

Um sistema operacional fornece o ambiente dentro do qual os programas são executados. Internamente, os sistemas operacionais variam muito em sua composição, já que estão organizados em muitas linhas diferentes. O projeto de um novo sistema operacional é uma tarefa de peso. É importante que os objetivos do sistema sejam bem definidos antes de o projeto começar. Esses objetivos formam a base das escolhas feitas entre vários algoritmos e estratégias.

Podemos considerar um sistema operacional segundo vários critérios. Um ponto de vista enfoca os serviços que o sistema fornece; outro, a interface que ele torna disponível para usuários e programadores; e um terceiro enfoca seus componentes e suas interconexões. Neste capítulo, exploramos todos os três aspectos dos sistemas operacionais, mostrando os pontos de vista de usuários, programadores e projetistas de sistemas operacionais. Consideramos os serviços que um sistema operacional fornece, como eles são fornecidos, como são depurados e que metodologias existem para o projeto desses sistemas. Para concluir, descrevemos como os sistemas operacionais são criados e como um computador inicia seu sistema operacional.

## OBJETIVOS DO CAPÍTULO

- Descrever os serviços que um sistema operacional fornece para usuários, processos e outros sistemas.
- Discutir as diversas formas de estruturação de um sistema operacional.
- Explicar como os sistemas operacionais são instalados e personalizados e como são inicializados.

### 2.1 Serviços do Sistema Operacional

Um sistema operacional fornece um ambiente para a execução de programas. Ele fornece certos serviços para programas e para os usuários desses programas. Os serviços específicos fornecidos diferem, obviamente, de um sistema operacional para outro, mas podemos identificar classes comuns. Esses serviços do sistema operacional são fornecidos visando à conveniência do programador, para tornar mais fácil a tarefa de programar. A Figura 2.1 mostra uma representação dos diversos serviços do sistema operacional e como eles estão relacionados.

Um conjunto de serviços do sistema operacional fornece funções que são úteis para o usuário.

- **Interface de usuário.** Quase todos os sistemas operacionais têm uma interface de usuário (UI — user interface). Essa interface pode assumir várias formas. Uma delas é a interface de linha de comando (CLI — command-line interface) que usa comandos de texto e um método para sua entrada (digamos, um teclado para a digitação de comandos em um formato específico com opções específicas). Outra é a interface batch em que os comandos e suas diretivas de controle são inseridos em arquivos, e esses arquivos são executados. O mais comum é o uso de uma interface gráfica de usuário (GUI — graphical user interface). Nesse caso, a interface é um sistema de janelas com um dispositivo apontador para direcionar o I/O, selecionar a partir de menus e escolher opções e um teclado para entrada de texto. Alguns sistemas fornecem duas dessas variações ou as três.

- **Execução de programas.** O sistema deve ser capaz de carregar um programa na memória e executar esse programa. O programa deve ser capaz de encerrar sua execução, normal ou anormalmente (indicando o erro).
- **Operações de I/O.** Um programa em execução pode requerer I/O, e isso pode envolver um arquivo ou um dispositivo de I/O. Para dispositivos específicos, funções especiais podem ser desejáveis (como a gravação em um drive de CD ou DVD ou a limpeza de uma tela). Para eficiência e proteção, os usuários geralmente não podem controlar os dispositivos de I/O diretamente. Portanto, o sistema operacional deve fornecer um meio para executar I/O.
- **Manipulação do sistema de arquivos.** O sistema de arquivos é de especial interesse. É claro que os programas precisam ler e gravar arquivos e diretórios. Eles também precisam criar e excluí-los pelo nome, procurar um arquivo específico e listar informações de arquivos. Para concluir, alguns sistemas operacionais incluem o gerenciamento de permissões para permitir ou negar acesso a arquivos ou diretórios com base no proprietário dos arquivos. Muitos sistemas operacionais fornecem uma variedade de sistemas de arquivos, algumas vezes para permitir a escolha pessoal e, outras vezes, para fornecer recursos específicos ou características de desempenho.
- **Comunicações.** Há muitas situações em que um processo precisa trocar informações com outro processo. Essa comunicação pode ocorrer entre processos sendo executados no mesmo computador ou entre processos sendo executados em sistemas de computação diferentes, conectados por uma rede de computadores. As comunicações podem ser implementadas por memória compartilhada, em que dois ou mais processos leem e gravam em uma seção compartilhada da memória, ou por troca de mensagens, em que pacotes de informações em formatos predefinidos são transmitidos entre processos pelo sistema operacional.



Figura 2.1 Uma visão dos serviços do sistema operacional.

- **Deteção de erros.** O sistema operacional precisa detectar e corrigir erros constantemente. Os erros podem ocorrer no hardware da CPU e da memória (como um erro de memória ou a falta de energia), em dispositivos de I/O (como um erro de paridade em disco, uma falha na conexão

de rede ou a falta de papel na impressora) e no programa do usuário (como um overflow aritmético, uma tentativa de acessar uma locação ilegal na memória, ou o uso excessivo de tempo da CPU). Para cada tipo de erro, o sistema operacional deve tomar a medida apropriada para assegurar a computação correta e consistente. Em algumas situações, ele não tem escolha, a não ser interromper o sistema. Em outras, pode encerrar um processo causador de erro, ou retornar um código de erro ao processo para que este detecte o erro e, possivelmente, o corrija.

Existe outro conjunto de funções do sistema operacional cujo objetivo não é ajudar o usuário, mas, sim, assegurar a operação eficiente do próprio sistema. Sistemas com múltiplos usuários podem ganhar eficiência compartilhando os recursos do computador entre os usuários.

- **Alocação de recursos.** Quando existem múltiplos usuários ou jobs ativos ao mesmo tempo, é necessário alocar recursos para cada um deles. O sistema operacional gerencia muitos tipos diferentes de recursos. Alguns (como os ciclos de CPU, a memória principal e o armazenamento em arquivos) podem ter um código especial de alocação, enquanto outros (como os dispositivos de I/O) podem ter um código muito mais genérico de solicitação e liberação. Por exemplo, para determinar a melhor forma de usar a CPU, os sistemas operacionais possuem rotinas de scheduling da CPU que levam em consideração a velocidade da CPU, os jobs que devem ser executados, o número de registradores disponíveis e outros fatores. Também podem existir rotinas de alocação de impressoras, drives de armazenamento USB e outros dispositivos periféricos.

- **Contabilização.** Queremos controlar que usuários utilizam que quantidade e que tipos de recursos do computador. Essa monitoração pode ser usada a título de contabilização (para que os usuários possam ser cobrados) ou, simplesmente, para acumulação de estatísticas de uso. As estatísticas de uso podem ser uma ferramenta valiosa para pesquisadores que desejem reconfigurar o sistema para melhorar os serviços de computação.

- **Proteção e segurança.** Os proprietários de informações armazenadas em um sistema de computação multiusuário ou em rede podem querer controlar o uso dessas informações. Quando vários processos separados são executados concorrentemente, um processo não pode interferir nos outros ou no próprio sistema operacional. Proteção significa garantir que qualquer acesso a recursos do sistema seja controlado. A segurança do sistema contra invasores também é importante. Tal segurança começa com a exigência de que cada usuário se autentique junto ao sistema, geralmente por meio de uma senha, para obter acesso aos recursos do sistema. Ela se estende à defesa de dispositivos externos de I/O, incluindo adaptadores de rede, contra tentativas de acesso ilegal, e à gravação de todas essas conexões para a detecção de invasões. Para um sistema estar protegido e seguro, precauções devem ser tomadas em toda a sua extensão. A força de uma corrente se mede pelo seu elo mais fraco.

## **2.2 Interface entre o Usuário e o Sistema Operacional**

Mencionamos, anteriormente, que há diversas maneiras para os usuários se comunicarem com o sistema operacional. Aqui, discutimos duas abordagens básicas. Uma fornece uma interface de linha de comando, ou interpretador de comandos, que permite aos usuários darem entrada, diretamente, em comandos a serem executados pelo sistema operacional. A outra permite que os usuários se comuniquem com o sistema operacional por meio de uma interface gráfica de usuário ou GUI.

### **2.2.1 Interpretadores de Comandos**

Alguns sistemas operacionais incluem o interpretador de comandos no kernel. Outros, como o Windows e o UNIX, tratam o interpretador de comandos como um programa especial que está sendo executado quando um job é iniciado ou quando um usuário faz login pela primeira vez (em sistemas interativos). Em sistemas em que é possível escolher entre vários interpretadores de comandos, esses interpretadores são conhecidos como shells. Por exemplo, em sistemas UNIX e Linux, um usuário pode escolher entre vários shells diferentes, inclusive o shell Bourne, shell C, shell Bourne-Again, shell Korn, e outros. Shells de terceiros e shells gratuitos criados por usuários também estão disponíveis. A maioria dos shells fornece funcionalidade semelhante, e a escolha que o usuário faz do shell a ser usado baseia-se, em geral, na preferência pessoal. A Figura 2.2 mostra o interpretador de comandos do shell Bourne sendo usado no Solaris 10.

A principal função do interpretador de comandos é capturar e executar o próximo comando especificado pelo usuário. Muitos dos comandos fornecidos, nesse nível, manipulam arquivos: criar, excluir, listar, imprimir, copiar, executar, e assim por diante. Os shells do MS-DOS e do UNIX operam dessa forma. Esses comandos podem ser implementados de duas maneiras gerais. Em uma abordagem, o próprio interpretador de comandos contém o código que executa o comando. Por exemplo, um comando que exclui um arquivo pode fazer o interpretador de comandos saltar para uma seção de seu código que configura os parâmetros e faz a chamada de sistema apropriada. Nesse caso, o número de comandos que pode ser fornecido determina o tamanho do interpretador de comandos, já que cada comando requer seu próprio código de implementação.

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

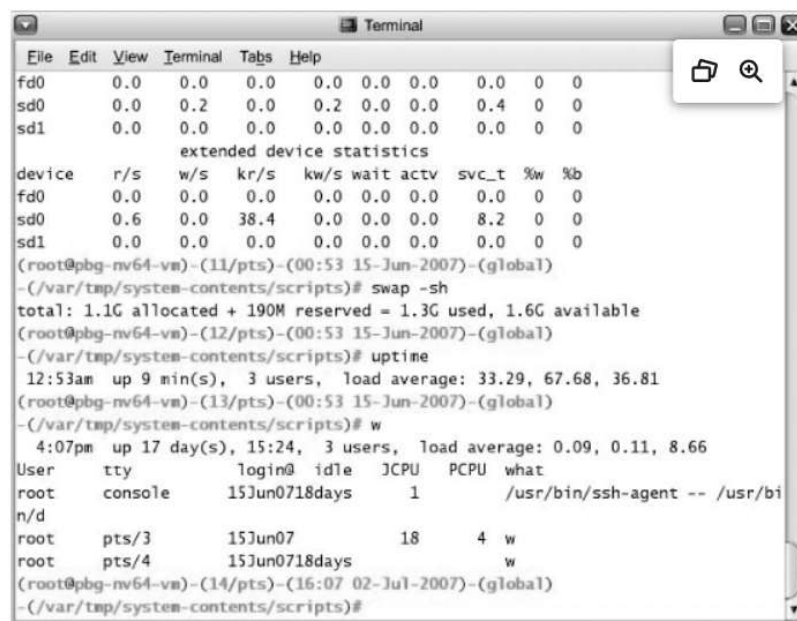
#### **rm file.txt**

buscaria um arquivo chamado rm, carregaria o arquivo na memória e o executaria com o parâmetro file.txt. A função associada ao comando rm seria totalmente definida pelo código existente no arquivo rm. Dessa forma, os programadores podem adicionar facilmente novos comandos ao sistema criando novos arquivos com os nomes apropriados. O programa interpretador de comandos que pode ser pequeno, não tem de ser alterado para novos comandos serem adicionados.

### 2.2.2 Interfaces Gráficas de Usuário

Uma segunda estratégia de comunicação com o sistema operacional é por meio de uma interface gráfica de usuário amigável ou GUI. Aqui, em vez de dar entrada em comandos, diretamente, por meio de uma interface de linha de comando, os usuários empregam um sistema de janelas e menus baseado em mouse, caracterizado por uma simulação de área de trabalho. O usuário movimenta o mouse para posicionar seu cursor em imagens, ou ícones, na tela (a área de trabalho), os quais representam programas, arquivos, diretórios e funções do sistema. Dependendo da localização do cursor do mouse, um clique em um de seus botões pode chamar um programa, selecionar um arquivo ou diretório — conhecido como pasta — ou abrir um menu contendo comandos.

As interfaces gráficas de usuário surgiram, em parte, por causa de pesquisas que ocorreram no início dos anos 1970, no centro de pesquisas Xerox PARC. A primeira GUI surgiu no computador Xerox Alto, em 1973. No entanto, as interfaces gráficas disseminaram-se nos anos 1980, com o advento dos computadores Apple Macintosh. A interface de usuário do sistema operacional Macintosh (Mac OS) passou por várias alterações através dos anos, sendo a mais significativa a adoção da interface Aqua, que surgiu com o Mac OS X. A primeira versão da Microsoft para o Windows — Versão 1.0 — baseou-se na adição de uma GUI ao sistema operacional MS-DOS. Versões posteriores do Windows forneceram alterações superficiais na aparência da GUI além de várias melhorias em sua funcionalidade.



```
File Edit View Terminal Tabs Help
fd0      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0  0
sd0      0.0    0.2    0.0    0.2    0.0    0.0    0.4    0  0
sd1      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0  0
extended device statistics
device   r/s    w/s    kr/s    kw/s    wait    actv    svc_t    %w    %b
fd0      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0  0
sd0      0.6    0.0   38.4    0.0    0.0    0.0    8.2    0  0
sd1      0.0    0.0    0.0    0.0    0.0    0.0    0.0    0  0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)# uptime
12:53am up 9 min(s),  3 users,  load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-/var/tmp/system-contents/scripts)# w
 4:07pm up 17 day(s), 15:24,  3 users,  load average: 0.09, 0.11, 8.66
User      tty          login@    idle    JCPU    PCPU    what
root      console      15Jun0718days    1      /usr/bin/ssh-agent -- /usr/bi
n/d
root      pts/3        15Jun07      18      4      w
root      pts/4        15Jun0718days    w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-/var/tmp/system-contents/scripts)#
```

Figura 2.2 O interpretador de comandos do shell Bourne no Solaris 10.



Figura 2.3 A tela sensível ao toque (touchscreen) do iPad.

Como o uso de um mouse é impraticável na maioria dos sistemas móveis, os smartphones e computadores tablets portáteis usam, tipicamente, uma interface sensível ao toque (touchscreen). Nesse caso, os usuários interagem fazendo gestos na tela sensível ao toque — por exemplo, pressionando e batendo com os dedos na tela. A Figura 2.3 ilustra a touchscreen do iPad da Apple. Enquanto os smartphones mais antigos incluíam um teclado físico, a maior parte dos smartphones atuais simula um teclado na touchscreen. Tradicionalmente, os sistemas UNIX têm sido dominados por interfaces de linha de comando. Várias GUIs estão disponíveis, no entanto. Elas incluem os sistemas Common Desktop Environment (CDE) e X-Windows, que são comuns em versões comerciais do UNIX, como o Solaris e o sistema AIX da IBM. Além disso, houve desenvolvimentos significativos na aparência das GUIs de vários projetos de código-fonte aberto, como o K Desktop Environment (ou KDE) e a área de trabalho do GNOME no projeto GNU. As áreas de trabalho do KDE e do GNOME são executadas no Linux e em vários sistemas UNIX e estão disponíveis sob licenças de código-fonte aberto, o que significa que seu código-fonte pode ser lido e modificado conforme os termos específicos da licença.

### 2.2.3 Escolha da Interface

A escolha entre usar uma interface de linha de comando ou uma GUI depende, em grande parte, de preferências pessoais. Administradores de sistemas que gerenciam computadores e usuários avançados que têm profundo conhecimento de um sistema usam, com frequência, a interface de linha de comando. Para eles, é mais eficiente, dando-lhes acesso mais rápido às atividades que precisam executar. Na verdade, em alguns sistemas, apenas um subconjunto de funções do sistema está disponível por meio da GUI, o que deixa as tarefas menos comuns para quem conhece a linha de comando. Além disso, as interfaces de linha de comando tornam, em geral, mais fácil a execução de tarefas repetitivas, em parte porque elas têm sua própria programabilidade. Por exemplo, se uma tarefa frequente precisa de um conjunto de passos de linha de comando, esses passos podem ser gravados em um arquivo, e o arquivo pode ser

executado como se fosse um programa. O programa não é compilado em código executável, mas interpretado pela interface de linha de comando. Esses scripts de shell são muito comuns em sistemas que são orientados à linha de comando, como o UNIX e o Linux.

Por outro lado, a maioria dos usuários do Windows gosta de usar o ambiente de GUI do sistema e quase nunca usa a interface de shell do MS-DOS. As diversas alterações pelas quais passaram os sistemas operacionais Macintosh também fornecem um estudo interessante. Historicamente, o Mac OS não fornecia uma interface de linha de comando, sempre demandando que seus usuários se comunicassem com o sistema operacional usando sua GUI. No entanto, com o lançamento do Mac OS X (que é, em parte, implementado com o uso de um kernel UNIX), o sistema operacional agora fornece tanto uma interface Aqua quanto uma interface de linha de comando. A Figura 2.4 é uma tomada de tela da GUI do Mac OS Big Sur.

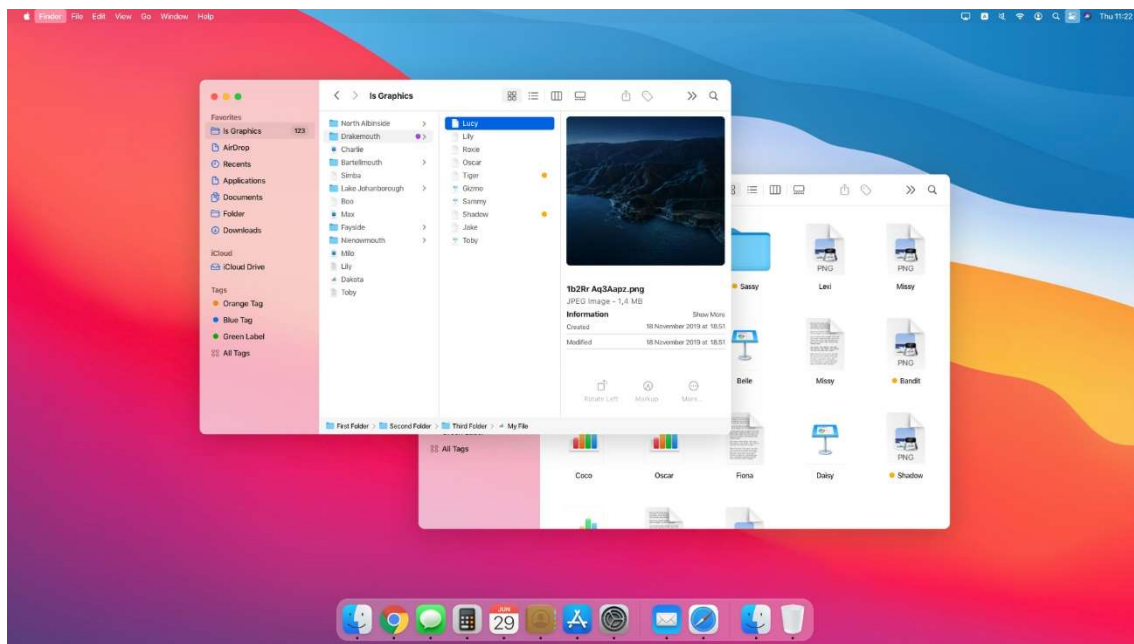


Figura 2.4 A GUI do Mac OS Big Sur.

A interface de usuário pode variar de um sistema para outro e, até mesmo, de usuário para usuário em um sistema. Ela costuma ser removida substancialmente da estrutura real do sistema. Portanto, o projeto de uma interface de usuário útil e amigável não depende diretamente do sistema operacional. Neste livro, estamos nos concentrando nos problemas básicos do fornecimento de serviço adequado para programas de usuário. Do ponto de vista do sistema operacional, não fazemos a distinção entre programas de usuário e programas de sistema.

## 2.3 Chamadas de Sistema

As chamadas de sistema fornecem uma interface com os serviços disponibilizados por um sistema operacional. Geralmente, essas chamadas estão disponíveis como rotinas escritas em C e C++, embora certas tarefas de baixo nível (por exemplo, tarefas em que o hardware deve ser

acessado diretamente) possam precisar ser escritas usando instruções em linguagem de montagem. Antes de discutirmos como um sistema operacional torna as chamadas de sistema disponíveis, vamos usar um exemplo para ilustrar como as chamadas de sistema são utilizadas: escrevendo um programa simples para ler dados em um arquivo e copiá-los em outro arquivo. A primeira entrada de que o programa precisará são os nomes dos dois arquivos: o arquivo de entrada e o arquivo de saída. Esses nomes podem ser especificados de muitas formas, dependendo do projeto do sistema operacional. Uma abordagem é aquela em que o programa solicita os nomes ao usuário. Em um sistema interativo, essa abordagem demandará uma sequência de chamadas de sistema, primeiro para exibir uma mensagem de alerta na tela e, em seguida, para ler a partir do teclado os caracteres que definem os dois arquivos. Em sistemas baseados em mouse e em ícones, é exibido, geralmente, um menu de nomes de arquivos em uma janela. O usuário pode, então, utilizar o mouse para selecionar o nome do arquivo de origem, e uma janela pode ser aberta para que o nome do arquivo de destino seja especificado. Essa sequência requer muitas chamadas de sistema de I/O.

Uma vez que os dois nomes de arquivo tenham sido obtidos, o programa deve abrir o arquivo de entrada e criar o arquivo de saída. Cada uma dessas operações requer outra chamada de sistema. Condições de erro que podem ocorrer, para cada operação, podem requerer chamadas de sistema adicionais. Quando o programa tentar abrir o arquivo de entrada, por exemplo, pode descobrir que não há arquivo com esse nome, ou que o arquivo está protegido contra acesso. Nesses casos, o programa deve exibir uma mensagem no console (outra sequência de chamadas de sistema) e, então, terminar anormalmente (outra chamada de sistema). Se o arquivo de entrada existe, devemos criar um novo arquivo de saída. Podemos descobrir que já existe um arquivo de saída com o mesmo nome. Essa situação pode fazer com que o programa aborte (uma chamada de sistema), ou podemos excluir o arquivo existente (outra chamada de sistema) e criar um novo (mais uma chamada de sistema). Outra opção, em um sistema interativo, é perguntar ao usuário (por meio de uma sequência de chamadas de sistema para exibir a mensagem de alerta e para ler a resposta a partir do terminal) se deseja substituir o arquivo existente ou abortar o programa.

Quando os dois arquivos estão definidos, entramos em um loop que lê o arquivo de entrada (uma chamada de sistema) e grava no arquivo de saída (outra chamada de sistema). Cada operação de leitura e gravação deve retornar informações de status referentes a várias condições de erro possíveis. Na entrada, o programa pode entender que o fim do arquivo foi alcançado ou que houve uma falha de hardware na leitura (como um erro de paridade). A operação de gravação pode encontrar vários erros, dependendo do dispositivo de saída (por exemplo, não há mais espaço em disco). Para concluir, após o arquivo inteiro ser copiado, o programa pode fechar os dois arquivos (outra chamada de sistema), exibir uma mensagem no console ou janela (mais chamadas de sistema) e, por fim, terminar normalmente (a última chamada de sistema). Essa sequência de chamadas de sistema é mostrada na Figura 2.5.



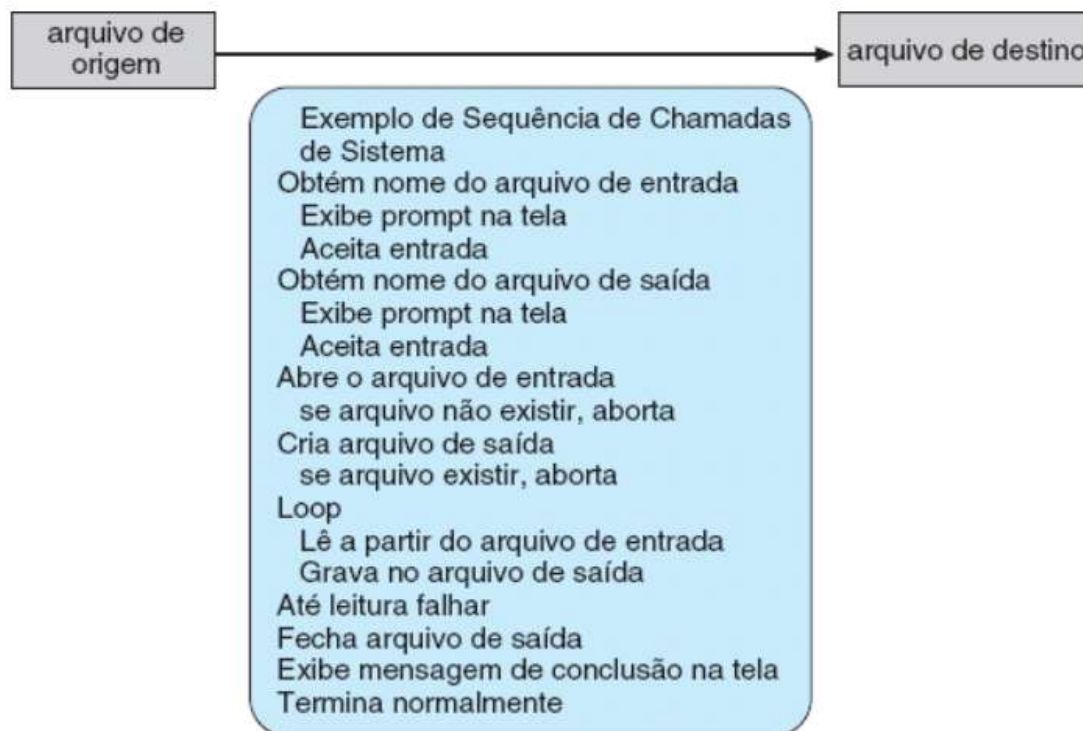


Figura 2.5 Exemplo de como as chamadas de sistema são usadas.

Como você pode ver, até mesmo programas simples podem fazer uso intenso do sistema operacional. Frequentemente, os sistemas executam milhares de chamadas de sistema por segundo. No entanto, a maioria dos programadores nunca vê esse nível de detalhe. Normalmente, os desenvolvedores de aplicações projetam programas de acordo com uma interface de programação de aplicações (API — application programming interface). A API especifica um conjunto de funções que estão disponíveis para um programador de aplicações, incluindo os parâmetros que são passados a cada função e os valores de retorno que o programador pode esperar. As três APIs mais comuns, disponíveis para programadores de aplicações, são a API Windows para sistemas Windows, a API POSIX para sistemas baseados em POSIX (que incluem virtualmente todas as versões do UNIX, Linux e Mac OS X) e a API Java para programas que são executados na máquina virtual Java. Um programador acessa uma API por meio de uma biblioteca de códigos fornecida pelo sistema operacional. No caso do UNIX e do Linux, para programas escritos na linguagem C, a biblioteca se chama `libc`. Observe que — a menos que especificado — os nomes das chamadas de sistema usados em todo este texto são exemplos genéricos. Cada sistema operacional tem seu próprio nome para cada chamada de sistema.

Em segundo plano, as funções que compõem uma API invocam tipicamente as chamadas de sistema reais em nome do programador de aplicações. Por exemplo, a função `CreateProcess ( )` do Windows (que, obviamente, é usada para criar um novo processo) na verdade invoca a chamada de sistema `NTCreateProcess ( )` no kernel do Windows. Por que um programador de aplicações iria preferir programar de acordo com uma API em vez de invocar chamadas de sistema reais? Há várias razões para fazer isso. Um benefício está relacionado com a portabilidade dos programas. Um programador de aplicações, ao projetar um programa usando uma API, espera que seu programa seja compilado e executado em qualquer sistema que dê suporte à mesma API (embora, na verdade, diferenças de arquitetura frequentemente tornem isso mais difícil do que parece). Além do mais, as chamadas de sistema reais costumam ser mais detalhadas e difíceis de manipular do que a API disponível para um programador de aplicações.

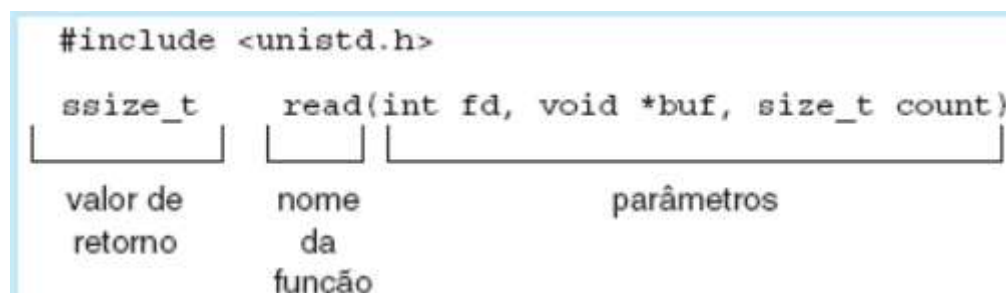
De qualquer forma, existe, com frequência, uma forte correlação entre uma função da API e a chamada de sistema associada a ela dentro do kernel. Na verdade, muitas das APIs POSIX e Windows são semelhantes às chamadas de sistema nativas fornecidas pelos sistemas operacionais UNIX, Linux e Windows.

## EXEMPLO DE API PADRÃO

Como exemplo de uma API padrão, considere a função `read ( )` que está disponível em sistemas UNIX e Linux. A API para essa função é obtida na página `man` invocando o comando

**man read**

na linha de comando. Uma descrição dessa API é mostrada a seguir:



Um programa que use a função `read ( )` deve incluir o arquivo de cabeçalho `unistd.h`, já que esse arquivo define os tipos de dados `ssize_t` e `size_t` (entre outras coisas). Os parâmetros passados para `read ( )` são os seguintes:

- `int fd` — o descritor de arquivo a ser lido
- `void *buf` — um buffer para o qual os dados serão lidos
- `size_t count` — o número máximo de bytes a serem lidos para o buffer

Em uma leitura bem-sucedida, o número de bytes lidos é retornado. Um valor de retorno igual a 0 indica fim de arquivo. Se ocorre um erro, `read ( )` retorna `-1`. Na maioria das linguagens de programação, o sistema de suporte ao tempo de execução (um conjunto de funções que faz parte das bibliotecas incluídas com o compilador) fornece uma interface de chamadas de sistema que serve como uma ponte para as chamadas de sistema disponibilizadas pelo sistema operacional. A interface de chamadas de sistema intercepta as chamadas de função da API e invoca as chamadas de sistema necessárias dentro do sistema operacional. Normalmente, um número é associado a cada chamada de sistema, e a interface de chamadas de sistema mantém uma tabela indexada de acordo com esses números. A interface de chamadas de sistema invoca,

então, a chamada de sistema desejada no kernel do sistema operacional e retorna o status da chamada de sistema e quaisquer valores de retorno.

O chamador não precisa saber coisa alguma sobre como a chamada de sistema é implementada ou o que ela faz durante a execução. Em vez disso, ele só precisa seguir a API e saber o que o sistema operacional fará como resultado da execução dessa chamada de sistema. Portanto, a maioria dos detalhes da interface do sistema operacional é oculta do programador pela API e gerenciada pela biblioteca de suporte ao tempo de execução. O relacionamento entre uma API, a interface de chamadas de sistema e o sistema operacional é mostrado na Figura 2.6, que ilustra como o sistema operacional manipula uma aplicação de usuário invocando a chamada de sistema `open ( )`.

As chamadas de sistema ocorrem de diferentes maneiras, dependendo do computador que estiver sendo usado. Geralmente, são necessárias mais informações do que simplesmente a identidade da chamada de sistema desejada. O tipo e o montante exatos das informações variam de acordo com a chamada e o sistema operacional específicos. Por exemplo, para obter entradas, podemos ter que especificar o arquivo ou dispositivo a ser usado como origem, assim como o endereço e o tamanho do buffer de memória para o qual a entrada deve ser lida. É claro que o dispositivo ou arquivo e o tamanho podem estar implícitos na chamada.

Três métodos gerais são usados para passar parâmetros ao sistema operacional. A abordagem mais simples é passar os parâmetros em registradores. Em alguns casos, no entanto, pode haver mais parâmetros do que registradores. Nesses casos, os parâmetros são, em geral, armazenados em um bloco, ou tabela, na memória, e o endereço do bloco é passado como parâmetro em um registrador (Figura 2.7). Essa é a abordagem adotada pelo Linux e o Solaris. Os parâmetros também podem ser colocados ou incluídos na pilha pelo programa e extraídos da pilha pelo sistema operacional. Alguns sistemas operacionais preferem o método do bloco ou da pilha porque essas abordagens não limitam a quantidade ou o tamanho dos parâmetros que estão sendo passados.

## **2.4 Tipos de Chamadas de Sistema**

As chamadas de sistema podem ser agrupadas, grosso modo, em seis categorias principais: controle de processos, manipulação de arquivos, manipulação de dispositivos, manutenção de informações, comunicações e proteção. Nas Seções 2.4.1 a 2.4.6, discutimos brevemente os tipos de chamadas de sistema que podem ser fornecidos por um sistema operacional. A maioria dessas chamadas de sistema dá suporte a, ou é suportada por, conceitos e funções que são discutidos em capítulos posteriores. A Figura 2.8 resume os tipos de chamadas de sistema normalmente fornecidos por um sistema operacional. Como mencionado neste texto, quase sempre nos referimos às chamadas de sistema usando nomes genéricos. No decorrer do texto, no entanto, fornecemos exemplos de contrapartidas reais às chamadas de sistemas do Windows, UNIX e Linux.

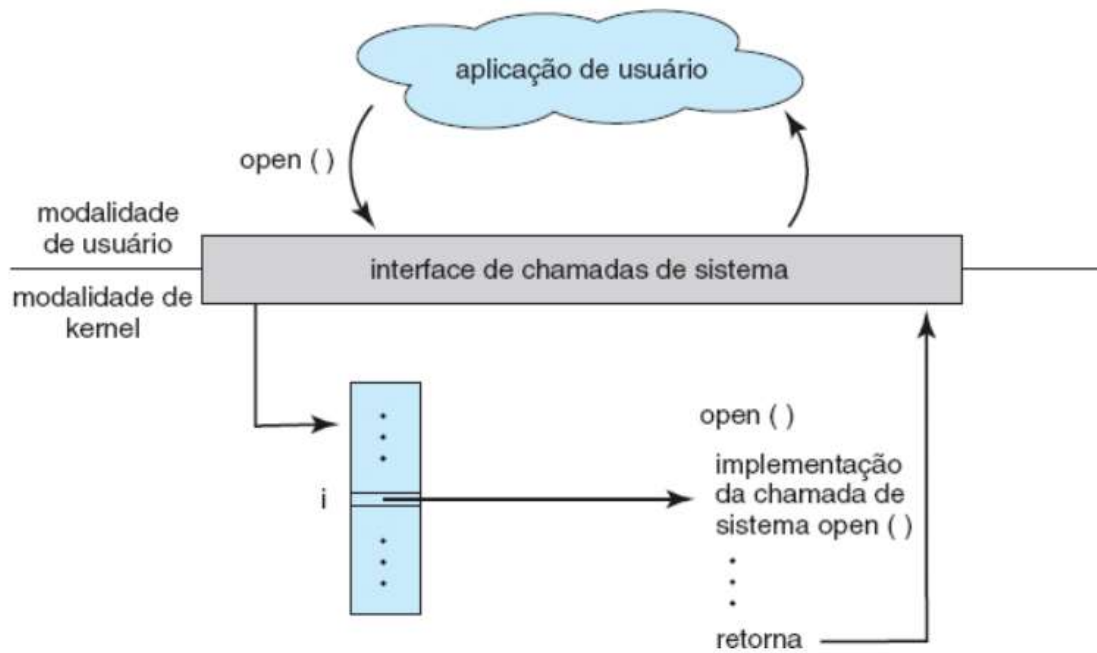
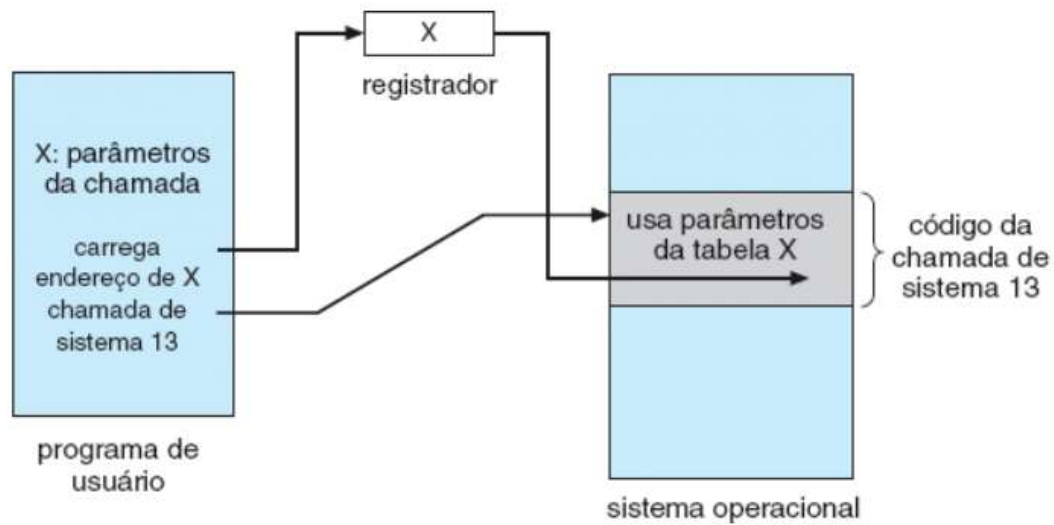


Figura 2.7 Passagem de parâmetros como uma tabela.



- Controle de processos
  - °encerrar, abortar
  - °carregar, executar
  - °criar processo, encerrar processo
  - °obter atributos do processo, definir atributos do processo
  - °esperar hora
  - °esperar evento, sinalizar evento
  - °alocar e liberar memória

- Gerenciamento de arquivos

- °criar arquivo, excluir arquivo

- °abrir, fechar

- °ler, gravar, reposicionar

- °obter atributos do arquivo, definir atributos do arquivo

- Gerenciamento de dispositivos

- °solicitar dispositivo, liberar dispositivo

- °ler, gravar, reposicionar

- °obter atributos do dispositivo, definir atributos do dispositivo

- °conectar ou desconectar dispositivos logicamente

- Manutenção de informações

- °obter a hora ou a data, definir a hora ou a data

- °obter dados do sistema, definir dados do sistema

- °obter atributos do processo, arquivo ou dispositivo

- °definir atributos do processo, arquivo ou dispositivo

- Comunicações

- °criar, excluir conexão de comunicações

- °enviar, receber mensagens

- °transferir informações de status

- °conectar ou desconectar dispositivos remotos

### 2.4.1 Controle de Processos

Um programa em execução precisa ser capaz de interromper sua operação, normal [end ( )] ou anormalmente [abort ( )]. Se uma chamada de sistema é feita para encerrar, anormalmente, o programa em execução corrente ou se o programa encontra um problema e causa uma exceção por erro, pode ocorrer um despejo da memória e a geração de uma mensagem de erro. O despejo é gravado em disco, podendo ser examinado por um depurador — um programa do sistema projetado para ajudar o programador a encontrar e corrigir erros ou bugs — para que seja determinada a causa do problema. Sob circunstâncias normais ou anormais, o sistema operacional deve transferir o controle ao interpretador de comandos invocador. O interpretador lê, então, o próximo comando. Em um sistema interativo, o interpretador de comandos simplesmente passa ao próximo comando; assume-se que o usuário emitirá um comando apropriado para responder a qualquer erro. Em um sistema de GUIs, uma janela pop-up pode alertar o usuário sobre o erro e solicitar orientação. Em um sistema batch, o interpretador de comandos usualmente encerra o job inteiro e continua com o próximo job. Alguns sistemas podem permitir ações de recuperação especiais em caso de erro. Se o programa descobrir um erro em sua entrada e quiser encerrar anormalmente, também pode querer definir um nível de erro. Erros mais graves podem ser indicados por um parâmetro de erro de nível mais alto. Assim, é possível combinar o encerramento normal e o anormal definindo um encerramento normal como um erro de nível 0. O interpretador de comandos ou o programa seguinte pode usar esse nível de erro para determinar, automaticamente, a próxima ação.

Um processo ou job executando um programa pode querer carregar (load) e executar (execute) outro programa. Esse recurso permite que o interpretador de comandos execute um programa a partir de um comando de usuário, um clique no mouse ou um comando batch, por exemplo. Uma questão interessante é para onde retornar o controle quando o programa carregado terminar. Essa questão está relacionada com se o programa existente foi perdido, salvo ou liberado para continuar a execução concorrentemente com o novo programa.

#### EXEMPLOS DE CHAMADAS DE SISTEMA DO WINDOWS E UNIX

EXEMPLOS DE CHAMADAS DE SISTEMA DO WINDOWS E UNIX		
	Windows	Unix
Controle de Processos	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
Manipulação de Arquivos	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Manipulação de Dispositivos	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Manutenção de Informações	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Comunicações	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shm_open() mmap()
Proteção	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Se o controle retorna ao programa existente quando o novo programa é encerrado, devemos salvar a imagem de memória do programa existente; assim, criamos efetivamente um mecanismo para que um programa chame outro programa. Se os dois programas continuarem concorrentemente, teremos criado um novo job ou processo a ser multiprogramado. Geralmente, há uma chamada de sistema especificamente para essa finalidade [`create_process ( )` ou `submit_job ( )`].

Se criamos um novo job ou processo, ou talvez até mesmo um conjunto de jobs ou processos, devemos ser capazes de controlar sua execução. Esse controle requer a habilidade de determinar e redefinir os atributos de um job ou processo, inclusive a prioridade do job, o tempo de execução máximo permitido, e assim por diante [`get_process_attributes ( )` e `set_process_attributes ( )`]. Também podemos querer encerrar um job ou processo que criamos [`terminate_process ( )`], se acharmos que ele está incorreto ou não é mais necessário.

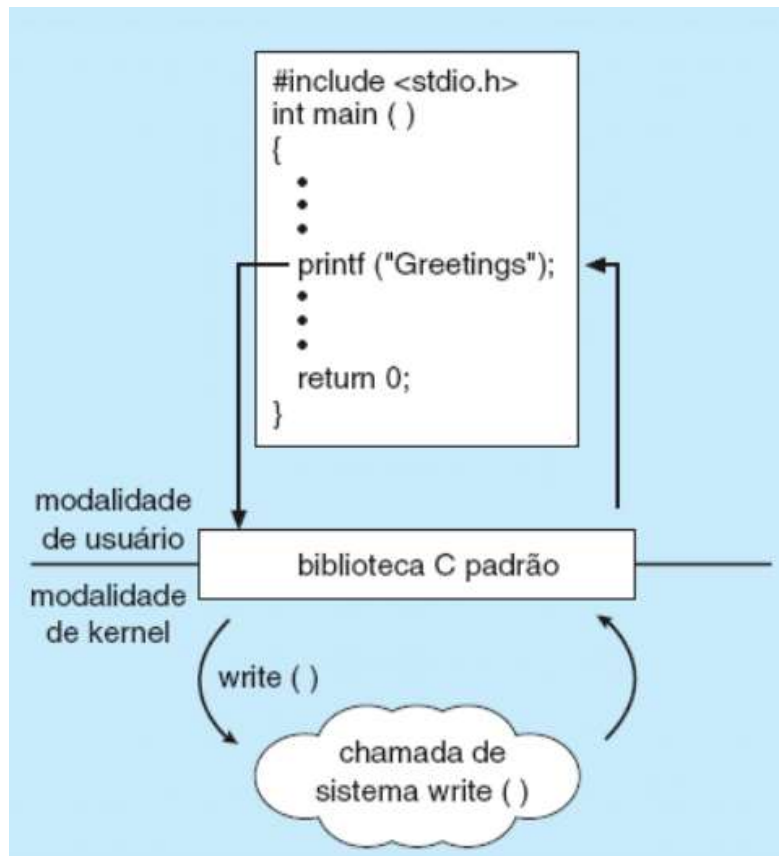
Tendo criado novos jobs ou processos, podemos ter de esperar o término de sua execução. Podemos desejar esperar por um determinado período de tempo [`wait_time ( )`]. O mais provável é que esperemos a ocorrência de um evento específico [`wait_event ( )`]. Os jobs ou processos devem, então, sinalizar quando esse evento tiver ocorrido [`signal_event ( )`].

Com bastante frequência, dois ou mais processos podem compartilhar dados. Para assegurar a integridade dos dados que estão sendo compartilhados, os sistemas operacionais costumam fornecer chamadas de sistema que permitem que um processo tranque (imponha um lock a) dados compartilhados. Assim, outros processos não podem acessar os dados até que o lock seja liberado. Normalmente, essas chamadas de sistema incluem `acquire_lock ( )` e `release_lock ( )`. Esses tipos de chamadas de sistema, que lidam com a coordenação de processos concorrentes, são discutidos em grandes detalhes no Capítulo 5.

Há tantas nuances e variações no controle de processos e jobs, que usamos, a seguir, dois exemplos — um envolvendo um sistema monotarefa e o outro, um sistema multitarefa — para esclarecer esses conceitos. O sistema operacional MS-DOS é um exemplo de sistema monotarefa. Ele tem um interpretador de comandos que é invocado quando o computador é iniciado [Figura 2.9(a)]. Como o MS-DOS é monotarefa, ele usa um método simples para executar um programa e não cria um novo processo. Ele carrega o programa na memória, gravando sobre grande parte de si próprio, para dar ao programa o máximo de memória possível [Figura 2.9(b)]. Em seguida, posiciona o ponteiro de instruções para a primeira instrução do programa. O programa é, então, executado, e então ou um erro causa uma exceção, ou o programa executa uma chamada de sistema para ser encerrado. De uma forma ou de outra, o código de erro é salvo na memória do sistema para uso posterior. Após essa ação, a pequena parte do interpretador de comandos que não foi sobreposta retoma a execução. Sua primeira tarefa é recarregar o resto do interpretador de comandos a partir do disco. Em seguida, o interpretador de comandos torna o código de erro anterior disponível para o usuário ou para o próximo programa.

## EXEMPLO DE BIBLIOTECA C PADRÃO

A biblioteca C padrão fornece parte da interface de chamadas de sistema de muitas versões do UNIX e Linux. Como exemplo, suponha que um programa em C invoque o comando `printf ( )`. A biblioteca C intercepta essa chamada e invoca a chamada (ou chamadas) de sistema necessária no sistema operacional — nesse exemplo, a chamada de sistema `write ( )`. A biblioteca C toma o valor retornado por `write ( )` e o passa de volta ao programa do usuário. Isso é mostrado abaixo:



O FreeBSD (derivado do UNIX de Berkeley) é um exemplo de sistema multitarefa. Quando um usuário faz login no sistema, o shell que ele escolheu é executado. Esse shell é semelhante ao shell do MS-DOS, já que aceita os comandos e executa os programas que o usuário solicita. No entanto, como o FreeBSD é um sistema multitarefa, o interpretador de comandos pode continuar em operação enquanto outro programa é executado (Figura 2.10). Para iniciar um novo processo, o shell executa uma chamada de sistema `fork ( )`. Em seguida, o programa selecionado é carregado na memória por meio de uma chamada de sistema `exec ( )` e é executado. Dependendo da maneira como o comando foi emitido, o shell espera que o processo termine ou o executa “em background”. No último caso, o shell solicita imediatamente outro comando. Quando um processo está sendo executado em background, ele não pode receber entradas diretamente do teclado porque o shell está usando esse recurso. Portanto, o I/O é executado por meio de arquivos ou de uma GUI. Enquanto isso, o usuário pode solicitar ao shell que execute outros programas, monitore o progresso do processo em execução, altere a prioridade desse programa, e assim por diante. Quando o processo é concluído, ele executa uma



chamada de sistema `exit ( )` para ser encerrado, retornando ao processo que o invocou, o código de status 0 ou um código de erro diferente de zero. Esse código de status ou de erro fica, então, disponível para o shell ou para outros programas. Os processos são discutidos no Capítulo 3 com um exemplo de programa que usa as chamadas de sistema `fork ( )` e `exec ( )`.

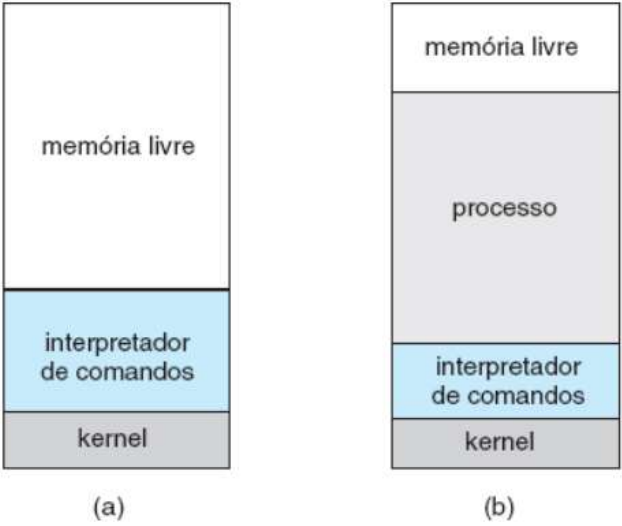


Figura 2.9 Execução do MS-DOS. (a) Na inicialização do sistema. (b) Executando um programa.



Figura 2.10 O FreeBSD executando múltiplos programas.

### 2.4.2 Gerenciamento de Arquivos

O sistema de arquivos é discutido com mais detalhes nos Capítulos 11 e 12. Podemos, no entanto, identificar várias chamadas de sistema comuns que lidam com arquivos. Primeiro, precisamos ser capazes de criar [`create ( )`] e excluir [`delete ( )`] arquivos. As duas chamadas de sistema precisam do nome do arquivo e, talvez, de alguns de seus atributos. Uma vez que o arquivo seja criado, precisamos abri-lo [`open ( )`] e usá-lo. Também podemos ler [`read ( )`], gravar [`write ( )`] ou reposicionar [`reposition ( )`] o arquivo (retornar ao início ou saltar para o fim do

arquivo, por exemplo). Para concluir, temos de fechar [close ( )] o arquivo, indicando que ele não está mais sendo usado.

Podemos precisar desses mesmos conjuntos de operações para diretórios se tivermos uma estrutura de diretórios para a organização de arquivos no sistema de arquivos. Além disso, tanto para arquivos como para diretórios, temos que ser capazes de determinar o valor de diversos atributos e de, talvez, redefini-los se necessário. Os atributos do arquivo incluem seu nome, seu tipo, códigos de proteção, informações de contabilização, e assim por diante. Pelo menos, duas chamadas de sistema, `get_file_attributes ( )` e `set_file_attributes ( )`, são requeridas para essa função. Alguns sistemas operacionais fornecem muitas outras chamadas, como as chamadas de movimentação [move ( )] e cópia [copy ( )] de arquivos. Outros podem fornecer uma API que execute essas operações usando código e chamadas de sistema diferentes; outros podem fornecer programas de sistema para executar essas tarefas. Se os programas de sistema puderem ser chamados por outros programas, cada um deles poderá ser considerado uma API por outros programas do sistema.

### 2.4.3 Gerenciamento de Dispositivos

Um processo pode precisar de vários recursos para ser executado — memória principal, drives de disco, acesso a arquivos, e assim por diante. Se os recursos estão disponíveis, eles podem ser cedidos e o controle pode ser retornado ao processo do usuário. Caso contrário, o processo terá de esperar até que recursos suficientes estejam disponíveis. Os diversos recursos controlados pelo sistema operacional podem ser considerados como dispositivos. Alguns desses dispositivos são dispositivos físicos (por exemplo, drives de disco), enquanto outros podem ser considerados dispositivos abstratos ou virtuais (por exemplo, arquivos). Um sistema com múltiplos usuários pode requerer que, primeiro, seja feita a solicitação [request ( )] de um dispositivo, para assegurar seu uso exclusivo. Quando o dispositivo não é mais necessário, ele pode ser liberado [release ( )]. Essas funções são semelhantes às chamadas de sistema `open ( )` e `close ( )` para arquivos. Outros sistemas operacionais permitem acesso não gerenciado a dispositivos. Nesse caso, o perigo é o potencial para a disputa por dispositivos e, talvez, para deadlocks que são descritos no Capítulo 7.

Uma vez que o dispositivo tenha sido solicitado (e alocado para nosso uso), podemos lê-lo [read ( )], gravá-lo [write ( )] e (possivelmente) reposicioná-lo [reposition ( )], da mesma forma que podemos fazer com os arquivos. Na verdade, a semelhança entre dispositivos de I/O e arquivos é tão grande que muitos sistemas operacionais, inclusive o UNIX, fundem os dois em uma estrutura combinada arquivo-dispositivo. Nesse caso, um conjunto de chamadas de sistema é usado tanto em arquivos quanto em dispositivos. Em algumas situações, dispositivos de I/O são identificados por nomes de arquivo especiais, localização de diretórios ou atributos de arquivo. A interface de usuário também pode fazer arquivos e dispositivos parecerem semelhantes, ainda que as chamadas de sistema subjacentes sejam diferentes. Esse é outro exemplo das muitas decisões de projeto que fazem parte da construção de um sistema operacional e da interface de usuário.

#### 2.4.4 Manutenção de Informações

Muitas chamadas de sistema existem, simplesmente, para fins de transferência de informações entre o programa do usuário e o sistema operacional. Por exemplo, a maioria dos sistemas tem uma chamada de sistema para retornar a hora [time ( )] e a data [date ( )] correntes. Outras chamadas de sistema podem retornar informações sobre o sistema, tais como o número de usuários correntes, o número de versão do sistema operacional, o montante de espaço livre na memória ou em disco, e assim por diante. Outro conjunto de chamadas de sistema é útil na depuração de um programa. Muitos sistemas fornecem chamadas de sistema para o despejo [dump ( )] da memória. Esse recurso é útil na depuração. Um rastreamento [trace ( )] de programa lista cada chamada de sistema enquanto ela é executada. Até mesmo microprocessadores fornecem uma modalidade de CPU conhecida como passo único, em que uma exceção é executada pela CPU após cada instrução. Geralmente, a exceção é capturada por um depurador.

Muitos sistemas operacionais fornecem um perfil de tempo de um programa que indica por quanto tempo ele é executado em uma localização específica ou conjunto de localidades. Um perfil de tempo requer um recurso de rastreamento ou interrupções periódicas de timer. A cada ocorrência de interrupção do timer, o valor do contador do programa é registrado. Com interrupções suficientemente frequentes do timer, um cenário estatístico do tempo gasto em várias partes do programa pode ser obtido.

Além disso, o sistema operacional mantém informações sobre todos os seus processos e chamadas de sistema utilizadas para acessar essas informações. Geralmente, chamadas também são usadas na redefinição das informações dos processos [get\_process\_attributes ( ) e set\_process\_attributes ( )]. Na Seção 3.1.3, discutimos as informações que costumam ser mantidas.

#### 2.4.5 Comunicação

Há dois modelos comuns de comunicação entre processos: o modelo de transmissão de mensagens e o modelo de memória compartilhada. No modelo de transmissão de mensagens, os processos em comunicação trocam mensagens uns com os outros para transferir informações. As mensagens podem ser trocadas entre os processos, direta ou indiretamente, por meio de uma caixa de correio comum. Antes que a comunicação possa ocorrer, uma conexão deve ser aberta. O nome do outro interlocutor deve ser conhecido, seja ele outro processo no mesmo sistema ou um processo em outro computador conectado por uma rede de comunicação. Cada computador de uma rede tem um nome de hospedeiro pelo qual é conhecido normalmente. O hospedeiro também tem um identificador de rede que pode ser um endereço IP. Da mesma forma, cada processo tem um nome de processo, e esse nome é traduzido para um identificador pelo qual o sistema operacional pode referenciar o processo. As chamadas de sistema get\_hostid ( ) e get\_processid ( ) fazem essa tradução. Os identificadores são, então, passados para as chamadas de uso geral open ( ) e close ( ), fornecidas pelo sistema de arquivos, ou para as chamadas específicas open\_connection ( ) e close\_connection ( ), dependendo do modelo de comunicação do sistema. Usualmente, o

processo receptor deve dar sua permissão para que a comunicação possa ocorrer, com uma chamada `accept_connection ( )`. A maioria dos processos que recebem conexões é de daemons de uso específico que são programas de sistema fornecidos para esse fim. Eles executam uma chamada `wait_for_connection ( )` e são ativados quando uma conexão é estabelecida. Em seguida, a fonte da comunicação, conhecida como cliente, e o daemon receptor, conhecido como servidor, trocam mensagens usando chamadas de sistema `read_message ( )` e `write_message ( )`. A chamada `close_connection ( )` encerra a comunicação.

No modelo de memória compartilhada, os processos usam chamadas de sistema `shared_memory_create ( )` e `shared_memory_attach ( )` para criar e obter acesso a regiões da memória ocupadas por outros processos. Lembre-se de que, normalmente, o sistema operacional tenta impedir que um processo acesse a memória de outro processo. A memória compartilhada exige que dois ou mais processos concordem com a remoção dessa restrição. Então, eles podem trocar informações lendo dados das áreas compartilhadas e gravando dados nessas áreas. A forma dos dados é determinada pelos processos e não fica sob controle do sistema operacional. Os processos também são responsáveis por garantir que não estão realizando gravações na mesma localização, simultaneamente. Tais mecanismos são discutidos no Capítulo 5. No Capítulo 4, examinamos uma variação do esquema de processos — os threads — em que a memória é compartilhada por default. Os dois modelos que acabamos de discutir são comuns nos sistemas operacionais, e a maioria dos sistemas implementa ambos. A troca de mensagens é útil na transmissão de quantidades menores de dados porque não é necessário evitar conflitos. Também é mais fácil de implementar do que a memória compartilhada para a comunicação entre computadores. A memória compartilhada proporciona velocidade máxima e conveniência na comunicação, já que pode ocorrer na velocidade de transferência da memória quando se dá dentro de um computador. No entanto, existem problemas nas áreas de proteção e sincronização entre os processos que compartilham memória.

#### **2.4.6 Proteção**

A proteção proporciona um mecanismo para o controle de acesso aos recursos fornecidos por um sistema de computação. Historicamente, a proteção era uma preocupação somente em sistemas de computação multiprogramados com vários usuários. No entanto, com o advento das redes e da Internet, todos os sistemas de computação, de servidores a dispositivos móveis, devem se preocupar com a proteção. Normalmente, as chamadas de sistema que fornecem proteção incluem `set_permission ( )` e `get_permission ( )`, que manipulam as definições de permissões para recursos, tais como arquivos e discos. As chamadas de sistema `allow_user ( )` e `deny_user ( )` especificam se determinados usuários podem — ou não podem — obter acesso a certos recursos.

Abordamos a proteção no Capítulo 14 e o tópico muito mais abrangente da segurança no Capítulo 15.

## 2.5 Programas de Sistema

Outra característica de um sistema moderno é seu conjunto de programas de sistema. Volte à Figura 1.1, que mostra a hierarquia lógica do computador. No nível mais baixo, está o hardware. Em seguida, está o sistema operacional, depois os programas de sistema e, finalmente, os programas de aplicação. Os programas de sistema, também conhecidos como utilitários de sistema, fornecem um ambiente conveniente para o desenvolvimento e a execução de programas. Alguns deles são, simplesmente, interfaces de usuário para chamadas de sistema. Outros são, consideravelmente, mais complexos. Eles podem ser divididos nas categorias a seguir:

- **Gerenciamento de arquivos.** Esses programas criam, excluem, copiam, renomeiam, imprimem, descarregam, listam e, geralmente, manipulam arquivos e diretórios.

- **Informações de status.** Alguns programas simplesmente solicitam ao sistema a data, a hora, o montante disponível de memória ou de espaço em disco, a quantidade de usuários ou informações de status semelhantes. Outros são mais complexos, fornecendo informações detalhadas sobre desempenho, registro em log e depuração. Normalmente, esses programas formatam e exibem a saída no terminal ou em outros dispositivos ou arquivos de saída, ou a exibem em uma janela da GUI. Alguns sistemas também dão suporte a um repositório de registros que é usado para armazenar e recuperar informações de configuração.

- **Modificação de arquivos.** Vários editores de texto podem estar disponíveis para criar e modificar o conteúdo de arquivos armazenados em disco ou em outros dispositivos de armazenamento. Também podem existir comandos especiais para a busca de conteúdos de arquivos ou a execução de alterações no texto.

- **Suporte a linguagens de programação.** Compiladores, montadores, depuradores e interpretadores de linguagens de programação comuns (como C, C++, Java e PERL) são, com frequência, fornecidos com o sistema operacional ou estão disponíveis para download em separado.

- **Carga e execução de programas.** Uma vez que um programa seja montado ou compilado, ele deve ser carregado na memória para ser executado. O sistema pode fornecer carregadores absolutos, carregadores relocáveis, linkage editors e carregadores de overlay. Sistemas de depuração para linguagens de alto nível ou linguagem de máquina também são necessários.

- **Comunicações.** Esses programas fornecem o mecanismo para a criação de conexões virtuais entre processos, usuários e sistemas de computação. Eles permitem que os usuários enviem mensagens para a tela uns dos outros, naveguem em páginas da web, enviem e-mails, façam login remotamente ou transfiram arquivos de uma máquina para outra.

- **Serviços de background.** Todos os sistemas de uso geral têm métodos que lançam certos processos de programas de sistema em tempo de inicialização. Alguns desses processos terminam após a conclusão de suas tarefas, enquanto outros continuam a ser executados até que o sistema seja interrompido. Processos de programas de sistema constantemente executados são conhecidos como serviços, subsistemas, ou daemons. Um exemplo é o daemon de rede discutido na Seção 2.4.5. Naquele exemplo, um sistema precisava de um serviço para escutar conexões de rede e conectar essas solicitações aos processos corretos. Outros exemplos incluem schedulers de processos que iniciam processos de acordo com um schedule especificado, serviços de monitoração de erros do sistema e servidores de impressão. Sistemas típicos têm vários daemons. Além disso, sistemas operacionais que executam atividades importantes no contexto do usuário, em vez de no contexto do kernel, podem usar daemons para executar essas atividades.

Além dos programas de sistema, a maioria dos sistemas operacionais vem com programas que são úteis na resolução de problemas comuns ou na execução de operações comuns. Esses programas de aplicação incluem navegadores web, processadores e formatadores de texto, planilhas, sistemas de bancos de dados, compiladores, pacotes de plotagem e análise estatística e jogos.

A visão que a maioria dos usuários tem do sistema operacional é definida pelos programas de aplicação e de sistema e não pelas chamadas de sistema reais. Considere o PC de um usuário. Quando o computador de um usuário está executando o sistema operacional Mac OS X, o usuário pode ver a GUI fornecendo uma interface baseada em mouse e janelas. Alternativamente, ou até mesmo em uma das janelas, o usuário pode ter um shell UNIX de linha de comando. Os dois usam o mesmo conjunto de chamadas de sistema, mas elas têm aparências diferentes e agem de maneira diferente. Para confundir mais a visão do usuário, considere a inicialização dual do Windows a partir do Mac OS X. Agora, o mesmo usuário no mesmo hardware tem duas interfaces totalmente diferentes e dois conjuntos de aplicações utilizando os mesmos recursos físicos. Portanto, no mesmo hardware um usuário pode ser exposto a múltiplas interfaces de usuário, sequencial ou concorrentemente.

## **2.6 Projeto e Implementação do Sistema Operacional**

Nesta seção, discutimos problemas que surgem no projeto e implementação de um sistema operacional. Não existem, obviamente, soluções definitivas para esses problemas, mas existem abordagens que se mostraram bem-sucedidas.

### **2.6.1 Objetivos do Projeto**

O primeiro problema ao projetar um sistema é a definição de objetivos e especificações. Em um nível mais alto, o projeto do sistema será afetado pela escolha do hardware e do tipo de sistema: batch, tempo compartilhado, monousuário, multiusuário, distribuído, tempo real ou uso geral.

Além desse nível mais alto de projeto, os requisitos podem ser muito mais difíceis de especificar. No entanto, esses requisitos podem ser divididos em dois grupos básicos: objetivos do usuário e objetivos do sistema. Usuários desejam certas propriedades óbvias em um sistema. O sistema deve ser conveniente para usar, fácil de aprender e utilizar, confiável, seguro e veloz. Naturalmente, essas especificações não são particularmente úteis ao projeto do sistema, já que não há um consenso geral sobre como alcançá-las.

Um conjunto semelhante de requisitos pode ser definido pelas pessoas responsáveis por projetar, criar, manter e operar o sistema. O sistema deve ser fácil de projetar, implementar e manter; e deve ser flexível, confiável, sem erros e eficiente. Novamente, esses requisitos são vagos e podem ser interpretados de várias maneiras.

Resumindo, não há uma solução única para o problema de definição dos requisitos de um sistema operacional. A grande quantidade de sistemas existente mostra que diferentes requisitos podem resultar em uma grande variedade de soluções para ambientes diferentes. Por exemplo, os requisitos do VxWorks, um sistema operacional de tempo real para sistemas embutidos, devem ter sido substancialmente diferentes dos definidos para o MVS, sistema operacional multiusuário e multiacesso, de grande porte, para mainframes IBM.

A especificação e o projeto de um sistema operacional é uma tarefa altamente criativa. Embora nenhum livro possa dizer a você como fazê-lo, foram desenvolvidos princípios gerais no campo da engenharia de software e passamos, agora, à discussão de alguns desses princípios.

## **2.6.2 Mecanismos e Políticas**

Um princípio importante é a separação entre política e mecanismo. Os mecanismos determinam como fazer algo; as políticas determinam o que será feito. Por exemplo, o construtor timer (consulte a Seção 1.5.2) é um mecanismo que assegura a proteção da CPU, mas a decisão de por quanto tempo ele deve ser posicionado, para um usuário específico, é uma decisão política.

A separação entre política e mecanismo é importante para a flexibilidade. As políticas podem mudar para locais diferentes ou com o passar do tempo. Na pior das hipóteses, cada mudança na política demandaria uma mudança no mecanismo subjacente. Um mecanismo genérico, insensível a mudanças na política, seria mais desejável. Uma mudança na política demandaria, então, a redefinição de somente certos parâmetros do sistema. Por exemplo, considere um mecanismo para dar prioridade a determinados tipos de programas sobre outros. Se o mecanismo for apropriadamente separado da política, ele poderá ser usado tanto para suportar a decisão política de que programas I/O-intensivos devem ter prioridade sobre os programas CPU-intensivos, como para dar suporte à política oposta. Sistemas operacionais baseados em microkernel (Seção 2.7.3) levam a separação entre mecanismo e política a um extremo, implementando um conjunto básico de blocos de construção primitivos. Esses blocos são quase independentes de política, permitindo que mecanismos e políticas mais avançados sejam adicionados por meio de módulos de kernel criados pelo usuário ou por meio dos próprios programas dos usuários. Como exemplo, considere a história do UNIX. Inicialmente, ele tinha um scheduler de tempo compartilhado. Na última versão do Solaris, o scheduling é controlado por tabelas carregáveis. Dependendo da tabela correntemente carregada, o sistema pode ser de tempo compartilhado, processamento batch, tempo real, compartilhamento justo, ou qualquer combinação. A definição do mecanismo de scheduling como de uso geral permite que amplas alterações sejam feitas na política com um único comando `load-new-table`. No outro

extremo, está um sistema como o Windows, em que tanto o mecanismo quanto a política são codificados no sistema para impor uma aparência global. Todas as aplicações têm interfaces semelhantes porque a própria interface é construída nas bibliotecas do kernel e do sistema. O sistema operacional Mac OS X tem funcionalidade semelhante.

As decisões políticas são importantes para qualquer alocação de recursos. Sempre que é necessário decidir se um recurso deve ou não ser alocado, uma decisão política deve ser tomada. Sempre que o problema é como em vez de o que, um mecanismo é que deve ser determinado.

### **2.6.3 Implementação**

Uma vez que o sistema operacional tenha sido projetado, ele deve ser implementado. Já que os sistemas operacionais são conjuntos de muitos programas, escritos por muitas pessoas durante um longo período de tempo, é difícil fazer afirmações gerais sobre como eles são implementados.

Os sistemas operacionais iniciais eram escritos em linguagem de montagem. Atualmente, embora alguns sistemas operacionais ainda sejam escritos em linguagem de montagem, a maioria é escrita em uma linguagem de mais alto nível como C ou de nível ainda mais alto como C++. Na verdade, um sistema operacional pode ser escrito em mais de uma linguagem. Os níveis mais baixos do kernel podem ser escritos em linguagem de montagem. As rotinas de nível mais alto podem ser escritas em C e os programas de sistema podem ser em C ou C++, em linguagens de script interpretadas como PERL ou Python, ou em scripts de shell. Na verdade, uma determinada distribuição do Linux provavelmente inclui programas escritos em todas essas linguagens.

O primeiro sistema não escrito em linguagem de montagem foi, provavelmente, o Master Control Program (MCP) para computadores Burroughs. O MCP foi escrito em uma variante de ALGOL. O MULTICS, desenvolvido no MIT, foi escrito, principalmente, na linguagem de programação de sistemas PL/1. Os kernels dos sistemas operacionais Linux e Windows, em sua maior parte, foram escritos em C, embora haja algumas pequenas seções de código de montagem para drivers de dispositivos e para salvar e restaurar o estado de registradores.

As vantagens do uso de uma linguagem de mais alto nível ou, pelo menos, de uma linguagem de implementação de sistemas, para implementar sistemas operacionais, são as mesmas obtidas quando a linguagem é usada para programas aplicativos: o código pode ser escrito mais rapidamente, é mais compacto e mais fácil de entender e depurar. Além disso, avanços na tecnologia dos compiladores melhoram o código gerado para o sistema operacional inteiro por meio de uma simples recompilação. Para concluir, um sistema operacional é muito mais fácil de portar — transportar para algum outro hardware — quando é escrito em uma linguagem de mais alto nível. Por exemplo, o MS-DOS foi escrito na linguagem de montagem Intel 8088. Como resultado, ele é executado, nativamente, apenas na família de CPUs Intel x86. (Observe que, embora o MS-DOS seja executado nativamente apenas no Intel x86, emuladores do conjunto de instruções do x86 permitem que o sistema operacional seja executado em outras CPUs — porém mais lentamente e com maior uso de recursos. Como mencionamos no Capítulo 1, emuladores são programas que duplicam a funcionalidade de um sistema em outro sistema.) O sistema operacional Linux, por outro lado, é escrito quase todo em C e está disponível nativamente em várias CPUs diferentes, inclusive no Intel x86, Oracle SPARC e IBM PowerPC.



Talvez as únicas desvantagens da implementação de um sistema operacional em uma linguagem de mais alto nível sejam a diminuição da velocidade e o aumento dos requisitos de armazenamento. No entanto, isso não é mais um grande problema nos sistemas atuais. Embora um programador especialista em linguagem de montagem possa produzir rotinas pequenas e eficientes, para programas grandes um compilador moderno pode executar análises complexas e aplicar otimizações sofisticadas que produzem excelente código. Os processadores modernos têm fortes interligações e várias unidades funcionais que podem manipular os detalhes de dependências complexas muito mais facilmente do que a mente humana.

Como ocorre em outros sistemas, é mais provável que os principais avanços no desempenho dos sistemas operacionais resultem de melhores estruturas de dados e algoritmos e não de um excelente código em linguagem de montagem. Além disso, embora os sistemas operacionais sejam grandes, apenas uma pequena parte do código é crítica para o alto desempenho; o manipulador de interrupções, o gerenciador de I/O, o gerenciador de memória e o scheduler da CPU são, provavelmente, as rotinas mais críticas. Depois que o sistema é escrito e está funcionando corretamente, rotinas que representem gargalos podem ser identificadas e substituídas por equivalentes em linguagem de montagem.

## **2.7 Estrutura do Sistema Operacional**

Um sistema tão grande e complexo, como um sistema operacional moderno, deve ser construído cuidadosamente para funcionar de maneira apropriada e ser facilmente modificável. Uma abordagem comum é a divisão da tarefa em componentes pequenos, ou módulos, em vez da criação de um sistema monolítico. Cada um desses módulos deve ser uma parte bem definida do sistema, com entradas, saídas e funções cuidadosamente estabelecidas. Já discutimos brevemente, no Capítulo 1, os componentes comuns dos sistemas operacionais. Nesta seção, discutimos como esses componentes são interconectados e combinados em um kernel.

### **2.7.1 Estrutura Simples**

Muitos sistemas operacionais não têm estruturas bem definidas. Frequentemente, tais sistemas começam como sistemas pequenos, simples e limitados e, então, crescem para além de seu escopo original. O MS-DOS é um exemplo desse tipo de sistema. Ele foi originalmente projetado e implementado por algumas pessoas que não tinham ideia de que se tornaria tão popular. Ele foi escrito para fornecer o máximo de funcionalidade no menor espaço e, portanto, não foi dividido cuidadosamente em módulos. A Figura 2.11 mostra sua estrutura.

No MS-DOS, as interfaces e níveis de funcionalidade não estão bem separados. Por exemplo, programas aplicativos podem acessar as rotinas básicas de I/O para gravar diretamente em tela e drives de disco. Tal liberdade deixa o MS-DOS vulnerável a programas errados (ou maliciosos), fazendo com que o sistema inteiro caia quando programas de usuário falham. Naturalmente, o MS-DOS também ficou limitado em razão do hardware de sua época. Já que o Intel 8088, para o qual ele foi escrito, não fornece modalidade dual e proteção de hardware, os projetistas do MS-DOS não tinham outra opção além de deixar o hardware básico acessível.

Outro exemplo de estrutura limitada é o sistema operacional UNIX original. Como o MS-DOS, o UNIX, inicialmente, foi limitado pela funcionalidade do hardware. Ele é composto por duas partes separadas: o kernel e os programas de sistema. Por sua vez, o kernel é separado em uma série de interfaces e drivers de dispositivos que foram sendo adicionados e expandidos com o passar dos anos, conforme o UNIX evoluía. Podemos considerar o sistema operacional UNIX tradicional como uma estrutura em camadas até certo ponto, como mostrado na Figura 2.12. Tudo que está abaixo da interface de chamadas de sistema e acima do hardware físico é o kernel. O kernel fornece o sistema de arquivos, o scheduling da CPU, o gerenciamento de memória e outras funções do sistema operacional por meio de chamadas de sistema. Tudo somado, isso corresponde a uma enorme quantidade de funcionalidades a serem combinadas em um único nível. Essa estrutura monolítica era difícil de implementar e manter. No entanto, apresentava uma vantagem peculiar para o desempenho: há muito pouco overhead na interface de chamadas de sistema ou na comunicação dentro do kernel. Ainda vemos evidências dessa estrutura simples e monolítica nos sistemas operacionais UNIX, Linux e Windows.

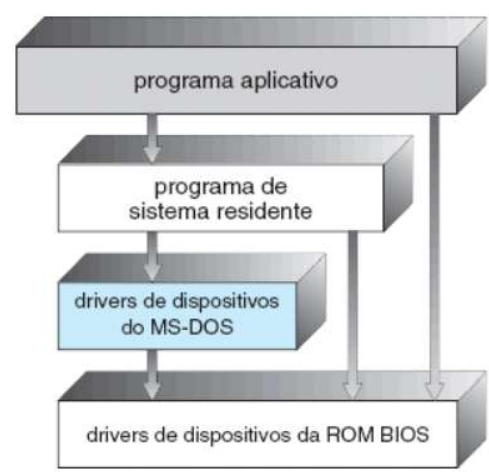


Figura 2.11 Estrutura em camadas do MS-DOS.



Figura 2.12 Estrutura tradicional de sistema UNIX.

### 2.7.2 Abordagem em Camadas

Com suporte de hardware apropriado, os sistemas operacionais podem ser divididos em partes que sejam menores e mais adequadas do que as permitidas pelos sistemas MS-DOS e UNIX originais. O sistema operacional pode, então, deter um controle muito maior sobre o computador e sobre as aplicações que fazem uso desse computador. Os implementadores têm mais liberdade para alterar os mecanismos internos do sistema e criar sistemas operacionais modulares. Em uma abordagem top-down, a funcionalidade e os recursos gerais são determinados e separados em componentes. A ocultação de informações também é importante porque deixa os programadores livres para implementar as rotinas de baixo nível como acharem melhor, contanto que a interface externa da rotina permaneça inalterada e que a rotina propriamente dita execute a tarefa anunciada.

Um sistema pode ser modularizado de várias maneiras. Um dos métodos é a abordagem em camadas, em que o sistema operacional é dividido em várias camadas (níveis). A camada inferior (camada 0) é o hardware; a camada mais alta (camada N) é a interface de usuário. Essa estrutura em camadas é mostrada na Figura 2.13.

A camada de um sistema operacional é a implementação de um objeto abstrato composto por dados e as operações que podem manipular esses dados. Uma camada típica de sistema operacional — digamos, a camada M — é composta por estruturas de dados e um conjunto de rotinas que podem ser invocadas por camadas de níveis mais altos. A camada M, por sua vez, pode invocar operações em camadas de níveis mais baixos.

A principal vantagem da abordagem em camadas é a simplicidade de construção e depuração. As camadas são selecionadas de modo que cada uma use funções (operações) e serviços somente de camadas de nível mais baixo. Essa abordagem simplifica a depuração e a verificação do sistema. A primeira camada pode ser depurada sem nenhuma preocupação com o resto do sistema porque, por definição, ela usa somente o hardware básico (que se supõe esteja correto) para implementar suas funções. Uma vez que a primeira camada seja depurada, seu funcionamento correto pode ser assumido enquanto a segunda camada é depurada, e assim por diante. Se é encontrado um erro durante a depuração de uma camada específica, ele deve estar nessa camada porque as camadas abaixo já estão depuradas. Portanto, o projeto e a implementação do sistema são simplificados.

Cada camada é implementada somente com as operações fornecidas por camadas de nível mais baixo. A camada não precisa saber como essas operações são implementadas; precisa saber apenas o que elas fazem. Portanto, cada camada oculta, das camadas de nível mais alto, a existência de certas estruturas de dados, operações e hardware.

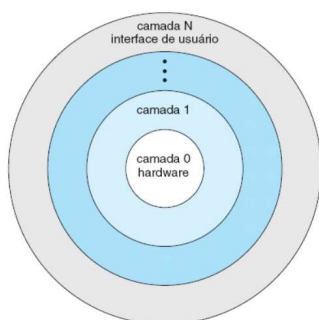


Figura 2.13 Um sistema operacional em camadas.

A principal dificuldade da abordagem em camadas envolve a definição apropriada das diversas camadas. Já que uma camada pode usar somente camadas de nível mais baixo, é necessário um planejamento cuidadoso. Por exemplo, o driver de dispositivos do backing store (espaço em disco usado por algoritmos de memória virtual) deve estar em um nível mais baixo do que as rotinas de gerenciamento da memória porque o gerenciamento da memória requer o uso do backing store.

Outros requisitos podem não ser tão óbvios. Normalmente, o driver do backing store fica acima do scheduler da CPU porque pode ter de esperar por I/O e a CPU pode sofrer um reschedule durante esse intervalo. No entanto, em um sistema grande, o scheduler da CPU pode ter mais informações sobre todos os processos ativos do que a memória pode armazenar. Portanto, essas informações podem ter que ser permutadas, inseridas na memória e dela extraídas, requerendo que a rotina do driver de backing store fique abaixo do scheduler da CPU.

Um problema final das implementações em camadas é que elas tendem a ser menos eficientes do que outras abordagens. Por exemplo, quando um programa de usuário executa uma operação de I/O, ele executa uma chamada de sistema que é interceptada para a camada de I/O que chama a camada de gerenciamento da memória que, por sua vez, chama a camada de scheduling da CPU que é, então, passada ao hardware. Em cada camada, os parâmetros podem ser modificados, dados podem ter de ser passados, e assim por diante. Cada camada adiciona overhead à chamada de sistema. O resultado final é uma chamada de sistema que demora mais do que em um sistema não estruturado em camadas.

Ultimamente, essas limitações têm causado alguma reação contra a estruturação em camadas. Menos camadas com mais funcionalidades estão sendo projetadas, fornecendo a maioria das vantagens do código modularizado, mas evitando os problemas de definição e interação de camadas.

### **2.7.3 Microkernels**

Já vimos que, conforme o UNIX se expandiu, o kernel tornou-se maior e mais difícil de gerenciar. Na metade dos anos 1980, pesquisadores da Universidade Carnegie Mellon desenvolveram um sistema operacional chamado Mach que modularizou o kernel usando a abordagem de microkernel. Esse método estrutura o sistema operacional removendo todos os componentes não essenciais do kernel e implementando-os como programas de nível de sistema e de usuário. O resultado é um kernel menor. Há pouco consenso sobre quais serviços devem permanecer no kernel e quais devem ser implementados no espaço do usuário. Normalmente, no entanto, os microkernels fornecem um gerenciamento mínimo dos processos e da memória, além de um recurso de comunicação. A Figura 2.14 ilustra a arquitetura de um microkernel típico. A principal função do microkernel é fornecer comunicação entre o programa cliente e os diversos serviços que também estão sendo executados no espaço do usuário. A comunicação é fornecida por transmissão de mensagens, que foi descrita na Seção 2.4.5. Por exemplo, se o programa cliente deseja acessar um arquivo, ele deve interagir com o servidor de arquivos. O programa cliente e o serviço nunca interagem diretamente. Em vez disso, eles se comunicam indiretamente trocando mensagens com o microkernel.

Um dos benefícios da abordagem de microkernel é que ela facilita a extensão do sistema operacional. Todos os serviços novos são adicionados ao espaço do usuário e, conseqüentemente, não requerem a modificação do kernel. Quando o kernel precisa ser modificado, as alterações tendem a ser minimizadas porque o microkernel é um kernel menor. O sistema operacional resultante é mais fácil de ser portado de um projeto de hardware para outro. O microkernel também fornece mais segurança e confiabilidade, já que a maioria dos serviços é executada como processos de usuário — e não do kernel. Se um serviço falha, o resto do sistema operacional permanece intocado.

Alguns sistemas operacionais contemporâneos têm usado a abordagem de microkernel. O Tru64 UNIX (antes conhecido como Digital UNIX) fornece uma interface UNIX para o usuário, mas é implementado com um kernel Mach. O kernel Mach mapeia chamadas de sistema UNIX em mensagens enviadas aos serviços de nível de usuário apropriados. O kernel do Mac OS X (também conhecido como Darwin) também se baseia, em parte, no microkernel Mach.

Outro exemplo é o QNX, um sistema operacional de tempo real para sistemas embutidos. O microkernel QNX Neutrino fornece serviços de transmissão de mensagens e scheduling de processos. Ele também manipula a comunicação de rede de baixo nível e as interrupções de hardware. Todos os outros serviços do QNX são fornecidos por processos-padrão executados fora do kernel em modalidade de usuário.

Infelizmente, o desempenho dos microkernels pode ser afetado pelo aumento do overhead de funções de sistema. Considere a história do Windows NT. A primeira versão tinha uma organização de microkernel em camadas. O desempenho dessa versão era baixo, se comparado ao do Windows 95. O Windows NT 4.0 corrigiu parcialmente o problema de desempenho movendo camadas do espaço do usuário para o espaço do kernel e integrando-as mais fortemente. Quando o Windows XP foi projetado, a arquitetura Windows tinha se tornado mais monolítica do que de microkernel.

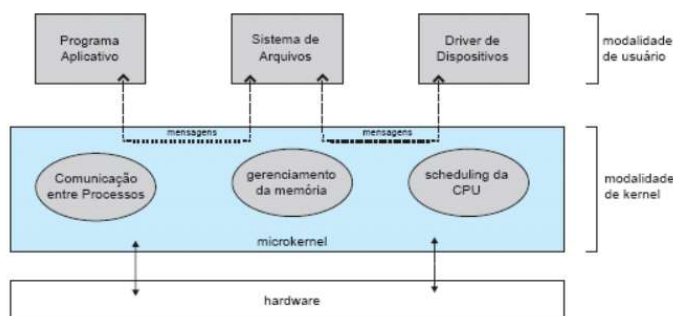
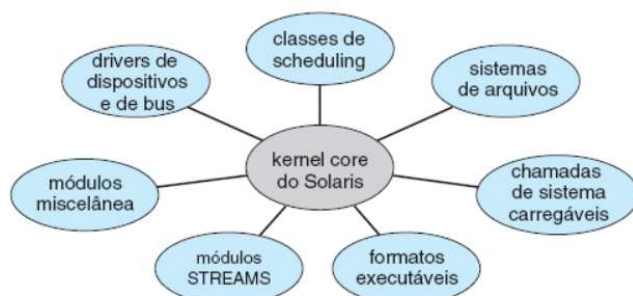


Figura 2.14 Arquitetura de um microkernel típico.



#### 2.7.4 Módulos

Talvez a melhor metodologia atual para o projeto de sistemas operacionais envolva o uso de módulos de kernel carregáveis. Aqui, o kernel tem um conjunto de componentes nucleares e vincula serviços adicionais por meio de módulos, tanto em tempo de inicialização quanto em tempo de execução. Esse tipo de projeto é comum em implementações modernas do UNIX e, também, do Solaris, do Linux e do Mac OS X, assim como do Windows.

A ideia do projeto é que o kernel forneça serviços nucleares enquanto outros serviços são implementados dinamicamente quando o kernel está em execução. É melhor vincular serviços dinamicamente do que adicionar novos recursos diretamente ao kernel, o que demanda a recompilação do kernel sempre que uma alteração é feita. Assim, podemos, por exemplo, construir algoritmos de scheduling da CPU e gerenciamento da memória diretamente no kernel e, então, adicionar o suporte a diferentes sistemas de arquivos por meio de módulos carregáveis.

O resultado final lembra um sistema em camadas em que cada seção do kernel tem interfaces definidas e protegidas; porém, ele é mais flexível do que um sistema em camadas porque um módulo pode chamar qualquer outro módulo. A abordagem também é semelhante à abordagem de microkernel, já que o módulo principal tem apenas funções nucleares e o conhecimento de como carregar e se comunicar com outros módulos; no entanto, é mais eficiente porque os módulos não precisam invocar a transmissão de mensagens para se comunicarem. A estrutura do sistema operacional Solaris, mostrada na Figura 2.15, é organizada ao redor de um kernel nuclear com sete tipos de módulos do kernel carregáveis:

1. Classes de scheduling
2. Sistemas de arquivos
3. Chamadas de sistema carregáveis
4. Formatos executáveis
5. Módulos STREAMS
6. Miscelâneas
7. Drivers de dispositivos e de bus

O Linux também usa módulos do kernel carregáveis, principalmente no suporte a drivers de dispositivos e sistemas de arquivos. Abordamos a criação de módulos do kernel carregáveis no Linux, como um exercício de programação, no fim deste capítulo.

### 2.7.5 Sistemas Híbridos

Na prática, muito poucos sistemas operacionais adotam uma estrutura única rigidamente definida. Em vez disso, eles combinam diferentes estruturas, resultando em sistemas híbridos que resolvem problemas de desempenho, segurança e usabilidade. Por exemplo, tanto o Linux quanto o Solaris são monolíticos porque o desempenho é muito mais eficiente quando o sistema operacional ocupa um único espaço de endereçamento. No entanto, eles também são modulares para que novas funcionalidades possam ser adicionadas ao kernel dinamicamente. O Windows também é amplamente monolítico (mais uma vez por questões de desempenho, principalmente), mas retém certo comportamento típico de sistemas de microkernel, inclusive fornecendo suporte a subsistemas separados (conhecidos como personalidades do sistema operacional) que são executados como processos de modalidade de usuário. Os sistemas Windows também fornecem suporte para módulos do kernel carregáveis dinamicamente. Fornecemos estudos de caso do Linux e do Windows 7 nos Capítulos 18 e 19, respectivamente. No resto desta seção, exploramos a estrutura de três sistemas híbridos: o sistema operacional Mac OS X da Apple e os dois sistemas operacionais móveis mais proeminentes — iOS e Android.

#### 2.7.5.1 Mac OS X

O sistema operacional Mac OS X da Apple usa uma estrutura híbrida. Como mostrado na Figura 2.16, ele é um sistema em camadas. As camadas do topo incluem a interface de usuário Aqua (Figura 2.4) e um conjunto de ambientes e serviços de aplicações. Em destaque, o ambiente Cocoa especifica uma API para a linguagem de programação Objective-C, usada para escrever aplicações do Mac OS X. Abaixo dessas camadas está o ambiente de kernel, que é composto, principalmente, pelo microkernel Mach e o kernel BSD UNIX. O Mach fornece gerenciamento da memória; suporte a chamadas de procedimento remotas (RPCs) e recursos de comunicação entre processos (IPC), incluindo transmissão de mensagens; e o scheduling de threads. O componente BSD fornece uma interface de linha de comando BSD, suporte à conexão de rede e sistemas de arquivos e uma implementação de APIs POSIX, incluindo o Pthreads. Além do MAC e do BSD, o ambiente do kernel fornece um kit de I/O para o desenvolvimento de drivers de dispositivos e de módulos carregáveis dinamicamente (que o Mac OS X chama de extensões do kernel). Como mostrado na Figura 2.16, o ambiente de aplicações BSD pode fazer uso dos recursos do BSD diretamente.

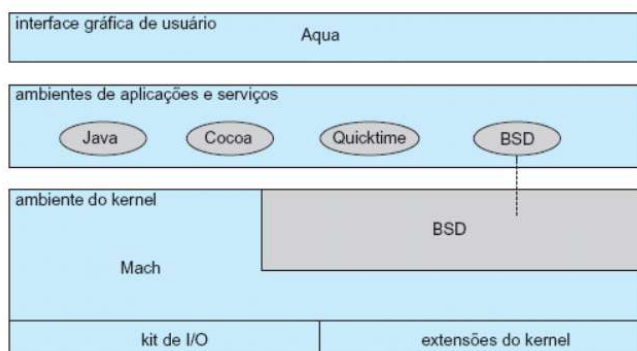


Figura 2.16 A estrutura do Mac OS X.

### 2.7.5.2 iOS

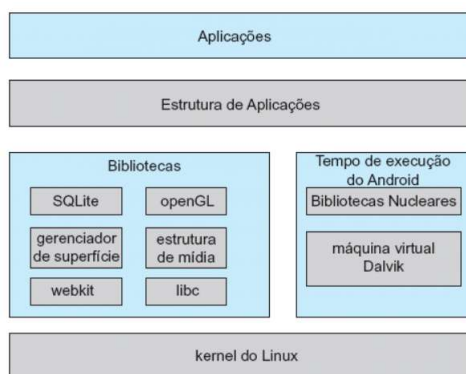
O iOS é um sistema operacional móvel projetado pela Apple para executar seu smartphone, o iPhone, bem como seu computador tablet, o iPad. O iOS foi estruturado sobre o sistema operacional Mac OS X, com funcionalidades adicionais pertinentes aos dispositivos móveis, mas não executa aplicativos do Mac OS X diretamente. A estrutura do iOS aparece na Figura 2.17.

Cocoa Touch é uma API para Objective-C que fornece várias estruturas para o desenvolvimento de aplicativos que executam em dispositivos iOS. A diferença fundamental entre o Cocoa, mencionado anteriormente, e o Cocoa Touch é que o último dá suporte a recursos de hardware exclusivos para dispositivos móveis, tais como as telas sensíveis ao toque. A camada de serviços de mídia fornece serviços para ambientes gráficos, áudio e vídeo.

A camada de serviços nucleares fornece uma variedade de recursos, incluindo suporte à computação em nuvem e a bancos de dados. A camada da base representa o sistema operacional nuclear que é baseado no ambiente de kernel mostrado na Figura 2.16.

### 2.7.5.3 Android

O sistema operacional Android foi projetado pela Open Handset Alliance (dirigida principalmente pela Google) e foi desenvolvido para smartphones e computadores tablets Android. Enquanto o iOS é projetado para execução em dispositivos móveis da Apple e seu código-fonte é fechado, o Android é executado em várias plataformas móveis e seu código-fonte é aberto, o que explica, em parte, sua rápida ascensão em popularidade. A estrutura do Android aparece na Figura 2.18. O Android é semelhante ao iOS por ser uma pilha de softwares em camadas que fornece um rico conjunto de estruturas para o desenvolvimento de aplicativos móveis. Na base dessa pilha de softwares, está o kernel Linux, embora ele tenha sido modificado pela Google e esteja, atualmente, fora da distribuição normal de versões do Linux.



O Linux é usado, principalmente, no suporte de hardware a processos, memória e drivers de dispositivos e tem sido expandido para incluir gerenciamento de energia. O ambiente de tempo de execução do Android inclui um conjunto básico de bibliotecas e a máquina virtual Dalvik. Os projetistas de software para dispositivos Android desenvolvem aplicativos na linguagem Java. No entanto, em vez de usar a API Java padrão, a Google projetou uma API Android separada para desenvolvimento em Java. Os arquivos de classes Java são compilados para bytecode Java e, depois, traduzidos para um arquivo executável que é operado na máquina virtual Dalvik. A



máquina virtual Dalvik foi projetada para o Android e é otimizada para dispositivos móveis com recursos limitados de memória e de processamento da CPU.

O conjunto de bibliotecas disponível para aplicativos Android inclui estruturas para desenvolvimento de navegadores web (webkit), suporte a banco de dados (SQLite) e multimídia. A biblioteca libc é semelhante à biblioteca C padrão, mas é muito menor e foi projetada para as CPUs mais lentas que caracterizam os dispositivos móveis.

## **2.8 Depuração do Sistema Operacional**

Fizemos menção frequente à depuração, neste capítulo. Aqui, nós a examinamos com mais detalhes. Em sentido amplo, depuração é a atividade de encontrar e corrigir erros em um sistema, tanto em hardware quanto em software. Problemas de desempenho são considerados bugs e, portanto, a depuração também pode incluir o ajuste de desempenho que tenta melhorar o desempenho removendo gargalos de processamento. Nesta seção, exploramos a depuração de erros em processos e no kernel e de problemas de desempenho. A depuração do hardware está fora do escopo deste texto.

### **2.8.1 Análise de Falhas**

Quando um processo falha, a maioria dos sistemas operacionais grava as informações de erro em um arquivo de log para alertar os operadores ou usuários do sistema de que o problema ocorreu. O sistema operacional também pode obter um despejo do núcleo — uma captura da memória do processo — e armazená-lo em um arquivo para análise posterior. (A memória era chamada de “núcleo” nos primórdios da computação.) Programas em execução e despejos do núcleo podem ser examinados por um depurador, o que permite ao programador examinar o código e a memória de um processo.

A depuração do código de processos de nível de usuário é um desafio. A depuração do kernel do sistema operacional é ainda mais complexa por causa do tamanho e complexidade do kernel, seu controle sobre o hardware e a falta de ferramentas de depuração de nível de usuário. Uma falha no kernel é chamada de desastre. Quando ocorre um desastre, as informações de erro são salvas em um arquivo de log, e o estado da memória é salvo em um despejo de desastre.

Geralmente, a depuração de sistemas operacionais e a depuração de processos usam ferramentas e técnicas diferentes em razão da natureza muito diferente dessas duas tarefas. Considere que uma falha de kernel no código do sistema de arquivos tornaria arriscado para o kernel tentar salvar seu estado em um arquivo do sistema de arquivos antes da reinicialização. Uma técnica comum é salvar o estado da memória do kernel em uma seção de disco sem sistema de arquivos e reservada para essa finalidade. Quando o kernel detecta um erro irreversível, ele grava todo o conteúdo da memória ou, pelo menos, as partes da memória do sistema que lhe pertencem, nessa área de disco. Quando o sistema reinicializa, um processo é executado para coletar os dados dessa área e gravá-los em um arquivo de despejo de desastre dentro de um sistema de arquivos, para análise. Obviamente, tais estratégias seriam desnecessárias para a depuração de processos em nível de usuário comum.

## LEI DE KERNIGHAN

“Depurar é duas vezes mais difícil do que escrever o código pela primeira vez. Portanto, se você escrever o código o mais inteligentemente possível, você não é, por definição, suficientemente inteligente para depurá-lo.”

### 2.8.2 Ajuste de Desempenho

Mencionamos, anteriormente, que o ajuste de desempenho procura melhorar o desempenho removendo gargalos no processamento. Para identificar gargalos, devemos ser capazes de monitorar o desempenho do sistema. Assim, o sistema operacional deve ter algum meio de calcular e exibir medidas de comportamento do sistema. Em vários sistemas, o sistema operacional faz isso produzindo listagens de rastreamento do comportamento do sistema. Todos os eventos de interesse são registrados em log, com sua duração e parâmetros importantes, e gravados em um arquivo. Posteriormente, um programa de análise pode processar o arquivo de log para determinar o desempenho do sistema e identificar gargalos e ineficiências. Esses mesmos rastreamentos podem ser executados como entradas para a simulação de uma proposta de sistema otimizado. Os rastreamentos também podem ajudar as pessoas a encontrar erros no comportamento do sistema operacional. Outra abordagem para o ajuste de desempenho utiliza ferramentas interativas e de uso específico que permitem a usuários e administradores questionarem o estado de vários componentes do sistema para procurar gargalos. Uma dessas ferramentas emprega o comando UNIX `top` para exibir os recursos usados no sistema, assim como uma lista ordenada dos processos “top” em relação ao uso de recursos. Outras ferramentas exibem o estado do I/O de disco, da alocação de memória e do tráfego de rede. O Gerenciador de Tarefas do Windows é uma ferramenta semelhante para sistemas Windows. O gerenciador de tarefas inclui informações sobre aplicativos e processos em execução corrente, uso de CPU e memória e estatísticas de rede. Uma tomada de tela do gerenciador de tarefas aparece na Figura 2.19.

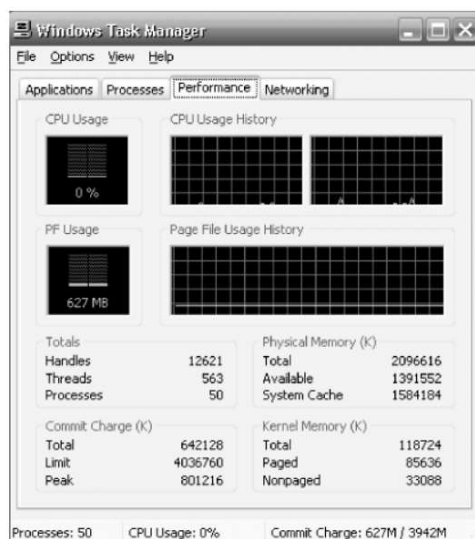


Figura 2.19 O gerenciador de tarefas do Windows.

Tornar os sistemas operacionais mais fáceis de entender, depurar e ajustar enquanto são executados é uma área ativa de pesquisa e implementação. Uma nova geração de ferramentas de análise de desempenho, habilitadas para o kernel, tem feito melhorias significativas no modo como esse objetivo pode ser atingido. A seguir, discutimos um exemplo pioneiro desse tipo de ferramenta: o recurso de rastreamento dinâmico DTrace, do Solaris 10.

### 2.8.3DTrace

O DTrace é um recurso que adiciona, dinamicamente, sondagens a um sistema em execução, tanto em processos de usuário quanto no kernel. Essas sondagens podem ser consultadas por meio da linguagem de programação D, para determinar uma quantidade enorme de informações sobre o kernel, o estado do sistema e atividades de processos. Por exemplo, a Figura 2.20 segue uma aplicação, enquanto ela executa uma chamada de sistema [`ioctl ( )`], e mostra as chamadas funcionais dentro do kernel enquanto elas são executadas para realizar a chamada. As linhas que terminam com “U” são executadas em modalidade de usuário, e as que terminam com “K”, em modalidade de kernel.

A depuração das interações entre código de nível de usuário e código de kernel é quase impossível sem um grupo de ferramentas que entenda os dois conjuntos de código e possa instrumentar as interações. Para que esse conjunto de ferramentas seja realmente útil, ele deve ser capaz de depurar qualquer área de um sistema, inclusive áreas que não foram escritas visando à depuração, e fazendo isso sem afetar a confiabilidade do sistema. Essa ferramenta também deve ter um impacto mínimo sobre o desempenho — o ideal é que ela não cause impactos quando não está em uso e cause um impacto proporcional durante o uso. A ferramenta DTrace atende a tais requisitos e fornece um ambiente de depuração dinâmico, seguro e de baixo impacto.

Antes de a estrutura e as ferramentas do DTrace se tornarem disponíveis no Solaris 10, a depuração do kernel era, usualmente, uma tarefa enigmática executada por meio de códigos e ferramentas casuais e arcaicos. Por exemplo, as CPUs têm um recurso de breakpoint que interrompe a execução e permite que o depurador examine o estado do sistema. Em seguida, a execução pode continuar até o próximo breakpoint ou o seu término. Esse método não pode ser usado em um kernel de sistema operacional multiusuário sem afetar negativamente todos os usuários do sistema. A geração de perfis que, periodicamente, coleta amostras do ponteiro de instruções para determinar qual o código que está sendo executado pode mostrar tendências estatísticas, mas não atividades individuais. Um código poderia ser incluído no kernel para a emissão de dados específicos sob determinadas circunstâncias, mas esse código torna o kernel lento e tende a não ser incluído na parte do kernel em que ocorre o problema específico que está sendo depurado.

Por outro lado, o DTrace é executado em sistemas de produção — sistemas que estão executando aplicações importantes ou críticas — e não causa danos ao sistema. Ele retarda as atividades enquanto está habilitado, mas, após a execução, restaura o sistema ao seu estado pré-depuração. Além disso, é uma ferramenta abrangente e profunda. Pode depurar de forma abrangente tudo que está acontecendo no sistema (tanto nos níveis do usuário e do kernel quanto entre as camadas de usuário e de kernel). Também pode investigar profundamente o código, exibindo instruções individuais da CPU ou atividades de sub-rotinas do kernel.

O DTrace é composto por um compilador, uma estrutura, provedores de sondagens escritos dentro dessa estrutura e consumidores dessas sondagens. Os provedores do DTrace criam sondagens. Existem estruturas no kernel que controlam todas as sondagens que os provedores criam. As sondagens são armazenadas em uma estrutura de dados de tabela hash a qual é aplicada uma função hash por nome, e indexada de acordo com identificadores de sondagem exclusivos. Quando uma sondagem é habilitada, um trecho de código da área a ser sondada é reescrito para chamar `dtrace_probe` (probe identifier) e, então, continuar com a operação original do código. Diferentes provedores criam diferentes tipos de sondagens. Por exemplo, uma sondagem de chamadas de sistema do kernel funciona de modo diferente de uma sondagem de processos do usuário que é diferente de uma sondagem de I/O.

O DTrace inclui um compilador que gera um bytecode executado no kernel. Esse código tem sua “segurança” assegurada pelo compilador. Por exemplo, loops não são permitidos, e apenas modificações específicas no estado do kernel são autorizadas quando especificamente solicitadas. Somente usuários com “privilégios” do DTrace (ou usuários “root”) podem usá-lo, já que ele pode recuperar dados privados do kernel (e modificar dados, se solicitado). O código gerado é executado no kernel e habilita sondagens. Ele também habilita consumidores em modalidade de usuário e comunicações entre os dois.

Um consumidor do DTrace é um código interessado em uma sondagem e em seus resultados. O consumidor solicita que o provedor crie uma ou mais sondagens. Quando uma sondagem é acionada, ela emite dados que são gerenciados pelo kernel. Dentro do kernel, ações chamadas blocos de controle de habilitação, ou ECBs (enabling control blocks), são executadas quando sondagens são acionadas. Uma sondagem pode fazer com que vários ECBs sejam executados se mais de um consumidor estiver interessado nessa sondagem. Cada ECB contém um predicado (“comando if”) que pode filtrá-lo. Caso contrário, a lista de ações do ECB é executada. A ação mais comum é a captura de algum fragmento de dados, como o valor de uma variável no ponto de execução da sondagem. Por meio da coleta desses dados, um cenário completo de uma ação do usuário ou do kernel pode ser construído. Além disso, sondagens acionadas, tanto a partir do espaço do usuário quanto a partir do kernel, podem mostrar como uma ação de nível de usuário causou reações no nível do kernel. Tais dados são inestimáveis para monitoração do desempenho e otimização do código.

Assim que o consumidor da sondagem termina, seus ECBs são removidos. Se não houver ECBs consumindo uma sondagem, a sondagem é removida. Isso envolve a reescrita do código para remover a chamada `dtrace_probe` ( ) e recolocar o código original. Portanto, antes de uma sondagem ser criada e após a mesma ser destruída, o sistema fica exatamente igual, como se nenhuma sondagem tivesse ocorrido.

O DTrace encarrega-se de assegurar que as sondagens não usem memória ou capacidade de CPU em demasia, o que poderia prejudicar o sistema que está em execução. Os buffers usados para armazenar os resultados da sondagem são monitorados para não excederem os limites default e máximo. O tempo de CPU para a execução de sondagens também é monitorado. Se limites são excedidos, o consumidor é encerrado, junto com as sondagens ofensivas. Buffers são alocados por CPU para evitar a disputa e a perda de dados. Um exemplo de código D junto com sua saída mostra parte de sua utilidade. O programa a seguir mostra o código DTrace que habilita sondagens no scheduler e registra o montante de tempo de CPU de cada processo em execução com ID de usuário 101 enquanto essas sondagens estão habilitadas (isto é, enquanto o programa

é executado): A saída do programa, exibindo os processos e quanto tempo (em nanossegundos) eles levam em execução nas CPUs, é mostrada na Figura 2.21.

Já que o DTrace faz parte da versão de código-fonte aberto OpenSolaris do sistema operacional Solaris 10, ele foi adicionado a outros sistemas operacionais que não têm contratos de licença conflitantes. Por exemplo, o DTrace foi adicionado ao Mac OS X e ao FreeBSD e, provavelmente, terá uma disseminação ainda maior em razão de seus recursos únicos. Outros sistemas operacionais, principalmente os derivados do Linux, também estão adicionando a funcionalidade de rastreamento do kernel. Ainda outros sistemas operacionais estão começando a incluir ferramentas de desempenho e rastreamento fomentadas por pesquisas em várias instituições, o que inclui o projeto Paradyn.

## 2.9 Geração do Sistema Operacional

É possível projetar, codificar e implementar um sistema operacional especificamente para um computador em um determinado sítio. Geralmente, no entanto, os sistemas operacionais são projetados para execução em qualquer máquina de uma classe de máquinas, em uma variedade de sítios, com inúmeras configurações periféricas. O sistema deve, então, ser configurado ou gerado para cada sítio específico de computadores, um processo também conhecido como geração do sistema (SYSGEN).

Normalmente, o sistema operacional é distribuído em disco, em CD-Rom ou DVD-Rom ou como uma imagem “ISO”, que é um arquivo no formato de um CD-Rom ou DVD-Rom. Para gerar um sistema, usamos um programa especial. Esse programa SYSGEN lê a partir de um arquivo fornecido ou solicita ao operador do sistema informações relacionadas com a configuração específica do sistema de hardware ou, ainda, sonda o hardware diretamente para determinar que componentes estão disponíveis. Os seguintes tipos de informações devem ser determinados.

- Qual é a CPU a ser usada? Que opções (conjuntos de instruções estendidas, aritmética de ponto flutuante, e assim por diante) estão instaladas? Para sistemas com múltiplas CPUs, cada uma das CPUs deve ser descrita.
- Como o disco de inicialização será formatado? Em quantas seções, ou “partições”, ele será dividido, e o que haverá em cada partição?
- Quanta memória está disponível? Alguns sistemas determinarão esse valor por si mesmos, referenciando locação a locação da memória, até que uma falha de “endereço ilegal” seja gerada. Esse procedimento define o último endereço legal e, portanto, o montante de memória disponível.
- Que dispositivos estão disponíveis? O sistema precisará saber como endereçar cada dispositivo (o número do dispositivo), seu número de interrupção, tipo e modelo, e qualquer característica especial do dispositivo.
- Que opções do sistema operacional são desejadas, ou que valores de parâmetros devem ser usados? Essas opções ou valores podem incluir a quantidade e o tamanho dos buffers a serem usados, que tipo de algoritmo de scheduling da CPU é desejado, a quantidade máxima de processos a ser suportada, e assim por diante.

Uma vez que tais informações sejam determinadas, elas poderão ser usadas de várias maneiras. Em um extremo, um administrador de sistemas pode usá-las para modificar uma cópia do código-fonte do sistema operacional. O sistema operacional é, então, totalmente compilado. Declarações de dados, inicializações e constantes, junto com uma compilação condicional, produzem uma versão-objeto do sistema operacional personalizada para o sistema descrito.

Em um nível de personalização um pouco menor, a descrição do sistema pode levar à criação de tabelas e à seleção de módulos em uma biblioteca pré-compilada. Esses módulos são vinculados para gerar o sistema operacional. A seleção permite que a biblioteca contenha os drivers de dispositivos para todos os dispositivos de I/O suportados, mas só os necessários são vinculados ao sistema operacional. Já que o sistema não é recompilado, sua geração é mais rápida, mas o sistema resultante pode ser excessivamente genérico.

No outro extremo, é possível construir um sistema totalmente dirigido por tabelas. O código inteiro faz parte do sistema o tempo todo, e a seleção ocorre em tempo de execução, em vez de em tempo de compilação ou vinculação. A geração do sistema envolve simplesmente a criação das tabelas apropriadas para descrevê-lo.

As principais diferenças entre essas abordagens são o tamanho e a generalidade do sistema gerado e a facilidade de modificá-lo quando a configuração do hardware muda. Considere o custo de modificação do sistema para dar suporte a um terminal gráfico recém-adquirido ou outro drive de disco. Como contraponto a esse custo, é claro, está a frequência (ou infrequência) de tais mudanças.

## **2.10 Inicialização do Sistema**

Após um sistema operacional ser gerado, ele deve ser disponibilizado para uso pelo hardware. Mas como o hardware sabe onde está o kernel ou como carregar esse kernel? O procedimento de iniciar um computador a partir da carga do kernel é conhecido como inicialização do sistema. Na maioria dos sistemas de computação, um pequeno bloco de código conhecido como programa bootstrap ou carregador bootstrap localiza o kernel, carrega-o na memória principal e inicia sua execução. Alguns sistemas de computação, como os PCs, usam um processo de dois passos em que um carregador bootstrap simples acessa um programa de inicialização mais complexo em disco que, por sua vez, carrega o kernel. Quando uma CPU recebe um evento de reinicialização — por exemplo, quando é ligada ou reinicializada — o registrador de instruções é carregado com uma locação de memória predefinida, e a execução começa daí. Nessa locação, está o programa bootstrap inicial. Esse programa encontra-se na forma de memória somente de leitura (ROM) porque a RAM está em um estado desconhecido na inicialização do sistema. A ROM é conveniente porque não precisa de inicialização e não pode ser infectada facilmente por um vírus de computador. O programa bootstrap pode executar várias tarefas. Geralmente, uma das tarefas é a execução de diagnósticos para determinar o estado da máquina. Se os diagnósticos forem bem-sucedidos, o programa poderá continuar com os passos de inicialização. Ele também pode inicializar todos os aspectos do sistema, dos registradores da CPU aos controladores de dispositivos e o conteúdo da memória principal. Assim que possível, ele inicia o sistema operacional. Alguns sistemas — como os telefones celulares, tablets e consoles de jogos — armazenam o sistema operacional inteiro em ROM. O armazenamento do sistema operacional em ROM é adequado para sistemas operacionais pequenos, hardware de suporte simples, e operação irregular. Um problema dessa abordagem é que a alteração do código bootstrap requer a mudança dos chips de hardware da ROM. Alguns sistemas resolvem

esse problema usando memória somente de leitura apagável e programável (EPROM), que é somente de leitura, exceto quando recebe explicitamente um comando para se tornar gravável. Todos os tipos de ROM também são conhecidos como firmware, já que suas características se encaixam em algum ponto entre as características de hardware e as de software. Um problema comum com o firmware é que a execução do código nesse local é mais lenta do que em RAM. Alguns sistemas armazenam o sistema operacional em firmware e o copiam em RAM para execução rápida. Um último problema do firmware é que ele é relativamente caro e, portanto, em geral, somente pequenos montantes ficam disponíveis.

Para sistemas operacionais grandes (inclusive a maioria dos sistemas operacionais de uso geral como o Windows, o Mac OS X e o UNIX) ou para sistemas que mudam com frequência, o carregador bootstrap é armazenado em firmware e o sistema operacional fica em disco. Nesse caso, o programa bootstrap executa diagnósticos e contém um trecho de código que pode ler um único bloco em uma localização fixa (por exemplo, o bloco zero) do disco, enviar para a memória e executar o código a partir desse bloco de inicialização. O programa armazenado no bloco de inicialização pode ser suficientemente sofisticado para carregar o sistema operacional inteiro na memória e começar sua execução. Mais comumente, é um código simples (já que cabe em um único bloco do disco) e conhece apenas o endereço em disco e o tamanho do resto do programa bootstrap. O GRUB é um exemplo de programa bootstrap de código-fonte aberto para sistemas Linux. Toda a inicialização baseada em disco e o próprio sistema operacional podem ser facilmente alterados com a criação de novas versões em disco. Um disco que contenha uma partição de inicialização (veja mais sobre isso na Seção 10.5.1) é chamado de disco de inicialização ou disco do sistema.

Agora que o programa bootstrap inteiro foi carregado, ele pode percorrer o sistema de arquivos para encontrar o kernel do sistema operacional, carregá-lo na memória e iniciar sua execução. É somente nesse ponto que o sistema é considerado em execução.

## **2.11 Resumo**

Os sistemas operacionais fornecem vários serviços. No nível mais baixo, chamadas de sistema permitem que um programa em execução faça solicitações diretamente ao sistema operacional. Em um nível mais alto, o interpretador de comandos ou shell fornece um mecanismo para o usuário emitir uma solicitação sem escrever um programa. Os comandos podem ser provenientes de arquivos durante a execução em modalidade batch ou diretamente de um terminal ou da GUI de um desktop quando em modalidade interativa ou de tempo compartilhado. Programas de sistema são fornecidos para atender a muitas solicitações comuns dos usuários.

Os tipos de solicitação variam de acordo com o nível. O nível de chamada de sistema deve fornecer as funções básicas, como controle de processos e manipulação de arquivos e dispositivos. Solicitações de nível mais alto, atendidas pelo interpretador de comandos ou por programas de sistema, são traduzidas em uma sequência de chamadas de sistema. Os serviços do sistema podem ser classificados em várias categorias: controle de programas, solicitações de status e solicitações de I/O. Os erros de programa podem ser considerados pedidos implícitos de serviço. O projeto de um novo sistema operacional é uma tarefa de peso. É importante que os objetivos do sistema sejam bem definidos antes de o projeto começar. O tipo de sistema desejado é a base das escolhas entre os vários algoritmos e estratégias que serão necessários. Durante todo o ciclo de projeto, devemos ter o cuidado de separar decisões políticas de detalhes

(mecanismos) de implementação. Essa separação permite flexibilidade máxima se as decisões políticas tiverem de ser alteradas posteriormente.

Uma vez que um sistema operacional seja projetado, ele deve ser implementado. Atualmente, os sistemas operacionais são quase sempre escritos em uma linguagem de implementação de sistemas ou em uma linguagem de mais alto nível. Essa característica melhora sua implementação, manutenção e portabilidade.

Um sistema tão grande e complexo, como os sistemas operacionais modernos, deve ser construído cuidadosamente. A modularidade é importante. Projetar um sistema como uma sequência de camadas ou usar um microkernel são consideradas boas técnicas. Muitos sistemas operacionais atuais suportam módulos carregados dinamicamente que permitem a inclusão de funcionalidades em um sistema operacional enquanto ele está em execução. Geralmente, os sistemas operacionais adotam uma abordagem híbrida que combina vários tipos de estruturas diferentes.

A depuração de falhas em processos e no kernel pode ser feita com o uso de depuradores e outras ferramentas que analisem despejos de memória. Ferramentas como o DTrace analisam sistemas de produção para encontrar gargalos e entender outros comportamentos do sistema. Para criar um sistema operacional para uma configuração de máquina específica, devemos realizar a geração do sistema. Para que o sistema de computação entre em execução, a CPU deve inicializar e começar a execução do programa bootstrap em firmware. O programa bootstrap pode executar o sistema operacional diretamente se este também estiver no firmware, ou pode completar uma sequência em que carregue progressivamente programas mais inteligentes a partir de firmware e disco até que o próprio sistema operacional seja carregado na memória e executado.