influência enorme em sistemas operacionais subsequentes (especialmente UNIX e seus derivativos, FreeBSD, Linux, iOS e Android). Ele é descrito em vários estudos e em um livro (CORBATÓ et al., 1972; CORBATÓ, VYSSOTSKY, 1965; DALEY e DENNIS, 1968; ORGANICK, 1972; e SALTZER, 1974). Ele também tem um site ativo em <*www.multicians.org*>, com muitas informações sobre o sistema, seus projetistas e seus usuários.

Outro importante desenvolvimento durante a terceira geração foi o crescimento fenomenal dos minicomputadores, começando com o DEC PDP-1 em 1961. O PDP-1 tinha apenas 4K de palavras de 18 bits, mas a US$ 120.000 por máquina (menos de 5% do preço de um 7094), vendeu como panqueca. Para determinado tipo de tarefas não numéricas, ele era quase tão rápido quanto o 7094 e deu origem a toda uma nova indústria. Ele foi logo seguido por uma série de outros PDPs (diferentemente da família IBM, todos incompatíveis), culminando no PDP-11.

Um dos cientistas de computação no Bell Labs que havia trabalhado no projeto MULTICS, Ken Thompson, descobriu subsequentemente um minicomputador pequeno PDP-7 que ninguém estava usando e decidiu escrever uma versão despojada e para um usuário do MULTICS. Esse trabalho mais tarde desenvolveu-se no sistema operacional **UNIX**, que se tornou popular no mundo acadêmico, em agências do governo e em muitas empresas.

A história do UNIX já foi contada em outras partes (por exemplo, SALUS, 1994). Parte da história será apresentada no Capítulo 10. Por ora, basta dizer que graças à ampla disponibilidade do código-fonte, várias organizações desenvolveram suas próprias versões (incompatíveis), o que levou ao caos. Duas versões importantes foram desenvolvidas, o **System V**, da AT&T, e o **BSD** (**Berkeley Software Distribution** — distribuição de software de Berkeley) da Universidade da Califórnia, em Berkeley. Elas tinham variantes menores também. Para tornar possível escrever programas que pudessem ser executados em qualquer sistema UNIX, o IEEE desenvolveu um padrão para o UNIX, chamado **POSIX** (*Portable Operating System Interface* — interface portátil para sistemas operacionais), ao qual a maioria das versões do UNIX dá suporte hoje em dia. O POSIX define uma interface minimalista de chamadas de sistema à qual os sistemas UNIX em conformidade devem dar suporte. Na realidade, alguns outros sistemas operacionais também dão suporte hoje em dia à interface POSIX.

Como um adendo, vale a pena mencionar que, em 1987, o autor lançou um pequeno clone do UNIX, chamado **MINIX**, para fins educacionais. Em termos funcionais, o MINIX é muito similar ao UNIX, incluindo o suporte ao POSIX. Desde então, a versão original evoluiu para o MINIX 3, que é bastante modular e focado em ser altamente confiável. Ele tem a capacidade de detectar e substituir módulos defeituosos ou mesmo danificados (como drivers de dispositivo de E/S) em funcionamento, sem reinicializá-lo e sem perturbar os programas em execução. O foco é proporcionar uma altíssima confiabilidade e disponibilidade. Um livro que descreve a sua operação interna e lista o código-fonte em um apêndice também se encontra disponível (TANENBAUM, WOODHULL, 2006). O sistema MINIX 3 está disponível gratuitamente (incluindo todo o código-fonte) na internet em <*www.minix3.org*>.

O desejo de produzir uma versão gratuita do MINIX (em vez de uma versão educacional) levou um estudante finlandês, Linus Torvalds, a escrever o **Linux**. Esse sistema foi diretamente inspirado pelo MINIX, desenvolvido sobre ele e originalmente fornecia suporte a vários aspectos do MINIX (por exemplo, o sistema de arquivos do MINIX). Desde então, foi ampliado de muitas maneiras por muitas pessoas, mas ainda mantém alguma estrutura subjacente comum ao MINIX e ao UNIX. Os leitores interessados em uma história detalhada do Linux e do movimento de código aberto (*open-source*) podem ler o livro de Glyn Moody (2001). A maior parte do que será dito sobre o UNIX neste livro se aplica, portanto, ao System V, MINIX, Linux e outras versões e clones do UNIX também.

1.2.4 A quarta geração (1980-presente): computadores pessoais

Com o desenvolvimento dos **circuitos integrados em larga escala** (*Large Scale Integration* — **LSI**) — que são chips contendo milhares de transistores em um centímetro quadrado de silicone —, surgiu a era do computador moderno. Em termos de arquitetura, computadores pessoais (no início chamados de **microcomputadores**) não eram tão diferentes dos minicomputadores da classe PDP-11, mas em termos de preço eles eram certamente muito diferentes. Enquanto o minicomputador tornou possível para um departamento em uma empresa ou universidade ter o

seu próprio computador, o chip microprocessador tornou possível para um único indivíduo ter o seu próprio computador pessoal.

Em 1974, quando a Intel lançou o 8080, a primeira CPU de 8 bits de uso geral, ela queria um sistema operacional para ele, em parte para poder testá-lo. A Intel pediu a um dos seus consultores, Gary Kildall, para escrever um. Kildall e um amigo primeiro construíram um controlador para o recém-lançado disco flexível de 8 polegadas da Shugart Associates e o inseriram no 8080, produzindo assim o primeiro microcomputador com um disco. Kildall escreveu então um sistema operacional baseado em disco chamado **CP/M** (**Control Program for Microcomputers** — programa de controle para microcomputadores) para ele. Como a Intel não achava que microcomputadores baseados em disco tinham muito futuro, quando Kildall solicitou os direitos sobre o CP/M, a Intel concordou. Ele formou então uma empresa, Digital Research, para desenvolver o CP/M e vendê-lo.

Em 1977, a Digital Research reescreveu o CP/M para torná-lo adequado para ser executado nos muitos microcomputadores que usavam o 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 da década de 1980, a IBM projetou o IBM PC e saiu à procura de um software para ser executado nele. O pessoal na IBM contatou Bill Gates para licenciar o seu interpretador BASIC. Eles também perguntaram se ele tinha conhecimento de um sistema operacional para ser executado no PC. Gates sugeriu que a IBM contatasse a Digital Research, então a empresa de sistemas operacionais dominante no mundo. Tomando a que certamente foi a pior decisão de negócios na história, Kildall recusou-se a se encontrar com a IBM, mandando um subordinado em seu lugar. Para piorar as coisas, seu advogado chegou a recusar-se a assinar o acordo de sigilo da IBM cobrindo o ainda não anunciado PC. Em consequência, a IBM voltou a Gates, perguntando se ele não lhes forneceria um sistema operacional.

Quando a IBM voltou, Gates se deu conta de que uma fabricante de computadores local, Seattle Computer Products, tinha um sistema operacional adequado, **DOS** (**Disk Operating System** — sistema operacional de disco). Ele os procurou e pediu para comprá-lo (supostamente por US$ 75.000), oferta que eles de pronto aceitaram. Gates ofereceu então à IBM um pacote DOS/BASIC, que a empresa aceitou. A IBM queria fazer algumas modificações, então Gates contratou a pessoa que havia escrito o DOS, Tim Paterson, como um empregado da empresa emergente de Gates, Microsoft, para fazê-las. O sistema revisado foi renomeado **MS-DOS** (**MicroSoft Disk Operating System** — Sistema operacional de disco da Microsoft) e logo passou a dominar o mercado do IBM PC. Um fator-chave aqui foi a decisão de Gates (em retrospectiva, extremamente sábia) de vender o MS-DOS às empresas de computadores em conjunto com o hardware, em comparação com a tentativa de Kildall de vender o CP/M aos usuários finais diretamente (pelo menos no início). Tempos depois de toda a história transparecer, Kildall morreu de maneira súbita e inesperada de causas que não foram completamente elucidadas.

Quando o sucessor do IBM PC, o IBM PC/AT, foi lançado em 1983 com o CPU Intel 80286, o MS-DOS estava firmemente estabelecido enquanto o CP/M vivia seus últimos dias. O MS-DOS mais tarde foi amplamente usado no 80386 e no 80486. Embora a versão inicial do MS-DOS fosse relativamente primitiva, as versões subsequentes incluíam aspectos mais avançados, muitos tirados do UNIX. (A Microsoft tinha plena consciência do UNIX, chegando até a vender uma versão em microcomputador dele chamada XENIX durante os primeiras anos da empresa.)

O CP/M, MS-DOS e outros sistemas operacionais para os primeiros microcomputadores eram todos baseados na digitação de comandos no teclado pelos usuários. Isto finalmente mudou por conta da pesquisa realizada por Doug Engelbert no Instituto de Pesquisa de Stanford na década de 1960. Engelbart inventou a Graphical User Interface (GUI — Interface Gráfica do Usuário), completa com janelas, ícones, menus e mouse. Essas ideias foram adotadas por pesquisadores na Xerox PARC e incorporadas nas máquinas que eles produziram.

Um dia, Steve Jobs, que coinventou o computador Apple em sua garagem, visitou a PARC, viu uma GUI e no mesmo instante percebeu o seu valor potencial, algo que o gerenciamento da Xerox notoriamente não fez. Esse erro estratégico de proporções gigantescas levou a um livro intitulado *Fumbling the Future* (SMITH e ALEXANDER, 1988). Jobs partiu então para a produção de um Apple com o GUI. O projeto levou ao Lisa, que era caro demais e fracassou comercialmente. A segunda tentativa de Jobs, o Apple Macintosh, foi um sucesso enorme, não apenas porque ele era muito mais barato que o Lisa, mas também por ser **amigável ao usuário**, significando que era dirigido a usuários que não apenas não sabiam nada sobre computadores como não tinham intenção alguma de aprender sobre eles. No mundo criativo do design gráfico, fotografia digital profissional e produção de vídeos digitais profissionais,

Macintoshes são amplamente utilizados e seus usuários entusiastas do seu desempenho. Em 1999, a Apple adotou um núcleo derivado do micronúcleo Mach da Universidade de Carnegie Mellon que foi originalmente desenvolvido para substituir o núcleo do BDS UNIX. Desse modo, o **MAC OS X** é um sistema operacional baseado no UNIX, embora com uma interface bastante distinta.

Quando decidiu produzir um sucessor para o MS-DOS, a Microsoft foi fortemente influenciada pelo sucesso do Macintosh. Ela produziu um sistema baseado em GUI chamado Windows, que originalmente era executado em cima do MS-DOS (isto é, era mais como um interpretador de comandos — shell — do que um sistema operacional de verdade). Por cerca de dez anos, de 1985 a 1995, o Windows era apenas um ambiente gráfico sobre o MS-DOS. Entretanto, começando em 1995, uma versão independente, Windows 95, foi lançada incorporando muitos aspectos de sistemas operacionais, usando o sistema MS-DOS subjacente apenas para sua inicialização e para executar velhos programas do MS-DOS. Em 1998, uma versão ligeiramente modificada deste sistema, chamada Windows 98, foi lançada. Não obstante isso, tanto o Windows 95 como o Windows 98 ainda continham uma grande quantidade da linguagem de montagem de 16 bits da Intel.

Outro sistema operacional da Microsoft, o **Windows NT** (em que o NT representa **New Technology**), era compatível com o Windows 95 até um determinado nível, mas internamente, foi completamente reescrito. Era um sistema de 32 bits completo. O principal projetista do Windows NT foi David Cutler, que também foi um dos projetistas do sistema operacional VAX VMS, de maneira que algumas ideias do VMS estão presentes no NT. Na realidade, tantas ideias do VMS estavam presentes nele, que seu proprietário, DEC, processou a Microsoft. O caso foi acordado extrajudicialmente por uma quantidade de dinheiro exigindo muitos dígitos para ser escrita. A Microsoft esperava que a primeira versão do NT acabaria com o MS-DOS e que todas as versões depois dele seriam um sistema vastamente superior, mas isso não aconteceu. Apenas com o Windows NT 4.0 o sistema enfim arrancou de verdade, especialmente em redes corporativas. A versão 5 do Windows NT foi renomeada Windows 2000 no início do ano de 1999. A intenção era que ela fosse a sucessora tanto do Windows 98, quanto do Windows NT 4.0.

Essa versão também não teve êxito, então a Microsoft produziu mais uma versão do Windows 98, chamada **Windows ME (Millenium Edition)**. Em 2001, uma versão ligeiramente atualizada do Windows 2000, chamada Windows XP foi lançada. Ela teve uma vida muito mais longa (seis anos), basicamente substituindo todas as versões anteriores do Windows.

Mesmo assim, a geração de versões continuou firme. Após o Windows 2000, a Microsoft dividiu a família Windows em uma linha de clientes e outra de servidores. A linha de clientes era baseada no XP e seus sucessores, enquanto a de servidores incluía o Windows Server 2003 e o Windows 2008. Uma terceira linha, para o mundo embutido, apareceu um pouco mais tarde. Todas essas versões do Windows aumentaram suas variações na forma de **pacotes de serviço (*service packs*)**. Foi o suficiente para deixar alguns administradores (e escritores de livros didáticos sobre sistemas operacionais) estupefatos.

Então, em janeiro de 2007, a Microsoft finalmente lançou o sucessor para o Windows XP, chamado Vista. Ele veio com uma nova interface gráfica, segurança mais firme e muitos programas para os usuários novos ou atualizados. A Microsoft esperava que ele substituísse o Windows XP completamente, mas isso nunca aconteceu. Em vez disso, ele recebeu muitas críticas e uma cobertura negativa da imprensa, sobretudo por causa das exigências elevadas do sistema, termos de licenciamento restritivos e suporte para o **Digital Rights Management**, técnicas que tornaram mais difícil para os usuários copiarem material protegido.

Com a chegada do Windows 7 — uma versão nova e muito menos faminta de recursos do sistema operacional —, muitas pessoas decidiram pular completamente o Vista. O Windows 7 não introduziu muitos aspectos novos, mas era relativamente pequeno e bastante estável. Em menos de três semanas, o Windows 7 havia conquistado um mercado maior do que o Vista em sete meses. Em 2012, a Microsoft lançou o sucessor, Windows 8, um sistema operacional com visual e sensação completamente diferentes, voltado para telas de toque. A empresa espera que o novo design se torne o sistema operacional dominante em uma série de dispositivos: computadores de mesa (*desktops*), laptops, notebooks, tablets, telefones e PCs de *home theater*. Até o momento, no entanto, a penetração de mercado é lenta em comparação ao Windows 7.

Outro competidor importante no mundo dos computadores pessoais é o UNIX (e os seus vários derivativos). O UNIX é mais forte entre servidores de rede e de empresas, mas também está presente em computadores de mesa, notebooks, tablets e smartphones. Em computadores baseados no x86, o Linux está se tornando uma alternativa popular ao Windows para estudantes e cada vez mais para muitos usuários corporativos.

Como nota, usaremos neste livro o termo **x86** para nos referirmos a todos os processadores modernos

baseados na família de arquiteturas de instruções que começaram com o 8086 na década de 1970. Há muitos processadores desse tipo, fabricados por empresas como a AMD e a Intel, e por dentro eles muitas vezes diferem consideravelmente: processadores podem ter 32 ou 64 bits com poucos ou muitos núcleos e pipelines que podem ser profundos ou rasos, e assim por diante. Não obstante, para o programador, todos parecem bastante similares e todos ainda podem ser executados no código 8086 que foi escrito 35 anos atrás. Onde a diferença for importante, vamos nos referir a modelos explícitos em vez disso — e usar o **x86-32** e o **x86-64** para indicar variantes de 32 bits e 64 bits.

O **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 (OS X). O UNIX também é padrão em estações de trabalho equipadas com chips RISC de alto desempenho. Seus derivados são amplamente usados em dispositivos móveis, os que executam iOS 7 ou Android.

Muitos usuários do UNIX, em especial programadores experientes, preferem uma interface baseada em comandos a uma GUI, de maneira que praticamente todos os sistemas UNIX dão suporte a um sistema de janelas chamado de **X Window System** (também conhecido como **X11**) produzido no M.I.T. Esse sistema cuida do gerenciamento básico de janelas, permitindo que os usuários criem, removam, movam e redimensionem as janelas usando o mouse. Muitas vezes uma GUI completa, como **Gnome** ou **KDE**, está disponível para ser executada em cima do X11, dando ao UNIX uma aparência e sensação semelhantes ao Macintosh ou Microsoft Windows, para aqueles usuários do UNIX que buscam isso.

Um desenvolvimento interessante que começou a ocorrer em meados da década de 1980 foi o crescimento das redes de computadores pessoais executando **sistemas operacionais de rede** e **sistemas operacionais distribuídos** (TANENBAUM e VAN STEEN, 2007). Em um sistema operacional de rede, os usuários estão conscientes 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 e tem seu próprio usuário local (ou usuários).

Sistemas operacionais de rede não são fundamentalmente diferentes de sistemas operacionais de um único processador. Eles precisam, óbvio, de um controlador de interface de rede e algum software de baixo nível para executá-los, assim como programas para conseguir realizar o login remoto e o acesso remoto a arquivos, mas esses acréscimos não mudam a estrutura essencial do sistema operacional.

Um sistema operacional distribuído, por sua vez, aparece para os seus usuários como um sistema monoprocessador tradicional, embora seja na realidade composto de múltiplos processadores. Os usuários não precisam saber onde os programas estão sendo executados ou onde estão localizados os seus arquivos; isso tudo deve ser cuidado automática e eficientemente pelo sistema operacional.

Sistemas operacionais de verdade exigem mais do que apenas acrescentar um pequeno código a um sistema operacional monoprocessador, pois sistemas distribuídos e centralizados diferem em determinadas maneiras críticas. Os distribuídos, por exemplo, muitas vezes permitem que aplicativos sejam executados em vários processadores ao mesmo tempo, demandando assim algoritmos mais complexos de escalonamento de processadores a fim de otimizar o montante de paralelismo.

Atrasos de comunicação dentro da rede muitas vezes significam que esses (e outros) algoritmos devem estar sendo executados com informações incorretas, desatualizadas ou incompletas. Essa situação difere radicalmente daquela em um sistema monoprocessador no qual o sistema operacional tem informações completas sobre o estado do sistema.

1.2.5 A quinta geração (1990-presente): computadores móveis

Desde os dias em que o detetive Dick Tracy começou a falar para o seu "rádio relógio de pulso" nos quadrinhos da década de 1940, as pessoas desejavam ardentemente um dispositivo de comunicação que elas pudessem levar para toda parte. O primeiro telefone móvel real apareceu em 1946 e pesava em torno de 40 quilos. Você podia levá-lo para toda parte, desde que você tivesse um carro para carregá-lo.

O primeiro telefone verdadeiramente móvel foi criado na década de 1970 e, pesando cerca de um quilo, era positivamente um peso-pena. Ele ficou conhecido carinhosamente como "o tijolo". Logo todos queriam um. Hoje, a penetração do telefone móvel está próxima de 90% da população global. Podemos fazer chamadas não somente com nossos telefones portáteis e relógios de pulso, mas logo com óculos e outros itens que você pode vestir. Além disso, a parte do telefone não é mais tão importante. Recebemos e-mail, navegamos na web, enviamos mensagens para nossos amigos, jogamos, encontramos o melhor caminho dirigindo — e não pensamos duas vezes a respeito disso.

Embora a ideia de combinar a telefonia e a computação em um dispositivo semelhante a um telefone exista desde a década de 1970 também, o primeiro smartphone de verdade não foi inventado até meados de 1990, quando a Nokia lançou o N9000, que literalmente combinava dois dispositivos mormente separados: um telefone e um **PDA** (**Personal Digital Assistant** — assistente digital pessoal). Em 1997, a Ericsson cunhou o termo *smartphone* para o seu "Penelope" GS88.

Agora que os smartphones tornaram-se onipresentes, a competição entre os vários sistemas operacionais tornou-se feroz e o desfecho é mais incerto ainda que no mundo dos PCs. No momento em que escrevo este livro, o Android da Google é o sistema operacional dominante, com o iOS da Apple sozinho em segundo lugar, mas esse nem sempre foi o caso e tudo pode estar diferente de novo em apenas alguns anos. Se algo está claro no mundo dos smartphones é que não é fácil manter-se no topo por muito tempo.

Afinal de contas, a maioria dos smartphones na primeira década após sua criação era executada em **Symbian** OS. Era o sistema operacional escolhido para as marcas populares como Samsung, Sony Ericsson, Motorola e especialmente Nokia. No entanto, outros sistemas operacionais como o Blackberry OS da **RIM** (introduzido para smartphones em 2002) e o iOS da Apple (lançado para o primeiro **iPhone** em 2007) começaram a ganhar mercado do Symbian. Muitos esperavam que o RIM dominasse o mercado de negócios, enquanto o iOS seria o rei dos dispositivos de consumo. A participação de mercado do Symbian desabou. Em 2011, a Nokia abandonou o Symbian e anunciou que se concentraria no Windows Phone como sua principal plataforma. Por algum tempo, a Apple e o RIM eram festejados por todos (embora não tão dominantes quanto o Symbian tinha sido), mas não levou muito tempo para o Android, um sistema operacional baseado no Linux lançado pelo Google em 2008, dominar os seus rivais.

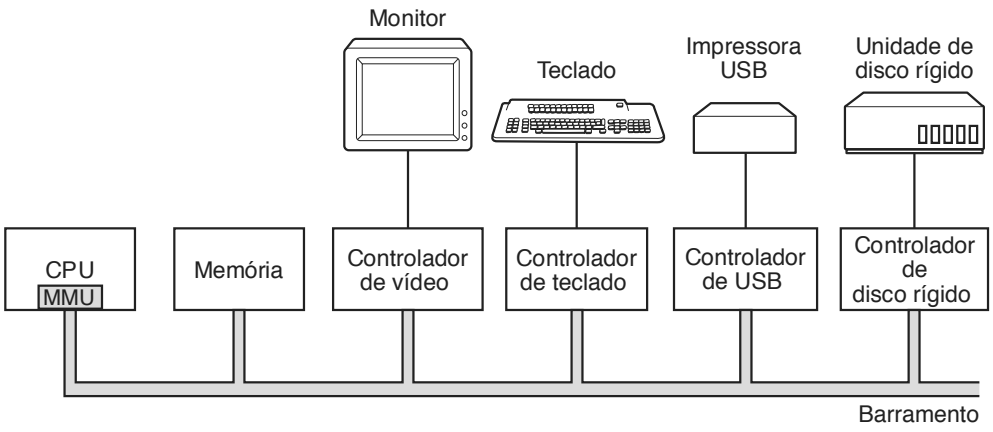
Para os fabricantes de telefone, o Android tinha a vantagem de ser um sistema aberto e disponível sob uma licença permissiva. Como resultado, podiam mexer nele e adaptá-lo a seu hardware com facilidade. Ele também tem uma enorme comunidade de desenvolvedores escrevendo aplicativos, a maior parte na popular linguagem de programação Java. Mesmo assim, os últimos anos mostraram que o domínio talvez não dure, e os competidores do Android estão ansiosos para retomar parte da sua participação de mercado. Examinaremos o Android detalhadamente na Seção 10.8.

1.3 Revisão sobre hardware de computadores

Um sistema operacional está intimamente ligado ao hardware do computador no qual ele é executado. Ele estende o conjunto de instruções do computador e gerencia seus recursos. Para funcionar, ele deve conhecer profundamente o hardware, pelo menos como aparece para o programador. Por esta razão, vamos revisar brevemente o hardware de computadores como encontrado nos computadores pessoais modernos. Depois, podemos começar a entrar nos detalhes do que os sistemas operacionais fazem e como eles funcionam.

Conceitualmente, um computador pessoal simples pode ser abstraído em um modelo que lembra a Figura 1.6. A CPU, memória e dispositivos de E/S estão todos conectados por um sistema de barramento e comunicam-se uns com os outros sobre ele. Computadores pessoais modernos têm uma estrutura mais complicada,

FIGURA 1.6 Alguns dos componentes de um computador pessoal simples.



envolvendo múltiplos barramentos, os quais examinaremos mais tarde. Por ora, este modelo será suficiente. Nas seções seguintes, revisaremos brevemente esses componentes e examinaremos algumas das questões de hardware que interessam aos projetistas de sistemas operacionais. Desnecessário dizer que este será um resumo bastante compacto. Muitos livros foram escritos sobre o tema hardware e organização de computadores. Dois títulos bem conhecidos foram escritos por Tanenbaum e Austin (2012) e Patterson e Hennessy (2013).

1.3.1 Processadores

O "cérebro" do computador é a CPU. Ela busca instruções da memória e as executa. O ciclo básico de toda CPU é buscar a primeira instrução da memória, decodificá-la para determinar o seu tipo e operandos, executá-la, e então buscar, decodificar e executar as instruções subsequentes. O ciclo é repetido até o programa terminar. É dessa maneira que os programas são executados.

Cada CPU tem um conjunto específico de instruções que ela consegue executar. Desse modo, um processador x86 não pode executar programas ARM e um processador ARM não consegue executar programas x86. Como o tempo para acessar a memória para buscar uma instrução ou palavra dos operandos é muito maior do que o tempo para executar uma instrução, todas as CPUs têm alguns registradores internos para armazenamento de variáveis e resultados temporários. Desse modo, o conjunto de instruções geralmente contém instruções para carregar uma palavra da memória para um registrador e armazenar uma palavra de um registrador para a memória. Outras instruções combinam dois operandos provenientes de registradores, da memória, ou ambos, para produzir um resultado como adicionar duas palavras e armazenar o resultado em um registrador ou na memória.

Além dos registradores gerais usados para armazenar variáveis e resultados temporários, a maioria dos computadores tem vários registradores especiais que são visíveis para o programador. Um desses é o **contador de programa**, que contém o endereço de memória da próxima instrução a ser buscada. Após essa instrução ter sido buscada, o contador de programa é atualizado para apontar a próxima instrução.

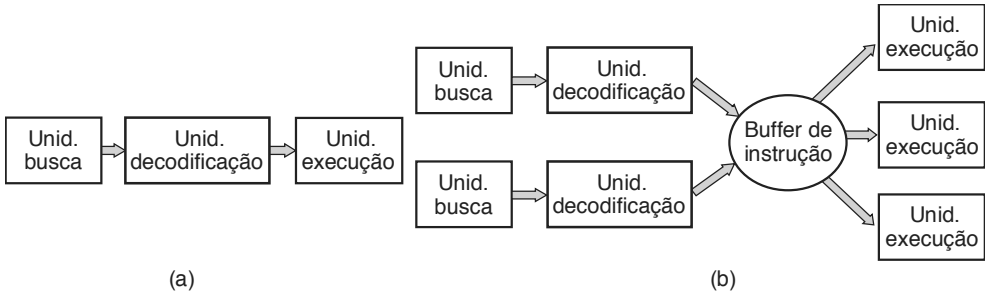
Outro registrador é o **ponteiro de pilha**, que aponta para o topo da pilha atual na memória. A pilha contém uma estrutura para cada rotina que foi chamada, mas ainda não encerrada. Uma estrutura de pilha de rotina armazena aqueles parâmetros de entrada, variáveis locais e variáveis temporárias que não são mantidas em registradores.

Outro registrador ainda é o **PSW** (**Program Status Word** — palavra de estado do programa). Esse registrador contém os bits do código de condições, que são estabelecidos por instruções de comparação, a prioridade da CPU, o modo de execução (usuário ou núcleo) e vários outros bits de controle. Programas de usuários normalmente podem ler todo o PSW, mas em geral podem escrever somente parte dos seus campos. O PSW tem um papel importante nas chamadas de sistema e em E/S.

O sistema operacional deve estar absolutamente ciente de todos os registros. Quando realizando a multiplexação de tempo da CPU, ele muitas vezes vai interromper o programa em execução para (re)começar outro. Toda vez que ele para um programa em execução, o sistema operacional tem de salvar todos os registradores de maneira que eles possam ser restaurados quando o programa for executado mais tarde.

Para melhorar o desempenho, os projetistas de CPU há muito tempo abandonaram o modelo simples de buscar, decodificar e executar uma instrução de cada vez. Muitas CPUs modernas têm recursos para executar mais de uma instrução ao mesmo tempo. Por exemplo, uma CPU pode ter unidades de busca, decodificação e execução separadas, assim enquanto ela está executando a instrução n, poderia também estar decodificando a instrução $n + 1$ e buscando a instrução $n + 2$. Uma organização com essas características é chamada de **pipeline** e é ilustrada na Figura 1.7(a) para um pipeline

FIGURA 1.7 (a) Um pipeline com três estágios. (b) Uma CPU superescalar.



com três estágios. Pipelines mais longos são comuns. Na maioria desses projetos, uma vez que a instrução tenha sido levada para o pipeline, ela deve ser executada, mesmo que a instrução anterior tenha sido um desvio condicional tomado. Pipelines provocam grandes dores de cabeça nos projetistas de compiladores e de sistemas operacionais, pois expõem as complexidades da máquina subjacente e eles têm de lidar com elas.

Ainda mais avançada que um projeto de pipeline é uma CPU **superescalar**, mostrada na Figura 1.7(b). Nesse projeto, unidades múltiplas de execução estão presentes. Uma unidade para aritmética de números inteiros, por exemplo, uma unidade para aritmética de ponto flutuante e uma para operações booleanas. Duas ou mais instruções são buscadas ao mesmo tempo, decodificadas e jogadas em um buffer de instrução até que possam ser executadas. Tão logo uma unidade de execução fica disponível, ela procura no buffer de instrução para ver se há uma instrução que ela pode executar e, se assim for, ela remove a instrução do buffer e a executa. Uma implicação desse projeto é que as instruções do programa são muitas vezes executadas fora de ordem. Em geral, cabe ao hardware certificar-se de que o resultado produzido é o mesmo que uma implementação sequencial conseguiria, mas como veremos adiante, uma quantidade incômoda de tarefas complexas é empurrada para o sistema operacional.

A maioria das CPUs — exceto aquelas muito simples usadas em sistemas embarcados, tem dois modos, núcleo e usuário, como mencionado anteriormente. Em geral, um bit no PSW controla o modo. Quando operando em modo núcleo, a CPU pode executar todas as instruções em seu conjunto de instruções e usar todos os recursos do hardware. Em computadores de mesa e servidores, o sistema operacional normalmente opera em modo núcleo, dando a ele acesso a todo o hardware. Na maioria dos sistemas embarcados, uma parte pequena opera em modo núcleo, com o resto do sistema operacional operando em modo usuário.

Programas de usuários sempre são executados em modo usuário, o que permite que apenas um subconjunto das instruções possa ser executado e um subconjunto dos recursos possa ser acessado. Geralmente, todas as instruções envolvendo E/S e proteção de memória são inacessíveis no modo usuário. Alterar o bit de modo PSW para modo núcleo também é proibido, claro.

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 é finalizado, o controle retorna para o programa do usuário na instrução posterior à chamada de sistema. Explicaremos os detalhes do mecanismo de chamada de sistema posteriormente neste capítulo. Por ora, 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. Como nota a respeito da tipografia, usaremos a fonte Helvética com letras minúsculas para indicar chamadas de sistema ao longo do texto, como: read.

Vale a pena observar que os computadores têm outras armadilhas ("traps") além da instrução para executar uma chamada de sistema. A maioria das outras armadilhas é causada pelo hardware para advertir sobre uma situação excepcional como uma tentativa de divisão por 0 ou um *underflow* (incapacidade de representação de um número muito pequeno) em ponto flutuante. Em todos os casos o sistema operacional assume o controle e tem de decidir o que fazer. Às vezes, o programa precisa ser encerrado por 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 anunciou com antecedência que ele quer lidar com determinados tipos de condições, o controle pode ser passado de volta ao programa para deixá-lo cuidar do problema.

Chips multithread e multinúcleo

A lei de Moore afirma que o número de transistores em um chip dobra a cada 18 meses. Tal "lei" não é nenhum tipo de lei da física, como a conservação do momento, mas é uma observação do cofundador da Intel, Gordon Moore, de quão rápido os engenheiros de processo nas empresas de semicondutores são capazes de reduzir o tamanho dos seus transistores. A lei de Moore se mantém há mais de três décadas até agora e espera-se que se mantenha por pelo menos mais uma. Após isso, o número de átomos por transistor tornar-se-á pequeno demais e a mecânica quântica começará a ter um papel maior, evitando uma redução ainda maior dos tamanhos dos transistores.

A abundância de transistores está levando a um problema: o que fazer com todos eles? Vimos uma abordagem acima: arquiteturas superescalares, com múltiplas unidades funcionais. Mas à medida que o número de transistores aumenta, mais ainda é possível. Algo óbvio a ser feito é colocar memórias cache maiores no chip da CPU. Isso de fato está acontecendo, mas finalmente o ponto de ganhos decrescentes será alcançado.

O próximo passo óbvio é replicar não apenas as unidades funcionais, mas também parte da lógica de controle.

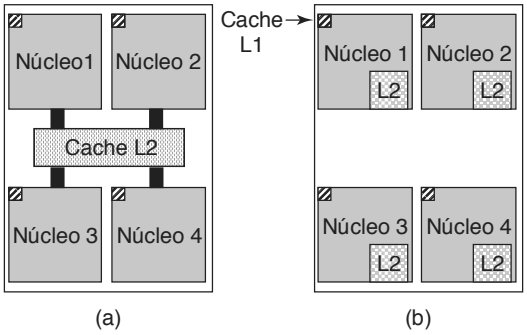
O Pentium 4 da Intel introduziu essa propriedade, chamada **multithreading** ou **hyperthreading** (o nome da Intel para ela), ao processador x86 e vários outros chips de CPU também o têm — incluindo o SPARC, o Power5, o Intel Xeon e a família Intel Core. Para uma primeira aproximação, o que ela faz é permitir que a CPU mantenha o estado de dois threads diferentes e então faça o chaveamento entre um e outro em uma escala de tempo de nanossegundos. (Um thread é um tipo de processo leve, o qual, por sua vez, é um programa de execução; entraremos nos detalhes no Capítulo 2.) Por exemplo, se um dos processos precisa ler uma palavra da memória (o que leva muitos ciclos de relógio), uma CPU multithread pode simplesmente fazer o chaveamento para outro thread. O multithreading não proporciona paralelismo real. Apenas um processo de cada vez é executado, mas o tempo de chaveamento de thread é reduzido para a ordem de um nanossegundo.

O multithreading tem implicações para o sistema operacional, pois cada thread aparece para o sistema operacional como uma CPU em separado. Considere um sistema com duas CPUs efetivas, cada uma com dois threads. O sistema operacional verá isso como quatro CPUs. Se há apenas trabalho suficiente para manter duas CPUs ocupadas em um determinado momento no tempo, ele pode escalonar inadvertidamente dois threads para a mesma CPU, com a outra completamente ociosa. Essa escolha é muito menos eficiente do que usar um thread para cada CPU.

Além do multithreading, muitos chips de CPU têm agora quatro, oito ou mais processadores completos ou **núcleos** neles. Os chips multinúcleo da Figura 1.8 efetivamente trazem quatro minichips, cada um com sua CPU independente. (As caches serão explicadas a seguir.) Alguns processadores, como o Intel Xeon Phi e o Tilera TilePro, já apresentam mais de 60 núcleos em um único chip. Fazer uso de um chip com múltiplos núcleos como esse definitivamente exigirá um sistema operacional de multiprocessador.

Incidentalmente, em termos de números absolutos, nada bate uma **GPU (Graphics Processing Unit** — unidade de processamento gráfico) moderna. Uma GPU é um processador com, literalmente, milhares de núcleos minúsculos. Eles são muito bons para realizar muitos pequenos cálculos feitos em paralelo, como reproduzir polígonos em aplicações gráficas. Não são tão bons em tarefas em série. Eles também são difíceis de programar. Embora GPUs possam ser úteis para sistemas operacionais (por exemplo, codificação ou processamento de tráfego de rede), não é provável que grande parte do sistema operacional em si vá ser executada nas GPUs.

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



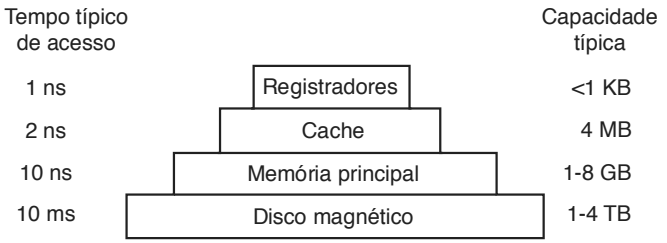
1.3.2 Memória

O segundo principal componente em qualquer computador é a memória. Idealmente, uma memória deve ser rápida ao extremo (mais rápida do que executar uma instrução, de maneira que a CPU não seja atrasada pela memória), abundantemente grande e muito barata. Nenhuma tecnologia atual satisfaz todas essas metas, assim uma abordagem diferente é tomada. O sistema de memória é construído como uma hierarquia de camadas, como mostrado na Figura 1.9. As camadas superiores têm uma velocidade mais alta, capacidade menor e um custo maior por bit do que as inferiores, muitas vezes por fatores de um bilhão ou mais.

A camada superior consiste em registradores internos à CPU. Eles são feitos do mesmo material que a CPU e são, desse modo, tão rápidos quanto ela. Em consequência, não há um atraso ao acessá-los. A capacidade de armazenamento disponível neles é tipicamente 32 × 32 bits em uma CPU de 32 bits e 64 × 64 bits em uma CPU de 64 bits. Menos de 1 KB em ambos os casos. Os programas devem gerenciar os próprios registradores (isto é, decidir o que manter neles) no software.

Em seguida, vem a memória cache, que é controlada principalmente pelo hardware. A memória principal é dividida em **linhas de cache**, tipicamente 64 bytes, com endereços 0 a 63 na linha de cache 0, 64 a 127 na linha de cache 1 e assim por diante. As linhas de cache mais

FIGURA 1.9 Uma hierarquia de memória típica. Os números são apenas aproximações.



utilizadas são mantidas em uma cache de alta velocidade localizada dentro ou muito próximo da CPU. Quando o programa precisa ler uma palavra de memória, o hardware de cache confere se a linha requisitada está na cache. Se ela estiver **presente na cache** (*cache hit*), a requisição é atendida e nenhuma requisição de memória é feita para a memória principal sobre o barramento. *Cache hits* costumam levar em torno de dois ciclos de CPU. Se a linha requisitada estiver ausente da cache (*cache miss*), uma requisição adicional é feita à memória, com uma penalidade de tempo substancial. A memória da cache é limitada em tamanho por causa do alto custo. Algumas máquinas têm dois ou três níveis de cache, cada um mais lento e maior do que o antecedente.

O conceito de *caching* exerce um papel importante em muitas áreas da ciência de computadores, não apenas na colocação de linhas de RAM na cache. Sempre que um recurso pode ser dividido em partes, algumas das quais são usadas com muito mais frequência que as outras, o *caching* é muitas vezes utilizado para melhorar o desempenho. Sistemas operacionais o utilizam seguidamente. Por exemplo, a maioria dos sistemas operacionais mantém (partes de) arquivos muito usados na memória principal para evitar ter de buscá-los do disco de modo repetido. Similarmente, os resultados da conversão de nomes de rota longa 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 buscas repetidas. Por fim, quando o endereço de uma página da web (URL) é convertido em um endereço de rede (endereço IP), o resultado pode ser armazenado em cache para uso futuro. Há muitos outros usos.

Em qualquer sistema de cache, muitas perguntas surgem relativamente rápido, incluindo:

1. Quando colocar um novo item na cache.
2. Em qual linha de cache colocar o novo item.
3. Qual item remover da cache quando for preciso espaço.
4. Onde colocar um item recentemente desalojado na memória maior.

Nem toda pergunta é relevante para toda situação de cache. Para linhas de cache da memória principal na cache da CPU, um novo item geralmente 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 de 32 bits, os bits 6 a 17 podem ser usados para especificar a linha de cache, com os bits de 0 a 5 especificando os bytes dentro da linha de cache. Aqui, o item a ser removido é o mesmo de onde os novos dados são inseridos, mas em outros sistemas este pode não ser o caso. Por fim, quando uma linha de cache é reescrita para a memória principal (se ela tiver sido modificada desde que foi colocada na cache), o lugar na memória para reescrevê-la é determinado unicamente pelo endereço em questão.

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 usadas com muita intensidade. As caches L1 são em geral de 16 KB cada. Além disso, há muitas vezes uma segunda cache, chamada de **cache L2**, que armazena vários megabytes de palavras de memória recentemente usadas. A diferença entre as caches L1 e L2 encontra-se na sincronização. O acesso à cache L1 é feito sem atraso algum, enquanto o acesso à cache L2 envolve um atraso de um ou dois ciclos de relógio.

Em chips de multinúcleo, os projetistas têm de decidir onde colocar as caches. Na Figura 1.8(a), uma única cache L2 é compartilhada por todos os núcleos. Essa abordagem é usada em chips de multinúcleo da Intel. Em comparação, na Figura 1.8(b), cada núcleo tem sua própria cache L2. Essa abordagem é usada pela AMD. Cada estratégia tem seus prós e contras. Por exemplo, a cache L2 compartilhada da Intel exige um controlador de cache mais complicado, mas o método AMD torna mais difícil manter a consistência entre as caches L2.

A memória principal vem a seguir na hierarquia da Figura 1.9. Trata-se da locomotiva do sistema de memória. A memória principal é normalmente chamada de **RAM** (**Random Access Memory** — memória de acesso aleatório). Os mais antigos às vezes a chamam de **memória de núcleo** (**core memory**), pois os computadores nas décadas de 1950 e 1960 usavam minúsculos núcleos de ferrite magnetizáveis como memória principal. Hoje, as memórias têm centenas de megabytes a vários gigabytes e vêm 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 têm uma pequena memória de acesso aleatório não volátil. Diferentemente da RAM, a memória não volátil não perde o seu conteúdo quando a energia é desligada. A **ROM** (**Read Only Memory** — memória somente de leitura) é programada na fábrica e não pode ser modificada depois. Ela é rápida e barata. Em alguns computadores, o carregador (*bootstrap loader*) usado para

inicializar o computador está contido na ROM. Também algumas placas de E/S vêm com a ROM para lidar com o controle de dispositivos de baixo nível.

A **EEPROM** (**Electrically Erasable PROM** — ROM eletricamente apagável) e a **memória flash** também são não voláteis, mas, diferentemente da ROM, podem ser apagadas e reescritas. No entanto, escrevê-las leva muito mais tempo do que escrever em RAM, então elas são usadas da mesma maneira que a ROM, apenas com a característica adicional de que é possível agora corrigir erros nos programas que elas armazenam mediante sua regravação.

A memória flash também é bastante usada como um meio de armazenamento em dispositivos eletrônicos portáteis. Ela serve como um filme em câmeras digitais e como disco em reprodutores de música portáteis, apenas como exemplo. A memória flash é intermediária em velocidade entre a RAM e o disco. Também, diferentemente da memória de disco, ela se desgasta quando apagada muitas vezes.

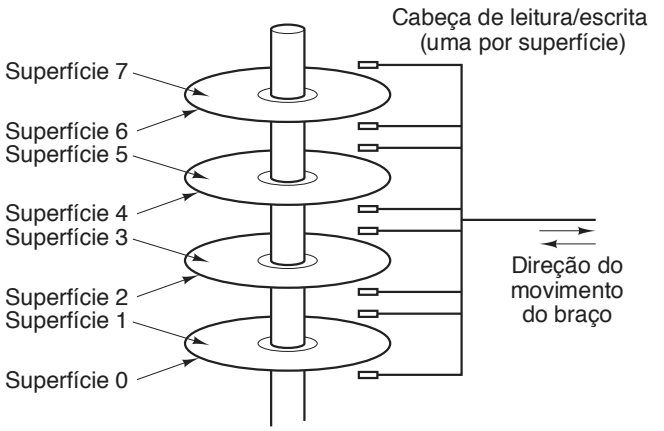
Outro tipo ainda de memória é a CMOS, que é volátil. Muitos computadores usam a memória CMOS para armazenar a hora e a data atualizadas. A memória CMOS e o circuito de relógio que incrementa o tempo registrado nela são alimentados por uma bateria pequena, então a hora é atualizada corretamente, mesmo quando o computador estiver desligado. A memória CMOS também pode conter os parâmetros de configuração, como de qual disco deve se carregar o sistema. A CMOS é usada porque consome tão pouca energia que a bateria original instalada na fábrica muitas vezes dura por vários anos. No entanto, quando ela começa a falhar, o computador pode parecer ter a doença de Alzheimer, esquecendo coisas que ele sabia há anos, como de qual disco rígido carregar o sistema operacional.

1.3.3 Discos

Em seguida na hierarquia está o disco magnético (disco rígido). O armazenamento de disco é duas ordens de magnitude mais barato, por bit, que o da RAM e frequentemente duas ordens de magnitude maior também. O único problema é que o tempo para acessar aleatoriamente os dados é próximo de três ordens de magnitude mais lento. Isso ocorre porque o disco é um dispositivo mecânico, como mostrado na Figura 1.10.

Um disco consiste em um ou mais pratos metálicos que rodam a 5.400, 7.200, 10.800 RPM, ou mais. Um braço mecânico move-se sobre esses pratos a partir da lateral, como o braço de toca-discos de um velho fonógrafo de 33 RPM para tocar discos de vinil. A

FIGURA 1.10 Estrutura de uma unidade de disco.



informação é escrita no disco em uma série de círculos concêntricos. Em qualquer posição do braço, cada uma das cabeças 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 determinado número de setores, com tipicamente 512 bytes por setor. Em discos modernos, os cilindros externos contêm mais setores do que os internos. Mover o braço de um cilindro para o próximo leva em torno de 1 ms. Movê-lo para um cilindro aleatório costuma levar de 5 a 10 ms, dependendo do dispositivo acionador. Uma vez que o braço esteja na trilha correta, o dispositivo acionador tem de esperar até que o setor desejado gire sob a cabeça, um atraso adicional de 5 a 10 ms, dependendo da RPM do dispositivo acionador. Assim que o setor estiver sob a cabeça, a leitura ou escrita ocorre a uma taxa de 50 MB/s em discos de baixo desempenho até 160 MB/s em discos mais rápidos.

Às vezes você ouvirá as pessoas falando sobre discos que não são discos de maneira alguma, como os **SSDs** (**Solid State Disks** — discos em estado sólido). SSDs não têm partes móveis, não contêm placas na forma de discos e armazenam dados na memória (flash). A única maneira pela qual lembram discos é que eles também armazenam uma quantidade grande de dados que não é perdida quando a energia é desligada.

Muitos computadores dão suporte a um esquema conhecido como **memória virtual**, que discutiremos de maneira mais aprofundada no Capítulo 3. Esse esquema torna possível executar programas maiores que a memória física colocando-os no disco e usando a memória principal como um tipo de cache para as partes mais intensivamente executadas. Esse esquema exige o remapeamento dos endereços de memória rapidamente para converter o endereço que o programa gerou para

o endereço físico em RAM onde a palavra está localizada. Esse mapeamento é feito por uma parte da CPU chamada **MMU** (**Memory Management Unit** — unidade de gerenciamento de memória), 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 de um programa para outro, às vezes chamado de um **chaveamento de contexto**, pode ser necessário limpar todos os blocos modificados da cache e mudar os registros de mapeamento na MMU. Ambas são operações caras, e os programadores fazem o que podem para evitá-las. Veremos algumas das implicações de suas táticas mais tarde.

1.3.4 Dispositivos de E/S

A CPU e a memória não são os únicos recursos que o sistema operacional tem de gerenciar. Dispositivos de E/S também interagem intensamente com o sistema operacional. Como vimos na Figura 1.6, dispositivos de E/S consistem em geral em duas partes: um controlador e o dispositivo em si. O controlador é um chip ou um conjunto de chips que controla fisicamente o dispositivo. Ele aceita comandos do sistema operacional, por exemplo, para ler dados do dispositivo, e os executa.

Em muitos casos, o controle real do dispositivo é complicado e detalhado, então faz parte do trabalho do controlador apresentar uma interface mais simples (mas mesmo assim muito complexa) para o sistema operacional. Por exemplo, um controlador de disco pode aceitar um comando para ler o setor 11.206 do disco 2. O controlador tem então de converter esse número do setor linear para um cilindro, setor e cabeça. Essa conversão pode ser complicada porque os cilindros exteriores têm mais setores do que os interiores, e alguns setores danificados foram remapeados para outros. Então o controlador tem de determinar em qual cilindro está o braço do disco e dar a ele um comando correspondente à distância em número de cilindros. Ele deve aguardar até que o setor apropriado tenha girado sob a cabeça e então começar a ler e a armazenar os bits à medida que eles saem do acionador, removendo o cabeçalho e conferindo a soma de verificação (*checksum*). Por fim, ele tem de montar os bits que chegam em palavras e armazená-las na memória. Para fazer todo esse trabalho, os controladores muitas vezes contêm pequenos computadores embutidos que são programados para realizar o seu trabalho.

A outra parte é o dispositivo real em si. Os dispositivos possuem interfaces relativamente simples, tanto porque eles não podem fazer muito, como para padronizá-los. A padronização é necessária para que qualquer controlador de disco SATA possa controlar qualquer disco SATA, por exemplo. **SATA** é a sigla para **Serial ATA**, e **ATA** por sua vez é a sigla para **AT Attachment**. Caso você esteja curioso para saber o significado de AT, esta foi a segunda geração da "Personal Computer Advanced Technology" (tecnologia avançada de computadores pessoais) da IBM, produzida em torno do então extremamente potente processador 80286 de 6 MHz que a empresa introduziu em 1984. O que aprendemos disso é que a indústria de computadores tem o hábito de incrementar continuamente os acrônimos existentes com novos prefixos e sufixos. Também aprendemos que um adjetivo como "avançado" deve ser usado com grande cuidado, ou você passará ridículo daqui a trinta anos.

O SATA é atualmente o tipo de disco padrão em muitos computadores. Dado que a interface do dispositivo real está escondida atrás do controlador, tudo o que o sistema operacional vê é a interface para o controlador, o que pode ser bastante diferente da interface para o dispositivo.

Como cada tipo de controlador é diferente, diversos softwares são necessários para controlar cada um. O software que conversa com um controlador, dando a ele comandos e aceitando respostas, é chamado de **driver de dispositivo**. Cada fabricante de controladores tem de fornecer um driver para cada sistema operacional a que dá suporte. Assim, um digitalizador de imagens pode vir com drivers para OS X, Windows 7, Windows 8 e Linux, por exemplo.

Para ser usado, o driver tem de ser colocado dentro do sistema operacional de maneira que ele possa ser executado em modo núcleo. Na realidade, drivers podem ser executados fora do núcleo, e sistemas operacionais como Linux e Windows hoje em dia oferecem algum suporte para isso. A vasta maioria dos drivers ainda opera abaixo do nível do núcleo. Apenas muito poucos sistemas atuais, como o MINIX 3, operam todos os drivers em espaço do usuário. Drivers no espaço do usuário precisam ter permissão de acesso ao dispositivo de uma maneira controlada, o que não é algo trivial.

Há três maneiras pelas quais o driver pode ser colocado no núcleo. A primeira é religar o núcleo com o novo driver e então reinicializar o sistema. Muitos sistemas UNIX mais antigos funcionam assim. A segunda maneira é adicionar uma entrada em um arquivo do sistema operacional dizendo-lhe que ele precisa do driver e então reinicializar o sistema. No momento da inicialização, o

sistema operacional vai e encontra os drivers que ele precisa e os carrega. O Windows funciona dessa maneira. A terceira maneira é capacitar o sistema operacional a aceitar novos drivers enquanto estiver sendo executado e instalá-los rapidamente sem a necessidade da reinicialização. Essa maneira costumava ser rara, mas está se tornando muito mais comum hoje. Dispositivos do tipo *hot-pluggable* (acoplados a quente), como dispositivos USB e IEEE 1394 (discutidos a seguir), sempre precisam de drivers carregados dinamicamente.

Todo controlador tem um pequeno número de registradores que são usados para comunicar-se com ele. Por exemplo, um controlador de discos mínimo pode ter registradores para especificar o endereço de disco, endereço de memória, contador de setores e direção (leitura ou escrita). Para ativar o controlador, o driver recebe um comando do sistema operacional, então o traduz para os valores apropriados a serem escritos nos registradores dos dispositivos. A reunião de todos esses registradores de dispositivos forma o **espaço de portas de E/S**, um assunto que retomaremos no Capítulo 5.

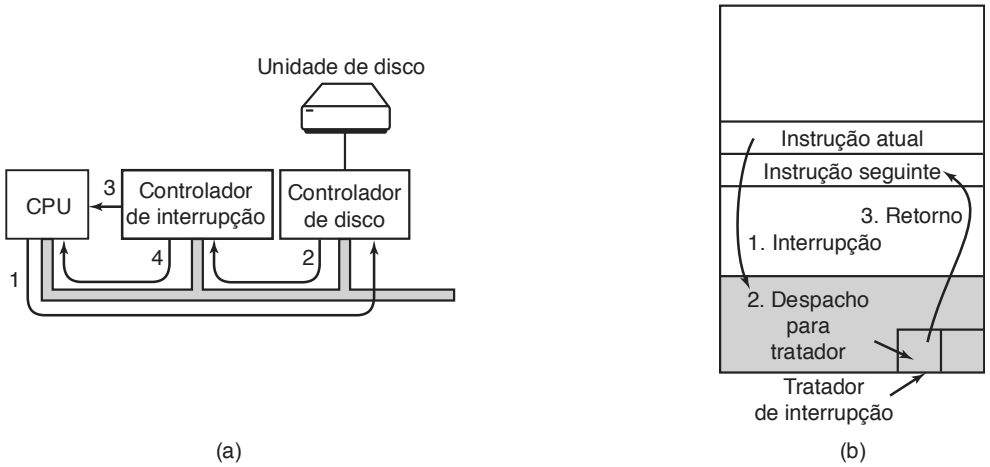
Em alguns computadores, os registradores dos dispositivos estão mapeados no espaço do endereço do sistema operacional (os endereços que ele pode usar), portanto podem ser lidos e escritos como palavras de memória comuns. Nesses computadores, não são necessárias instruções de E/S especiais e os programas de usuários podem ser mantidos distantes do hardware deixando esses endereços de memória fora de seu alcance (por exemplo, pelo uso de registradores-base e limite). Em outros computadores, os registradores dos dispositivos são colocados em um espaço de porta E/S especial, com cada registrador tendo um endereço de porta. Nessas máquinas, instruções especiais IN e OUT estão disponíveis em modo núcleo para permitir que os drivers leiam e escrevam nos registradores. O primeiro esquema elimina a necessidade para instruções de E/S especiais, mas consome parte do espaço do endereço. O segundo esquema não utiliza espaço do endereço, mas exige instruções especiais. Ambos os sistemas 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, que o núcleo traduz em uma chamada de rotina para o driver apropriado. O driver então inicia a E/S e aguarda usando um laço curto, inquirindo continuamente o dispositivo para ver se ele terminou a operação (em geral há algum bit que indica que o dispositivo ainda está ocupado). Quando a operação de E/S termina, o driver coloca os dados (se algum) onde eles são necessários e retorna. O sistema operacional então retorna o controle a quem o chamou. Esse método é chamado de **espera ocupada** e tem a desvantagem de manter a CPU ocupada interrogando o dispositivo até o término da operação de E/S.

No segundo método, o driver inicia o dispositivo e pede a ele que o interrompa quando tiver terminado. Nesse ponto, o driver retorna. O sistema operacional bloqueia então o programa que o chamou, se necessário, e procura por mais trabalho para fazer. Quando o controlador detecta o fim da transferência, ele gera uma **interrupção** para sinalizar o término.

Interrupções são muito importantes nos sistemas operacionais, então vamos examinar a ideia mais de perto. Na Figura 1.11(a), vemos um processo de quatro passos para a E/S. No passo 1, o driver diz para o controlador o que fazer escrevendo nos seus registradores de dispositivo. O controlador então inicia o dispositivo. Quando o controlador termina de ler ou escrever o

FIGURA 1.11 (a) Os passos para iniciar um dispositivo de E/S e obter uma interrupção. (b) O processamento de interrupção envolve obter a interrupção, executar o tratador de interrupção e retornar ao programa do usuário.



número de bytes que lhe disseram para transferir, ele sinaliza o chip controlador de interrupção usando determinadas linhas de barramento no passo 2. Se o controlador de interrupção está pronto para aceitar a interrupção (o que ele talvez não esteja, se estiver ocupado lidando com uma interrupção de maior prioridade), ele sinaliza isso à CPU no passo 3. No passo 4, o controlador de interrupção insere o número do dispositivo no barramento de maneira que a CPU possa lê-lo e saber qual dispositivo acabou de terminar (muitos dispositivos podem estar sendo executados 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) normalmente são empilhados na pilha atual e a CPU chaveada para o modo núcleo. O número do dispositivo pode ser usado como um índice para parte da memória para encontrar o endereço do tratador de interrupção (*interrupt handler*) para esse dispositivo. Essa parte da memória é chamada de **vetor de interrupção**. Uma vez que o tratador de interrupção (parte do driver para o dispositivo de interrupção) tenha iniciado, ele remove o contador de programa e PSW empilhados e os salva, e então indaga o dispositivo para saber como está a sua situação. Assim que o tratador de interrupção tenha sido encerrado, ele retorna para o programa do usuário previamente executado para a primeira instrução que ainda não tenha sido executada. Esses passos são mostrados na Figura 1.11 (b).

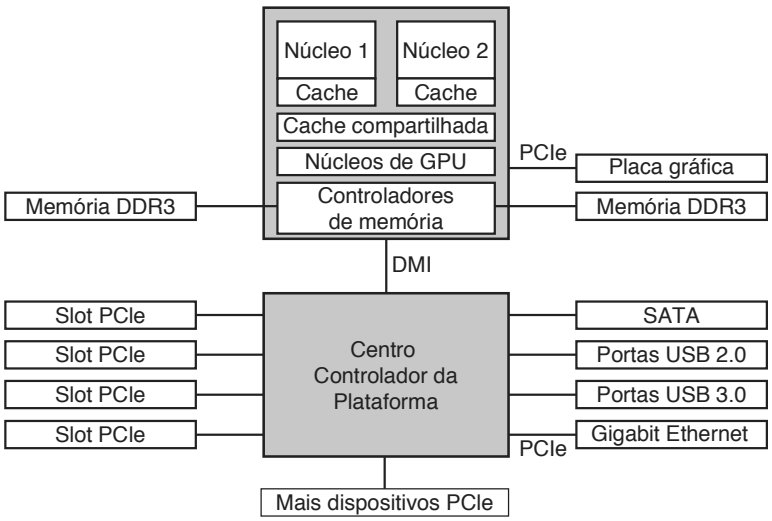
O terceiro método para implementar E/S faz uso de um hardware especial: um chip **DMA (Direct Memory Access** — acesso direto à memória) que pode controlar o fluxo de bits entre a memória e algum controlador sem a intervenção da CPU constante. A CPU configura o chip DMA, dizendo a ele quantos bytes transferir, o dispositivo e endereços de memória envolvidos, e a direção, e então o deixa executar. Quando o chip de DMA tiver finalizado a sua tarefa, ele causa uma interrupção, que é tratada como já descrito. Os hardwares de DMA e E/S em geral serão discutidos mais detalhadamente no Capítulo 5.

Interrupções podem (e muitas vezes isso ocorre) acontecer em momentos altamente inconvenientes, por exemplo, enquanto outro tratador de interrupção estiver em execução. Por essa razão, a CPU tem uma maneira para desabilitar interrupções e então reabilitá-las depois. Enquanto as interrupções estiverem desabilitadas, quaisquer 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 múltiplos dispositivos finalizarem enquanto as interrupções estiverem desabilitadas, o controlador de interrupção decide qual deixar passar primeiro, normalmente baseado em prioridades estáticas designadas para cada dispositivo. O dispositivo de maior prioridade vence e é servido primeiro. Os outros precisam esperar.

1.3.5 Barramentos

A organização da Figura 1.6 foi usada em microcomputadores por anos e também no IBM original. No entanto, à medida que os processadores e as memórias foram ficando mais rápidos, a capacidade de um único barramento (e certamente o barramento do PC IBM) de lidar com todo o tráfego foi exigida até o limite. Algo tinha de ceder. Como resultado, barramentos adicionais foram acrescentados, tanto para dispositivos de E/S mais rápidos quanto para o tráfego CPU para memória. Como consequência dessa evolução, um sistema x86 grande atualmente se parece com algo como a Figura 1.12.

FIGURA 1.12 A estrutura de um sistema x86 grande.



Este sistema tem muitos barramentos (por exemplo, cache, memória, PCIe, PCI, USB, SATA e DMI), cada um com uma taxa de transferência e função diferentes. O sistema operacional precisa ter ciência de todos eles para configuração e gerenciamento. O barramento principal é o **PCIe** (**Peripheral Component Interconnect Express** — interconexão expressa de componentes periféricos).

O PCIe foi inventado pela Intel como um sucessor para o barramento **PCI** mais antigo, que por sua vez foi uma substituição para o barramento **ISA** (**Industry Standard Architecture** — arquitetura padrão industrial). Capaz de transferir dezenas de gigabits por segundo, o PCIe é muito mais rápido que os seus predecessores. Ele também é muito diferente em sua natureza. Uma **arquitetura de barramento compartilhado** significa que múltiplos dispositivos usam os mesmos fios para transferir dados. Assim, quando múltiplos dispositivos têm dados para enviar, você precisa de um árbitro para determinar quem pode utilizar o barramento. Em comparação, o PCIe faz uso de conexões dedicadas de ponto a ponto. Uma **arquitetura de barramento paralela** como usada no PCI tradicional significa que você pode enviar uma palavra de dados através de múltiplos fios. Por exemplo, em barramentos PCI regulares, um único número de 32 bits é enviado através de 32 fios paralelos. Em comparação com isso, o PCIe usa uma **arquitetura de barramento serial** e envia todos os bits em uma mensagem através de uma única conexão, chamada faixa, de maneira muito semelhante a um pacote de rede. Isso é muito mais simples, pois você não tem de assegurar que todos os 32 bits cheguem ao destino exatamente ao mesmo tempo. O paralelismo ainda é usado, pois você pode ter múltiplas faixas em paralelo. Por exemplo, podemos usar 32 faixas para carregar 32 mensagens em paralelo. À medida que a velocidade de dispositivos periféricos como cartões de rede e adaptadores de gráficos aumenta rapidamente, o padrão PCIe é atualizado a cada 3-5 anos. Por exemplo, 16 faixas de PCIe 2.0 oferecem 64 gigabits por segundo. Atualizar para PCIe 3.0 dará a você duas vezes aquela velocidade e o PCIe 4.0 dobrará isso novamente.

Enquanto isso, ainda temos muitos dispositivos de legado do padrão PCI mais antigo. Como vemos na Figura 1.12, esses dispositivos estão ligados a um centro processador em separado. No futuro, quando virmos o PCI não mais como meramente *velho*, mas *ancestral*, é possível que todos os dispositivos PCI vão se ligar a mais um centro ainda que, por sua vez, vai conectá-los ao centro principal, criando uma árvore de barramentos.

Nessa configuração, a CPU se comunica com a memória por meio de um barramento DDR3 rápido, com um dispositivo gráfico externo através do PCIe e com todos os outros dispositivos via um centro controlador usando um barramento **DMI** (**Direct Media Interface** — interface de mídia direta). O centro por sua vez conecta-se com todos os outros dispositivos, usando o Barramento Serial Universal para conversar com os dispositivos USB, o barramento SATA para interagir com discos rígidos e acionadores de DVD, e o PCIe para transferir quadros (frames) Ethernet. Já mencionamos os dispositivos PCI que usam um barramento PCI tradicional.

Além disso, cada um dos núcleos tem uma cache dedicada e uma muito maior que é compartilhada entre eles. Cada uma dessas caches introduz outro barramento.

O **USB** (**Universal Serial Bus** — barramento serial universal) foi inventado para conectar todos os dispositivos de E/S lentos, como o teclado e o mouse, ao computador. No entanto, chamar um dispositivo USB 3.0 zunindo a 5 Gbps de "lento" pode não soar natural para a geração que cresceu com o ISA de 8 Mbps como o barramento principal nos primeiros PCs da IBM. O USB usa um pequeno conector com quatro a onze fios (dependendo da versão), alguns dos quais fornecem energia elétrica para os dispositivos USB ou conectam-se com o terra. O USB é um barramento centralizado no qual um dispositivo-raiz interroga todos os dispositivos de E/S a cada 1 ms para ver se eles têm algum tráfego. O USB 1.0 pode lidar com uma carga agregada de 12 Mbps, o USB 2.0 aumentou a velocidade para 480 Mbps e o USB 3.0 chega a não menos que 5 Gbps. Qualquer dispositivo USB pode ser conectado a um computador e ele funcionará imediatamente, sem exigir uma reinicialização, algo que os dispositivos pré-USB exigiam para a consternação de uma geração de usuários frustrados.

O barramento **SCSI** (**Small Computer System Interface** — interface pequena de sistema computacional) é um barramento de alto desempenho voltado para discos rápidos, digitalizadores de imagens e outros dispositivos que precisam de uma considerável largura de banda. Hoje em dia, eles são encontrados na maior parte das vezes em servidores e estações de trabalho, e podem operar a até 640 MB/s.

Para trabalhar em um ambiente como o da Figura 1.12, o sistema operacional tem de saber quais dispositivos periféricos estão conectados ao computador e configurá-los. Essa exigência levou a Intel e a Microsoft a projetar um sistema para o PC chamado de **plug and play**, baseado em um conceito similar primeiro

implementado no Apple Macintosh. 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 interrupção 1 e usava endereços 0x60 a 0x64, o controlador de disco flexível era a interrupção 6 e usava endereços de E/S 0x3F0 a 0x3F7, e a impressora era a interrupção 7 e usava os endereços de E/S 0x378 a 0x37A, e assim por diante.

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 entravam em conflito e não funcionavam juntas. A solução era incluir chaves DIP ou jumpers em todas as placas de E/S e instruir o usuário a ter o cuidado de configurá-las para selecionar o nível de interrupção e endereços dos dispositivos de E/S que não entrassem em conflito com quaisquer outros no sistema do usuário. Adolescentes que devotaram a vida às complexidades do hardware do PC podiam fazê-lo às vezes sem cometer erros. Infelizmente, ninguém mais conseguia, levando ao caos.

O plug and play faz o sistema coletar automaticamente informações sobre os dispositivos de E/S, atribuir centralmente níveis de interrupção e endereços desses dispositivos e, então, informar a cada placa quais são os seus números. Esse trabalho está relacionado de perto à inicialização do computador, então vamos examinar essa questão. Ela não é completamente trivial.

1.3.6 Inicializando o computador

De modo bem resumido, o processo de inicialização funciona da seguinte maneira: todo PC contém uma parentboard (placa-pais) (que era chamada de placa-mãe antes da onda politicamente correta atingir a indústria de computadores). Na placa-pais há um programa chamado de sistema **BIOS** (**Basic Input Output System** — sistema básico de entrada e saída). O BIOS conta com rotinas de E/S de baixo nível, incluindo procedimentos para ler o teclado, escrever na tela e realizar a E/S no disco, entre outras coisas. Hoje, ele fica em um flash RAM, que é não volátil, mas que pode ser atualizado pelo sistema operacional quando erros são encontrados no BIOS.

Quando o computador é inicializado, o BIOS começa a executar. Primeiro ele confere para ver quanta RAM está instalada e se o teclado e os outros dispositivos básicos estão instalados e respondendo corretamente. Ele segue varrendo os barramentos PCIe e PCI para detectar todos os dispositivos ligados a ele. Se os dispositivos presentes forem diferentes de quando o sistema foi inicializado pela última vez, os novos dispositivos são configurados.

O BIOS então determina o dispositivo de inicialização tentando uma lista de dispositivos armazenados na memória CMOS. O usuário pode mudar essa lista entrando em um programa de configuração do BIOS logo após a inicialização. Tipicamente, é feita uma tentativa para inicializar a partir de uma unidade de CD-ROM (ou às vezes USB), se houver uma. Se isso não der certo, o sistema inicializa a partir do disco rígido. O primeiro setor do dispositivo de inicialização é lido na memória e executado. Ele contém um programa que normalmente 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 então o BIOS para conseguir as informações de configuração. Para cada dispositivo, ele confere para ver se possui o driver do dispositivo. Se não possuir, pede para o usuário inserir um CD-ROM contendo o driver (fornecido pelo fabricante do dispositivo) ou para baixá-lo da internet. Assim que todos os drivers dos dispositivos estiverem disponíveis, o sistema operacional os carrega no núcleo. Então ele inicializa suas tabelas, cria os processos de segundo plano necessários e inicia um programa de identificação (*login*) ou uma interface gráfica GUI.

1.4 O zoológico dos sistemas operacionais

Os sistemas operacionais existem há mais de meio século. Durante esse tempo, uma variedade bastante significativa deles foi desenvolvida, nem todos bastante conhecidos. Nesta seção abordaremos brevemente nove deles. Voltaremos a alguns desses tipos diferentes de sistemas mais tarde no livro.

1.4.1 Sistemas operacionais de computadores de grande porte

No topo estão os sistemas operacionais para computadores de grande porte (*mainframes*), aquelas máquinas do tamanho de uma sala ainda encontradas nos centros de processamento de dados de grandes corporações. Esses computadores diferem dos computadores pessoais em termos de sua capacidade de E/S.

Um computador de grande porte com 1.000 discos e milhões de gigabytes de dados não é incomum; um computador pessoal com essas especificações causaria inveja aos seus amigos. Computadores de grande porte também estão retornando de certa maneira como servidores sofisticados da web, para sites de comércio eletrônico em larga escala e para transações entre empresas (business-to-business).

Os sistemas operacionais para computadores de grande porte são intensamente orientados para o processamento de muitas tarefas ao mesmo tempo, a maioria delas exigindo quantidades prodigiosas de E/S. Eles em geral oferecem três tipos de serviços: em lote (batch), processamento de transações e tempo compartilhado (timesharing). Um sistema em lote processa tarefas rotineiras sem qualquer usuário interativo presente. O processamento de apólices em uma companhia de seguros ou relatórios de vendas para uma cadeia de lojas é tipicamente feito em modo de lote. Sistemas de processamento de transações lidam com grandes números de pedidos pequenos, por exemplo, processamento de cheques em um banco ou reservas de companhias aéreas. Cada unidade de trabalho é pequena, mas o sistema tem de lidar com centenas ou milhares por segundo. Sistemas de tempo compartilhado permitem que múltiplos usuários remotos executem tarefas no computador ao mesmo tempo, como na realização de consultas a um grande banco de dados. Essas funções são proximamente relacionadas; sistemas operacionais em computadores de grande porte muitas vezes executam todas elas. Um exemplo de sistema operacional de computadores de grande porte é o OS/390, um descendente do OS/360. No entanto, sistemas operacionais de computadores de grande porte estão pouco a pouco sendo substituídos por variantes 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 mesmo computadores de grande porte. Eles servem a múltiplos usuários ao mesmo tempo por meio de uma rede e permitem que os usuários compartilhem recursos de hardware e software. Servidores podem fornecer serviços de impressão, de arquivo ou de web. Provedores de acesso à internet utilizam várias máquinas servidoras para dar suporte aos clientes, e sites usam servidores para armazenar páginas e lidar com as requisições que chegam. Sistemas operacionais típicos de servidores são Solaris, FreeBSD, Linux e Windows Server 201x.

1.4.3 Sistemas operacionais de multiprocessadores

Uma maneira cada vez mais comum de se obter potência computacional para valer é conectar múltiplas CPUs a um único sistema. Dependendo de como precisamente eles são conectados e o que é compartilhado, esses sistemas são chamados de computadores paralelos, multicomputadores ou multiprocessadores. Eles precisam de sistemas operacionais especiais, porém muitas vezes esses são variações dos sistemas operacionais de servidores, com aspectos especiais para comunicação, conectividade e consistência.

Com o advento recente de chips multinúcleo para computadores pessoais, mesmo sistemas operacionais de computadores de mesa e notebooks convencionais estão começando a lidar com pelo menos multiprocessadores de pequena escala, e é provável que o número de núcleos cresça com o tempo. Felizmente, já sabemos bastante a respeito de sistemas operacionais de multiprocessadores de anos de pesquisa anteriores, de maneira que utilizar esse conhecimento em sistemas multinúcleo não deverá ser difícil. A parte difícil será fazer com que os aplicativos usem toda essa potência computacional. Muitos sistemas operacionais populares, incluindo Windows e Linux, são executados em multiprocessadores.

1.4.4 Sistemas operacionais de computadores pessoais

A próxima categoria é a do sistema operacional de computadores pessoais. Todos os computadores modernos dão suporte à multiprogramação, muitas vezes com dezenas de programas iniciados no momento da inicialização do sistema. Seu trabalho é proporcionar um bom apoio para um único usuário. Eles são amplamente usados para o processamento de texto, planilhas e acesso à internet. Exemplos comuns são o Linux, o FreeBSD, o Windows 7, o Windows 8 e o OS X da Apple. Sistemas operacionais de computadores pessoais são tão conhecidos que provavelmente é necessária pouca introdução. Na realidade, a maioria das pessoas nem sabe que existem outros tipos.

1.4.5 Sistemas operacionais de computadores portáteis

Seguindo com sistemas cada vez menores, chegamos aos tablets, smartphones e outros computadores portáteis. Um computador portátil, originalmente conhecido

como um **PDA (Personal Digital Assistant** — assistente pessoal digital), é um computador pequeno que pode ser seguro na mão durante a operação. Smartphones e tablets são os exemplos mais conhecidos. Como já vimos, esse mercado está dominado pelo Android do Google e o iOS da Apple, mas eles têm muitos competidores. A maioria deles conta com CPUs multinúcleo, GPS, câmeras e outros sensores, quantidades enormes de memória e sistemas operacionais sofisticados. Além disso, todos eles têm mais aplicativos ("**apps**") de terceiros que você possa imaginar.

1.4.6 Sistemas operacionais embarcados

Sistemas embarcados são executados em computadores que controlam dispositivos que não costumam ser vistos como computadores e que não aceitam softwares instalados pelo usuário. Exemplos típicos são os fornos de micro-ondas, os aparelhos de televisão, os carros, os aparelhos de DVD, os telefones tradicionais e os MP3 players. A principal propriedade que distingue sistemas embarcados dos portáteis é a certeza de que nenhum software não confiável vá ser executado nele um dia. Você não consegue baixar novos aplicativos para o seu forno de micro-ondas – todo o software está na memória ROM. Isso significa que não há necessidade para proteção entre os aplicativos, levando a simplificações no design. Sistemas como o Embedded Linux, 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 para uma série de finalidades. Esses nós são computadores minúsculos que se comunicam entre si e com uma estação-base usando comunicação sem fio. Redes de sensores são usadas para proteger os perímetros de prédios, guardar fronteiras nacionais, detectar incêndios em florestas, medir a temperatura e a precipitação para a previsão de tempo, colher informações sobre a movimentação de inimigos nos campos de batalha e muito mais.

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

Cada nó sensor é um computador verdadeiro, com uma CPU, RAM, ROM e um ou mais sensores ambientais. Ele executa um sistema operacional pequeno, mas verdadeiro, em geral orientado a eventos, respondendo a eventos externos ou tomando medidas periodicamente com base em um relógio interno. O sistema operacional tem de ser pequeno e simples, pois os nós têm uma RAM pequena e a duração da bateria é uma questão fundamental. Também, como com os sistemas embarcados, todos os programas são carregados antecipadamente; os usuários não inicializam subitamente os programas que eles baixaram da internet, o que torna o design muito mais simples. TinyOS é um sistema operacional bem conhecido para um nó sensor.

1.4.8 Sistemas operacionais de tempo real

Outro tipo de sistema operacional é o sistema de tempo real. Esses sistemas são caracterizados por ter o tempo como um parâmetro-chave. Por exemplo, em sistemas de controle de processo industrial, computadores em tempo real têm de coletar dados a respeito do processo de produção e usá-los para controlar máquinas na fábrica. Muitas vezes há prazos rígidos a serem cumpridos. Por exemplo, se um carro está seguindo pela linha de montagem, determinadas ações têm de ocorrer em dados instantes. Se, por exemplo, um robô soldador fizer as soldas cedo demais ou tarde demais, o carro será arruinado. Se a ação *tem* de ocorrer absolutamente em um determinado momento (ou dentro de uma dada faixa de tempo), temos um **sistema de tempo real crítico**. Muitos desses sistemas são encontrados no controle de processos industriais, aviônica, militar e áreas de aplicação semelhantes. Esses sistemas têm de fornecer garantias absolutas de que uma determinada ação ocorrerá em um determinado momento.

Um **sistema de tempo real não crítico** é aquele em que perder um prazo ocasional, embora não desejável, é aceitável e não causa danos permanentes. Sistemas de multimídia ou áudio digital caem nesta categoria. Smartphones também são sistemas de tempo real não críticos.

Tendo em vista que cumprir prazos é algo crucial nos sistemas de tempo real (críticos), às vezes o sistema operacional é nada mais que uma biblioteca conectada com os programas aplicativos, com todas as partes do sistema estreitamente acopladas e sem nenhuma proteção entre si. Um exemplo desse tipo de sistema de tempo real é o eCos.

As categorias de sistemas portáteis, embarcados e de tempo real se sobrepõem consideravelmente. Quase

todas elas têm pelo menos algum aspecto de tempo real não crítico. Os sistemas de tempo real e embarcado executam apenas softwares inseridos pelos projetistas do sistema; usuários não podem acrescentar seu próprio software, o que torna a proteção mais fácil. Os sistemas portáteis e embarcados são direcionados para os consumidores, ao passo que os sistemas de tempo real são mais voltados para o uso industrial. Mesmo assim, eles têm aspectos em comum.

1.4.9 Sistemas operacionais de cartões inteligentes (*smartcard*)

Os menores sistemas operacionais são executados em cartões inteligentes, que são dispositivos do tamanho de cartões de crédito contendo um chip de CPU. Possuem severas restrições de memória e processamento de energia. Alguns obtêm energia por contatos no leitor no qual estão inseridos, mas cartões inteligentes sem contato obtêm energia por indução, o que limita muito o que eles podem fazer. Alguns deles conseguem realizar somente uma função, como pagamentos eletrônicos, mas outros podem realizar múltiplas funções. Muitas vezes são sistemas proprietários.

Alguns cartões inteligentes são orientados a Java. Isso significa que o ROM no cartão inteligente contém um interpretador para a Java Virtual Machine (JVM — Máquina virtual Java). Os aplicativos pequenos (*applets*) Java são baixados para o cartão e são interpretados pelo JVM. Alguns desses cartões podem lidar com múltiplos *applets* Java ao mesmo tempo, levando à multiprogramação e à necessidade de escaloná-los. O gerenciamento de recursos e a proteção também se tornam um problema quando dois ou mais applets estão presentes ao mesmo tempo. Essas questões devem ser tratadas pelo sistema operacional (em geral extremamente primitivo) presente no cartão.

1.5 Conceitos de sistemas operacionais

A maioria dos sistemas operacionais fornece determinados conceitos e abstrações básicos, como processos, espaços de endereços e arquivos, que são fundamentais para compreendê-los. Nas seções a seguir, examinaremos alguns desses conceitos básicos de maneira bastante breve, como uma introdução. Voltaremos a cada um deles detalhadamente mais tarde neste livro. Para ilustrar esses conceitos, de tempos em tempos usaremos exemplos, geralmente tirados do UNIX. No entanto, exemplos similares existem em outros sistemas também, e estudaremos alguns deles mais tarde.

1.5.1 Processos

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

Voltaremos para o conceito de processo com muito mais detalhes no Capítulo 2. Por ora, a maneira mais fácil de compreender intuitivamente um processo é pensar a respeito do sistema de multiprogramação. O usuário pode ter inicializado um programa de edição de vídeo e o instruído a converter um vídeo de uma hora para um determinado formato (algo que pode levar horas) e então partido para navegar na web. Enquanto isso, um processo em segundo plano que desperta de tempos em tempos para conferir o e-mail que chega pode ter começado a ser executado. 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 começa a executar outro, talvez porque o primeiro utilizou mais do que sua parcela de tempo da CPU no último segundo ou dois.

Quando um processo é suspenso temporariamente assim, ele deve ser reiniciado mais tarde no exato mesmo estado em que estava quando foi parado. Isso significa que todas as informações a respeito do processo precisam ser explicitamente salvas em algum lugar durante a suspensão. Por exemplo, o processo pode ter vários arquivos abertos para leitura ao mesmo tempo. Há um ponteiro associado com cada um desses arquivos dando a posição atual (isto é, o número do byte ou registro a ser lido em seguida). Quando um processo está temporariamente suspenso, todos esses ponteiros têm de ser salvos de maneira que uma chamada read executada após o processo ter sido reiniciado vá ler os dados

corretos. Em muitos sistemas operacionais, todas as informações a respeito de cada processo, fora o conteúdo do seu próprio espaço de endereçamento, estão armazenadas em uma tabela do sistema operacional chamada de **tabela de processos**, que é um arranjo de estruturas, uma para cada processo existente no momento.

Desse modo, um processo (suspenso) consiste em seu espaço de endereçamento, em geral chamado de **imagem do núcleo** (em homenagem às memórias de núcleo magnético usadas antigamente), e de sua entrada na tabela de processo, que armazena os conteúdos de seus registradores e muitos outros itens necessários para reiniciar o processo mais tarde.

As principais chamadas de sistema de gerenciamento de processos são as que lidam com a criação e o término de processos. Considere um exemplo típico. Um processo chamado de **interpretador de comandos** ou shell lê os comandos de um terminal. O usuário acabou de digitar um comando requisitando que um programa seja compilado. O shell tem de criar agora um novo processo que vai executar o compilador. Quando 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 de **processos filhos**), e estes por sua vez podem criar processos filhos, chegamos logo à estrutura da árvore de processo da Figura 1.13. Processos relacionados que estão cooperando para finalizar alguma tarefa muitas vezes precisam comunicar-se entre si e sincronizar as atividades. Essa comunicação é chamada de **comunicação entre processos**, e será analisada detalhadamente no Capítulo 2.

Outras chamadas de sistemas de processos permitem requisitar mais memória (ou liberar memória não utilizada), esperar que um processo filho termine e sobrepor seu programa por um diferente.

Há ocasionalmente uma necessidade de se transmitir informação para um processo em execução que não está parado esperando por ela. Por exemplo, um processo que está se comunicando com outro em um computador diferente envia mensagens para o processo remoto por intermédio de uma rede de computadores. Para evitar a possibilidade de uma mensagem ou de sua resposta ser perdida, o emissor pode pedir para o seu próprio sistema operacional notificá-lo após um número especificado de segundos, de maneira que ele possa retransmitir a mensagem se nenhuma confirmação tiver sido recebida ainda. Após ligar esse temporizador, o programa pode continuar executando outra tarefa.

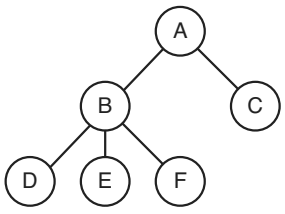
Decorrido o número especificado de segundos, o sistema operacional envia um **sinal de alarme** para o processo. O sinal faz que o processo suspenda por algum tempo o que quer que ele esteja fazendo, salve seus registradores na pilha e comece a executar uma rotina especial para tratamento desse sinal, por exemplo, para retransmitir uma mensagem presumivelmente perdida. Quando a rotina de tratamento desse sinal encerra sua ação, o processo em execução é reiniciado no estado em que se encontrava um instante antes do sinal. Sinais são os análogos em software das interrupções em hardwares e podem ser gerados por uma série de causas além de temporizadores expirando. Muitas armadilhas detectadas por hardwares, como executar uma instrução ilegal ou utilizar um endereço inválido, também são convertidas em sinais para o processo culpado.

A cada pessoa autorizada a usar um sistema é designada uma **UID** (**User IDentification** — identificação do usuário) pelo administrador do sistema. Todo processo iniciado tem a UID da pessoa que o iniciou. Um processo filho tem a mesma UID que o seu processo pai. Usuários podem ser membros de grupos, cada qual com uma **GID** (**Group IDentification** — identificação do grupo).

Uma UID, chamada de **superusuário** (em UNIX), ou **Administrador** (no Windows), tem um poder especial e pode passar por cima de muitas das regras de proteção. Em grandes instalações, apenas o administrador do sistema sabe a senha necessária para tornar-se um superusuário, mas muitos dos usuários comuns (especialmente estudantes) devotam um esforço considerável buscando falhas no sistema que permitam que eles se tornem superusuários sem a senha.

Estudaremos processos e comunicações entre processos no Capítulo 2.

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*.



1.5.2 Espaços de endereçamento

Todo computador tem alguma memória principal que ele usa para armazenar programas em execução. Em um sistema operacional muito simples, apenas um programa de cada 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 evitar que interfiram entre si (e com o sistema operacional), algum tipo de mecanismo de proteção é necessário. Embora esse mecanismo deva estar no hardware, ele é controlado pelo sistema operacional.

Este último ponto de vista diz respeito ao gerenciamento e à proteção da memória principal do computador. Uma questão diferente relacionada à memória, mas igualmente importante, é o gerenciamento de espaços de endereçamento dos processos. Em geral, cada processo tem algum conjunto de endereços que ele pode usar, tipicamente indo de 0 até algum máximo. No caso mais simples, a quantidade máxima de espaço de endereços que um processo tem é menor do que a memória principal. Dessa maneira, um processo pode preencher todo o seu espaço de endereçamento e haverá espaço suficiente na memória principal para armazená-lo inteiramente.

No entanto, em muitos computadores os endereços são de 32 ou 64 bits, dando um espaço de endereçamento de 2^{32} e 2^{64}, respectivamente. O que acontece se um processo tem mais espaço de endereçamento do que o computador tem de memória principal e o processo quer usá-lo inteiramente? Nos primeiros computadores, ele não teria sorte. Hoje, existe uma técnica chamada memória virtual, como já mencionado, na qual o sistema operacional mantém parte do espaço de endereçamento na memória principal e parte no disco, enviando trechos entre eles para lá e para cá conforme a necessidade. Na 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 espaço de endereçamento é desacoplado da memória física da máquina e pode ser maior ou menor do que a memória física. O gerenciamento de espaços de endereçamento e da memória física forma uma parte importante do que faz um sistema operacional, de maneira que todo o Capítulo 3 é devotado a esse tópico.

1.5.3 Arquivos

Outro conceito fundamental que conta com o suporte de virtualmente todos os sistemas operacionais é o sistema de arquivos. Como já foi observado, uma função importante do sistema operacional é esconder as peculiaridades dos discos e outros dispositivos de E/S e apresentar ao programador um modelo agradável e claro de arquivos que sejam independentes dos dispositivos. Chamadas de sistema são obviamente necessárias para criar, remover, ler e escrever arquivos. Antes que um arquivo possa ser lido, ele deve ser localizado no disco e aberto, e após ter sido lido, deve ser fechado, assim as chamadas de sistema são fornecidas para fazer essas coisas.

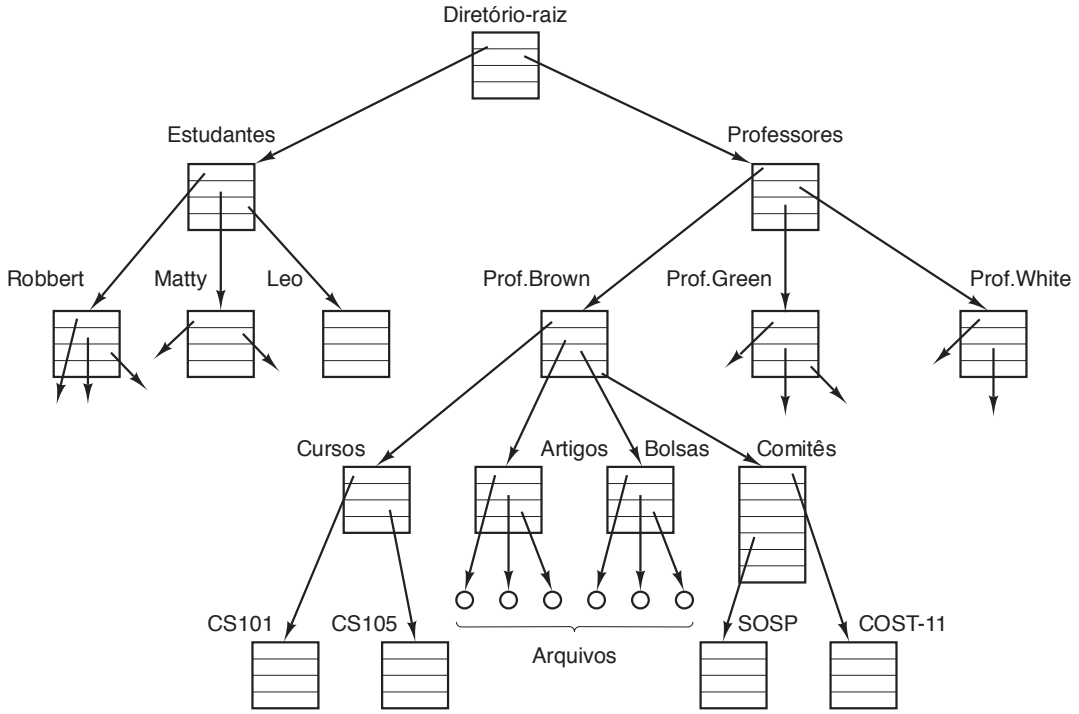
Para fornecer um lugar para manter os arquivos, a maioria dos sistemas operacionais de PCs tem o conceito de um **diretório** como uma maneira de agrupar os arquivos. Um estudante, por exemplo, pode ter um diretório para cada curso que ele estiver seguindo (para os programas necessários para aquele curso), outro para o correio eletrônico e ainda um para sua página na web. Chamadas de sistema são então necessárias para criar e remover diretórios. Chamadas também são fornecidas para colocar um arquivo existente em um diretório e para remover um arquivo de um diretório. Entradas de diretório podem ser de arquivos ou de outros diretórios. Esse modelo também dá origem a uma hierarquia — o sistema de arquivos — como mostrado na Figura 1.14.

Ambas as hierarquias de processos e arquivos são organizadas como árvores, mas a similaridade para aí. Hierarquias de processos em geral não são muito profundas (mais do que três níveis é incomum), enquanto hierarquias de arquivos costumam ter quatro, cinco, ou mesmo mais níveis de profundidade. Hierarquias de processos tipicamente têm vida curta, em geral minutos no máximo, enquanto hierarquias de diretórios podem existir por anos. Propriedade e proteção também diferem para processos e arquivos. Normalmente, apenas um processo pai pode controlar ou mesmo acessar um processo filho, mas quase sempre existem mecanismos para permitir que arquivos e diretórios sejam lidos por um grupo mais amplo do que apenas o proprietário.

Todo arquivo dentro de uma hierarquia de diretório pode ser especificado fornecendo o seu **nome de caminho** a partir do topo da hierarquia do diretório, o **diretório-raiz**. Esses nomes de caminho absolutos consistem na lista de diretórios que precisam ser percorridos a partir do diretório-raiz para se chegar ao 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, começando no diretório-raiz. Como nota, no Windows, o caractere barra invertida (\) é usado como o separador em vez do caractere da barra (/) por razões históricas, então o caminho do arquivo acima seria escrito como *\Professores\Prof.Brown\Cursos\CS101*. Ao longo deste livro geralmente usaremos a convenção UNIX para os caminhos.

A todo instante, cada processo tem um **diretório de trabalho** atual, no qual são procurados nomes de caminhos que não começam com uma barra. Por

FIGURA 1.14 Um sistema de arquivos para um departamento universitário.



exemplo, na Figura 1.14, se */Professores/Prof.Brown* fosse o diretório de trabalho, o uso do caminho *Cursos/CS101* resultaria no mesmo arquivo que o nome de caminho absoluto dado anteriormente. Os processos podem mudar seu diretório de trabalho emitindo uma chamada de sistema especificando o novo diretório de trabalho.

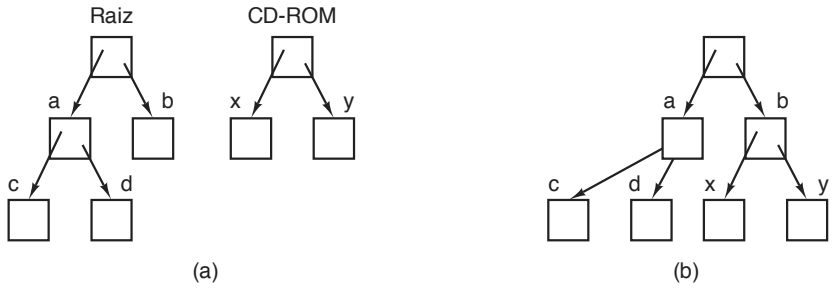
Antes que um arquivo possa ser lido ou escrito, ele precisa ser aberto, momento em que as permissões são conferidas. Se o acesso for permitido, o sistema retorna um pequeno valor inteiro, chamado **descritor de arquivo**, para usá-lo em operações subsequentes. Se o acesso for proibido, um código de erro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

Outro conceito importante em UNIX é o de montagem do sistema de arquivos. A maioria dos computadores de mesa tem uma ou mais unidades de discos óticos nas quais CD-ROMs, DVDs e discos de Blu-ray podem ser inseridos. Eles quase sempre têm portas USB, nas quais dispositivos de memória USB (na realidade, unidades de disco em estado sólido) podem ser conectados, e alguns computadores têm discos flexíveis ou discos rígidos externos. Para fornecer uma maneira elegante de lidar com essa mídia removível, a UNIX permite que o sistema de arquivos no disco ótico seja agregado à árvore principal. Considere a situação da Figura 1.15(a). Antes da chamada mount, o **sistema de arquivos-raiz** no disco rígido e um segundo sistema de arquivos, em um CD-ROM, estão separados e desconexos.

No entanto, o sistema de arquivos no CD-ROM não pode ser usado, pois não há como especificar nomes de caminhos nele. O UNIX não permite que nomes de caminhos sejam prefixados por um nome ou número de um dispositivo acionador; esse seria precisamente o tipo de dependência de dispositivos que os sistemas operacionais deveriam eliminar. Em vez disso, a chamada de sistema mount permite que o sistema de arquivos no CD-ROM seja agregado ao sistema de arquivos-raiz sempre que seja pedido pelo programa. Na Figura 1.15(b) o sistema de arquivos no CD-ROM foi montado no diretório *b*, permitindo assim acesso aos arquivos */b/x* e */b/y*. Se o diretório *b* contivesse quaisquer arquivos, eles não seriam acessíveis enquanto o CD-ROM estivesse montado, tendo em vista que */b* se referiria ao diretório-raiz do CD-ROM. (A impossibilidade de acessar esses arquivos não é tão sério quanto possa parecer em um primeiro momento: sistemas de arquivos são quase sempre montados em diretórios vazios). Se um sistema contém múltiplos discos rígidos, todos eles podem ser montados em uma única árvore também.

Outro conceito importante em UNIX é o **arquivo especial**. Arquivos especiais permitem que dispositivos de E/S se pareçam com arquivos. Dessa maneira, eles podem ser lidos e escritos com as mesmas chamadas de sistema que são usadas para ler e escrever arquivos. Existem dois tipos especiais: **arquivos especiais de bloco** e **arquivos especiais de caracteres**. Arquivos

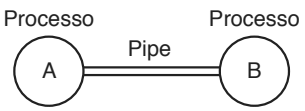
FIGURA 1.15 (a) Antes da montagem, os arquivos no CD-ROM não estão acessíveis. (b) Depois da montagem, eles fazem parte da hierarquia de arquivos.



especiais de bloco são usados para modelar dispositivos que consistem em uma coleção de blocos aleatoriamente endereçáveis, como discos. Ao abrir um arquivo especial de bloco e ler, digamos, bloco 4, um programa pode acessar diretamente o quarto bloco no dispositivo, sem levar em consideração a estrutura do sistema de arquivo contido nele. De modo similar, arquivos especiais de caracteres são usados para modelar impressoras, modems e outros dispositivos que aceitam ou enviam um fluxo de caracteres. Por convenção, os arquivos especiais são mantidos no diretório */dev*. Por exemplo, */dev/lp* pode ser a impressora — que um dia já foi chamada de impressora de linha (line printer).

O último aspecto que discutiremos nesta visão geral relaciona-se tanto com os processos quanto com os arquivos: os pipes. Um **pipe** é uma espécie de pseudoarquivo que pode ser usado para conectar dois processos, como mostrado na Figura 1.16. Se os processos *A* e *B* querem conversar usando um pipe, eles têm de configurá-lo antes. Quando o processo *A* quer enviar dados para o processo *B*, ele escreve no pipe como se ele fosse um arquivo de saída. Na realidade, a implementação de um pipe lembra muito a de um arquivo. O processo *B* pode ler os dados a partir do pipe como se ele fosse um arquivo de entrada. Desse modo, a comunicação entre os processos em UNIX se parece muito com a leitura e escrita de arquivos comuns. É ainda mais forte, pois a única maneira pela qual um processo pode descobrir se o arquivo de saída em que ele está escrevendo não é realmente um arquivo, mas um pipe, é fazendo uma chamada de sistema especial. Sistemas de arquivos são muito importantes. Teremos muito para falar a respeito deles no Capítulo 4 e também nos capítulos 10 e 11.

FIGURA 1.16 Dois processos conectados por um pipe.



1.5.4 Entrada/Saída

Todos os computadores têm dispositivos físicos para obter entradas e produzir saídas. Afinal, para que serviria um computador se os usuários não pudessem dizer a ele o que fazer e não pudessem receber os resultados após ele ter feito o trabalho pedido? Existem muitos tipos de dispositivos de entrada e de saída, incluindo teclados, monitores, impressoras e assim por diante. Cabe ao sistema operacional gerenciá-los.

Em consequência, todo sistema operacional tem um subsistema de E/S para gerenciar os dispositivos de E/S. Alguns softwares de E/S são independentes do dispositivo, isto é, aplicam-se igualmente bem a muitos ou a todos dispositivos de E/S. Outras partes dele, como drivers de dispositivo, são específicos a dispositivos de E/S particulares. No Capítulo 5 examinaremos o software de E/S.

1.5.5 Proteção

Computadores contêm grandes quantidades de informações que os usuários muitas vezes querem proteger e manter confidenciais. Essas informações podem incluir e-mails, planos de negócios, declarações fiscais e muito mais. Cabe ao sistema operacional gerenciar a segurança do sistema de maneira que os arquivos, por exemplo, sejam acessíveis somente 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 designando-se a cada arquivo um código de proteção binário de 9 bits. O código de proteção consiste de três campos de 3 bits, um para o proprietário, um para os outros membros do grupo do proprietário (usuários são divididos em grupos pelo administrador do sistema) e um para todos os demais usuários. Cada campo tem um bit de permissão de leitura, um bit de permissão de escrita e um bit

de permissão de execução. Esses 3 bits são conhecidos como os **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 que todos os demais podem executar (mas não ler ou escrever) o arquivo. Para um diretório, *x* indica permissão de busca. Um traço significa que a permissão correspondente está ausente.

Além da proteção ao arquivo, há muitas outras questões de segurança. Proteger o sistema de intrusos indesejados, humanos ou não (por exemplo, vírus) é uma delas. Examinaremos várias questões 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*), programas utilitários e interpretadores de comandos definitivamente não fazem parte do sistema operacional, mesmo que sejam importantes e úteis. Correndo o risco de confundir as coisas de certa maneira, nesta seção examinaremos brevemente o interpretador de comandos UNIX, o shell. Embora não faça parte do sistema operacional, ele faz um uso intensivo de muitos aspectos do sistema operacional e serve assim como um bom exemplo de como as chamadas de sistema são usadas. Ele também é a principal interface entre um usuário sentado no seu terminal e o sistema operacional, a não ser que o usuário esteja usando uma interface de usuário gráfica. Muitos shells existem, incluindo, *sh, csh, ksh* e *bash*. Todos eles dão suporte à funcionalidade descrita a seguir, derivada do shell (*sh*) original.

Quando qualquer usuário se conecta, um shell é iniciado. O shell tem o terminal como entrada-padrão e saída-padrão. Ele inicia emitindo um caractere de **prompt**, um caractere como o cifrão do dólar, que diz ao usuário que o shell está esperando para aceitar um comando. Se o usuário agora digitar

```
date
```

por exemplo, o shell cria um processo filho e executa o programa *date* como um filho. Enquanto o processo filho estiver em execução, o shell espera que ele termine. Quando o filho termina, o shell emite o sinal de prompt de novo e tenta ler a próxima linha de entrada.

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

```
date >file
```

De modo similar, a entrada-padrão pode ser redirecionada, como em

```
sort <file1 >file2
```

que invoca o programa sort com a entrada vindo de *file1* e a saída enviada para *file2*.

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

```
cat file1 file2 file3 | sort >/dev/lp
```

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

Se um usuário coloca um & após um comando, o shell não espera que ele termine. Em vez disso, ele dá um prompt imediatamente. Em consequência,

```
cat file1 file2 file3 | sort >/dev/lp &
```

inicia o *sort* como uma tarefa de segundo plano, permitindo que o usuário continue trabalhando normalmente enquanto o ordenamento prossegue. O shell tem uma série de outros aspectos interessantes, mas que não temos espaço para discuti-los aqui. A maioria dos livros em UNIX discute o shell mais detalhadamente (por exemplo, KERNIGHAN e PIKE, 1984; QUIGLEY, 2004; ROBBINS, 2005).

A maioria dos computadores pessoais usa hoje uma interface gráfica GUI. Na realidade, a GUI é apenas um programa sendo executado em cima do sistema operacional, como um shell. Nos sistemas Linux, esse fato é óbvio, pois o usuário tem uma escolha de (pelo menos) duas GUIs: Gnome e KDE ou nenhuma (usando uma janela de terminal no X11). No Windows, também é possível substituir a área de trabalho com interface GUI padrão (*Windows Explorer*) por um programa diferente alterando alguns programas no registro, embora poucas pessoas o façam.

1.5.7 A ontogenia recapitula a filogenia

Após o livro de Charles Darwin *A origem das espécies* ter sido publicado, o zoólogo alemão Ernst Haeckel declarou que "a ontogenia recapitula a filogenia". Com isso ele queria dizer que o desenvolvimento de um embrião (ontogenia) repete (isto é, recapitula) a evolução da espécie (filogenia). Em outras palavras, após a fertilização, um ovo humano passa pelos estágios de ser um peixe, um porco e assim por diante, antes de transformar-se em um bebê humano. Biólogos modernos

consideram isso uma simplificação grosseira, mas ainda há alguma verdade nela.

Algo vagamente análogo aconteceu na indústria de computadores. Cada nova espécie (computador de grande porte, minicomputador, computador pessoal, portátil, embarcado, cartões inteligentes etc.) parece passar pelo mesmo desenvolvimento que seus antecessores, tanto em hardware quanto em software. Muitas vezes esquecemos que grande parte do que acontece no negócio dos computadores e em um monte de outros campos é impelido pela tecnologia. A razão por que os romanos antigos não tinham carros não era por eles gostarem tanto de caminhar. É porque não sabiam como construir carros. Computadores pessoais existem *não* porque milhões de pessoas têm um desejo contido de centenas de anos de ter um computador, mas porque agora é possível fabricá-los barato. Muitas vezes esquecemos o quanto a tecnologia afeta nossa visão dos sistemas e vale a pena refletir sobre isso de vez em quando.

Em particular, acontece com frequência de uma mudança na tecnologia tornar uma ideia obsoleta e ela rapidamente desaparece. No entanto, outra mudança na tecnologia poderia revivê-la. Isso é especialmente verdadeiro quando a mudança tem a ver com o desempenho relativo de diferentes partes do sistema. Por exemplo, quando as CPUs se tornaram muito mais rápidas do que as memórias, caches se tornaram importantes para acelerar a memória "lenta". Se a nova tecnologia de memória algum dia tornar as memórias muito mais rápidas do que as CPUs, as caches desaparecerão. E se uma nova tecnologia de CPU torná-las mais rápidas do que as memórias novamente, as caches reaparecerão. Na biologia, a extinção é para sempre, mas, na ciência de computadores, às vezes ela é apenas por alguns anos.

Como uma consequência dessa impermanência, examinaremos de tempos em tempos neste livro conceitos "obsoletos", isto é, ideias que não são as melhores para a tecnologia atual. No entanto, mudanças na tecnologia podem trazer de volta alguns dos chamados "conceitos obsoletos". Por essa razão, é importante compreender por que um conceito é obsoleto e quais mudanças no ambiente podem trazê-lo de volta.

Para esclarecer esse ponto, vamos considerar um exemplo simples. Os primeiros computadores tinham conjuntos de instruções implementados no hardware. As instruções eram executadas diretamente pelo hardware e não podiam ser mudadas. Então veio a microprogramação (introduzida pela primeira vez em grande escala com o IBM 360), no qual um interpretador subjacente executava as "instruções do hardware" no software. A execução implementada no hardware tornou-se obsoleta. Ela não era suficientemente flexível. Então os computadores RISC foram inventados, e a microprogramação (isto é, execução interpretada) tornou-se obsoleta porque a execução direta era mais rápida. Agora estamos vendo o ressurgimento da interpretação na forma de applets Java, que são enviados pela internet e interpretados na chegada. A velocidade de execução nem sempre é crucial, pois os atrasos de rede são tão grandes que eles tendem a predominar. Desse modo, o pêndulo já oscilou vários ciclos entre a execução direta e a interpretação e pode ainda oscilar novamente no futuro.

Memórias grandes

Vamos examinar agora alguns desenvolvimentos históricos em hardware e como eles afetaram o software repetidamente. Os primeiros computadores de grande porte tinham uma memória limitada. Um IBM 7090 ou um 7094 completamente carregados, que eram os melhores computadores do final de 1959 até 1964, tinha apenas um pouco mais de 128 KB de memória. Em sua maior parte, eram programados em linguagem de montagem e seu sistema operacional era escrito nessa linguagem para poupar a preciosa memória.

Com o passar do tempo, compiladores para linguagens como FORTRAN e COBOL tornaram-se tão bons que a linguagem de montagem foi abandonada. Mas quando o primeiro minicomputador comercial (o PDP-1) foi lançado, ele tinha apenas 4.096 palavras de 18 bits de memória, e a linguagem de montagem fez um retorno surpreendente. Por fim, os minicomputadores adquiriram mais memória e as linguagens de alto nível tornaram-se prevalentes neles.

Quando os microcomputadores tornaram-se um sucesso no início da década de 1980, os primeiros tinham memórias de 4 KB e a programação de linguagem de montagem foi ressuscitada. Computadores embarcados muitas vezes usam os mesmos chips de CPU que os microcomputadores (8080s, Z80s e mais tarde 8086s) e também inicialmente foram programados em linguagem de montagem. Hoje, seus descendentes, os computadores pessoais, têm muita memória e são programados em C, C++, Java e outras linguagens de alto nível. Cartões inteligentes estão passando por um desenvolvimento similar, embora a partir de um determinado tamanho, os cartões inteligentes tenham um interpretador Java e executem os programas Java de maneira interpretativa, em vez de ter o Java compilado para a linguagem de máquina do cartão inteligente.

Hardware de proteção

Os primeiros computadores de grande porte, como o IBM 7090/7094, não tinham hardware de proteção, de maneira que eles executavam apenas um programa de cada vez. Um programa defeituoso poderia acabar com o sistema operacional e facilmente derrubar a máquina. Com a introdução do IBM 360, uma forma primitiva de proteção de hardware tornou-se disponível. Essas máquinas podiam então armazenar vários programas na memória ao mesmo tempo e deixá-los que se alternassem na execução (multiprogramação). A monoprogramação tornou-se obsoleta.

Pelo menos até o primeiro minicomputador aparecer — sem hardware de proteção — a multiprogramação não era possível. Embora o PDP-1 e o PDP-8 não tivessem hardware de proteção, finalmente o PDP-11 teve, e esse aspecto levou à multiprogramação e por fim ao UNIX.

Quando os primeiros microcomputadores foram construídos, eles usavam o chip de CPU Intel 8080, que não tinha proteção de hardware, então estávamos de volta à monoprogramação — um programa na memória de cada vez. Foi somente com o chip 80286 da Intel que o hardware de proteção foi acrescentado e a multiprogramação tornou-se possível. Até hoje, muitos sistemas embarcados não têm hardware de proteção e executam apenas um único programa.

Agora vamos examinar os sistemas operacionais. Os primeiros computadores de grande porte inicialmente não tinham hardware de proteção e nenhum suporte para multiprogramação, então sistemas operacionais simples eram executados neles. Esses sistemas lidavam com apenas um programa carregado manualmente por vez. Mais tarde, eles adquiriram o suporte de hardware e sistema operacional para lidar com múltiplos programas ao mesmo tempo, e então capacidades de compartilhamento de tempo completas.

Quando os minicomputadores apareceram pela primeira vez, eles também não tinham hardware de proteção e os programas carregados manualmente eram executados um a um, mesmo com a multiprogramação já bem estabelecida no mundo dos computadores de grande porte. Pouco a pouco, eles adquiriram hardware de proteção e a capacidade de executar dois ou mais programas ao mesmo tempo. Os primeiros microcomputadores também eram capazes de executar apenas um programa de cada vez, porém mais tarde adquiriram a capacidade de multiprogramar. Computadores portáteis e cartões inteligentes seguiram o mesmo caminho.

Em todos os casos, o desenvolvimento do software foi ditado pela tecnologia. Os primeiros microcomputadores, por exemplo, tinham algo como 4 KB de memória e nenhum hardware de proteção. Linguagens de alto nível e a multiprogramação eram simplesmente demais para um sistema tão pequeno lidar. À medida que os microcomputadores evoluíram para computadores pessoais modernos, eles adquiriam o hardware necessário e então o software necessário para lidar com aspectos mais avançados. É provável que esse desenvolvimento vá continuar por muitos anos ainda. Outros campos talvez também tenham esse ciclo de reencarnação, mas na indústria dos computadores ele parece girar mais rápido.

Discos

Os primeiros computadores de grande porte eram em grande parte baseados em fitas magnéticas. Eles liam um programa a partir de uma fita, compilavam-no e escreviam os resultados de volta para 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) em 1956. Ele ocupava cerca de 4 m² de espaço e podia armazenar 5 milhões de caracteres de 7 bits, o suficiente para uma foto digital de resolução média. Mas com uma taxa de aluguel anual de US$ 35.000, reunir um número suficiente deles para armazenar o equivalente a um rolo de filme tornava-se caro rapidamente. Mas por fim os preços baixaram e os sistemas de arquivos primitivos foram desenvolvidos.

Representativo desses novos desenvolvimentos foi o CDC 6600, introduzido em 1964 e, por anos, de longe o computador mais rápido no mundo. Usuários podiam criar os chamados "arquivos permanentes" dando a eles nomes e esperando que nenhum outro usuário tivesse decidido que, digamos, "dados" fosse um nome adequado para um arquivo. Tratava-se de um diretório de um único nível. Por fim, computadores de grande porte desenvolveram sistemas de arquivos hierárquicos complexos, talvez culminando no sistema de arquivos MULTICS.

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

Memória virtual

A memória virtual (discutida no Capítulo 3) proporciona a capacidade de executar programas maiores do que a memória física da máquina, rapidamente movendo pedaços entre a memória RAM e o disco. Ela passou por um desenvolvimento similar, primeiro aparecendo nos computadores de grande porte, então passando para os minis e os micros. A memória virtual também permitiu que um programa se conectasse dinamicamente a uma biblioteca no momento da execução em vez de fazê-lo na compilação. O MULTICS foi o primeiro sistema a permitir isso. Por fim, a ideia propagou-se adiante e agora é amplamente usada na maioria dos sistemas UNIX e Windows.

Em todos esses desenvolvimentos, vemos ideias inventadas em um contexto e mais tarde jogadas fora quando o contexto muda (programação em linguagem de montagem, monoprogramação, diretórios em nível único etc.) apenas para reaparecer em um contexto diferente muitas vezes uma década mais tarde. Por essa razão, neste livro às vezes veremos ideias e algoritmos que talvez pareçam datados nos PCs de gigabytes de hoje, mas que podem voltar logo em computadores embarcados e cartões inteligentes.

1.6 Chamadas de sistema

Vimos que os sistemas operacionais têm duas funções principais: fornecer abstrações para os programas de usuários e gerenciar os recursos do computador. Em sua maior parte, a interação entre programas de usuários e o sistema operacional lida com a primeira; por exemplo, criar, escrever, ler e deletar arquivos. A parte de gerenciamento de arquivos é, em grande medida, transparente para os usuários e feita automaticamente. Desse modo, a interface entre os programas de usuários e o sistema operacional diz respeito fundamentalmente a abstrações. Para compreender de verdade o que os sistemas operacionais fazem, temos de examinar essa interface de perto. As chamadas de sistema disponíveis na interface variam de um sistema para outro (embora os conceitos subjacentes tendam a ser similares).

Somos então forçados a fazer uma escolha entre (1) generalidades vagas ("sistemas operacionais têm chamadas de sistema para ler arquivos") e (2) algum sistema específico ("UNIX possui uma chamada de sistema read com três parâmetros: um para especificar o arquivo, um para dizer onde os dados devem ser colocados e outro para dizer quantos bytes devem ser lidos").

Escolhemos a segunda abordagem. Ela é mais trabalhosa, mas proporciona um entendimento melhor sobre o que os sistemas operacionais realmente fazem. Embora essa discussão se refira especificamente ao POSIX (International Standard 9945-1), em consequência também o UNIX, System V, BSD, Linux, MINIX 3 e assim por diante, a maioria dos outros sistemas operacionais modernos tem chamadas de sistema que desempenham as mesmas funções, mesmo que os detalhes difiram. Como os mecanismos reais de emissão de uma chamada de sistema são altamente dependentes da máquina e muitas vezes devem ser expressos em código de montagem, uma biblioteca de rotinas é fornecida para tornar possível fazer chamadas de sistema de programas C e muitas vezes de outras linguagens também.

Convém ter o seguinte em mente. Qualquer computador de uma única CPU pode executar apenas uma instrução de cada vez. Se um processo estiver executando um programa de usuário em modo de usuário e precisa de um serviço de sistema, como ler dados de um arquivo, ele tem de executar uma instrução de armadilha (*trap*) para transferir o controle para o sistema operacional. O sistema operacional verifica os parâmetros e descobre o que o processo que está chamando quer. Então ele executa a chamada de sistema e retorna o controle para a instrução seguinte à chamada de sistema. De certa maneira, fazer uma chamada de sistema é como fazer um tipo especial de chamada de rotina, apenas que as chamadas de sistema entram no núcleo e as chamadas de rotina, não.

Para esclarecer o mecanismo de chamada de sistema, vamos fazer uma análise rápida da chamada de sistema read. Como mencionado anteriormente, ela tem três parâmetros: o primeiro especificando o arquivo, o segundo é um ponteiro para o buffer e o terceiro dá o número de bytes a ser lido. Como quase todas as chamadas de sistema, ele é invocado de programas C chamando uma rotina de biblioteca com o mesmo nome que a chamada de sistema: *read*. Uma chamada de um programa C pode parecer desta forma:

```
contador = read(fd, buffer, nbytes)
```

A chamada de sistema (e a rotina de biblioteca) retornam o número de bytes realmente lidos em *contador*. Esse valor é normalmente o mesmo que *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 por causa de um parâmetro inválido ou de um erro de disco, o *contador* passa a valer −1, e o número de erro é colocado em uma variável global, *errno*. Os programas

devem sempre conferir os resultados de uma chamada de sistema para ver se um erro ocorreu.

Chamadas de sistema são realizadas em uma série de passos. Para deixar o conceito mais claro, vamos examinar a chamada read discutida anteriormente. Em preparação para chamar a rotina de biblioteca *read*, que na realidade é quem faz a chamada de sistema read, o programa de chamada primeiro empilha os parâmetros, como mostrado nos passos 1 a 3 na Figura 1.17.

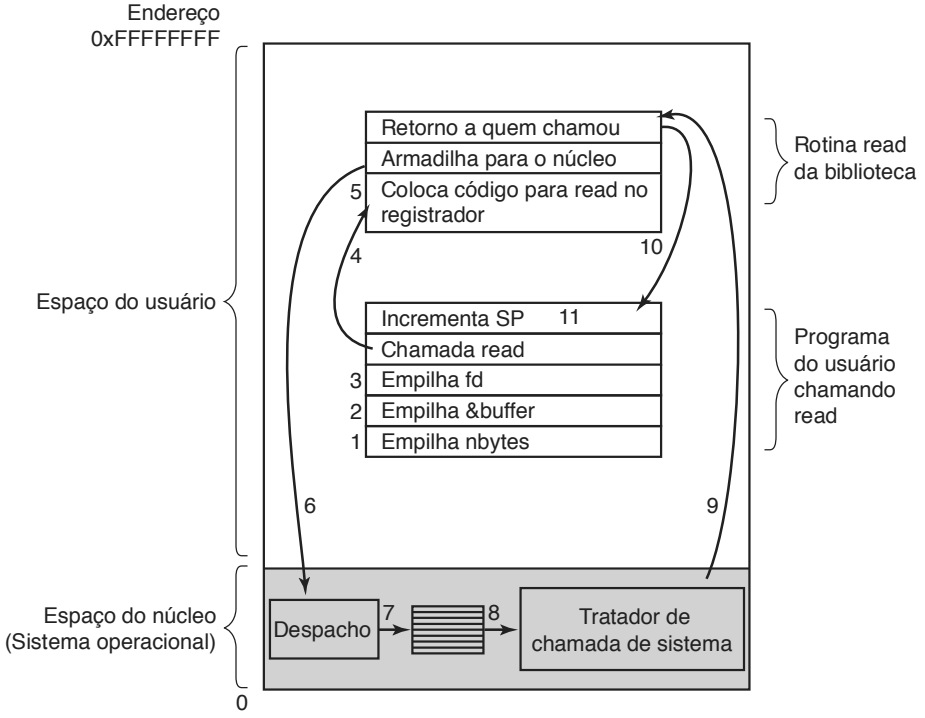
Os compiladores C e C++ empilham os parâmetros em ordem inversa por razões históricas (a ideia é fazer o primeiro parâmetro de *printf*, a cadeia de caracteres do formato, aparecer no topo da pilha). O primeiro e o terceiro parâmetros são chamados por valor, mas o segundo parâmetro é passado por referência, significando que o endereço do buffer (indicado por &) é passado, não seu conteúdo. Então vem a chamada real para a 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 de montagem, tipicamente coloca o número da chamada de sistema em um lugar onde o sistema operacional a espera, como um registro (passo 5). Então ela executa uma instrução TRAP para passar do modo usuário para o modo núcleo e começar a execução em um endereço fixo dentro do núcleo (passo 6). A instrução TRAP é na realidade relativamente similar à instrução de chamada de rotina no sentido de que a instrução que a segue é tirada de um local distante e o endereço de retorno é salvo na pilha para ser usado depois.

Entretanto, a instrução TRAP também difere da instrução de chamada de rotina de duas maneiras fundamentais. Primeiro, como efeito colateral, ela troca para o modo núcleo. A instrução de chamada de rotina não muda o modo. Segundo, em vez de dar um endereço relativo ou absoluto onde a rotina está localizada, a instrução TRAP não pode saltar para um endereço arbitrário. Dependendo da arquitetura, ela salta para um único local fixo, ou há um campo de 8 bits na instrução fornecendo o índice para uma tabela na memória contendo endereços para saltar, ou algo equivalente.

O código de núcleo que se inicia seguindo a instrução TRAP examina o número da chamada de sistema e então o despacha para o tratador correto da chamada de sistema, normalmente através de uma tabela de ponteiros que designam as rotinas de tratamento de chamadas de sistema indexadas pelo número da chamada (passo 7). Nesse ponto, é executado o tratamento de chamada de sistema (passo 8). Uma vez que ele tenha completado o seu trabalho, o controle pode ser retornado para a rotina de biblioteca no espaço do usuário na instrução após a instrução TRAP (passo 9). Essa rotina retorna para o programa do usuário da maneira usual que as chamadas de rotina retornam (passo 10).

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



Para terminar a tarefa, o programa do usuário tem de limpar a pilha, como ele faz após qualquer chamada de rotina (passo 11). Presumindo que a pilha cresce para baixo, como muitas vezes é o caso, o código compilado incrementa o ponteiro da pilha exatamente o suficiente para remover os parâmetros empilhados antes da chamada *read*. O programa está livre agora para fazer o que quiser em seguida.

No passo 9, dissemos "pode ser retornado para a rotina de biblioteca no espaço do usuário" por uma boa razão. A chamada de sistema pode bloquear quem a chamou, impedindo-o de seguir. Por exemplo, se ele está tentando ler do teclado e nada foi digitado ainda, ele tem de ser bloqueado. Nesse caso, o sistema operacional vai procurar à sua volta para ver se algum outro processo pode ser executado em seguida. Mais tarde, quando a entrada desejada estiver disponível, esse processo receberá a atenção do sistema e executará os passos 9-11.

Nas seções a seguir, examinaremos algumas das chamadas de sistema POSIX mais usadas, ou mais especificamente, as rotinas de biblioteca que fazem uso dessas chamadas de sistema. POSIX tem cerca de 100 chamadas de rotina. Algumas das mais importantes estão listadas na Figura 1.18, agrupadas por conveniência em quatro categorias. No texto, examinaremos brevemente cada chamada para ver o que ela faz.

Em grande medida, os serviços oferecidos por essas chamadas determinam a maior parte do que o sistema operacional tem de fazer, tendo em vista que o gerenciamento de recursos em computadores pessoais é mínimo (pelo menos comparado a grandes máquinas com múltiplos usuários). Os serviços incluem coisas como criar e finalizar processos, criar, excluir, ler e escrever arquivos, gerenciar diretórios e realizar entradas e saídas.

Como nota, vale a pena observar que o mapeamento de chamadas de rotina POSIX em chamadas de sistema não é de uma para uma. O padrão POSIX especifica uma série de procedimentos que um sistema em conformidade com esse padrão deve oferecer, mas ele não especifica se elas são chamadas de sistema, chamadas de biblioteca ou algo mais. Se uma rotina pode ser executada sem invocar uma chamada de sistema (isto é, sem um desvio para o núcleo), normalmente ela será realizada no espaço do usuário por questões de desempenho. No entanto, a maioria das rotinas POSIX invoca chamadas de sistema, em geral com uma rotina mapeando diretamente uma chamada de sistema. Em alguns casos — especialmente onde várias rotinas exigidas são apenas pequenas variações umas das outras — uma chamada de sistema lida com mais de uma chamada de biblioteca.

1.6.1 Chamadas de sistema para gerenciamento de processos

O primeiro grupo de chamadas na Figura 1.18 lida com o gerenciamento de processos. A chamada fork é um bom ponto para se começar a discussão. A chamada fork é a única maneira para se criar um processo novo em POSIX. Ela cria uma cópia exata do processo original, incluindo todos os descritores de arquivos, registradores — tudo. Após a fork, o processo original e a cópia (o processo pai e o processo filho) seguem seus próprios caminhos separados. Todas as variáveis têm valores idênticos no momento da 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 os processos pai e filho). A chamada fork retorna um valor, que é zero no processo filho e igual ao **PID (Process IDentifier** — identificador de processo) do processo filho no processo pai. Usando o PID retornado, os dois processos podem ver qual é o processo pai e qual é o filho.

Na maioria dos casos, após uma fork, o processo filho precisará executar um código diferente do processo pai. Considere o caso do shell. Ele lê um comando do terminal, cria um processo filho, espera que ele execute o comando e então lê o próximo comando quando o processo filho termina. Para esperar que o processo filho termine, o processo pai executa uma chamada de sistema waitpid, que apenas espera até o processo filho terminar (qualquer processo filho se mais de um existir). Waitpid pode esperar por um processo filho específico ou por qualquer filho mais velho configurando-se o primeiro parâmetro em –1. Quando waitpid termina, o endereço apontado pelo segundo parâmetro, *statloc*, será configurado como estado de saída do processo filho (término normal ou anormal e valor de saída). Várias opções também são fornecidas, especificadas pelo terceiro parâmetro. Por exemplo, retornar imediatamente se nenhum processo filho já tiver terminado.

Agora considere como a fork é usada pelo shell. Quando um comando é digitado, o shell cria um novo processo. Esse processo filho tem de executar o comando de usuário. Ele o faz usando a chamada de sistema execve, que faz que toda a sua imagem de núcleo seja substituída pelo arquivo nomeado no seu primeiro parâmetro. (Na realidade, a chamada de sistema em si é exec, mas várias rotinas de biblioteca a chamam com parâmetros diferentes e nomes ligeiramente diferentes. Nós as trataremos aqui como chamadas de sistema.) Um shell altamente simplificado ilustrando o uso de fork, waitpid e execve é mostrado na Figura 1.19.

FIGURA 1.18 Algumas das principais chamadas de sistema 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 um deslocamento no interior do arquivo e *seconds* é o tempo decorrido. Os parâmetros são explicados no texto.

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

FIGURA 1.19 Um interpretador de comandos simplificado. Neste livro, presume-se que *TRUE* seja definido como 1.*

```
#define TRUE 1

while (TRUE) {                                  /* repita para sempre */
      type_prompt( );                           /* mostra prompt na tela */
      read_command(command, parameters);        /* le entrada do terminal */

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

* A linguagem C não permite o uso de caracteres com acentos, por isso os comentários neste e em outros códigos C estão sem acentuação. (N.R.T.)

No caso mais geral, execve possui três parâmetros: o nome do arquivo a ser executado, um ponteiro para o arranjo de argumentos e um ponteiro para o arranjo de ambiente. Esses parâmetros serão descritos brevemente. Várias rotinas de biblioteca, incluindo *execl, execv, execle* e *execve*, são fornecidas para permitir que os parâmetros sejam omitidos ou especificados de várias maneiras. Neste livro, usaremos o nome exec para representar a chamada de sistema invocada por todas essas rotinas.

Vamos considerar o caso de um comando como

```
cp fd1 fd2
```

usado para copiar o *fd1* para o *fd2*. Após o shell ter criado 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 programa principal da maioria dos outros programas C) contém a declaração

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

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

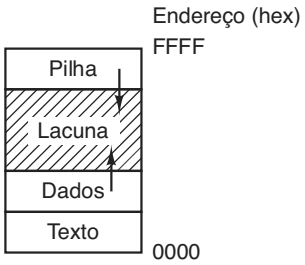
O segundo parâmetro, *argv*, é um ponteiro para um arranjo. O elemento *i* do arranjo é um ponteiro para a *i*-ésima cadeia de caracteres na linha de comando. Em nosso exemplo, *argv*[0] apontaria para a cadeia de caracteres "cp", *argv*[1] apontaria para a "fd1" e *argv*[2] apontaria para a "fd2".

O terceiro parâmetro do *main*, *envp*, é um ponteiro para o ambiente, um arranjo de cadeias de caracteres contendo atribuições da forma *nome* = *valor* usadas para passar informações como o tipo de terminal e o nome do diretório home para programas. Há rotinas de biblioteca que os programas podem chamar para conseguir as variáveis de ambiente, as quais são muitas vezes usadas para personalizar como um usuário quer desempenhar determinadas tarefas (por exemplo, a impressora padrão a ser utilizada). Na Figura 1.19, nenhum ambiente é passado para o processo filho, então o terceiro parâmetro de *execve* é um zero.

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

Processos em UNIX têm sua memória dividida em três segmentos: o **segmento de texto** (isto é, código de programa), o **segmento de dados** (isto é, as variáveis) e o **segmento de pilha**. O segmento de dados cresce para cima e a pilha cresce para baixo, como mostrado na Figura 1.20. Entre eles há uma lacuna de espaço de endereço não utilizado. A pilha cresce na lacuna automaticamente, na medida do necessário, mas a expansão do segmento de dados é feita explicitamente pelo uso de uma chamada de sistema, brk, que especifica o novo endereço onde o segmento de dados deve terminar. Essa chamada, no entanto, não é definida pelo padrão POSIX, tendo em vista que os programadores são encorajados a usar a rotina de biblioteca *malloc* para alocar dinamicamente memória, e a implementação subjacente de *malloc* não foi vista como um assunto adequado para padronização, pois poucos programadores a usam diretamente e é questionável se alguém mesmo percebe que brk não está no POSIX.

FIGURA 1.20 Os processos têm três segmentos: texto, dados e pilha.



1.6.2 Chamadas de sistema para gerenciamento de arquivos

Muitas chamadas de sistema relacionam-se ao sistema de arquivos. Nesta seção examinaremos as chamadas que operam sobre arquivos individuais; na próxima, examinaremos as que envolvem diretórios ou o sistema de arquivos como um todo.

Para ler ou escrever um arquivo, é preciso primeiro abri-lo. Essa chamada especifica o nome do arquivo a ser aberto, seja como um nome de caminho absoluto ou relativo ao diretório de trabalho, assim como um código de *O_RDONLY*, *O_WRONLY*, ou *O_RDWR*, significando aberto para leitura, escrita ou ambos. Para criar um novo arquivo, o parâmetro *O_CREAT* é usado.

O descritor de arquivos retornado pode então ser usado para leitura ou escrita. Em seguida, o arquivo pode ser fechado por close, que torna o descritor disponível para ser reutilizado em um open subsequente.

As chamadas mais intensamente usadas são, sem dúvida, read e write. Já vimos read. Write tem os mesmos parâmetros.

Embora a maioria dos programas leia e escreva arquivos sequencialmente, alguns programas de aplicativos precisam ser capazes de acessar qualquer parte de um arquivo de modo aleatório. Associado a cada arquivo há um ponteiro que indica a posição atual no arquivo. Quando lendo (escrevendo) sequencialmente, ele em geral aponta para o próximo byte a ser lido (escrito). A chamada lseek muda o valor do ponteiro de posição, de maneira que chamadas subsequentes para ler ou escrever podem começar em qualquer parte no arquivo.

Lseek tem três parâmetros: o primeiro é o descritor de arquivo para o arquivo, o segundo é uma posição do arquivo e o terceiro diz se a posição do arquivo é relativa ao começo, à posição atual ou ao fim do arquivo. O valor retornado por lseek é a posição absoluta no arquivo (em bytes) após mudar o ponteiro.

Para cada arquivo, UNIX registra o tipo do arquivo (regular, especial, diretório, e assim por diante), tamanho, hora da última modificação e outras informações. Os programas podem pedir para ver essas informações através de uma chamada de sistema stat. O primeiro parâmetro 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 para um arquivo aberto.

1.6.3 Chamadas de sistema para gerenciamento de diretórios

Nesta seção examinaremos algumas chamadas de sistema que se relacionam mais aos diretórios ou o sistema de arquivos como um todo, em vez de apenas um arquivo específico como na seção anterior. As primeiras duas chamadas, mkdir e rmdir, criam e removem diretórios vazios, respectivamente. A próxima chamada é link. Sua finalidade é permitir que o mesmo arquivo apareça sob dois ou mais nomes, muitas vezes em diretórios diferentes. Um uso típico é permitir que vários membros da mesma equipe de programação compartilhem um arquivo comum, com cada um deles tendo o arquivo aparecendo no seu próprio diretório, possivelmente sob nomes diferentes. Compartilhar um arquivo não é o mesmo que dar a cada membro da equipe uma cópia particular; ter um arquivo compartilhado significa que as mudanças feitas por qualquer membro da equipe são instantaneamente visíveis para os outros membros, mas há apenas um arquivo. Quando cópias de um arquivo são feitas, mudanças subsequentes feitas para uma cópia não afetam as outras.

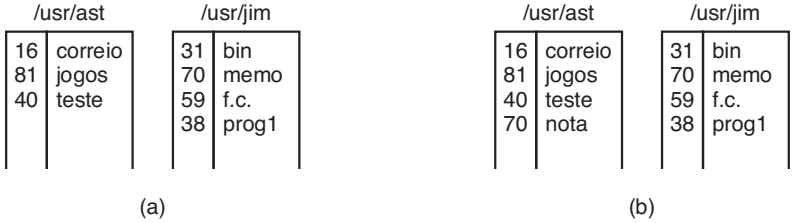
Para vermos como link funciona, considere a situação da Figura 1.21(a). Aqui há dois usuários, *ast* e *jim*, cada um com o seu próprio diretório com alguns arquivos. Se *ast* agora executa um programa contendo a chamada de sistema

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

o arquivo *memo* no diretório de *jim* estará aparecendo agora no diretório de *ast* sob o nome *note*. Daí em diante, */usr/jim/memo* e */usr/ast/note* referem-se ao mesmo arquivo. Como uma nota, se os diretórios serão mantidos em */usr*, */user*, */home,* ou em outro lugar é apenas uma decisão tomada pelo administrador do sistema local.

Compreender como link funciona provavelmente tornará mais claro o que ele faz. Todo arquivo UNIX tem um número único, o seu i-número, que o identifica. Esse i-número é um índice em uma tabela de **i-nós**, um por arquivo, dizendo quem possui o arquivo, onde seus blocos de disco estão e assim por diante. Um diretório é apenas um arquivo contendo um conjunto de pares (i-número, nome em ASCII). Nas primeiras versões do UNIX, cada entrada de diretório tinha 16 bytes — 2 bytes para o i-número e 14 bytes para o nome. Agora, uma estrutura mais complicada é necessária para dar suporte a nomes longos de arquivos, porém conceitualmente, um diretório ainda é um conjunto de pares (i-número, nome em ASCII). Na Figura 1.21, *mail* tem o i-número 16 e assim por diante. O que link faz é nada mais que criar uma entrada de diretório nova com um

FIGURA 1.21 (a) Dois diretórios antes da ligação de */usr/jim/memo* ao diretório *ast*. (b) Os mesmos diretórios depois dessa ligação.



nome (possivelmente novo), usando o i-número de um arquivo existente. Na Figura 1.21(b), duas entradas têm o mesmo i-número (70) e desse modo, referem-se ao mesmo arquivo. Se qualquer uma delas for removida mais tarde, usando a chamada de sistema unlink, a outra permanece. Se ambas são removidas, UNIX vê que não existem entradas para o arquivo (um campo no i-nó registra o número de entradas de diretório apontando para o arquivo), assim o arquivo é removido do disco.

Como mencionamos antes, a chamada de sistema mount permite que dois sistemas de arquivos sejam fundidos em um. Uma situação comum é ter o sistema de arquivos-raiz, contendo as versões (executáveis) binárias dos comandos comuns e outros arquivos intensamente usados, em uma (sub)partição de disco rígido e os arquivos do usuário em outra (sub)partição. Posteriormente o usuário pode ainda inserir um disco USB com arquivos para serem lidos.

Ao executar a chamada de sistema mount, o sistema de arquivos USB pode ser anexado ao sistema de arquivos-raiz, como mostrado na Figura 1.22. Um comando típico em C para realizar essa montagem é

```
mount("/dev/sdb0", "/mnt", 0);
```

onde o primeiro parâmetro é o nome de um arquivo especial de blocos para a unidade de disco 0, o segundo é o lugar na árvore onde ele deve ser montado, e o terceiro diz se o sistema de arquivos deve ser montado como leitura e escrita ou somente leitura.

Após a chamada mount, um arquivo na unidade de disco 0 pode ser acessado usando o seu caminho do diretório-raiz ou do diretório de trabalho, sem levar em consideração em qual unidade de disco ele está. Na realidade, a segunda, terceira e quarta unidades de disco também podem ser montadas em qualquer parte na árvore. A chamada mount torna possível integrar meios removíveis em uma única hierarquia de arquivos integrada, sem precisar preocupar-se em qual dispositivo se encontra um arquivo. Embora esse exemplo envolva CD-ROMs, porções de discos rígidos (muitas vezes chamadas **partições** ou **dispositivos secundários**) também podem ser montadas dessa maneira, assim como discos rígidos externos e *pen drives* USB. Quando um sistema de arquivos não é mais necessário, ele pode ser desmontado com a chamada de sistema umount.

1.6.4 Chamadas de sistema diversas

Existe também uma variedade de outras chamadas de sistema. Examinaremos apenas quatro delas aqui. A chamada chdir muda o diretório de trabalho atual. Após a chamada

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

uma abertura no arquivo *xyz* abrirá */usr/ast/test/xyz*. O conceito de um diretório de trabalho elimina a necessidade de digitar (longos) nomes de caminhos absolutos a toda hora.

Em UNIX todo arquivo tem um modo usado para proteção. O modo inclui os bits de leitura-escrita-execução para o proprietário, para o grupo e para os outros. A chamada de sistema chmod torna possível mudar o modo de um arquivo. Por exemplo, para tornar um arquivo como somente de leitura para todos, exceto o proprietário, poderia ser executado

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

A chamada de sistema kill é a maneira pela qual os usuários e os processos de usuários enviam sinais. Se um processo está preparado para capturar um sinal em particular, então, quando ele chega, uma rotina de tratamento desse sinal é executada. Se o processo não está preparado para lidar com um sinal, então sua chegada mata o processo (daí seu nome).

O POSIX define uma série de rotinas para lidar com o tempo. Por exemplo, time retorna o tempo atual somente em segundos, com 0 correspondendo a 1º de janeiro, 1970, à meia-noite (bem como se o dia estivesse começando, não terminando). Em computadores usando palavras de 32 bits, o valor máximo que time pode retornar é $2^{32} - 1$ s (presumindo que um inteiro sem sinal esteja sendo usado). Esse valor corresponde a um pouco mais de 136 anos. Desse modo, no ano 2106, sistemas UNIX de 32 bits entrarão em pane, de maneira

FIGURA 1.22 (a) O sistema de arquivos antes da montagem. (b) O sistema de arquivos após a montagem.

(a) (b)

semelhante ao famoso problema Y2K que causaria um estrago enorme com os computadores do mundo em 2000, não fosse o esforço enorme realizado pela indústria de TI para resolver o problema. Se hoje você possui um sistema UNIX de 32 bits, aconselhamos que você o troque por um de 64 bits em algum momento antes do ano de 2106.

1.6.5 A API Win32 do Windows

Até aqui nos concentramos fundamentalmente no UNIX. Agora chegou o momento para examinarmos com brevidade o Windows. O Windows e o UNIX diferem de uma maneira fundamental em seus respectivos modelos de programação. Um programa UNIX consiste de um código que faz uma coisa ou outra, fazendo chamadas de sistema para ter determinados serviços realizados. Em comparação, um programa Windows é normalmente direcionado por eventos. O programa principal espera por algum evento acontecer, então chama uma rotina para lidar com ele. Eventos típicos são teclas sendo pressionadas, o mouse sendo movido, um botão do mouse acionado, ou um disco flexível inserido. Tratadores são então chamados para processar o evento, atualizar a tela e o estado do programa interno. Como um todo, isso leva a um estilo de certa maneira diferente de programação do que com o UNIX, mas tendo em vista que o foco deste livro está na função e estrutura do sistema operacional, esses modelos de programação diferentes não nos dizem mais respeito.

É claro, o Windows também tem chamadas de sistema. Com o UNIX, há quase uma relação de um para um 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 aproximadamente uma rotina de biblioteca que é chamada para invocá-la, como indicado na Figura 1.17. Além disso, POSIX tem apenas em torno de 100 chamadas de rotina.

Com o Windows, a situação é radicalmente diferente. Para começo de conversa, as chamadas de biblioteca e as chamadas de sistema reais são altamente desacopladas. A Microsoft definiu um conjunto de rotinas chamadas de **API Win32 (Application Programming Interface** — interface de programação de aplicativos) que se espera que os programadores usem para acessar os serviços do sistema operacional. Essa interface tem contado com o suporte (parcial) de todas as versões do Windows desde o Windows 95. Ao desacoplar a interface API das chamadas de sistema reais, a Microsoft retém a capacidade de mudar as chamadas de sistema reais a qualquer tempo (mesmo de um lançamento para outro) sem invalidar os programas existentes. O que de fato constitui o Win32 também é um tanto ambíguo, pois versões recentes do Windows têm muitas chamadas novas que não estavam disponíveis anteriormente. Nesta seção, Win32 significa a interface que conta com o suporte de todas as versões do Windows. A Win32 proporciona compatibilidade entre as versões do Windows.

O número de chamadas API Win32 é extremamente grande, chegando a milhares. Além disso, enquanto muitas delas invocam chamadas de sistema, um número substancial é executado inteiramente no espaço do usuário. Como consequência, com o Windows é impossível de se ver o que é uma chamada de sistema (isto é, realizada pelo núcleo) e o que é apenas uma chamada de biblioteca do espaço do usuário. Na realidade, o que é uma chamada de sistema em uma versão do Windows pode ser feito no espaço do usuário em uma versão diferente, e vice-versa. Quando discutirmos as chamadas de sistema do Windows neste livro, usaremos as rotinas Win32 (quando for apropriado), já que a Microsoft garante que essas rotinas seguirão estáveis com o tempo. Mas vale a pena lembrar que nem todas elas são verdadeiras chamadas de sistema (isto é, levam o controle para o núcleo).

A API Win32 tem um número enorme de chamadas para gerenciar janelas, figuras geométricas, texto, fontes, barras de rolagem, caixas de diálogo, menus e outros aspectos da interface gráfica GUI. Na medida em que o subsistema gráfico é executado no núcleo (uma verdade em algumas versões do Windows, mas não todas), elas são chamadas de sistema; do contrário, são apenas chamadas de biblioteca. Deveríamos discutir essas chamadas neste livro ou não? Tendo em vista que elas não são realmente relacionadas à função do sistema operacional, decidimos que não, embora elas possam ser executadas pelo núcleo. Leitores interessados na API Win32 devem consultar um dos muitos livros sobre o assunto (por exemplo, HART, 1997; RECTOR e NEWCOMER, 1997; e SIMON, 1997).

Mesmo introduzir todas as chamadas API Win32 aqui está fora de questão, então vamos nos restringir àquelas chamadas que correspondem mais ou menos à funcionalidade das chamadas UNIX listadas na Figura 1.18. Estas estão listadas na Figura 1.23.

Vamos agora repassar brevemente a lista da Figura 1.23. CreateProcess cria um novo processo, realizando o trabalho combinado de fork e execve em UNIX. Possui muitos parâmetros especificando as propriedades do processo recentemente criado. O Windows não tem uma hierarquia de processo como o UNIX, então não há um conceito de um processo pai e um processo filho. Após

um processo ser criado, o criador e criatura são iguais. WaitForSingleObject é usado para esperar por um evento. É possível se esperar por muitos eventos com essa chamada. Se o parâmetro especifica um processo, então quem chamou espera pelo processo especificado terminar, o que é feito usando ExitProcess.

As próximas seis chamadas operam em arquivos e são funcionalmente similares a suas correspondentes do UNIX, embora difiram nos parâmetros e detalhes. Ainda assim, os arquivos podem ser abertos, fechados, lidos e escritos de uma maneira bastante semelhante ao UNIX. As chamadas SetFilePointer e GetFileAttributesEx estabelecem a posição do arquivo e obtêm alguns de seus atributos.

O Windows tem diretórios e eles são criados com chamadas API CreateDirectory e RemoveDirectory, respectivamente. Há também uma noção de diretório atual, determinada por SetCurrentDirectory. A hora atual do dia é conseguida usando GetLocalTime.

A interface Win32 não tem links para os arquivos, tampouco sistemas de arquivos montados, segurança ou sinais, de maneira que não existem as chamadas correspondentes ao UNIX. É claro, Win32 tem um número enorme de outras chamadas que o UNIX não tem, em especial para gerenciar a interface gráfica GUI. O Windows Vista tem um sistema de segurança elaborado e também dá suporte a links de arquivos. Os Windows 7 e 8 acrescentam ainda mais aspectos e chamadas de sistema.

Uma última nota a respeito do Win32 talvez valha a pena ser feita. O Win32 não é uma interface realmente uniforme ou consistente. A principal culpada aqui foi a necessidade de ser retroativamente compatível com a interface anterior de 16 bits usada no Windows 3.x.

1.7 Estrutura de sistemas operacionais

Agora que vimos como os sistemas operacionais parecem por fora (isto é, a interface do programador), é hora de darmos uma olhada por dentro. Nas seções a seguir, examinaremos seis estruturas diferentes que foram tentadas, a fim de termos alguma ideia do espectro de possibilidades. Isso não quer dizer que esgotaremos o assunto, mas elas dão uma ideia de alguns projetos que foram tentados na prática. Os seis projetos que discutiremos aqui são sistemas monolíticos, sistemas de camadas, micronúcleos, sistemas cliente-servidor, máquinas virtuais e exonúcleos.

FIGURA 1.23 As chamadas da API Win32 que correspondem aproximadamente às chamadas UNIX da Figura 1.18. Vale a pena enfatizar que o Windows tem um número muito grande de outras chamadas de sistema, a maioria das quais não corresponde a nada no UNIX.

UNIX	Win32	Descrição
fork	CreateProcess	Cria um novo processo
waitpid	WaitForSingleObject	Pode esperar que um processo termine
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 ligações
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

1.7.1 Sistemas monolíticos

De longe a organização mais comum, na abordagem monolítica todo o sistema operacional é executado como um único programa em modo núcleo. O sistema operacional é escrito como uma coleção de rotinas, ligadas a um único grande programa binário executável. Quando a técnica é usada, cada procedimento no sistema é livre para chamar qualquer outro, se este oferecer alguma computação útil de que o primeiro precisa. Ser capaz de chamar qualquer procedimento que você quer é muito eficiente, mas ter milhares de procedimentos que podem chamar um ao outro sem restrições pode também levar a um sistema difícil de lidar e compreender. Também, uma quebra em qualquer uma dessas rotinas derrubará todo o sistema operacional.

Para construir o programa objeto real do sistema operacional quando essa abordagem é usada, é preciso primeiro compilar todas as rotinas individuais (ou os arquivos contendo as rotinas) e então juntá-las em um único arquivo executável usando o ligador (*linker*) do sistema. Em termos de ocultação de informações, essencialmente não há nenhuma — toda rotina é visível para toda outra rotina (em oposição a uma estrutura contendo módulos ou pacotes, na qual grande parte da informação é escondida dentro de módulos, e apenas os pontos de entrada oficialmente designados podem ser chamados de fora do módulo).

Mesmo em sistemas monolíticos, no entanto, é possível se ter alguma estrutura. Os serviços (chamadas de sistema) providos pelo sistema operacional são requisitados colocando-se os parâmetros em um local bem definido (por exemplo, em uma pilha) 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 no passo 6 na Figura 1.17. O sistema operacional então busca 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 de serviço requisitada.
2. Um conjunto de rotinas de serviço que executam as chamadas de sistema.
3. Um conjunto de rotinas utilitárias que ajudam as rotinas de serviço.

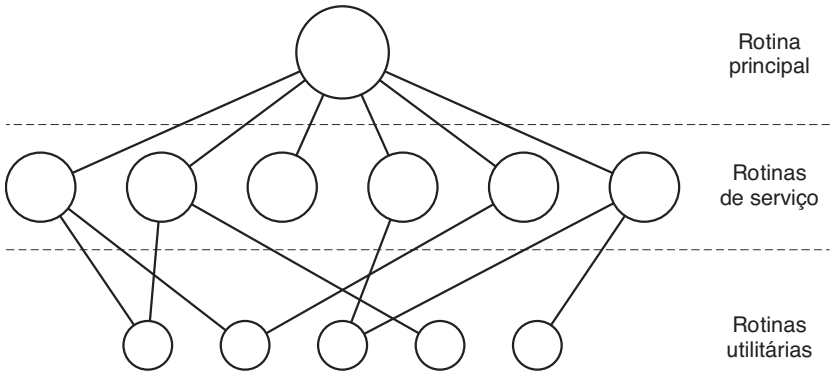
Nesse modelo, para cada chamada de sistema há uma rotina de serviço que se encarrega dela e a executa. As rotinas utilitárias fazem coisas que são necessárias para várias rotinas de serviços, como buscar dados de programas dos usuários. Essa divisão em três camadas é mostrada na Figura 1.24.

Além do sistema operacional principal que é carregado quando o computador é inicializado, muitos sistemas operacionais dão suporte a extensões carregáveis, como drivers de dispositivos de E/S e sistemas de arquivos. Esses componentes são carregados conforme a demanda. No UNIX eles são chamados de **bibliotecas compartilhadas**. No Windows são chamados de **DLLs (Dynamic Link Libraries** — bibliotecas de ligação dinâmica). Eles têm a extensão de arquivo *.dll* e o diretório *C:\Windows\system32* nos sistemas Windows tem mais de 1.000 deles.

1.7.2 Sistemas de camadas

Uma generalização da abordagem da Figura 1.24 é organizar o sistema operacional como uma hierarquia de camadas, cada uma construída sobre a camada abaixo dela.

FIGURA 1.24 Um modelo de estruturação simples para um sistema monolítico.



O primeiro sistema construído dessa maneira foi o sistema THE desenvolvido na Technische Hogeschool Eindhoven na Holanda por E. W. Dijkstra (1968) e seus estudantes. O sistema THE era um sistema em lote simples para um computador holandês, o Electrologica X8, que tinha 32 K de palavras de 27 bits (bits eram caros na época).

O sistema tinha seis camadas, com mostrado na Figura 1.25. A camada 0 lidava com a alocação do processador, realizando o chaveamento de processos quando ocorriam interrupções ou quando os temporizadores expiravam. Acima da camada 0, o sistema consistia em processos sequenciais e cada um deles podia ser programado sem precisar preocupar-se com o fato de que múltiplos processos estavam sendo executados em um único processador. Em outras palavras, a camada 0 fornecia a multiprogramação básica da CPU.

A camada 1 realizava o gerenciamento de memória. Ela alocava espaço para processos na memória principal e em um tambor magnético de 512 K palavras usado para armazenar partes de processos (páginas) para as quais não havia espaço na memória principal. Acima da camada 1, os processos não precisavam se preocupar se eles estavam na memória ou no tambor magnético; o software da camada 1 certificava-se de que as páginas fossem trazidas à memória no momento em que eram necessárias e removidas quando não eram mais.

A camada 2 encarregava-se da comunicação entre cada processo e o console de operação (isto é, o usuário). Acima dessa camada cada processo efetivamente tinha o 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 ou vinham desses dispositivos. Acima da camada 3, cada processo podia lidar com dispositivos de E/S abstratos mais acessíveis, em vez de dispositivos reais com muitas peculiaridades. A camada 4 era onde os programas dos usuários eram encontrados. Eles não precisavam se preocupar com o gerenciamento de processo, memória, console ou E/S. O processo operador do sistema estava localizado na camada 5.

FIGURA 1.25 Estrutura do sistema operacional THE.

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

Outra generalização do conceito de camadas estava presente no sistema MULTICS. Em vez de camadas, MULTICS foi descrito como tendo uma série de anéis concêntricos, com os anéis internos sendo mais privilegiados do que os externos (o que é efetivamente a mesma coisa). Quando um procedimento em um anel exterior queria chamar um procedimento em um anel interior, ele tinha de fazer o equivalente de uma chamada de sistema, isto é, uma instrução de desvio, TRAP, cujos parâmetros eram cuidadosamente conferidos por sua validade antes de a chamada ter permissão para prosseguir. Embora todo o sistema operacional fosse parte do espaço de endereço de cada processo de usuário em MULTICS, o hardware tornou possível que se designassem rotinas individuais (segmentos de memória, na realidade) como protegidos contra leitura, escrita ou execução.

Enquanto o esquema de camadas THE era na realidade somente um suporte para o projeto, pois em última análise todas as partes do sistema estavam unidas em um único programa executável, em MULTICS, o mecanismo de anéis estava bastante presente no momento de execução e imposto pelo hardware. A vantagem do mecanismo de anéis é que ele pode ser facilmente estendido para estruturar subsistemas de usuário. Por exemplo, um professor poderia escrever um programa para testar e atribuir notas a programas de estudantes executando-o no anel n, com os programas dos estudantes seriam executados no anel $n + 1$, de maneira que eles não pudessem mudar suas notas.

1.7.3 Micronúcleos

Com a abordagem de camadas, os projetistas têm uma escolha de onde traçar o limite núcleo-usuário. Tradicionalmente, todas as camadas entram no núcleo, mas isso não é necessário. Na realidade, um forte argumento pode ser defendido para a colocação do mínimo possível no modo núcleo, pois erros no código do núcleo podem derrubar o sistema instantaneamente. Em comparação, processos de usuário podem ser configurados para ter menos poder, de maneira que um erro possa não ser fatal.

Vários pesquisadores estudaram repetidamente o número de erros por 1.000 linhas de código (por exemplo, BASILLI e PERRICONE, 1984; OSTRAND e WEYUKER, 2002). A densidade de erros depende do tamanho do módulo, idade do módulo etc., mas um número aproximado para sistemas industriais sérios fica entre dois e dez erros por mil linhas de código. Isso significa que em um sistema operacional monolítico de cinco milhões de linhas de código é provável que

contenha entre 10.000 e 50.000 erros no núcleo. Nem todos são fatais, é claro, tendo em vista que alguns erros podem ser coisas como a emissão de uma mensagem de erro incorreta em uma situação que raramente ocorre. Mesmo assim, sistemas operacionais são a tal ponto sujeitos a erros, que os fabricantes de computadores colocam botões de reinicialização neles (muitas vezes no painel da frente), algo que os fabricantes de TVs, aparelhos de som e carros não o fazem, apesar da grande quantidade de software nesses dispositivos.

A ideia básica por trás do projeto de micronúcleo é atingir uma alta confiabilidade através da divisão do sistema operacional em módulos pequenos e bem definidos, apenas um dos quais — o micronúcleo — é executado em modo núcleo e o resto é executado como processos de usuário comuns relativamente sem poder. Em particular, ao se executar cada driver de dispositivo e sistema de arquivos como um processo de usuário em separado, um erro em um deles pode derrubar esse componente, mas não consegue derrubar o sistema inteiro. Desse modo, um erro no driver de áudio fará que o som fique truncado ou pare, mas não derrubará o computador. Em comparação, em um sistema monolítico, com todos os drivers no núcleo, um driver de áudio com problemas pode facilmente referenciar um endereço de memória inválido e provocar uma parada dolorosa no sistema instantaneamente.

Muitos micronúcleos foram implementados e empregados por décadas (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; e ZUBERI et al., 1999). Com a exceção do OS X, que é baseado no micronúcleo Mach (ACETTA et al., 1986), sistemas operacionais de computadores de mesa comuns não usam micronúcleos. No entanto, eles são dominantes em aplicações de tempo real, industriais, de aviônica e militares, que são cruciais para missões e têm exigências de confiabilidade muito altas. Alguns dos micronúcleos mais conhecidos incluem Integrity, K42, L4, PikeOS, QNX, Symbian e MINIX 3. Daremos agora uma breve visão geral do MINIX 3, que levou a ideia da modularidade até o limite, decompondo a maior parte do sistema operacional em uma série de processos de modo usuário independentes. MINIX 3 é um sistema em conformidade com o POSIX, de código aberto e gratuitamente disponível em <*www.minix3.org*> (GIUFFRIDA et al., 2012; GIUFFRIDA et al., 2013; HERDER et al., 2006; HERDER et al., 2009; e HRUBY et al., 2013).

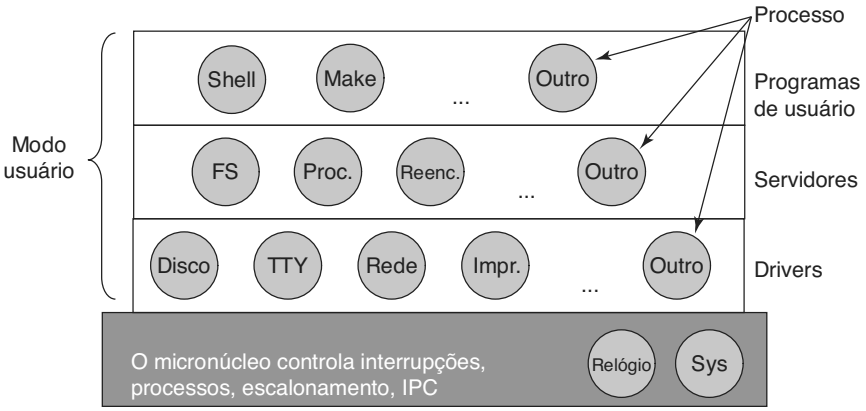
O micronúcleo MINIX 3 tem apenas em torno de 12.000 linhas de C e cerca de 1.400 linhas de assembler para funções de nível muito baixo como capturar interrupções e chavear processos. O código C gerencia e escalona processos, lida com a comunicação entre eles (passando mensagens entre processos) e oferece um conjunto de mais ou menos 40 chamadas de núcleo que permitem que o resto do sistema operacional faça o seu trabalho. Essas chamadas realizam funções como associar os tratadores às interrupções, transferir dados entre espaços de endereços e instalar mapas de memória para processos novos. A estrutura de processo de MINIX 3 é mostrada na Figura 1.26, com os tratadores de chamada de núcleo rotulados *Sys*. O driver de dispositivo para o relógio também está no núcleo, pois o escalonador interage de perto com ele. Os outros drivers de dispositivos operam como processos de usuário em separado.

Fora do núcleo, o sistema é estruturado como três camadas de processos, todos sendo executados em modo usuário. A camada mais baixa contém os drivers de dispositivos. Como são executados em modo usuário, eles não têm acesso físico ao espaço da 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 quais valores escrever para quais portas de E/S e faz uma chamada de núcleo dizendo para o núcleo fazer a escrita. Essa abordagem significa que o núcleo pode conferir para ver que o driver está escrevendo (ou lendo) a partir da E/S que ele está autorizado a usar. Em consequência (e diferentemente de um projeto monolítico), um driver de áudio com erro não consegue escrever por acidente no disco.

Acima dos drivers há outra camada no modo usuário contendo os servidores, que fazem a maior parte do trabalho do sistema operacional. Um ou mais servidores de arquivos gerenciam o(s) sistema(s) de arquivos, o gerente de processos cria, destrói e gerencia processos, e assim por diante. Programas de usuários obtêm serviços de sistemas operacionais enviando mensagens curtas para os servidores solicitando as chamadas de sistema POSIX. Por exemplo, um processo precisando fazer uma `read`, envia uma mensagem para um dos servidores de arquivos dizendo a ele o que ler.

Um servidor interessante é o **servidor de reencarnação**, cujo trabalho é conferir se os outros servidores e drivers estão funcionando corretamente. No caso da detecção de um servidor ou driver defeituoso, ele é automaticamente substituído sem qualquer intervenção do usuário. Dessa maneira, o sistema está regenerando a si mesmo e pode atingir uma alta confiabilidade.

FIGURA 1.26 Estrutura simplificada do sistema MINIX.



O sistema tem muitas restrições limitando o poder de cada processo. Como mencionado, os drivers podem tocar apenas portas de E/S autorizadas, mas o acesso às chamadas de núcleo também é controlado processo a processo, assim como a capacidade de enviar mensagens para outros processos. Processos também podem conceder uma permissão limitada para outros processos para que o núcleo acesse seus espaços de endereçamento. Como exemplo, um sistema de arquivos pode conceder uma permissão para que a unidade de disco deixe o núcleo colocar uma leitura recente de um bloco do disco em um endereço específico dentro do espaço de endereço do sistema de arquivos. A soma de todas essas restrições é que cada driver e servidor têm exatamente o poder de fazer o seu trabalho e nada mais, dessa maneira limitando muito o dano que um componente com erro pode provocar.

Uma ideia de certa maneira relacionada a ter um núcleo mínimo é colocar o **mecanismo** para fazer algo no núcleo, mas não a **política**. Para esclarecer esse ponto, considere o escalonamento de processos. Um algoritmo de escalonamento relativamente simples é designar uma prioridade numérica para todo processo e então fazer que o núcleo execute o processo mais prioritário e que seja executável. O mecanismo — no núcleo — é procurar pelo processo mais prioritário e executá-lo. A política — designar prioridades para processos — pode ser implementada por processos de modo usuário. Dessa maneira, política e mecanismo podem ser desacoplados e o núcleo tornado menor.

1.7.4 O modelo cliente-servidor

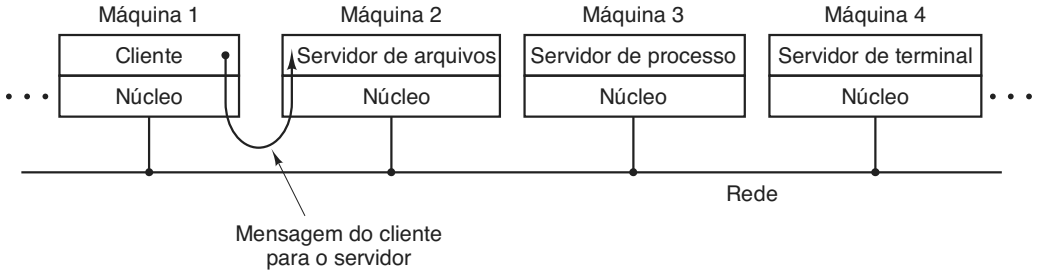
Uma ligeira variação da ideia do micronúcleo é distinguir duas classes de processos, os **servidores**, que prestam algum serviço, e os **clientes**, que usam esses serviços. Esse modelo é conhecido como o modelo **cliente-servidor**. Muitas vezes, a camada mais baixa é a do micronúcleo, mas isso não é necessário. A essência encontra-se na presença de processos clientes e processos servidores.

A comunicação entre clientes e servidores é realizada muitas vezes pela troca de mensagens. Para obter um serviço, um processo cliente constrói uma mensagem dizendo o que ele quer e a envia ao serviço apropriado. O serviço então realiza o trabalho e envia de volta a resposta. Se acontecer de o cliente e o servidor serem executados na mesma máquina, determinadas otimizações são possíveis, mas conceitualmente, ainda estamos falando da troca de mensagens aqui.

Uma generalização óbvia dessa ideia é ter os clientes e servidores sendo executados em computadores diferentes, conectados por uma rede local ou de grande área, como descrito na Figura 1.27. Tendo em vista que os clientes comunicam-se com os servidores enviando mensagens, os clientes não precisam saber se as mensagens são entregues localmente em suas próprias máquinas, ou se são enviadas através de uma rede para servidores em uma máquina remota. No que diz respeito ao cliente, a mesma coisa acontece em ambos os casos: pedidos são enviados e as respostas retornadas. Desse modo, o modelo cliente-servidor é uma abstração que pode ser usada para uma única máquina ou para uma rede de máquinas.

Cada vez mais, muitos sistemas envolvem usuários em seus PCs em casa como clientes e grandes máquinas em outra parte operando como servidores. Na realidade, grande parte da web opera dessa maneira. Um PC pede uma página na web para um servidor e ele a entrega. Esse é o uso típico do modelo cliente-servidor em uma rede.

FIGURA 1.27 O modelo cliente-servidor em uma rede.



1.7.5 Máquinas virtuais

Os lançamentos iniciais do OS/360 foram estritamente sistemas em lote. Não obstante isso, muitos usuários do 360 queriam poder trabalhar interativamente em um terminal, de maneira que vários grupos, tanto dentro quanto fora da IBM, decidiram escrever sistemas de compartilhamento de tempo para ele. O sistema de compartilhamento de tempo oficial da IBM, TSS/360, foi lançado tarde, e quando enfim chegou, era tão grande e lento que poucos converteram-se a ele. Ele foi finalmente abandonado após o desenvolvimento ter consumido algo em torno de US$ 50 milhões (GRAHAM, 1970). Mas um grupo no Centro Científico da IBM em Cambridge, Massachusetts, produziu um sistema radicalmente diferente que a IBM por fim aceitou como produto. Um descendente linear, chamado **z/VM**, é hoje amplamente usado nos computadores de grande porte da IBM, os zSeries, que são intensamente usados em grandes centros de processamento de dados corporativos, por exemplo, como servidores de comércio eletrônico que lidam com centenas ou milhares de transações por segundo e usam bancos de dados cujos tamanhos chegam a milhões de gigabytes.

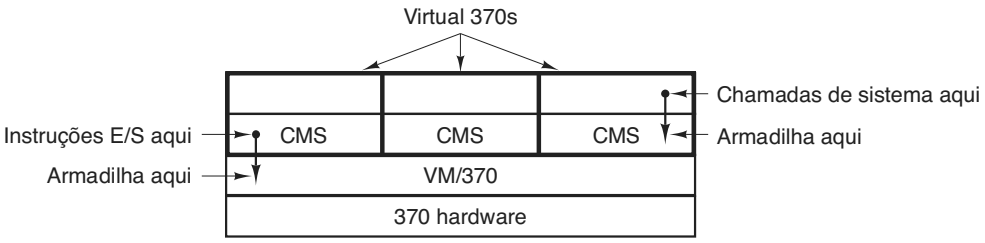
VM/370

Esse sistema, na origem chamado CP/CMS e mais tarde renomeado VM/370 (SEAWRIGHT e MacKINNON, 1979), foi baseado em uma observação astuta: um sistema de compartilhamento de tempo fornece (1) multiprogramação e (2) uma máquina estendida com uma interface mais conveniente do que apenas o hardware. A essência do VM/370 é separar completamente essas duas funções.

O cerne do sistema, conhecido como o **monitor de máquina virtual**, opera direto no hardware e realiza a multiprogramação, fornecendo não uma, mas várias máquinas virtuais para a camada seguinte, como mostrado na Figura 1.28. No entanto, diferentemente de todos os outros sistemas operacionais, essas máquinas virtuais não são máquinas estendidas, com arquivos e outros aspectos interessantes. Em vez disso, elas são cópias *exatas* do hardware exposto, incluindo modos núcleo/usuário, E/S, interrupções e tudo mais que a máquina tem.

Como cada máquina virtual é idêntica ao hardware original, cada uma delas pode executar qualquer sistema operacional capaz de ser executado diretamente sobre o hardware. Máquinas virtuais diferentes podem — e frequentemente o fazem — executar diferentes sistemas operacionais. No sistema VM/370 original da IBM, em algumas é executado o sistema operacional OS/360 ou um dos outros sistemas operacionais de processamento de transações ou em lote grande, enquanto em outras é executado um sistema operacional monousuário interativo chamado **CMS (Conversational Monitor System** — sistema monitor conversacional), para usuários interativos em tempo compartilhado. Esse sistema era popular entre os programadores.

FIGURA 1.28 A estrutura do VM/370 com CMS.



Quando um programa CMS executava uma chamada de sistema, ela era desviada para o sistema operacional na sua própria máquina virtual, não para o VM/370, como se estivesse executando em uma máquina real em vez de uma virtual. O CMS então emitia as instruções de E/S normais de hardware para leitura do seu disco virtual ou o que quer que fosse necessário para executar a chamada. Essas instruções de E/S eram desviadas pelo VM/370, que então as executava como parte da sua simulação do hardware real. Ao separar completamente as funções da multiprogramação e da provisão de uma máquina estendida, cada uma das partes podia ser muito mais simples, mais flexível e muito mais fácil de manter.

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

Máquinas virtuais redescobertas

Embora a IBM tenha um produto de máquina virtual disponível há quatro décadas, e algumas outras empresas, incluindo a Oracle e Hewlett-Packard, tenham recentemente acrescentado suporte de máquina virtual para seus servidores empreendedores de alto desempenho, a ideia da virtualização foi em grande parte ignorada no mundo dos PCs até há pouco tempo. Mas nos últimos anos, uma combinação de novas necessidades, novo software e novas tecnologias combinaram-se para torná-la um tópico de alto interesse.

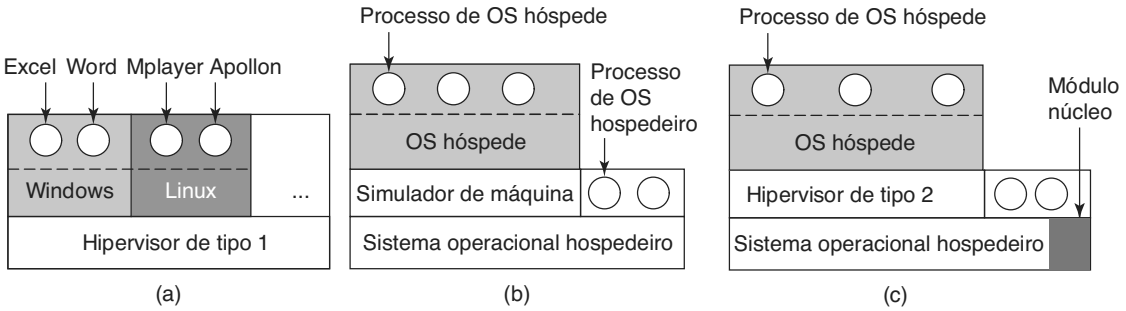
Primeiro as necessidades. Muitas empresas tradicionais executavam seus próprios servidores de correio, de web, de FTP e outros servidores em computadores separados, às vezes com sistemas operacionais diferentes. Elas veem a virtualização como uma maneira de executar todos eles na mesma máquina sem correr o risco de um travamento em um servidor derrubar a todos.

A virtualização também é popular no mundo da hospedagem de páginas da web. Sem a virtualização, os clientes de hospedagem na web são obrigados a escolher entre a **hospedagem compartilhada** (que dá a eles uma conta de acesso a um servidor da web, mas nenhum controle sobre o software do servidor) e a hospedagem dedicada (que dá a eles a própria máquina, que é muito flexível, mas cara para sites de pequeno a médio porte). Quando uma empresa de hospedagem na web oferece máquinas virtuais para alugar, uma única máquina física pode executar muitas máquinas virtuais, e cada uma delas parece ser uma máquina completa. Clientes que alugam uma máquina virtual podem executar qualquer sistema operacional e software que eles quiserem, mas a uma fração do custo de um servidor dedicado (pois a mesma máquina física dá suporte a muitas máquinas virtuais ao mesmo tempo).

Outro uso da virtualização é por usuários finais que querem poder executar dois ou mais sistemas operacionais ao mesmo tempo, digamos Windows e Linux, pois alguns dos seus pacotes de aplicativos favoritos são executados em um sistema e outros no outro sistema. Essa situação é ilustrada na Figura 1.29(a), onde o termo "monitor de máquina virtual" foi renomeado como **hipervisor tipo 1**, que é bastante usado hoje, pois "monitor de máquina virtual" exige mais toques no teclado do que as pessoas estão preparadas para suportar agora. Observe que muitos autores usam os dois termos naturalmente.

Embora hoje ninguém discuta a atratividade das máquinas virtuais, o problema então era de implementação. A fim de executar um software de máquina virtual em um computador, a sua CPU tem de ser virtualizável (POPEK e GOLDBERG, 1974). Resumindo, eis o problema. Quando um sistema operacional sendo executado em uma máquina virtual (em modo usuário) executa uma instrução privilegiada, como modificar a PSW ou

FIGURA 1.29 (a) Um hipervisor de tipo 1. (b) Um hipervisor de tipo 2 puro. (c) Um hipervisor de tipo 2 na prática.



realizar uma E/S, é essencial que o hardware crie uma armadilha que direcione para o monitor da máquina virtual, de maneira que a instrução possa ser emulada em software. Em algumas CPUs — notadamente a Pentium, suas predecessoras e seus clones —, tentativas de executar instruções privilegiadas em modo usuário são simplesmente ignoradas. Essa propriedade impossibilitou ter máquinas virtuais nesse hardware, o que explica a falta de interesse no mundo x86. É claro, havia interpretadores para o Pentium, como *Bochs*, que eram executados nele, porém com uma perda de desempenho de uma ou duas ordens de magnitude, eles não eram úteis para realizar trabalhos sérios.

Essa situação mudou em consequência de uma série de projetos de pesquisa acadêmica na década de 1990 e nos primeiros anos deste milênio, notavelmente Disco em Stanford (BUGNION et al., 1997) e Xen na Universidade de Cambridge (BARHAM et al., 2003). Essas pesquisas levaram a vários produtos comerciais (por exemplo, VMware Workstation e Xen) e um renascimento do interesse em máquinas virtuais. Além do VMware e do Xen, hipervisores populares hoje em dia incluem KVM (para o núcleo Linux), VirtualBox (da Oracle) e Hyper-V (da Microsoft).

Alguns desses primeiros projetos de pesquisa melhoraram o desempenho de interpretadores como o *Bochs* ao traduzir blocos de código rapidamente, armazenando-os em uma cache interna e então reutilizando-os se eles fossem executados de novo. Isso melhorou bastante o desempenho, e levou ao que chamaremos de **simuladores de máquinas**, como mostrado na Figura 1.29(b). No entanto, embora essa técnica, conhecida como **tradução binária**, tenha melhorado as coisas, os sistemas resultantes, embora bons o suficiente para terem estudos publicados em conferências acadêmicas, ainda não eram rápidos o suficiente para serem usados em ambientes comerciais onde o desempenho é muito importante.

O passo seguinte para a melhoria do desempenho foi acrescentar um módulo núcleo para fazer parte do trabalho pesado, como mostrado na Figura 1.29(c). Na prática agora, todos os hipervisores disponíveis comercialmente, como o VMware Workstation, usam essa estratégia híbrida (e têm muitas outras melhorias também). Eles são chamados de **hipervisores tipo 2** por todos, então acompanharemos (de certa maneira a contragosto) e usaremos esse nome no resto deste livro, embora preferíssemos chamá-los de hipervisores tipo 1.7 para refletir o fato de que eles não são inteiramente programas de modo usuário. No Capítulo 7, descreveremos em detalhes o funcionamento do VMware Workstation e o que as suas várias partes fazem.

Na prática, a distinção real entre um hipervisor tipo 1 e um hipervisor tipo 2 é que o tipo 2 usa um **sistema operacional hospedeiro** e o seu sistema de arquivos para criar processos, armazenar arquivos e assim por diante. Um hipervisor tipo 1 não tem suporte subjacente e precisa realizar todas essas funções sozinho.

Após um hipervisor tipo 2 ser inicializado, ele lê o CD-ROM de instalação (ou arquivo de imagem CD-ROM) para o **sistema operacional hóspede** escolhido e o instala em um disco virtual, que é apenas um grande arquivo no sistema de arquivos do sistema operacional hospedeiro. Hipervisores tipo 1 não podem realizar isso porque não há um sistema operacional hospedeiro para armazenar os arquivos. Eles têm de gerenciar sua própria armazenagem em uma partição de disco bruta.

Quando o sistema operacional hóspede é inicializado, ele faz o mesmo que no hardware de verdade, tipicamente iniciando alguns processos de segundo plano e então uma interface gráfica GUI. Para o usuário, o sistema operacional hóspede comporta-se como quando está sendo executado diretamente no hardware, embora não seja o caso aqui.

Uma abordagem diferente para o gerenciamento de instruções de controle é modificar o sistema operacional para removê-las. Essa abordagem não é a verdadeira virtualização, mas a **paravirtualização**. Discutiremos a virtualização em mais detalhes no Capítulo 7.

A máquina virtual Java

Outra área onde as máquinas virtuais são usadas, mas de uma maneira de certo modo diferente, é na execução de programas Java. Quando a Sun Microsystems inventou a linguagem de programação Java, ela também inventou uma máquina virtual (isto é, uma arquitetura de computadores) chamada de **JVM (Java Virtual Machine** — máquina virtual Java). O compilador Java produz código para a JVM, que então é executado por um programa interpretador da JVM. A vantagem dessa abordagem é que o código JVM pode ser enviado pela internet para qualquer computador que tenha um interpretador JVM e ser executado lá. Se o compilador tivesse produzido programas binários x86 ou SPARC, por exemplo, eles não poderiam ser enviados e executados em qualquer parte tão facilmente. (É claro, a Sun poderia ter produzido um compilador que produzisse binários SPARC e então distribuído um interpretador SPARC, mas a JVM é uma arquitetura muito mais simples de interpretar.) Outra vantagem de se usar a JVM é que se o interpretador for implementado da maneira adequada, o que não é algo completamente trivial, os programas

JVM que chegam podem ser verificados, por segurança, e então executados em um ambiente protegido para que não possam roubar dados ou causar qualquer dano.

1.7.6 Exonúcleos

Em vez de clonar a máquina real, como é feito com as máquinas virtuais, outra estratégia é dividi-la, ou em outras palavras, dar a cada usuário um subconjunto dos recursos. Desse modo, uma máquina virtual pode obter os blocos de disco de 0 a 1.023, a próxima pode ficar com os blocos 1.024 a 2.047 e assim por diante.

Na camada de baixo, executando em modo núcleo, há um programa chamado **exonúcleo** (ENGLER et al., 1995). Sua tarefa é alocar recursos às máquinas virtuais e então conferir tentativas de usá-las para assegurar-se de que nenhuma máquina esteja tentando usar os recursos de outra pessoa. Cada máquina virtual no nível do usuário pode executar seu próprio sistema operacional, como na VM/370 e no modo virtual 8086 do Pentium, exceto que cada uma está restrita a usar apenas os recursos que ela pediu e foram alocados.

A vantagem do esquema do exonúcleo é que ele poupa uma camada de mapeamento. Nos outros projetos, cada máquina virtual pensa que ela tem seu próprio disco, com blocos sendo executados de 0 a algum máximo, de maneira que o monitor da máquina virtual tem de manter tabelas para remapear os endereços de discos (e todos os outros recursos). Com o exonúcleo, esse remapeamento não é necessário. O exonúcleo precisa apenas manter o registro de para qual máquina virtual foi atribuído qual recurso. Esse método ainda tem a vantagem de separar a multiprogramação (no exonúcleo) do código do sistema operacional do usuário (em espaço do usuário), mas com menos sobrecarga, tendo em vista que tudo o que o exonúcleo precisa fazer é manter as máquinas virtuais distantes umas das outras.

1.8 O mundo de acordo com a linguagem C

Sistemas operacionais normalmente são grandes programas C (ou às vezes C++) consistindo em muitas partes escritas por muitos programadores. O ambiente usado para desenvolver os sistemas operacionais é muito diferente do que os indivíduos (tais como estudantes) estão acostumados quando estão escrevendo programas pequenos Java. Esta seção é uma tentativa de fazer uma introdução muito breve para o mundo da escrita de um sistema operacional para programadores Java ou Python modestos.

1.8.1 A linguagem C

Este não é um guia para a linguagem C, mas um breve resumo de algumas das diferenças fundamentais entre C e linguagens como **Python** e especialmente Java. Java é baseado em C, portanto há muitas similaridades entre as duas. Python é de certa maneira diferente, mas ainda assim ligeiramente similar. Por conveniência, focaremos em Java. Java, Python e C são todas linguagens imperativas com tipos de dados, variáveis e comandos de controle, por exemplo. Os tipos de dados primitivos em C são inteiros (incluindo curtos e longos), caracteres e números de ponto flutuante. Os tipos de dados compostos em C são similares àqueles em Java, incluindo os comandos if, switch, for e while. Funções e parâmetros são mais ou menos os mesmos em ambas as linguagens.

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

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

que declara *c1* e *c2* como variáveis de caracteres e *p* como sendo uma variável que aponta para (isto é, contém o endereço de) um caractere. A primeira atribuição armazena o código ASCII para o caractere "c" na variável *c1*. A segunda designa o endereço de *c1* para a variável do ponteiro *p*. A terceira designa o conteúdo da variável apontada por *p* para a variável *c2*, de maneira que após esses comandos terem sido executados, *c2* também contém o código ASCII para "c". Na teoria, ponteiros possuem tipos, assim não se supõe que você vá designar o endereço de um número em ponto flutuante a um ponteiro de caractere, porém na prática compiladores aceitam tais atribuições, embora algumas vezes com um aviso. Ponteiros são uma construção muito poderosa, mas também uma grande fonte de erros quando usados de modo descuidado

Algumas coisas que C não tem incluem cadeias de caracteres incorporadas, *threads*, pacotes, classes, objetos, segurança de tipos e coletor de lixo. Todo armazenamento em C é estático ou explicitamente alocado e liberado pelo programador, normalmente com as funções de biblioteca *malloc* e *free*. É a segunda propriedade — controle do programador total sobre a memória — junto com ponteiros explícitos que torna C atraente para a escrita de sistemas operacionais. Sistemas operacionais são, até certo ponto, basicamente sistemas em

tempo real, até mesmo sistemas com propósito geral. Quando uma interrupção ocorre, o sistema operacional pode ter apenas alguns microssegundos para realizar alguma ação ou perder informações críticas. A entrada do coletor de lixo em um momento arbitrário é algo intolerável.

1.8.2 Arquivos de cabeçalho

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

```
#define BUFFER_SIZE 4096
```

que permitem ao programador nomear constantes, assim, quando *BUFFER_SIZE* é usado no código, ele é substituído durante a compilação pelo número 4096. Uma boa prática de programação C é nomear todas as constantes, com exceção de 0, 1 e −1, e às vezes até elas. Macros podem ter parâmetros como em

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

que permite ao programador escrever

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

e obter

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

para armazenar o maior entre *j* e *k+1* em *i*. Cabeçalhos também podem conter uma compilação condicional, por exemplo

```
#ifdef X86
intel_int_ack();
#endif
```

que compila uma chamada para a função *intel_int_ack* se o macro *X86* for definido e nada mais de outra forma. A compilação condicional é intensamente usada para isolar códigos dependentes de arquitetura, assim um determinado código é inserido apenas quando o sistema for compilado no X86, outro código é inserido somente quando o sistema é compilado em um SPARC e assim por diante. Um arquivo *.c* pode incluir conjuntamente zero ou mais arquivos de cabeçalho usando a diretiva *#include*. Há também muitos arquivos de cabeçalho que são comuns a quase todos os *.c* e são armazenados em um diretório central.

1.8.3 Grandes projetos de programação

Para construir o 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 mais tarde diretamente executados pela CPU. Não há nada semelhante ao bytecode Java ou o bytecode Python no mundo C.

O primeiro passo do compilador C é chamado de **pré-processador C**. Quando lê cada arquivo *.c*, toda vez que ele atinge uma diretiva *#include*, ele vai e pega o arquivo cabeçalho nomeado nele e o processa, expandindo macros, lidando com a compilação condicional (e determinadas outras coisas) e passando os resultados ao próximo passo do compilador como se eles estivessem fisicamente incluídos.

Tendo em vista que os sistemas operacionais são muito grandes (cinco milhões de linhas de código não é incomum), ter de recompilar tudo cada vez que um arquivo é alterado seria insuportável. Por outro lado, mudar um arquivo de cabeçalho chave que esteja incluído em milhares de outros arquivos não exige recompilar esses arquivos. Acompanhar quais arquivos-objeto depende de quais arquivos de cabeçalho seria completamente impraticável sem ajuda.

Ainda bem que os computadores são muito bons precisamente nesse tipo de coisa. Nos sistemas UNIX, há um programa chamado *make* (com inúmeras variantes como *gmake*, *pmake* etc.) que lê o *Makefile*, que diz a ele quais arquivos são dependentes de quais outros arquivos. O que o *make* faz é ver quais arquivos-objeto são necessários para construir o binário do sistema operacional e para cada um conferir para ver se algum dos arquivos dos quais ele depende (o código e os cabeçalhos) foi modificado depois da última vez que o arquivo-objeto foi criado. Se isso ocorreu, esse arquivo-objeto deve ser recompilado. Quando *make* determinar quais arquivos *.c* precisam ser recompilados, ele então invoca o compilador C para compilá-los novamente, reduzindo assim o número de compilações ao mínimo possível. Em grandes projetos, a criação do *Makefile* é propensa a erros, portanto existem ferramentas que fazem isso automaticamente.

Uma vez que todos os arquivos *.o* estejam prontos, eles são passados para um programa chamado **ligador (*linker*)** para combinar todos eles em um único arquivo binário executável. Quaisquer funções de biblioteca chamadas também são incluídas nesse ponto, referências interfuncionais resolvidas e endereços de máquinas relocados conforme a necessidade. Quando o ligador é

terminado, o resultado é um programa executável, tradicionalmente chamado *a.out* em sistemas UNIX. Os vários componentes desse processo estão ilustrados na Figura 1.30 para um programa com três arquivos C e dois arquivos de cabeçalho. Embora estejamos discutindo o desenvolvimento de sistemas operacionais aqui, tudo isso se aplica ao desenvolvimento de qualquer programa de grande porte.

1.8.4 O modelo de execução

Uma vez que os binários do sistema operacional tenham sido ligados, o computador pode ser reinicializado e o novo sistema operacional carregado. Ao ser executado, ele pode carregar dinamicamente partes que não foram estaticamente incluídas no sistema binário, como drivers de dispositivo e sistemas de arquivos. No tempo de execução, o sistema operacional pode consistir de múltiplos segmentos, para o texto (o código de programa), os dados e a pilha. O segmento de texto é em geral imutável, não se alterando durante a execução. O segmento de dados começa em um determinado tamanho e é inicializado com determinados valores, mas pode mudar e crescer conforme a necessidade. A pilha inicia vazia, mas cresce e diminui conforme as funções são chamadas e retornadas. Muitas vezes o segmento de texto é colocado próximo à parte inferior da memória, o segmento 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, mas sistemas diferentes funcionam diferentemente.

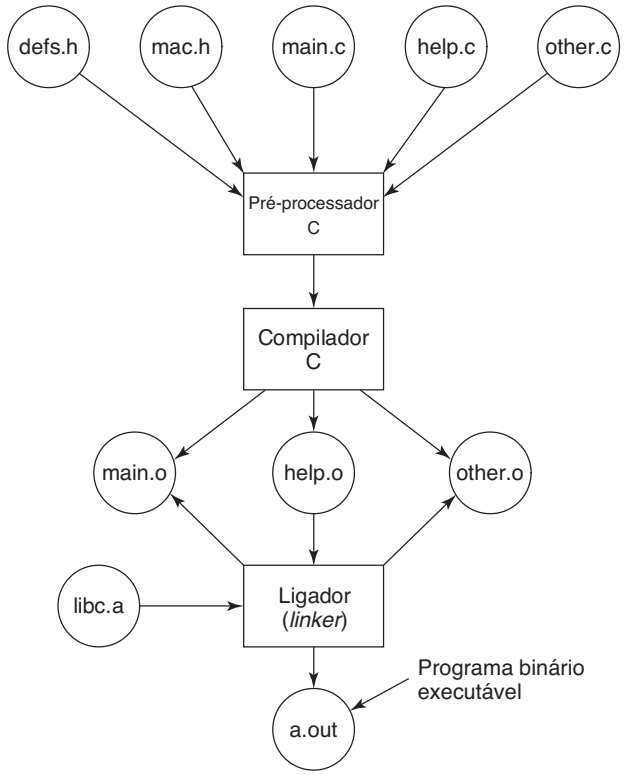
Em todos os casos, o código do sistema operacional é diretamente executado pelo hardware, sem interpretadores ou compilação just-in-time, como é normal com Java.

1.9 Pesquisa em sistemas operacionais

A ciência de computação é um campo que avança rapidamente e é difícil de prever para onde ele está indo. Pesquisadores em universidades e laboratórios de pesquisa industrial estão constantemente pensando em novas ideias, algumas das quais não vão a parte alguma, mas outras tornam-se a pedra fundamental de produtos futuros e têm um impacto enorme sobre a indústria e usuários. Diferenciar umas das outras é mais fácil depois do momento em que são lançadas. Separar o joio do trigo é especialmente difícil, pois muitas vezes são necessários de 20 a 30 anos para uma ideia causar um impacto.

Por exemplo, quando o presidente Eisenhower criou a ARPA (Advanced Research Projects Agency — Agência

FIGURA 1.30 O processo de compilação de C e arquivos de cabeçalho para criar um arquivo executável.



de Projetos de Pesquisa Avançada) do Departamento de Defesa em 1958, ele estava tentando evitar que o Exército tomasse conta do orçamento de pesquisa do Pentágono, deixando de fora a Marinha e a Força Aérea. Ele não estava tentando inventar a internet. Mas uma das coisas que a ARPA fez foi financiar alguma pesquisa universitária sobre o então obscuro conceito de comutação de pacotes, que levou à primeira rede de comutação de pacotes, a ARPANET. Ela foi criada em 1969. Não levou muito tempo e outras redes de pesquisa financiadas pela ARPA estavam conectadas à ARPANET, e a internet foi criada. A internet foi então usada alegremente pelos pesquisadores acadêmicos para enviar e-mails uns para os outros por 20 anos. No início da década de 1990, Tim Berners-Lee inventou a World Wide Web no laboratório de pesquisa CERN em Genebra e Marc Andreessen criou um navegador gráfico para ela na Universidade de Illinois. De uma hora para outra a internet estava cheia de adolescentes batendo papo. O presidente Eisenhower está provavelmente rolando em sua sepultura.

A pesquisa em sistemas operacionais também levou a mudanças dramáticas em sistemas práticos. Como discutimos antes, os primeiros sistemas de computadores comerciais eram todos em lote, até que o M.I.T. inventou o tempo compartilhado interativo no início da década de 1960. Computadores eram todos baseados em texto até que Doug Engelbart inventou o mouse e a interface gráfica com o usuário no Instituto de Pesquisa Stanford no fim da década de 1960. Quem sabe o que virá por aí?

Nesta seção e em seções comparáveis neste livro, examinaremos brevemente algumas das pesquisas que foram feitas sobre sistemas operacionais nos últimos cinco a dez anos, apenas para dar um gosto do que pode vir pela frente. Esta introdução certamente não é abrangente. Ela é baseada em grande parte nos estudos que foram publicados nas principais conferências de pesquisa, pois essas ideias pelo menos passaram por um processo rigoroso de análise de seus pares a fim de serem publicados. Observe que na ciência de computação — em comparação com outros campos científicos — a maior parte da pesquisa é publicada em conferências, não em periódicos. A maioria dos estudos citados nas seções de pesquisa foi publicada pela ACM, a IEEE Computer Society ou USENIX, e estão disponíveis na internet para membros (estudantes) dessas organizações. Para mais informações sobre essas organizações e suas bibliotecas digitais, ver

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

Virtualmente todos os pesquisadores de sistemas operacionais sabem que os sistemas operacionais atuais são enormes, inflexíveis, inconfiáveis, inseguros e carregados de erros, uns mais que os outros (*os nomes não são citados aqui para proteger os culpados*). Consequentemente, há muita pesquisa sobre como construir sistemas operacionais melhores. Trabalhos foram publicados recentemente sobre erros em códigos e sua correção (RENZELMANN et al., 2012; e ZHOU et al., 2012), recuperação de travamentos (CORREIA et al., 2012; MA et al., 2013; ONGARO et al., 2011; e YEH e CHENG, 2012), gerenciamento de energia (PATHAK et al., 2012; PETRUCCI e LOQUES, 2012; e SHEN et al., 2013), sistemas de armazenamento e de arquivos (ELNABLY e WANG, 2012; NIGHTINGALE et al., 2012; e ZHANG et al., 2013a), E/S de alto desempenho (De BRUIJN et al., 2011; LI et al., 2013a; e RIZZO, 2012), *hiper-threading* e *multi-threading* (LIU et al., 2011), atualização ao vivo (GIUFFRIDA et al., 2013), gerenciando GPUs (ROSSBACH et al., 2011), gerenciamento de memória (JANTZ et al., 2013; e JEONG et al., 2013), sistemas operacionais com múltiplos núcleos (BAUMANN et al., 2009; KAPRITSOS, 2012; LACHAIZE et al., 2012; e WENTZLAFF et al., 2012), corretude de sistemas operacionais (ELPHINSTONE et al., 2007; YANG et al., 2006; e KLEIN et al., 2009), confiabilidade de sistemas operacionais (HRUBY et al., 2012; RYZHYK et al., 2009, 2011 e ZHENG et al., 2012), privacidade e segurança (DUNN et al., 2012; GIUFFRIDA et al., 2012; LI et al., 2013b; LORCH et al., 2013; ORTOLANI e CRISPO, 2012; SLOWINSKA et al., 2012; e UR et al., 2012), uso e monitoramento de desempenho (HARTER et al., 2012; e RAVINDRANATH et al., 2012), e virtualização (AGESEN et al., 2012; BEN-YEHUDA et al., 2010; COLP et al., 2011; DAI et al., 2013; TARASOV et al., 2013; e WILLIAMS et al., 2012) entre muitos outros tópicos.

1.10 Delineamento do resto deste livro

Agora completamos a nossa introdução e visão panorâmica do sistema operacional. É chegada a hora de entrarmos nos detalhes. Como já mencionado, do ponto de vista do programador, a principal finalidade de um sistema operacional é fornecer algumas abstrações fundamentais, das quais as mais importantes são os processos e threads, espaços de endereçamento e arquivos. Portanto, os próximos três capítulos são devotados a esses tópicos críticos.

O Capítulo 2 trata de processos e threads. Ele discute as suas propriedades e como eles se comunicam uns com os outros. Ele também dá uma série de exemplos detalhados de como a comunicação entre processos funciona e como evitar algumas de suas armadilhas.

No Capítulo 3, estudaremos detalhadamente os espaços de endereçamento e seu complemento e o gerenciamento de memória. O tópico importante da memória virtual será examinado, juntamente com conceitos proximamente relacionados, como a paginação e segmentação.

Então, no Capítulo 4, chegaremos ao tópico tão importante dos sistemas de arquivos. Em grande parte, o que o usuário mais vê é o sistema de arquivos. Examinaremos tanto a interface como a implementação de sistemas de arquivos.

A Entrada/Saída é coberta no Capítulo 5. Os conceitos de independência e dependência de dispositivos serão examinados. Vários dispositivos importantes, incluindo discos, teclados e monitores, serão usados como exemplos.

O Capítulo 6 aborda os impasses. Mostramos brevemente o que são os impasses neste capítulo, mas há muito mais para se dizer a respeito deles. São discutidas maneiras de prevenir e evitá-los.

A essa altura teremos completado nosso estudo dos princípios básicos de sistemas operacionais de uma única CPU. No entanto, há mais a ser dito, em especial sobre tópicos avançados. No Capítulo 7, examinaremos a virtualização. Discutiremos em detalhes tanto os princípios quanto algumas das soluções de virtualização existentes. Tendo em vista que a virtualização é intensamente usada na computação na nuvem, também observaremos os sistemas de nuvem que há por aí. Outro tópico avançado diz respeito aos sistemas de multiprocessadores, incluindo múltiplos núcleos, computadores paralelos e sistemas distribuídos. Esses assuntos são cobertos no Capítulo 8.

Um assunto importantíssimo é o da segurança de sistemas operacionais, que é coberto no Capítulo 9. Entre os tópicos discutidos nesse capítulo está o das ameaças (por exemplo, vírus e vermes), mecanismos de proteção e modelos de segurança.

Em seguida temos alguns estudos de caso de sistemas operacionais reais. São eles: UNIX, Linux e Android (Capítulo 10) e Windows 8 (Capítulo 11). O texto conclui com alguma sensatez e reflexões sobre projetos de sistemas operacionais no Capítulo 12.

1.11 Unidades métricas

Para evitar qualquer confusão, vale a pena declarar explicitamente que neste livro, como na ciência de computação em geral, as unidades métricas são usadas em vez das unidades inglesas tradicionais (o sistema *furlong-stone-furlong*). Os principais prefixos métricos são listados na Figura 1.31. Os prefixos são abreviados por suas primeiras letras, com as unidades maiores que 1 em letras maiúsculas. Desse modo, 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}s. Tendo em vista que tanto mili quanto micro começam com a letra "m", uma escolha tinha de ser feita. Normalmente, "m" é para mili e "μ" (a letra grega mu) é para micro.

Também vale a pena destacar que, em comum com a prática da indústria, as unidades para mensurar tamanhos da memória têm significados ligeiramente diferentes. O quilo corresponde a 2^{10} (1.024) em vez de 10^3 (1.000), pois as memórias são sempre expressas em potências de dois. Desse modo, uma memória de 1 KB contém 1.024 bytes, não 1.000 bytes. Similarmente, 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. No entanto, uma linha de comunicação de 1 Kbps transmite 1.000 bits por segundo e uma LAN de 10 Mbps transmite a

FIGURA 1.31 Os principais prefixos métricos.

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

10.000.000 bits/s, pois essas velocidades não são potências de dois. Infelizmente, muitas pessoas tendem a misturar os dois sistemas, em especial para tamanhos de discos. Para evitar ambiguidade, usaremos neste livro 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 e 10^9 bits/s, respectivamente.

1.12 Resumo

Sistemas operacionais podem ser vistos de dois pontos de vista: como gerenciadores de recursos e como máquinas estendidas. Como gerenciador de recursos, o trabalho do sistema operacional é gerenciar as diferentes partes do sistema de maneira eficiente. Como máquina estendida, o trabalho do sistema é proporcionar aos usuários abstrações que sejam mais convenientes para usar do que a máquina real. Essas incluem processos, espaços de endereçamento e arquivos.

Sistemas operacionais têm uma longa história, começando nos dias quando substituíam o operador, até os sistemas de multiprogramação modernos. Destaques incluem os primeiros sistemas em lote, sistemas de multiprogramação e sistemas de computadores pessoais.

Como os sistemas operacionais interagem intimamente com o hardware, algum conhecimento sobre o hardware 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 os processos, o gerenciamento de memória, o gerenciamento de E/S, o sistema de arquivos e a segurança. Cada um deles será tratado em um capítulo subsequente.

O coração de qualquer sistema operacional é o conjunto de chamadas de sistema com que ele consegue lidar. Essas chamadas dizem o que o sistema operacional realmente faz. Para UNIX, examinamos quatro grupos de chamadas de sistema. O primeiro grupo diz respeito à criação e ao término de processos. O segundo é para a leitura e escrita de arquivos. O terceiro é para o gerenciamento de diretórios. O quarto grupo contém chamadas diversas.

Sistemas operacionais podem ser estruturados de várias maneiras. As mais comuns são o sistema monolítico, a hierarquia de camadas, o micronúcleo, o cliente-servidor, a máquina virtual e o exonúcleo.

PROBLEMAS

1. Quais são as duas principais funções de um sistema operacional?
2. Na Seção 1.4, nove tipos diferentes de sistemas operacionais são descritos. Dê uma lista das aplicações para cada um desses sistemas (uma para cada tipo de sistema operacional).
3. Qual é a diferença entre sistemas de compartilhamento de tempo e de multiprogramação?
4. Para usar a memória de cache, a memória principal é dividida em linhas de cache, em geral de 32 a 64 bytes de comprimento. Uma linha inteira é capturada em cache de uma só vez. Qual é a vantagem de fazer isso com uma linha inteira em vez de um único byte ou palavra de cada vez?
5. Nos primeiros computadores, cada byte de dados lido ou escrito era executado pela CPU (isto é, não havia DMA). Quais implicações isso tem para a multiprogramação?
6. Instruções relacionadas ao acesso a dispositivos de E/S são tipicamente instruções privilegiadas, isto é, podem ser executadas em modo núcleo, mas não em modo usuário. Dê uma razão de por que essas instruções são privilegiadas.
7. A ideia de família de computadores foi introduzida na década de 1960 com os computadores de grande porte System/360 da IBM. Essa ideia está ultrapassada ou ainda é válida?
8. Uma razão para a adoção inicialmente lenta das GUIs era o custo do hardware necessário para dar suporte a elas. Quanta RAM de vídeo é necessária para dar suporte a uma tela de texto monocromo de 25 linhas $\times$ 80 colunas de caracteres? E para um bitmap colorido de 24 bits de 1.200 $\times$ 900 pixels? Qual era o custo desta RAM em preços de 1980 (US$ 5/KB)? Quanto é agora?
9. Há várias metas de projeto na construção de um sistema operacional, por exemplo, utilização de recursos, oportunidade, robustez e assim por diante. Dê um exemplo de duas metas de projeto que podem contradizer uma à outra.

10. Qual é a diferença entre modo núcleo e modo usuário? Explique como ter dois modos distintos ajuda no projeto de um sistema operacional.

11. Um disco de 255 GB tem 65.536 cilindros com 255 setores por faixa e 512 bytes por setor. Quantos pratos e cabeças esse disco tem? Presumindo um tempo de busca de cilindro médio de 11 ms, atraso rotacional médio de 7 ms e taxa de leitura de 100 MB/s, calcule o tempo médio que será necessário para ler 400 KB de um setor.

12. Quais das instruções a seguir devem ser deixadas somente em modo núcleo?

(a) Desabilitar todas as interrupções.

(b) Ler o relógio da hora do dia.

(c) Configurar o relógio da hora do dia.

(d) Mudar o mapa de memória.

13. Considere um sistema que tem duas CPUs, cada uma tendo duas threads (hiper-threading). Suponha que três programas, *P0*, *P1* e *P2*, sejam iniciados com tempos de execução de 5, 10 e 20 ms, respectivamente. Quanto tempo levará para completar a execução desses programas? Presuma que todos os três programas sejam 100% ligados à CPU, não bloqueiem durante a execução e não mudem de CPUs uma vez escolhidos.

14. Um computador tem um pipeline com quatro estágios. Cada estágio leva um tempo para fazer seu trabalho, a saber, 1 ns. Quantas instruções por segundo essa máquina consegue executar?

15. Considere um sistema de computador que tem uma memória de cache, memória principal (RAM) e disco, e um sistema operacional que usa memória virtual. É necessário 1 ns para acessar uma palavra da cache, 10 ns para acessar uma palavra da RAM e 10 ms para acessar uma palavra do disco. Se o índice de acerto da cache é 95% e o índice de acerto da memória principal (após um erro de cache) 99%, qual é o tempo médio para acessar uma palavra?

16. Quando um programa de usuário faz uma chamada de sistema para ler ou escrever um arquivo de disco, ele fornece uma indicação de qual arquivo ele quer, um ponteiro para o buffer de dados e o contador. O controle é então transferido para o sistema operacional, que chama o driver apropriado. Suponha que o driver começa o disco e termina quando ocorre uma interrupção. No caso da leitura do disco, obviamente quem chamou terá de ser bloqueado (pois não há dados para ele). E quanto a escrever para o disco? Quem chamou precisa ser bloqueado esperando o término da transferência de disco?

17. O que é uma instrução? Explique o uso em sistemas operacionais.

18. Por que a tabela de processos é necessária em um sistema de compartilhamento de tempo? Ela também é necessária em sistemas de computadores pessoais executando UNIX ou Windows com um único usuário?

19. Existe alguma razão para que você quisesse montar um sistema de arquivos em um diretório não vazio? Se a resposta for sim, por quê?

20. Para cada uma das chamadas de sistema a seguir, dê uma condição que a faça falhar: fork, exec e unlink.

21. Qual tipo de multiplexação (tempo, espaço ou ambos) pode ser usado para compartilhar os seguintes recursos: CPU, memória, disco, placa de rede, impressora, teclado e monitor?

22. A chamada

```
count = write(fd, buffer, nbytes);
```

pode retornar qualquer valor em *count* fora *nbytes*? Se a resposta for sim, por quê?

23. Um arquivo cujo descritor é *fd* contém a sequência de bytes: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. As chamadas de sistema a seguir são feitas:

```
lseek(fd, 3, SEEK_SET);
read(fd, &buffer, 4);
```

onde a chamada lseek faz uma busca para o byte 3 do arquivo. O que o *buffer* contém após a leitura ter sido feita?

24. Suponha que um arquivo de 10 MB esteja armazenado em um disco na mesma faixa (faixa 50) em setores consecutivos. O braço do disco está atualmente situado sobre o número da faixa 100. Quanto tempo ele levará para retirar esse arquivo do disco? Presuma que ele leve em torno de 1 ms para mover o braço de um cilindro para o próximo e em torno de 5 ms para o setor onde o início do arquivo está armazenado para girar sob a cabeça. Também, presuma que a leitura ocorra a uma taxa de 200 MB/s.

25. Qual é a diferença essencial entre um arquivo especial de bloco e um arquivo especial de caractere?

26. No exemplo dado na Figura 1.17, a rotina de biblioteca é chamada *read* e a chamada de sistema em si é chamada read. É fundamental que ambas tenham o mesmo nome? Se não, qual é a mais importante?

27. Sistemas operacionais modernos desacoplam o espaço de endereçamento do processo da memória física da máquina. Liste duas vantagens desse projeto.

28. Para um programador, uma chamada de sistema parece com qualquer outra chamada para uma rotina de biblioteca. É importante que um programador saiba quais rotinas de biblioteca resultam em chamadas de sistema? Em quais circunstâncias e por quê?

29. A Figura 1.23 mostra que uma série de chamadas de sistema UNIX não possuem equivalentes na API Win32. Para cada uma das chamadas listadas como não tendo um equivalente Win32, quais são as consequências para

um programador de converter um programa UNIX para ser executado sob o Windows?

30. Um sistema operacional portátil é um sistema que pode ser levado de uma arquitetura de sistema para outra sem nenhuma modificação. Explique por que é impraticável construir um sistema operacional que seja completamente portátil. Descreva duas camadas de alto nível que você terá ao projetar um sistema operacional que seja altamente portátil.

31. Explique como a separação da política e mecanismo ajuda na construção de sistemas operacionais baseados em micronúcleos.

32. Máquinas virtuais tornaram-se muito populares por uma série de razões. Não obstante, elas têm alguns problemas. Cite um.

33. A seguir algumas questões para praticar conversões de unidades:

(a) Quantos segundos há em um nanoano?

(b) Micrômetros são muitas vezes chamados de mícrons. Qual o comprimento de um megamícron?

(c) Quantos bytes existem em uma memória de 1 PB?

(d) A massa da Terra é 6.000 yottagramas. Quanto é isso em quilogramas?

34. Escreva um shell que seja similar à Figura 1.19, mas contenha código suficiente para que ela realmente funcione, de maneira que você possa testá-lo. Talvez você queira acrescentar alguns aspectos como o redirecionamento de entrada e saída, pipes e tarefas de segundo plano.

35. Se você tem um sistema tipo UNIX (Linux, MINIX 3, FreeBSD etc.) disponível que possa seguramente derrubar e reinicializar, escreva um script de shell que tente criar um número ilimitado de processos filhos e observe o que acontece. Antes de realizar o experimento, digite sync para o shell para limpar os buffers de sistema de arquivos para disco para evitar arruinar o sistema de arquivos. Você também pode fazer o experimento seguramente em uma máquina virtual.

Nota: não tente fazer isso em um sistema compartilhado sem antes conseguir a permissão do administrador do sistema. As consequências serão de imediato óbvias, então é provável que você seja pego e sofra sanções.

36. Examine e tente interpretar os conteúdos de um diretório tipo UNIX ou Windows com uma ferramenta como o programa UNIX *od*. (*Dica*: como você vai fazer isso depende do que o sistema operacional permitir. Um truque que pode funcionar é criar um diretório em um pen drive com um sistema operacional e então ler os dados brutos do dispositivo usando um sistema operacional diferente que permita esse acesso.)