Capítulo 2
Processos e threads

Vamos agora iniciar um estudo detalhado sobre 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 depende desse conceito e é importante que o projetista (e o estudante) de sistemas operacionais tenha um entendimento completo do que é um processo, o mais cedo possível.

Processos são uma das mais antigas e importantes abstrações que o sistema operacional oferece. Eles mantêm a capacidade de operações (pseudo)concorrentes, mesmo quando há apenas uma CPU disponível. Eles transformam uma única CPU em múltiplas CPUs virtuais. Sem a abstração de processos, a ciência da computação moderna não existiria. Neste capítulo, examinaremos em detalhes processos e seus primos irmãos, os threads.

2.1 Processos

Todos os computadores modernos são capazes de fazer várias coisas ao mesmo tempo. As pessoas acostumadas a trabalhar com computadores pessoais podem não estar completamente cientes desse fato; portanto, alguns exemplos podem torná-lo mais claro. Primeiro considere um servidor da Web. Solicitações de páginas da Web chegam de toda parte. Quando uma solicitação chega, o servidor verifica se a página necessária está na cache. Se estiver, é enviada de volta; se não, uma solicitação de acesso ao disco é iniciada para buscá-la. Entretanto, do ponto de vista da CPU, as solicitações de acesso ao disco duram uma eternidade. Enquanto espera que a solicitação de acesso ao disco seja concluída, muitas outras solicitações podem chegar. Se há múltiplos discos presentes, algumas delas ou todas elas podem ser enviadas rapidamente a outros discos muito antes de a primeira solicitação ser atendida. Evidentemente, é necessário algum modo de modelar e controlar essa simultaneidade. Os processos (e especialmente os threads) podem ajudar aqui.

Agora considere um usuário de PC. Quando o sistema é inicializado, muitos processos muitas vezes desconhecidos ao usuário começam secretamente. Por exemplo, um processo pode ser iniciado para espera de e-mails que chegam. Outro processo pode ser executado pelo programa de antivírus para verificar periodicamente se há novas definições de antivírus disponíveis. Além disso, processos de usuários explícitos podem estar sendo executados, imprimindo arquivos e gravando um CD-ROM, tudo enquanto o usuário está navegando na Web. Toda essa atividade tem de ser administrada, e um sistema multiprogramado que sustente múltiplos processos é bastante útil nesse caso.

Em qualquer sistema multiprogramado, a CPU chaveia de programa para programa, executando cada um deles por dezenas ou centenas de milissegundos. Estritamente falando, enquanto a cada instante a CPU executa somente um programa, no decorrer de um segundo ela pode trabalhar sobre vários programas, dando aos usuários a ilusão de paralelismo. Algumas vezes, nesse contexto, fala-se de **pseudoparalelismo**, para contrastar com o verdadeiro paralelismo de hardware dos sistemas **multiprocessadores** (que têm duas ou mais CPUs que compartilham simultaneamente a mesma memória física). Ter controle sobre múltiplas atividades em paralelo é algo difícil para as pessoas. Contudo, projetistas de sistemas operacionais vêm desenvolvendo ao longo dos anos um modelo conceitual (processos sequenciais) que facilita o paralelismo. Esse modelo, seu uso e algumas de suas consequências compõem o assunto deste capítulo.

2.1.1 O modelo de processo

Nesse modelo, todos os softwares que podem ser executados em um computador — inclusive, algumas vezes, o próprio sistema operacional — são organizados em vários **processos sequenciais** (ou, para simplificar, **processos**). Um processo é apenas um programa em execução, acompanhado dos valores atuais do contador de programa, dos registradores e das variáveis. Conceitualmente, cada processo tem sua própria CPU virtual. É claro que, na realidade, a CPU troca, a todo momento, de um processo para outro, mas, para entender o sistema, é muito mais fácil pensar em um conjunto de processos executando (pseudo) paralelamente do que tentar controlar o modo como a CPU faz esses chaveamentos. Esse mecanismo de trocas rápidas é chamado de **multiprogramação**, conforme visto no Capítulo 1.

Na Figura 2.1(a), vemos um computador multiprogramado com quatro programas na memória. Na Figura 2.1(b) estão quatro processos, cada um com seu próprio fluxo de controle (isto é, seu próprio contador de programa lógico) e executando independentemente dos outros. Claro, há somente um contador de programa físico, de forma que, quando cada processo é executado, seu contador de programa lógico é carregado no contador de programa real. Quando acaba o tempo de CPU alocado para um processo, o contador de programa físico é salvo no contador de programa lógico do processo na memória. Na Figura 2.1(c) vemos que, por um intervalo de tempo suficientemente longo, todos os processos estão avançando, mas, a cada instante, apenas um único processo está realmente executando.

Neste capítulo, supomos que haja apenas uma CPU. Cada vez mais, entretanto, essa suposição não é verdadeira, visto que os novos chips são muitas vezes multinúcleo (multicore), com duas, quatro ou mais CPUs. Examinaremos chips multinúcleo e multiprocessadores em geral no Capítulo 8, mas, por ora, é mais simples pensar em uma CPU de cada vez. Assim, quando dizemos que uma CPU pode de fato executar apenas um processo por vez, se houver dois núcleos (ou duas CPUs), cada um deles pode executar apenas um processo por vez.

Com o rápido chaveamento da CPU entre os processos, a taxa na qual o processo realiza sua computação não será uniforme e provavelmente não será nem reproduzível se os mesmos processos forem executados novamente. Desse modo, os processos não devem ser programados com hipóteses predefinidas sobre a temporização. Considere, por exemplo, um processo de E/S que inicia uma fita magnética para que sejam restaurados arquivos de backup; ele executa dez mil vezes um laço ocioso para aguardar que uma rotação seja atingida e então executa um comando para ler o primeiro registro. Se a CPU decidir chavear para um outro processo durante a execução do laço ocioso, o processo da fita pode não estar sendo executado quando a cabeça de leitura chegar ao primeiro registro. Quando um processo tem restrições críticas de tempo real como essas — isto é, eventos específicos devem ocorrer dentro de um intervalo de tempo prefixado de milissegundos —, é preciso tomar medidas especiais para que esses eventos ocorram. Contudo, em geral a maioria dos processos não é afetada pelo aspecto inerente de multiprogramação da CPU ou pelas velocidades relativas dos diversos processos.

A diferença entre um processo e um programa é sutil, mas crucial. Uma analogia pode ajudar. Imagine um cientista da computação com dotes culinários e que está assando um bolo de aniversário para sua filha. Ele tem uma receita de bolo de aniversário e uma cozinha bem suprida, com todos os ingredientes: farinha, ovos, açúcar, essência de baunilha, entre outros. Nessa analogia, a receita é o programa (isto é, um algoritmo expresso por uma notação adequada), o cientista é o processador (CPU) e os ingredientes do bolo são os dados de entrada. O processo é a atividade desempenhada pelo nosso confeiteiro de ler a receita, buscar os ingredientes e assar o bolo.

Agora imagine que o filho do cientista chegue chorando, dizendo que uma abelha o picou. O cientista registra onde ele estava na receita (o estado atual do processo é salvo), busca um livro de primeiros socorros e começa a seguir as instruções contidas nele. Nesse ponto, vemos que o processador está sendo alternado de um processo (assar o bolo) para um processo de prioridade mais alta (fornecer cuidados médicos), cada um em um programa diferente (receita *versus* livro de primeiros socorros). Quando a picada da abelha tiver sido tratada, o cientista voltará ao seu bolo, continuando do ponto em que parou.

A ideia principal é que um processo constitui uma atividade. Ele possui programa, entrada, saída e um estado. Um único processador pode ser compartilhado entre os vários processos, com algum algoritmo de escalonamento usado para determinar quando parar o trabalho sobre um processo e servir outro.

Convém notar que, se um programa está sendo executado duas vezes, isso conta como dois processos. Por exemplo, frequentemente 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

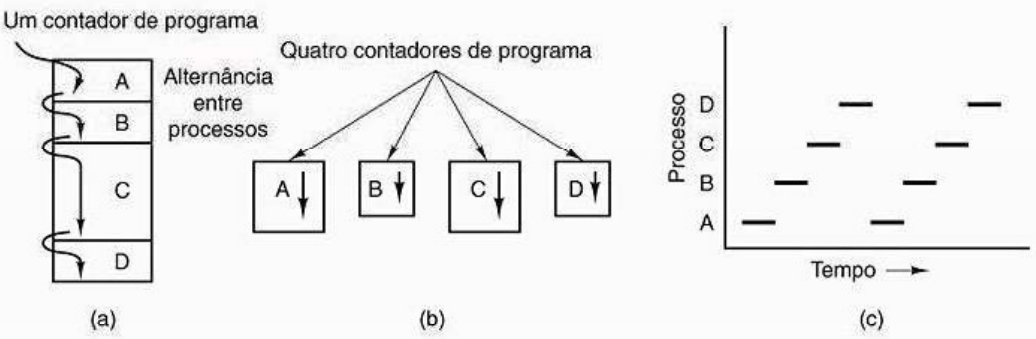

Figura 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Somente um programa está ativo a cada momento.

programa não importa; eles são processos diferentes. O sistema operacional pode compartilhar o código entre eles e, desse modo, apenas uma cópia está na memória, mas esse é um detalhe técnico que não altera a situação conceitual dos dois processos sendo executados.

2.1.2 Criação de processos

Os sistemas operacionais precisam de mecanismos para criar processos. Em sistemas muito simples, ou em sistemas projetados para executar apenas uma única aplicação (por exemplo, o controlador do forno de micro-ondas), pode ser possível que todos os processos que serão necessários sejam criados quando o sistema é ligado. Contudo, em sistemas de propósito geral, é necessário algum mecanismo para criar e terminar processos durante a operação, quando for preciso. Veremos agora alguns desses tópicos.

Há quatro eventos principais que fazem com que processos sejam criados:

1. Início do sistema.
2. Execução de uma chamada de sistema de criação de processo por um processo em execução.
3. Uma requisição do usuário para criar um novo processo.
4. Início de uma tarefa em lote (*batch job*).

Quando um sistema operacional é carregado, em geral criam-se vários processos. Alguns deles são processos em foreground (primeiro plano), ou seja, que interagem com usuários (humanos) e realizam tarefas para eles. Outros são processos em background (segundo plano), que não estão associados a usuários em particular, mas que apresentam alguma função específica. Por exemplo, um processo em background (segundo plano) pode ser designado a aceitar mensagens eletrônicas sendo recebidas, ficando inativo na maior parte do dia, mas surgindo de repente quando uma mensagem chega. Outro processo em background (segundo plano) pode ser destinado a aceitar solicitações que chegam para páginas da Web hospedadas naquela máquina, despertando quando uma requisição chega pedindo o serviço. Processos que ficam em background com a finalidade de lidar com alguma atividade como mensagem eletrônica, páginas da Web, notícias, impressão, entre outros, são chamados de **daemons**. É comum os grandes sistemas lançarem mão de dezenas deles. No UNIX, o programa *ps* pode ser usado para relacionar os processos que estão executando. No Windows, o gerenciador de tarefas pode ser usado.

Além dos processos criados durante a carga do sistema operacional, novos processos podem ser criados depois disso. Muitas vezes, um processo em execução fará chamadas de sistema (*system calls*) para criar um ou mais novos processos para ajudá-lo em seu trabalho. Criar novos processos é particularmente interessante quando a tarefa a ser executada puder ser facilmente dividida em vários processos relacionados, mas interagindo de maneira independente. Por exemplo, se uma grande quantidade de dados estiver sendo trazida via rede para que seja subsequentemente processada, poderá ser conveniente criar um processo para trazer esses dados e armazená-los em um local compartilhado da memória, enquanto um segundo processo remove os dados e os processa. Em um sistema multiprocessador, permitir que cada processo execute em uma CPU diferente também torna o trabalho mais rápido.

Em sistemas interativos, os usuários podem inicializar um programa digitando um comando ou clicando (duas vezes) um ícone. Cada uma dessas ações inicia um novo processo e executa nele o programa selecionado. Em sistemas UNIX baseados em comandos que executam o X, o novo processo toma posse da janela na qual ele foi disparado. No Microsoft Windows, quando um processo é disparado, ele não tem uma janela, mas pode criar uma (ou mais de uma), e a maioria deles cria. Nos dois sistemas, os usuários podem ter múltiplas janelas abertas ao mesmo tempo, cada uma executando algum processo. Usando o mouse, o usuário seleciona uma janela e interage com o processo — por exemplo, fornecendo a entrada quando for necessário.

A última situação na qual processos são criados aplica-se somente a sistemas em lote encontrados em computadores de grande porte. Nesses sistemas, usuários podem submeter (até remotamente) tarefas em lote para o sistema. Quando julgar que tem recursos para executar outra tarefa, o sistema operacional criará um novo processo e executará nele a próxima tarefa da fila de entrada.

Tecnicamente, em todos esses casos, um novo processo (processo filho) é criado por um processo existente (processo pai) executando uma chamada de sistema para a criação de processo. Esse processo (processo pai) pode ser um processo de usuário que está executando, um processo de sistema invocado a partir do teclado ou do mouse ou um processo gerenciador de lotes. O que o processo (pai) faz é executar uma chamada de sistema para criar um novo processo (filho) e assim indica, direta ou indiretamente, qual programa executar nele.

No UNIX, há somente uma chamada de sistema para criar um novo processo: fork. Essa chamada cria um clone idêntico ao processo que a chamou. Depois da 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 executa, em seguida, 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 *sort* no interpretador de comandos, este se bifurca gerando um processo filho, e o processo filho executa o *sort*. A razão para esse processo de dois passos é permitir que o filho manipule seus descritores de arquivos depois da fork, mas antes da execve, para conseguir redirecionar a entrada-padrão, a saída-padrão e a saída de erros-padrão.

Por outro lado, no Windows, uma única chamada de função do Win32, CreateProcess, trata tanto do pro-

cesso de criação quanto da carga do programa correto no novo processo. Essa chamada possui dez parâmetros, incluindo o programa a ser executado, os parâmetros da linha de comando que alimentam esse programa, vários atributos de segurança, os bits que controlam se os arquivos abertos são herdados, informação sobre prioridade, uma especificação da janela a ser criada para o processo (se houver) e um ponteiro para uma estrutura na qual a informação sobre o processo recém-criado é retornada para quem chamou. Além do CreateProcess, o Win32 apresenta cerca de cem outras funções para gerenciar e sincronizar processos e tópicos afins.

Tanto no UNIX quanto no Windows, depois que um processo é criado, o pai e o filho têm seus próprios espaços de endereçamento distintos. Se um dos dois processos alterar uma palavra em seu espaço de endereçamento, a mudança não será visível ao outro processo. No UNIX, o espaço de endereçamento inicial do filho é uma *cópia* do espaço de endereçamento do pai, mas há dois espaços de endereçamento distintos envolvidos; nenhuma memória para escrita é compartilhada (algumas implementações UNIX compartilham o código do programa entre os dois, já que não podem ser alteradas). Contudo, é possível que um processo recentemente criado compartilhe algum de seus recursos com o processo que o criou, como arquivos abertos. No Windows, os espaços de endereçamento do pai e do filho são diferentes desde o início.

2.1.3 Término de processos

Depois de criado, um processo começa a executar e faz seu trabalho. Contudo, nada é para sempre, nem mesmo os processos. Mais cedo ou mais tarde o novo processo terminará, normalmente em razão de alguma das seguintes condições:

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

Na maioria das vezes, os processos terminam porque fizeram seu trabalho. Quando acaba de compilar o programa atribuído a ele, o compilador executa uma chamada de sistema para dizer ao sistema operacional que ele terminou. Essa chamada é a exit no UNIX e a ExitProcess no Windows. Programas baseados em tela também suportam o término voluntário. Processadores de texto, visualizadores da Web (browsers) e programas similares sempre têm um ícone ou um item de menu no qual o usuário pode clicar para dizer ao processo que remova quaisquer arquivos temporários que ele tenha aberto e, então, termine.

O segundo motivo para término é 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 esse arquivo não existe, o compilador simplesmente termina a execução. Processos interativos com base em tela geralmente não fecham quando parâmetros errados são fornecidos. Em vez disso, uma caixa de diálogo emerge e pergunta ao usuário se ele quer tentar novamente.

A terceira razão para o término é um erro causado pelo processo, muitas vezes por um erro de programa. Entre os vários exemplos estão a execução de uma instrução ilegal, a referência à memória inexistente ou a divisão por zero. Em alguns sistemas (por exemplo, UNIX), um processo pode dizer ao sistema operacional que deseja, ele mesmo, tratar certos erros. Nesse caso, o processo é sinalizado (interrompido) em vez de finalizado pela ocorrência de erros.

A quarta razão pela qual um processo pode terminar se dá quando um processo executa uma chamada de sistema dizendo ao sistema operacional para cancelar algum outro processo. No UNIX, essa chamada é a kill. A função Win32 correspondente é a TerminateProcess. Em ambos os casos, o processo que for efetuar o cancelamento deve ter a autorização necessária para fazê-lo. Em alguns sistemas, quando um processo termina, voluntariamente ou não, todos os processos criados por ele também são imediatamente cancelados. Contudo, nem o UNIX nem 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, de certa maneira, associados. O próprio processo filho pode gerar mais processos, formando uma hierarquia de processos. Observe que isso é diferente do que ocorre com plantas e animais, que utilizam a reprodução sexuada, pois um processo tem apenas um pai (mas pode ter nenhum, um, dois ou mais filhos).

No UNIX, um processo, todos os seus filhos e descendentes formam um grupo de processos. Quando um usuário envia um sinal do teclado, o sinal é entregue a todos os membros do grupo de processos associado com o teclado (normalmente todos os processos ativos que foram criados na janela atual). Individualmente, cada processo pode capturar o sinal, ignorá-lo ou tomar a ação predefinida, isto é, ser finalizado pelo sinal.

Outro exemplo da atuação dessa hierarquia pode ser observado no início do UNIX, quando o computador é ligado. Um processo especial, chamado *init*, está presente na imagem de carga do sistema. Quando começa a executar, ele lê um arquivo dizendo quantos terminais existem. Então ele se bifurca várias vezes para ter um novo processo para cada terminal. Esses processos esperam por alguma conexão de usuário. Se algum usuário se conectar, o processo de conexão executará um interpretador de comandos para aceitar comandos dos usuários. Esses comandos podem iniciar mais processos, e assim por diante. Desse modo, todos os processos em todo o sistema pertencem a uma única árvore, com o *init* na raiz.

Por outro lado, o Windows não apresenta nenhum conceito de hierarquia de processos. Todos os processos são iguais. Algo parecido com uma hierarquia de processos ocorre somente quando um processo é criado. Ao pai é dado um identificador especial (chamado **handle**), que ele pode usar para controlar o filho. Contudo, ele é livre para passar esse identificador para alguns outros processos, invalidando, assim, a hierarquia. Os processos no 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, muitas vezes os processos precisam interagir com outros. Um processo pode gerar uma saída que outro processo usa como entrada. No interpretador de comandos,

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

o primeiro processo, que executa *cat*, gera como saída a concatenação dos três arquivos. O segundo processo, que executa *grep*, seleciona todas as linhas contendo a palavra 'tree'. Dependendo das velocidades relativas dos dois processos (atreladas tanto à complexidade relativa dos programas quanto ao tempo de CPU que cada um deteve), pode ocorrer que o *grep* esteja pronto para executar, mas não haja entrada para ele. Ele deve, então, bloquear até que alguma entrada esteja disponível.

Um processo bloqueia porque obviamente não pode prosseguir — em geral porque está esperando por uma entrada ainda não disponível. É possível também que um processo conceitualmente pronto e capaz de executar esteja bloqueado porque o sistema operacional decidiu alocar a CPU para outro processo por algum tempo. Essas duas condições são completamente diferentes. No primeiro caso, a suspensão é inerente ao problema (não se pode processar a linha de comando do usuário enquanto ele não digitar nada). O segundo é uma tecnicalidade do sistema (não há CPUs suficientes para dar a cada processo um processador exclusivo). Na Figura 2.2, podemos ver um diagrama de estados mostrando os três estados de um processo:

1. Em execução (realmente usando a CPU naquele instante).
2. Pronto (executável; temporariamente parado para dar lugar a outro processo).
3. Bloqueado (incapaz de executar enquanto não ocorrer um evento externo).

Logicamente, os dois primeiros estados são similares. Em ambos os casos o processo vai executar, só que no segundo não há, temporariamente, CPU disponível para ele. O terceiro estado é diferente dos dois primeiros, pois o processo não pode executar, mesmo que a CPU não tenha nada para fazer.

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

Figura 2.2 Um processo pode estar nos estados em execução, bloqueado ou pronto. As transições entre esses estados são mostradas.

Quatro transições são possíveis entre esses três estados, conforme se vê na figura. A transição 1 ocorre quando o sistema operacional descobre que um processo não pode prosseguir. Em alguns sistemas, o processo precisa executar uma chamada de sistema, como pause, para entrar no estado bloqueado. Em outros sistemas, inclusive no UNIX, quando um processo lê de um pipe ou de um arquivo especial (por exemplo, um terminal) e não há entrada disponível, o processo é automaticamente bloqueado.

As transições 2 e 3 são causadas pelo escalonador de processos — uma parte do sistema operacional —, sem que o processo saiba disso. A transição 2 ocorre quando o escalonador decide que o processo em execução já teve tempo suficiente de CPU e é momento de deixar outro processo ocupar o tempo da CPU. A transição 3 ocorre quando todos os outros processos já compartilharam a CPU, de uma maneira justa, e é hora de o primeiro processo obter novamente a CPU. O escalonamento — isto é, a decisão sobre quando e por quanto tempo cada processo deve executar — é um tópico muito importante e será estudado depois, neste mesmo capítulo. Muitos algoritmos vêm sendo desenvolvidos na tentativa de equilibrar essa competição, que exige eficiência para o sistema como um todo e igualdade para os processos individuais. Estudaremos alguns deles neste capítulo.

A transição 4 ocorre quando acontece um evento externo pelo qual um processo estava aguardando (como a chegada de alguma entrada). Se nenhum outro processo estiver executando naquele momento, a transição 3 será disparada e o processo começará a executar. Caso contrário, ele poderá ter de aguardar em estado de *pronto* por um pequeno intervalo de tempo, até que a CPU esteja disponível e chegue sua vez.

Com o modelo de processo, torna-se muito mais fácil saber o que está ocorrendo dentro do sistema. Alguns dos processos chamam programas que executam comandos digitados por um usuário. Outros processos são parte do sistema e manejam tarefas como fazer requisições por 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 a decisão de parar de executar o processo corrente e retomar o processo do disco que foi bloqueado para aguardar essa interrupção. Assim, em vez de pensar em interrupções, podemos pensar em processos de usuário, de disco, de terminais ou outros, que bloqueiam quando estão à espera de que algo aconteça. Finalizada a leitura do disco ou a digitação de um caractere, o processo que aguarda por isso é desbloqueado e torna-se disponível para executar novamente.

Essa visão dá origem ao modelo mostrado na Figura 2.3. Nele, o nível mais baixo do sistema operacional é o escalonador, com diversos processos acima dele. Todo o tratamento de interrupção e detalhes sobre a inicialização e o bloqueio de processos estão ocultos naquilo que é chamado aqui de escalonador, que, na verdade, não tem muito código. O restante do sistema operacional é bem estruturado na forma de processos. Contudo, poucos sistemas reais são tão bem estruturados como esse.

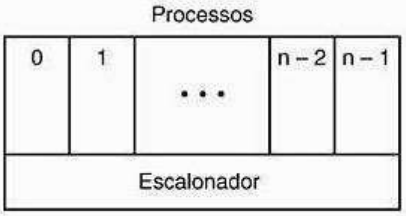

Figura 2.3 O nível mais baixo de um sistema operacional estruturado em processos controla interrupções e escalonamento. Acima desse nível estão processos sequenciais.

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 processo. (Alguns autores chamam essas entradas de **process control blocks** — blocos de controle de processo.) Essa entrada contém informações sobre o estado do processo, seu contador de programa, o ponteiro da pilha, a alocação de memória, os estados de seus arquivos abertos, sua informação sobre contabilidade e escalonamento e tudo o mais sobre o processo que deva ser salvo quando o processo passar do estado *em execução* para o estado *pronto* ou *bloqueado*, para que ele possa ser reiniciado depois, como se nunca tivesse sido bloqueado.

A Tabela 2.1 mostra alguns dos campos mais importantes de um sistema típico. Os campos na primeira coluna relacionam-se com o gerenciamento do processo. As outras duas colunas são relativas ao gerenciamento de memória e ao gerenciamento de arquivos, respectivamente. Deve-se observar que a exatidão dos campos da tabela de processos é altamente dependente do sistema, mas essa figura dá uma ideia geral dos tipos necessários de informação.

Agora que vimos a tabela de processos, é possível explicar um pouco mais sobre como é mantida a ilusão de múltiplos processos sequenciais, em uma máquina com uma (ou cada) CPU e muitos dispositivos de E/S. Associada a cada classe de dispositivos de E/S (por exemplo, discos flexíveis ou rígidos, temporizadores, terminais) está uma parte da memória (geralmente próxima da parte mais baixa da memória), chamada de **arranjo de interrupções**. Esse arranjo contém os endereços das rotinas dos serviços de interrupção. Suponha que o processo do usuário 3 esteja

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 texto	Descritores de arquivo
Ponteiro da pilha		ID do usuário
Estado do processo	Ponteiro para informações sobre o segmento de texto	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		

Tabela 2.1 Alguns dos campos de um processo típico de entrada na tabela.

executando quando ocorre uma interrupção de disco. O contador de programa do processo do usuário 3, palavra de status do programa e, possivelmente, um ou mais registradores são colocados na pilha (atual) pelo hardware de interrupção. O computador, então, desvia a execução para o endereço especificado no arranjo de interrupções. Isso é tudo o que hardware faz. Dali em diante, é papel do software, em particular, fazer a rotina de serviço da interrupção prosseguir.

Todas as interrupções começam salvando os registradores, muitas vezes na entrada da tabela de processos referente ao processo corrente. Então a informação colocada na pilha pela interrupção é removida e o ponteiro da pilha é alterado para que aponte para uma pilha temporária usada pelo manipulador dos processos (process handler). Ações como salvar os registradores e alterar o ponteiro de pilha não podem ser expressas em linguagens de alto nível como C. Assim, elas são implementadas por uma pequena rotina em linguagem assembly (linguagem de montagem). Normalmente é a mesma rotina para todas as interrupções, já que o trabalho de salvar os registradores é idêntico, não importando o que causou a interrupção.

Quando termina, a rotina assembly chama uma rotina em C para fazer o restante do trabalho desse tipo específico de interrupção. (Vamos supor que o sistema operacional esteja escrito em C, a escolha usual para todos os sistemas operacionais reais.) Quando essa tarefa acaba, possivelmente colocando algum processo em estado de 'pronto', o escalonador é chamado para verificar qual é o próximo processo a executar. Depois disso, o controle é passado de volta para o código em linguagem assembly para carregar os registradores e o mapa de memória do novo processo corrente e inicializar sua execução. O tratamento de interrupção e o escalonamento são resumidos na Tabela 2.2. Convém observar que os detalhes variam de sistema para sistema.

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 procedimento 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.

Tabela 2.2 O esqueleto do que o nível mais baixo do sistema operacional faz quando ocorre uma interrupção.

Quando o processo termina, o sistema operacional exibe um caractere de prompt (prontidão) e espera um novo comando. Quando recebe o comando, carrega um novo programa na memória, sobrescrevendo o primeiro.

2.1.7 | Modelando a multiprogramação

Quando a multiprogramação é usada, a utilização da CPU pode ser aumentada. De modo geral, se o processo médio computa apenas durante 20 por cento do tempo em que está na memória, com cinco processos na memória a cada vez, a CPU deveria estar ocupada o tempo todo. Esse modelo é otimista e pouco realista, entretanto, uma vez que supõe tacitamente que nenhum dos cinco processos estará esperando por dispositivos de E/S ao mesmo tempo.

Um modelo melhor é examinar o emprego da CPU do ponto de vista probabilístico. Imagine que um processo passe uma fração p de seu tempo esperando que os dispositivos de E/S sejam concluídos. Com n processos na memória simultaneamente, a probabilidade de que todos os n processos estejam esperando por dispositivos de E/S (caso no qual a CPU estaria ociosa) é p^n. A utilização da CPU é, portanto, dada pela fórmula

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

A Figura 2.4 mostra a utilização da CPU como função de n, que é chamada de **grau de multiprogramação**.

De acordo com a figura, fica claro que, se os processos passam 80 por cento de seu tempo esperando por dispositivos de E/S, pelo menos dez processos devem estar na memória simultaneamente para que a CPU desperdice menos de 10 por cento. Se você já notou que um processo interativo esperando que um usuário digite algo em um terminal está em estado de espera de E/S, então deveria ficar claro que tempos de espera de E/S de 80 por cento ou mais não são incomuns. Mas, mesmo nos servidores, os processos executando muitas operações de E/S em discos muitas vezes terão porcentagem igual ou superior a essa.

Para garantir exatidão completa, deve-se assinalar que o modelo probabilístico descrito é apenas uma aproxima-

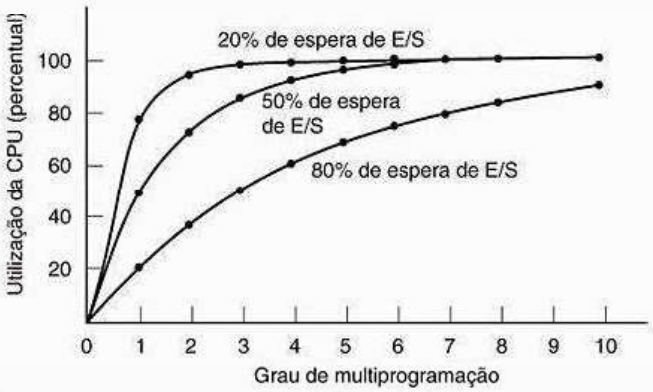

Figura 2.4 Utilização da CPU como função do número de processos na memória.

ção. Ele supõe implicitamente que todos os processos *n* são independentes, o que significa que é bastante aceitável que um sistema com cinco processos em memória tenha três sendo executados e dois esperando. Mas, com uma única CPU, não podemos ter três processos sendo executados ao mesmo tempo, de forma que um processo que foi para o estado 'pronto' enquanto a CPU está ocupada terá de esperar. Desse modo, os processos não são independentes. Um modelo mais preciso pode ser construído utilizando a teoria das filas, mas o nosso argumento — a multiprogramação permite que os processos usem a CPU quando, em outras circunstâncias, ela se tornaria ociosa — ainda é, naturalmente, válido, mesmo que as curvas verdadeiras da Figura 2.4 sejam ligeiramente diferentes das mostradas na figura.

Embora muito simples, o modelo da Figura 2.4 pode, mesmo assim, ser usado para previsões específicas, ainda que aproximadas, de desempenho da CPU. Suponha, por exemplo, que um computador tenha 512 MB de memória, com um sistema operacional que use 128 MB, e que cada programa de usuário também empregue 128 MB. Esses tamanhos possibilitam que três programas de usuário estejam simultaneamente na memória. Considerando-se que, em média, um processo passa 80 por cento de seu tempo em espera por E/S, tem-se uma utilização da CPU (ignorando o gasto extra — overhead — causado pelo sistema operacional) de $1 - 0{,}8^3$, ou cerca de 49 por cento. A adição de mais 512 MB de memória permite que o sistema aumente seu grau de multiprogramação de 3 para 7, elevando assim a utilização da CPU para 79 por cento. Em outras palavras, a adição de 512 MB aumentará a utilização da CPU em 30 por cento.

Adicionando ainda outros 512 MB, a utilização da CPU aumenta apenas de 79 por cento para 91 por cento, elevando, dessa forma, a utilização da CPU em apenas 12 por cento. Esse modelo permite que o dono de um computador decida que a primeira adição de memória é um bom investimento, mas não a segunda.

2.2 Threads

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

2.2.1 O uso de thread

Por que alguém desejaria ter um tipo de processo dentro de um processo? Constata-se que há várias razões para existirem esses miniprocessos, chamados **threads**. Examinemos alguns deles agora. A principal razão para existirem threads é que em muitas aplicações ocorrem múltiplas atividades ao mesmo tempo. Algumas dessas atividades podem ser bloqueadas de tempos em tempos. O modelo de programação se torna mais simples se decompomos uma aplicação em múltiplos threads sequenciais que executam em quase paralelo.

Já vimos esse argumento antes. É precisamente o mesmo argumento para a existência dos processos. Em vez de pensarmos em interrupções, temporizadores e chaveamento de contextos, podemos pensar em processos paralelos. Só que agora, com os threads, adicionamos um novo elemento: a capacidade de entidades paralelas compartilharem de um espaço de endereçamento e todos os seus dados entre elas mesmas. Isso é essencial para certas aplicações, nas quais múltiplos processos (com seus espaços de endereçamento separados) não funcionarão.

Um segundo argumento para a existência de threads é que eles são mais fáceis (isto é, mais rápidos) de criar e destruir que os processos, pois não têm quaisquer recursos associados a eles. Em muitos sistemas, criar um thread é cem vezes mais rápido do que criar um processo. É útil ter essa propriedade quando o número de threads necessários se altera dinâmica e rapidamente.

Uma terceira razão é também um argumento de desempenho. O uso de threads não resulta em ganho de desempenho quando todos eles são CPU-bound (limitados pela CPU, isto é, muito processamento com pouca E/S). No entanto, quando há grande quantidade de computação e de E/S, os threads permitem que essas atividades se sobreponham e, desse modo, aceleram a aplicação.

Finalmente, os threads são úteis em sistemas com múltiplas CPUs, para os quais o paralelismo real é possível. Voltaremos a esse assunto no Capítulo 8.

A maneira mais fácil de perceber a utilidade dos threads é apresentar exemplos concretos. Como um primeiro exemplo, considere um processador de textos. A maioria dos processadores de texto mostra o documento em criação na tela, formatado exatamente como ele aparecerá em uma página impressa. Mais especificamente, todas as quebras de linha e de página estão na posição correta e final para que o usuário possa conferi-las e alterar o documento, se for necessário (por exemplo, eliminar linhas viúvas e órfãs — linhas incompletas no início e no final de uma página, que são consideradas esteticamente desagradáveis).

Suponha que o usuário esteja escrevendo um livro. Do ponto de vista do autor, é mais fácil manter o livro inteiro como um arquivo único para tornar mais fácil a busca por tópicos, realizar substituições gerais e assim por diante. Mas há a alternativa de cada capítulo constituir um arquivo separado. Contudo, ter cada seção e subseção como um arquivo separado constitui um sério problema quando é necessário fazer alterações globais em todo o livro, já que, para isso, centenas de arquivos deverão ser editados indi-