CAPÍTULO 2

# PROCESSOS E THREADS

Estamos prestes a embarcar agora em um estudo detalhado de como os sistemas operacionais são projetados e construídos. O conceito mais central em qualquer sistema operacional é o *processo*: uma abstração de um programa em execução. Tudo o mais depende desse conceito, e o projetista (e estudante) do sistema operacional deve ter uma compreensão profunda do que é um processo o mais cedo possível.

Processos são uma das mais antigas e importantes abstrações que os sistemas operacionais proporcionam. Eles dão suporte à possibilidade de haver operações (pseudo) concorrentes mesmo quando há apenas uma CPU disponível, transformando uma única CPU em múltiplas CPUs virtuais. Sem a abstração de processo, a computação moderna não poderia existir. Neste capítulo, examinaremos detalhadamente os processos e seus "primos", os threads.

## 2.1 Processos

Todos os computadores modernos frequentemente realizam várias tarefas ao mesmo tempo. As pessoas acostumadas a trabalhar com computadores talvez não estejam totalmente cientes desse fato, então alguns exemplos podem esclarecer este ponto. Primeiro, considere um servidor da web, em que solicitações de páginas da web chegam de toda parte. Quando uma solicitação chega, o servidor confere para ver se a página requisitada está em cache. Se estiver, ela é enviada de volta; se não, uma solicitação de acesso ao disco é iniciada para buscá-la. No entanto, do ponto de vista da CPU, as solicitações de acesso ao disco levam uma eternidade. Enquanto espera que uma solicitação de acesso ao disco seja concluída, muitas outras solicitações podem chegar. Se há múltiplos discos presentes, algumas ou todas as solicitações mais recentes podem ser enviadas para os outros discos muito antes de a primeira solicitação ter sido concluída. Está claro que algum método é necessário para modelar e controlar essa concorrência. Processos (e especialmente threads) podem ajudar nisso.

Agora considere um PC de usuário. Quando o sistema é inicializado, muitos processos são secretamente iniciados, quase sempre desconhecidos para o usuário. Por exemplo, um processo pode ser inicializado para esperar pela chegada de e-mails. Outro pode ser executado em prol do programa antivírus para conferir periodicamente se há novas definições de vírus disponíveis. Além disso, processos explícitos de usuários podem ser executados, imprimindo arquivos e salvando as fotos do usuário em um pen-drive, tudo isso enquanto o usuário está navegando na Web. Toda essa atividade tem de ser gerenciada, e um sistema de multiprogramação que dê suporte a múltiplos processos é muito útil nesse caso.

Em qualquer sistema de multiprogramação, a CPU muda de um processo para outro rapidamente, executando cada um por dezenas ou centenas de milissegundos. Enquanto, estritamente falando, em qualquer dado instante a CPU está executando apenas um processo, no curso de 1s ela pode trabalhar em vários deles, dando a ilusão do paralelismo. Às vezes, as pessoas falam em **pseudoparalelismo** neste contexto, para diferenciar do verdadeiro paralelismo de hardware dos sistemas multiprocessadores (que têm duas ou mais CPUs compartilhando a mesma memória física). Ter controle sobre múltiplas atividades em paralelo é algo difícil para as pessoas realizarem. Portanto, projetistas de sistemas

operacionais através dos anos desenvolveram um modelo conceitual (processos sequenciais) que torna o paralelismo algo mais fácil de lidar. Esse modelo, seus usos e algumas das suas consequências compõem o assunto deste capítulo.

### 2.1.1 O modelo de processo

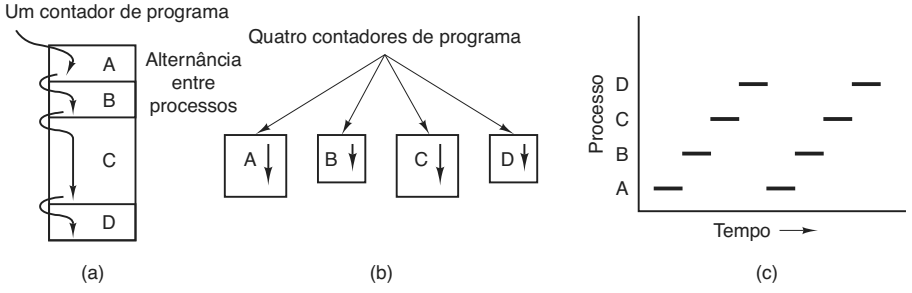
Nesse modelo, todos os softwares executáveis no computador, às vezes incluindo o sistema operacional, são organizados em uma série de **processos sequenciais**, ou, simplesmente, **processos**. Um processo é apenas uma instância de um programa em execução, incluindo os valores atuais do contador do programa, registradores e variáveis. Conceitualmente, cada processo tem sua própria CPU virtual. Na verdade, a CPU real troca a todo momento de processo em processo, mas, para compreender o sistema, é muito mais fácil pensar a respeito de uma coleção de processos sendo executados em (pseudo) paralelo do que tentar acompanhar como a CPU troca de um programa para o outro. Esse mecanismo de trocas rápidas é chamado de **multiprogramação**, como vimos no Capítulo 1.

Na Figura 2.1(a) vemos um computador multiprogramando quatro programas na memória. Na Figura 2.1(b) vemos quatro processos, cada um com seu próprio fluxo de controle (isto é, seu próprio contador de programa lógico) e sendo executado independente dos outros. É claro que há apenas um contador de programa físico, de maneira que, quando cada processo é executado, o seu contador de programa lógico é carregado para o contador de programa real. No momento em que ele é concluído, o contador de programa físico é salvo no contador de programa lógico do processo na memória. Na Figura 2.1(c) vemos que, analisados durante um intervalo longo o suficiente, todos os processos tiveram progresso, mas a qualquer dado instante apenas um está sendo de fato executado.

Neste capítulo, presumiremos que há apenas uma CPU. Cada vez mais, no entanto, essa suposição não é verdadeira, tendo em vista que os chips novos são muitas vezes *multinúcleos* (*multicore*), com dois, quatro ou mais núcleos. Examinaremos os chips multinúcleos e multiprocessadores em geral no Capítulo 8, mas, por ora, é mais simples pensar em apenas uma CPU de cada vez. Então quando dizemos que uma CPU pode na realidade executar apenas um processo de cada vez, se há dois núcleos (ou CPUs) cada um deles pode ser executado apenas um processo de cada vez.

Com o chaveamento rápido da CPU entre os processos, a taxa pela qual um processo realiza a sua computação não será uniforme e provavelmente nem reproduzível se os mesmos processos forem executados outra vez. Desse modo, processos não devem ser programados com suposições predefinidas sobre a temporização. Considere, por exemplo, um processo de áudio que toca música para acompanhar um vídeo de alta qualidade executado por outro dispositivo. Como o áudio deve começar um pouco depois do que o vídeo, ele sinaliza ao servidor do vídeo para começar a execução, e então realiza um laço ocioso 10.000 vezes antes de executar o áudio. Se o laço for um temporizador confiável, tudo vai correr bem, mas se a CPU decidir trocar para outro processo durante o laço ocioso, o processo de áudio pode não ser executado de novo até que os quadros de vídeo correspondentes já tenham vindo e ido embora, e o vídeo e o áudio ficarão irritantemente fora de sincronia. Quando um processo tem exigências de tempo real, críticas como essa, isto é, eventos particulares, *têm* de ocorrer dentro de um número específico de milissegundos e medidas especiais precisam ser tomadas para assegurar que elas ocorram. Em geral, no entanto, a maioria dos processos não é afetada pela multiprogramação

**FIGURA 2.1** (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Apenas um programa está ativo de cada vez.



subjacente da CPU ou as velocidades relativas de processos diferentes.

A diferença entre um processo e um programa é sutil, mas absolutamente crucial. Uma analogia poderá ajudá-lo aqui: considere um cientista de computação que gosta de cozinhar e está preparando um bolo de aniversário para sua filha mais nova. Ele tem uma receita de um bolo de aniversário e uma cozinha bem estocada com todas as provisões: farinha, ovos, açúcar, extrato de baunilha etc. Nessa analogia, a receita é o programa, isto é, o algoritmo expresso em uma notação adequada, o cientista de computação é o processador (CPU) e os ingredientes do bolo são os dados de entrada. O processo é a atividade consistindo na leitura da receita, busca de ingredientes e preparo do bolo por nosso cientista.

Agora imagine que o filho do cientista de computação aparece correndo chorando, dizendo que foi picado por uma abelha. O cientista de computação registra onde ele estava na receita (o estado do processo atual é salvo), pega um livro de primeiros socorros e começa a seguir as orientações. Aqui vemos o processador sendo trocado de um processo (preparo do bolo) para um processo mais prioritário (prestar cuidado médico), cada um tendo um programa diferente (receita *versus* livro de primeiros socorros). Quando a picada de abelha tiver sido cuidada, o cientista de computação volta para o seu bolo, continuando do ponto onde ele havia parado.

A ideia fundamental aqui é que um processo é uma atividade de algum tipo. Ela tem um programa, uma entrada, uma saída e um estado. Um único processador pode ser compartilhado entre vários processos, com algum algoritmo de escalonamento sendo usado para determinar quando parar o trabalho em um processo e servir outro. Em comparação, um programa é algo que pode ser armazenado em disco sem fazer nada.

Vale a pena observar que se um programa está sendo executado duas vezes, é contado como dois processos. Por exemplo, muitas vezes é possível iniciar um processador de texto duas vezes ou imprimir dois arquivos ao mesmo tempo, se duas impressoras estiverem disponíveis. O fato de que dois processos em execução estão operando o mesmo programa não importa, eles são processos distintos. O sistema operacional pode ser capaz de compartilhar o código entre eles de maneira que apenas uma cópia esteja na memória, mas isso é um detalhe técnico que não muda a situação conceitual de dois processos sendo executados.

### 2.1.2 Criação de processos

Sistemas operacionais precisam de alguma maneira para criar processos. Em sistemas muito simples, ou em sistemas projetados para executar apenas uma única aplicação (por exemplo, o controlador em um forno micro-ondas), pode ser possível ter todos os processos que serão em algum momento necessários quando o sistema for ligado. Em sistemas para fins gerais, no entanto, alguma maneira é necessária para criar e terminar processos, na medida do necessário, durante a operação. Vamos examinar agora algumas das questões.

Quatro eventos principais fazem com que os processos sejam criados:

1. Inicialização do sistema.
2. Execução de uma chamada de sistema de criação de processo por um processo em execução.
3. Solicitação de um usuário para criar um novo processo.
4. Início de uma tarefa em lote.

Quando um sistema operacional é inicializado, em geral uma série de processos é criada. Alguns desses processos são de primeiro plano, isto é, processos que interagem com usuários (humanos) e realizam trabalho para eles. Outros operam no segundo plano e não estão associados com usuários em particular, mas em vez disso têm alguma função específica. Por exemplo, um processo de segundo plano pode ser projetado para aceitar e-mails, ficando inativo a maior parte do dia, mas subitamente entrando em ação quando chega um e-mail. Outro processo de segundo plano pode ser projetado para aceitar solicitações de páginas da web hospedadas naquela máquina, despertando quando uma solicitação chega para servir àquele pedido. Processos que ficam em segundo plano para lidar com algumas atividades, como e-mail, páginas da web, notícias, impressão e assim por diante, são chamados de **daemons**. Grandes sistemas comumente têm dúzias deles: no UNIX,[1] o programa *ps* pode ser usado para listar os processos em execução; no Windows, o gerenciador de tarefas pode ser usado.

Além dos processos criados durante a inicialização do sistema, novos processos podem ser criados depois também. Muitas vezes, um processo em execução emitirá chamadas de sistema para criar um ou mais processos novos para ajudá-lo em seu trabalho. Criar processos novos é particularmente útil quando o trabalho a ser feito pode ser facilmente formulado em termos de vários

[1] Neste capítulo, o UNIX deve ser interpretado como incluindo quase todos os sistemas baseados em POSIX, incluindo Linux, FreeBSD, OS X, Solaris etc., e, até certo ponto, Android e iOS também. (N. A.)

processos relacionados, mas de outra forma interagindo de maneira independente. Por exemplo, se uma grande quantidade de dados está sendo buscada através de uma rede para processamento subsequente, pode ser conveniente criar um processo para buscar os dados e colocá-los em um local compartilhado de memória enquanto um segundo processo remove os itens de dados e os processa. Em um multiprocessador, permitir que cada processo execute em uma CPU diferente também pode fazer com que a tarefa seja realizada mais rápido.

Em sistemas interativos, os usuários podem começar um programa digitando um comando ou clicando duas vezes sobre um ícone. Cada uma dessas ações inicia um novo processo e executa nele o programa selecionado. Em sistemas UNIX baseados em comandos que executam X, o novo processo ocupa a janela na qual ele foi iniciado. No Windows, quando um processo é iniciado, ele não tem uma janela, mas ele pode criar uma (ou mais), e a maioria o faz. Em ambos os sistemas, os usuários têm múltiplas janelas abertas de uma vez, cada uma executando algum processo. Utilizando o mouse, o usuário pode selecionar uma janela e interagir com o processo, por exemplo, fornecendo a entrada quando necessário.

A última situação na qual processos são criados aplica-se somente aos sistemas em lote encontrados em grandes computadores. Pense no gerenciamento de estoque ao fim de um dia em uma cadeia de lojas, nesse caso usuários podem submeter tarefas em lote ao sistema (possivelmente de maneira remota). Quando o sistema operacional decide que ele tem os recursos para executar outra tarefa, ele cria um novo processo e executa a próxima tarefa a partir da fila de entrada nele.

Tecnicamente, em todos esses casos, um novo processo é criado por outro já existente executando uma chamada de sistema de criação de processo. Esse outro processo pode ser um processo de usuário sendo executado, um processo de sistema invocado do teclado ou mouse, ou um processo gerenciador de lotes. O que esse processo faz é executar uma chamada de sistema para criar o novo processo. Essa chamada de sistema diz ao sistema operacional para criar um novo processo e indica, direta ou indiretamente, qual programa executar nele.

No UNIX, há apenas uma chamada de sistema para criar um novo processo: fork. Essa chamada cria um clone exato do processo que a chamou. Após a fork, os dois processos, o pai e o filho, têm a mesma imagem de memória, as mesmas variáveis de ambiente e os mesmos arquivos abertos. E isso é tudo. Normalmente, o processo filho então executa execve ou uma chamada de sistema similar para mudar sua imagem de memória e executar um novo programa. Por exemplo, quando um usuário digita um comando, por exemplo, *sort*, para o shell, este se bifurca gerando um processo filho, e o processo filho executa *sort*. O objetivo desse processo em dois passos é permitir que o processo filho manipule seus descritores de arquivos depois da fork, mas antes da execve, a fim de conseguir o redirecionamento de entrada padrão, saída padrão e erro padrão.

No Windows, em comparação, uma única chamada de função Win32, CreateProcess, lida tanto com a criação do processo, quanto com o carga do programa correto no novo processo. Essa chamada tem 10 parâmetros, que incluem o programa a ser executado, os parâmetros de linha de comando para alimentar aquele programa, vários atributos de segurança, bits que controlam se os arquivos abertos são herdados, informações sobre prioridades, uma especificação da janela a ser criada para o processo (se houver alguma) e um ponteiro para uma estrutura na qual as informações sobre o processo recentemente criado é retornada para quem o chamou. Além do CreateProcess, Win32 tem mais ou menos 100 outras funções para gerenciar e sincronizar processos e tópicos relacionados.

Tanto no sistema UNIX quanto no Windows, após um processo ser criado, o pai e o filho têm os seus próprios espaços de endereços distintos. Se um dos dois processos muda uma palavra no seu espaço de endereço, a mudança não é visível para o outro processo. No UNIX, o espaço de endereço inicial do filho é uma *cópia* do espaço de endereço do pai, mas há definitivamente dois espaços de endereços distintos envolvidos; nenhuma memória para escrita é compartilhada. Algumas implementações UNIX compartilham o programa de texto entre as duas, tendo em vista que isso não pode ser modificado. Alternativamente, o filho pode compartilhar toda a memória do pai, mas nesse caso, a memória é compartilhada no sistema **copy-on-write (cópia-na-escrita)**, o que significa que sempre que qualquer uma das duas quiser modificar parte da memória, aquele pedaço da memória é explicitamente copiado primeiro para certificar-se de que a modificação ocorra em uma área de memória privada. Novamente, nenhuma memória que pode ser escrita é compartilhada. É possível, no entanto, que um processo recentemente criado compartilhe de alguns dos outros recursos do seu criador, como arquivos abertos. No Windows, os espaços de endereços do pai e do filho são diferentes desde o início.

### 2.1.3 Término de processos

Após um processo ter sido criado, ele começa a ser executado e realiza qualquer que seja o seu trabalho. No

entanto, nada dura para sempre, nem mesmo os processos. Cedo ou tarde, o novo processo terminará, normalmente devido a uma das condições a seguir:

1. Saída normal (voluntária).
2. Erro fatal (involuntário).
3. Saída por erro (voluntária).
4. Morto por outro processo (involuntário).

A maioria dos processos termina por terem realizado o seu trabalho. Quando um compilador termina de traduzir o programa dado a ele, o compilador executa uma chamada para dizer ao sistema operacional que ele terminou. Essa chamada é exit em UNIX e ExitProcess no Windows. Programas baseados em tela também dão suporte ao término voluntário. Processadores de texto, visualizadores da internet e programas similares sempre têm um ícone ou item no menu em que o usuário pode clicar para dizer ao processo para remover quaisquer arquivos temporários que ele tenha aberto e então concluí-lo.

A segunda razão para o término é a que o processo descobre um erro fatal. Por exemplo, se um usuário digita o comando

```
cc foo.c
```

para compilar o programa *foo.c* e não existe esse arquivo, o compilador simplesmente anuncia esse fato e termina a execução. Processos interativos com base em tela geralmente não fecham quando parâmetros ruins são dados. Em vez disso, eles abrem uma caixa de diálogo e pedem ao usuário para tentar de novo.

A terceira razão para o término é um erro causado pelo processo, muitas vezes decorrente de um erro de programa. Exemplos incluem executar uma instrução ilegal, referenciar uma memória não existente, ou dividir por zero. Em alguns sistemas (por exemplo, UNIX), um processo pode dizer ao sistema operacional que ele gostaria de lidar sozinho com determinados erros, nesse caso o processo é sinalizado (interrompido), em vez de terminado quando ocorrer um dos erros.

A quarta razão pela qual um processo pode ser finalizado ocorre quando o processo executa uma chamada de sistema dizendo ao sistema operacional para matar outro processo. Em UNIX, essa chamada é kill. A função Win32 correspondente é TerminateProcess. Em ambos os casos, o processo que mata o outro processo precisa da autorização necessária para fazê-lo. Em alguns sistemas, quando um processo é finalizado, seja voluntariamente ou de outra maneira, todos os processos que ele criou são de imediato mortos também. No entanto, nem o UNIX, tampouco o Windows, funcionam dessa maneira.

### 2.1.4 Hierarquias de processos

Em alguns sistemas, quando um processo cria outro, o processo pai e o processo filho continuam a ser associados de certas maneiras. O processo filho pode em si criar mais processos, formando uma hierarquia de processos. Observe que, diferentemente das plantas e dos animais que usam a reprodução sexual, um processo tem apenas um pai (mas zero, um, dois ou mais filhos). Então um processo lembra mais uma hidra do que, digamos, uma vaca.

Em UNIX, um processo e todos os seus filhos e demais descendentes formam juntos um grupo de processos. Quando um usuário envia um sinal do teclado, o sinal é entregue a todos os membros do grupo de processos associados com o teclado no momento (em geral todos os processos ativos que foram criados na janela atual). Individualmente, cada processo pode pegar o sinal, ignorá-lo, ou assumir a ação predefinida, que é ser morto pelo sinal.

Como outro exemplo de onde a hierarquia de processos tem um papel fundamental, vamos examinar como o UNIX se inicializa logo após o computador ser ligado. Um processo especial, chamado *init*, está presente na imagem de inicialização do sistema. Quando começa a ser executado, ele lê um arquivo dizendo quantos terminais existem, então ele se bifurca em um novo processo para cada terminal. Esses processos esperam que alguém se conecte. Se uma conexão é bem-sucedida, o processo de conexão executa um shell para aceitar os comandos. Esses comandos podem iniciar mais processos e assim por diante. Desse modo, todos os processos no sistema inteiro pertencem a uma única árvore, com *init* em sua raiz.

Em comparação, o Windows não tem conceito de uma hierarquia de processos. Todos os processos são iguais. O único indício de uma hierarquia ocorre quando um processo é criado e o pai recebe um identificador especial (chamado de **handle**) que ele pode usar para controlar o filho. No entanto, ele é livre para passar esse identificador para algum outro processo, desse modo invalidando a hierarquia. Processos em UNIX não podem deserdar seus filhos.

### 2.1.5 Estados de processos

Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, processos muitas vezes precisam interagir entre si. Um processo pode gerar alguma saída que outro processo usa como entrada. No comando shell

```
cat chapter1 chapter2 chapter3 | grep tree
```

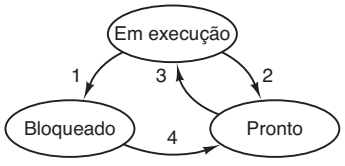
o primeiro processo, executando *cat*, gera como saída a concatenação dos três arquivos. O segundo processo, executando *grep*, seleciona todas as linhas contendo a palavra "tree". Dependendo das velocidades relativas dos dois processos (que dependem tanto da complexidade relativa dos programas, quanto do tempo de CPU que cada um teve), pode acontecer que *grep* esteja pronto para ser executado, mas não haja entrada esperando por ele. Ele deve então ser bloqueado até que alguma entrada esteja disponível.

Quando um processo bloqueia, ele o faz porque logicamente não pode continuar, em geral porque está esperando pela entrada que ainda não está disponível. Também é possível que um processo que esteja conceitualmente pronto e capaz de executar seja bloqueado porque o sistema operacional decidiu alocar a CPU para outro processo por um tempo. Essas duas condições são completamente diferentes. No primeiro caso, a suspensão é inerente ao problema (você não pode processar a linha de comando do usuário até que ela tenha sido digitada). No segundo caso, trata-se de uma tecnicalidade do sistema (não há CPUs suficientes para dar a cada processo seu próprio processador privado). Na Figura 2.2 vemos um diagrama de estado mostrando os três estados nos quais um processo pode se encontrar:

1. Em execução (realmente usando a CPU naquele instante).
2. Pronto (executável, temporariamente parado para deixar outro processo ser executado).
3. Bloqueado (incapaz de ser executado até que algum evento externo aconteça).

Claro, os primeiros dois estados são similares. Em ambos os casos, o processo está disposto a ser executado, apenas no segundo temporariamente não há uma CPU disponível para ele. O terceiro estado é fundamentalmente diferente dos dois primeiros, pois o processo não pode ser executado, mesmo que a CPU esteja ociosa e não tenha nada mais a fazer.

**FIGURA 2.2** Um processo pode estar nos estados em execução, bloqueado ou pronto. Transições entre esses estados ocorrem como mostrado.




1. O processo é bloqueado aguardando uma entrada
2. O escalonador seleciona outro processo
3. O escalonador seleciona esse processo
4. A entrada torna-se disponível

Como apresentado na Figura 2.2, quatro transições são possíveis entre esses três estados. A transição 1 ocorre quando o sistema operacional descobre que um processo não pode continuar agora. Em alguns sistemas o processo pode executar uma chamada de sistema, como em pause, para entrar em um estado bloqueado. Em outros, incluindo UNIX, quando um processo lê de um pipe ou de um arquivo especial (por exemplo, um terminal) e não há uma entrada disponível, o processo é automaticamente bloqueado.

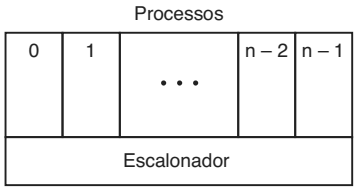
As transições 2 e 3 são causadas pelo escalonador de processos, uma parte do sistema operacional, sem o processo nem saber a respeito delas. A transição 2 ocorre quando o escalonador decide que o processo em andamento foi executado por tempo suficiente, e é o momento de deixar outro processo ter algum tempo de CPU. A transição 3 ocorre quando todos os outros processos tiveram sua parcela justa e está na hora de o primeiro processo chegar à CPU para ser executado novamente. O escalonamento, isto é, decidir qual processo deve ser executado, quando e por quanto tempo, é um assunto importante; nós o examinaremos mais adiante neste capítulo. Muitos algoritmos foram desenvolvidos para tentar equilibrar as demandas concorrentes de eficiência para o sistema como um todo e justiça para os processos individuais. Estudaremos algumas delas ainda neste capítulo.

A transição 4 se verifica quando o evento externo pelo qual um processo estava esperando (como a chegada de alguma entrada) acontece. Se nenhum outro processo estiver sendo executado naquele instante, a transição 3 será desencadeada e o processo começará a ser executado. Caso contrário, ele talvez tenha de esperar no estado de *pronto* por um intervalo curto até que a CPU esteja disponível e chegue sua vez.

Usando o modelo de processo, torna-se muito mais fácil pensar sobre o que está acontecendo dentro do sistema. Alguns dos processos executam programas que levam adiante comandos digitados pelo usuário. Outros processos são parte do sistema e lidam com tarefas como levar adiante solicitações para serviços de arquivos ou gerenciar os detalhes do funcionamento de um acionador de disco ou fita. Quando ocorre uma interrupção de disco, o sistema toma uma decisão para parar de executar o processo atual e executa o processo de disco, que foi bloqueado esperando por essa interrupção. Assim, em vez de pensar a respeito de interrupções, podemos pensar sobre os processos de usuários, processos de disco, processos terminais e assim por diante, que bloqueiam quando estão esperando que algo aconteça. Quando o disco foi lido ou o caractere digitado, o processo esperando por ele é desbloqueado e está disponível para ser executado novamente.

Essa visão dá origem ao modelo mostrado na Figura 2.3. Nele, o nível mais baixo do sistema operacional é o escalonador, com uma variedade de processos acima dele. Todo o tratamento de interrupções e detalhes sobre o início e parada de processos estão ocultos naquilo que é chamado aqui de escalonador, que, na verdade, não tem muito código. O resto do sistema operacional é bem estruturado na forma de processos. No entanto, poucos sistemas reais são tão bem estruturados como esse.

**FIGURA 2.3** O nível mais baixo de um sistema operacional estruturado em processos controla interrupções e escalonamento. Acima desse nível estão processos sequenciais.




## 2.1.6 Implementação de processos

Para implementar o modelo de processos, o sistema operacional mantém uma tabela (um arranjo de estruturas) chamada de **tabela de processos**, com uma entrada para cada um deles. (Alguns autores chamam essas entradas de **blocos de controle de processo**.) Essas entradas contêm informações importantes sobre o estado do processo, incluindo o seu contador de programa, ponteiro de pilha, alocação de memória, estado dos arquivos abertos, informação sobre sua contabilidade e escalonamento e tudo o mais que deva ser salvo quando o processo é trocado do estado *em execução* para *pronto* ou *bloqueado*, de maneira que ele possa ser reiniciado mais tarde como se nunca tivesse sido parado.

A Figura 2.4 mostra alguns dos campos fundamentais em um sistema típico: os campos na primeira coluna relacionam-se ao gerenciamento de processo. Os outros dois relacionam-se ao gerenciamento de memória e de arquivos, respectivamente. Deve-se observar que precisamente quais campos cada tabela de processo tem é algo altamente dependente do sistema, mas esse número dá uma ideia geral dos tipos de informações necessárias.

Agora que examinamos a tabela de processo, é possível explicar um pouco mais sobre como a ilusão de múltiplos processos sequenciais é mantida em uma (ou cada) CPU. Associada com cada classe de E/S há um local (geralmente em um local fixo próximo da parte inferior da memória) chamado de **vetor de interrupção**. Ele contém o endereço da rotina de serviço de interrupção. Suponha que o processo do usuário 3 esteja sendo executado quando ocorre uma interrupção de disco. O contador de programa do processo do usuário 3, palavra de estado de programa, e, às vezes, um ou mais registradores são colocados na pilha (atual) pelo hardware de interrupção. O computador, então, desvia a execução

**FIGURA 2.4** Alguns dos campos de uma entrada típica na tabela de processos.

| Gerenciamento de processo | Gerenciamento de memória | Gerenciamento de arquivo |
|---|---|---|
| Registros | Ponteiro para informações sobre o segmento de texto | Diretório-raiz |
| Contador de programa | | Diretório de trabalho |
| Palavra de estado do programa | Ponteiro para informações sobre o segmento de dados | Descritores de arquivo |
| Ponteiro da pilha | | ID do usuário |
| Estado do processo | Ponteiro para informações sobre o segmento de pilha | ID do grupo |
| Prioridade | | |
| Parâmetros de escalonamento | | |
| ID do processo | | |
| Processo pai | | |
| Grupo de processo | | |
| Sinais | | |
| Momento em que um processo foi iniciado | | |
| Tempo de CPU usado | | |
| Tempo de CPU do processo filho | | |
| Tempo do alarme seguinte | | |

para o endereço especificado no vetor de interrupção. Isso é tudo o que o hardware faz. Daqui em diante, é papel do software, em particular, realizar a rotina do serviço de interrupção.

Todas as interrupções começam salvando os registradores, muitas vezes na entrada da tabela de processo para o processo atual. Então a informação empurrada para a pilha pela interrupção é removida e o ponteiro de pilha é configurado para apontar para uma pilha temporária usada pelo tratador de processos. Ações como salvar os registradores e configurar o ponteiro da pilha não podem ser expressas em linguagens de alto nível, como C, por isso elas são desempenhadas por uma pequena rotina de linguagem de montagem, normalmente a mesma para todas as interrupções, já que o trabalho de salvar os registros é idêntico, não importa qual seja a causa da interrupção.

Quando essa rotina é concluída, ela chama uma rotina C para fazer o resto do trabalho para esse tipo específico de interrupção. (Presumimos que o sistema operacional seja escrito em C, a escolha mais comum para todos os sistemas operacionais reais). Quando o trabalho tiver sido concluído, possivelmente deixando algum processo agora pronto, o escalonador é chamado para ver qual é o próximo processo a ser executado. Depois disso, o controle é passado de volta ao código de linguagem de montagem para carregar os registradores e mapa de memória para o processo agora atual e iniciar a sua execução. O tratamento e o escalonamento de interrupção estão resumidos na Figura 2.5. Vale a pena observar que os detalhes variam de alguma maneira de sistema para sistema.

**FIGURA 2.5** O esqueleto do que o nível mais baixo do sistema operacional faz quando ocorre uma interrupção.

1. O hardware empilha o contador de programa etc.
2. O hardware carrega o novo contador de programa a partir do arranjo de interrupções.
3. O vetor de interrupções em linguagem de montagem salva os registradores.
4. O procedimento em linguagem de montagem configura uma nova pilha.
5. O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).
6. O escalonador decide qual processo é o próximo a executar.
7. O procedimento em C retorna para o código em linguagem de montagem.
8. O procedimento em linguagem de montagem inicia o novo processo atual.

Um processo pode ser interrompido milhares de vezes durante sua execução, mas a ideia fundamental é que, após cada interrupção, o processo retorne precisamente para o mesmo estado em que se encontrava antes de ser interrompido.

### 2.1.7 Modelando a multiprogramação

Quando a multiprogramação é usada, a utilização da CPU pode ser aperfeiçoada. Colocando a questão de maneira direta, se o processo médio realiza computações apenas 20% do tempo em que está na memória, então com cinco processos ao mesmo tempo na memória, a CPU deve estar ocupada o tempo inteiro. Entretanto, esse modelo é irrealisticamente otimista, tendo em vista que ele presume de modo tácito que todos os cinco processos jamais estarão esperando por uma E/S ao mesmo tempo.

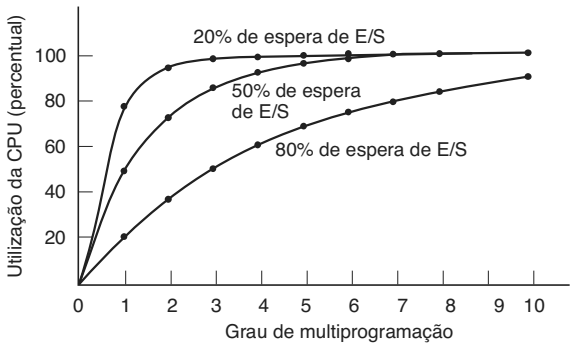
Um modelo melhor é examinar o uso da CPU a partir de um ponto de vista probabilístico. Suponha que um processo passe uma fração $p$ de seu tempo esperando que os dispositivos de E/S sejam concluídos. Com $n$ processos na memória ao mesmo tempo, a probabilidade de que todos os processos $n$ estejam esperando para E/S (caso em que a CPU estará ociosa) é $p^n$. A utilização da CPU é então dada pela fórmula

$$\text{Utilização da CPU} = 1 - p^n$$

A Figura 2.6 mostra a utilização da CPU como uma função de $n$, que é chamada de **grau de multiprogramação**.

Segundo a figura, fica claro que se os processos passam 80% do tempo esperando por dispositivos de E/S, pelo menos 10 processos devem estar na memória ao mesmo tempo para que a CPU desperdice menos de 10%. Quando você percebe que um processo interativo esperando por um usuário para digitar algo em um terminal (ou clicar em um ícone) está no estado de espera

**FIGURA 2.6** Utilização da CPU como uma função do número de processos na memória.



de E/S, deve ficar claro que tempos de espera de E/S de 80% ou mais não são incomuns. Porém mesmo em servidores, processos executando muitas operações de E/S em disco muitas vezes terão essa percentagem ou mais.

Levando em consideração a precisão, deve ser destacado que o modelo probabilístico descrito há pouco é apenas uma aproximação. Ele presume implicitamente que todos os *n* processos são independentes, significando que é bastante aceitável para um sistema com cinco processos na memória ter três em execução e dois esperando. Mas com uma única CPU, não podemos ter três processos sendo executados ao mesmo tempo, portanto o processo que ficar pronto enquanto a CPU está ocupada terá de esperar. Então, os processos não são independentes. Um modelo mais preciso pode ser construído usando a teoria das filas, mas o ponto que estamos sustentando — a multiprogramação deixa que os processos usem a CPU quando ela estaria em outras circunstâncias ociosa — ainda é válido, mesmo que as verdadeiras curvas da Figura 2.6 sejam ligeiramente diferentes daquelas mostradas na imagem.

Embora o modelo da Figura 2.6 seja bastante simples, ele pode ser usado para realizar previsões específicas, embora aproximadas, a respeito do desempenho da CPU. Suponha, por exemplo, que um computador tenha 8 GB de memória, com o sistema operacional e suas tabelas ocupando 2 GB e cada programa de usuário também ocupando 2 GB. Esses tamanhos permitem que três programas de usuários estejam na memória simultaneamente. Com uma espera de E/S média de 80%, temos uma utilização de CPU (ignorando a sobrecarga do sistema operacional) de $1 - 0{,}8^3$ ou em torno de 49%. Acrescentar outros 8 GB de memória permite que o sistema aumente seu grau de multiprogramação de três para sete, aumentando desse modo a utilização da CPU para 79%. Em outras palavras, os 8 GB adicionais aumentarão a utilização da CPU em 30%.

Acrescentar outros 8 GB ainda aumentaria a utilização da CPU apenas de 79% para 91%, desse modo elevando a utilização da CPU em apenas 12% a mais. Usando esse modelo, o proprietário do computador pode decidir que a primeira adição foi um bom investimento, mas a segunda, não.

## 2.2 Threads

Em sistemas operacionais tradicionais, cada processo tem um espaço de endereçamento e um único thread de controle. Na realidade, essa é quase a definição de um processo. Não obstante isso, em muitas situações, é desejável ter múltiplos threads de controle no mesmo espaço de endereçamento executando em quase paralelo, como se eles fossem (quase) processos separados (exceto pelo espaço de endereçamento compartilhado). Nas seções a seguir, discutiremos essas situações e suas implicações.

### 2.2.1 Utilização de threads

Por que alguém iria querer ter um tipo de processo dentro de um processo? Na realidade, há várias razões para a existência desses miniprocessos, chamados **threads**. Vamos examinar agora algumas delas. A principal razão para se ter threads é que em muitas aplicações múltiplas atividades estão ocorrendo simultaneamente e algumas delas podem bloquear de tempos em tempos. Ao decompormos uma aplicação dessas em múltiplos threads sequenciais que são executados em quase paralelo, o modelo de programação torna-se mais simples.

Já vimos esse argumento antes. É precisamente o argumento para se ter processos. Em vez de pensar a respeito de interrupções, temporizadores e chaveamentos de contextos, podemos pensar a respeito de processos em paralelo. Apenas agora com os threads acrescentamos um novo elemento: a capacidade para as entidades em paralelo compartilharem um espaço de endereçamento e todos os seus dados entre si. Essa capacidade é essencial para determinadas aplicações, razão pela qual ter múltiplos processos (com seus espaços de endereçamento em separado) não funcionará.

Um segundo argumento para a existência dos threads é que como eles são mais leves do que os processos, eles são mais fáceis (isto é, mais rápidos) para criar e destruir do que os processos. Em muitos sistemas, criar um thread é algo de 10 a 100 vezes mais rápido do que criar um processo. Quando o número necessário de threads muda dinâmica e rapidamente, é útil se contar com essa propriedade.

Uma terceira razão para a existência de threads também é o argumento do desempenho. O uso de threads não resulta em um ganho de desempenho quando todos eles são limitados pela CPU, mas quando há uma computação substancial e também E/S substancial, contar com threads permite que essas atividades se sobreponham, acelerando desse modo a aplicação.

Por fim, threads são úteis em sistemas com múltiplas CPUs, onde o paralelismo real é possível. Voltaremos a essa questão no Capítulo 8.

É mais fácil ver por que os threads são úteis observando alguns exemplos concretos. Como um primeiro