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

exemplo, considere um processador de texto. Processadores de texto em geral exibem o documento que está sendo criado em uma tela formatada exatamente como aparecerá na página impressa. Em particular, todas as quebras de linha e quebras de página estão em suas posições finais e corretas, de maneira que o usuário pode inspecioná-las e mudar o documento se necessário (por exemplo, eliminar viúvas e órfãos — linhas incompletas no início e no fim das páginas, que são consideradas esteticamente desagradáveis).

Suponha que o usuário esteja escrevendo um livro. Do ponto de vista de um autor, é mais fácil manter o livro inteiro como um único arquivo para tornar mais fácil buscar por tópicos, realizar substituições globais e assim por diante. Como alternativa, cada capítulo pode ser um arquivo em separado. No entanto, ter cada seção e subseção como um arquivo em separado é um verdadeiro inconveniente quando mudanças globais precisam ser feitas para o livro inteiro, visto que centenas de arquivos precisam ser individualmente editados, um de cada vez. Por exemplo, se o padrão xxxx proposto é aprovado um pouco antes de o livro ser levado para impressão, todas as ocorrências de "Padrão provisório xxxx" têm de ser modificadas para "Padrão xxxx" no último minuto. Se o livro inteiro for um arquivo, em geral um único comando pode realizar todas as substituições. Em comparação, se o livro estiver dividido em mais de 300 arquivos, cada um deve ser editado separadamente.

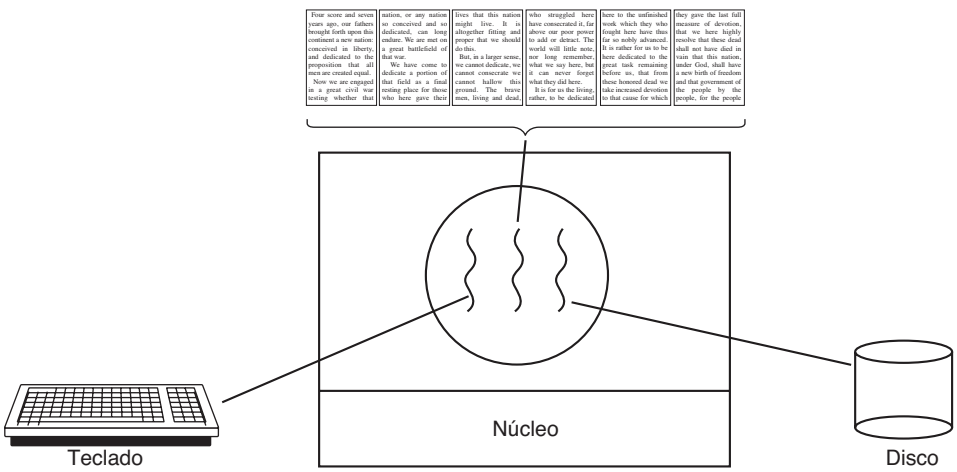
Agora considere o que acontece quando o usuário subitamente apaga uma frase da página 1 de um livro de 800 páginas. Após conferir a página modificada para assegurar-se de que está corrigida, ele agora quer fazer outra mudança na página 600 e digita um comando dizendo ao processador de texto para ir até aquela página (possivelmente procurando por uma frase ocorrendo apenas ali). O processador de texto agora é forçado a reformatar o livro inteiro até a página 600, algo difícil, pois ele não sabe qual será a primeira linha da página 600 até ter processado todas as páginas anteriores. Pode haver um atraso substancial antes que a página 600 seja exibida, resultando em um usuário infeliz.

Threads podem ajudar aqui. Suponha que o processador de texto seja escrito como um programa com dois threads. Um thread interage com o usuário e o outro lida com a reformatação em segundo plano. Tão logo a frase é apagada da página 1, o thread interativo diz ao de reformatação para reformatar o livro inteiro. Enquanto isso, o thread interativo continua a ouvir o teclado e o mouse e responde a comandos simples como rolar a página 1 enquanto o outro thread está trabalhando com afinco no segundo plano. Com um pouco de sorte, a reformatação será concluída antes que o usuário peça para ver a página 600, então ela pode ser exibida instantaneamente.

Enquanto estamos nesse exemplo, por que não acrescentar um terceiro thread? Muitos processadores de texto têm a capacidade de salvar automaticamente o arquivo inteiro para o disco em intervalos de poucos minutos para proteger o usuário contra o perigo de perder um dia de trabalho caso o programa ou o sistema trave ou falte luz. O terceiro thread pode fazer *backups* de disco sem interferir nos outros dois. A situação com os três threads é mostrada na Figura 2.7.

Se o programa tivesse apenas um thread, então sempre que um *backup* de disco fosse iniciado, comandos do teclado e do mouse seriam ignorados até que o *backup* tivesse sido concluído. O usuário certamente perceberia isso como um desempenho lento. Como alternativa,

FIGURA 2.7 Um processador de texto com três threads.



eventos do teclado e do mouse poderiam interromper o *backup* do disco, permitindo um bom desempenho, mas levando a um modelo de programação complexo orientado à interrupção. Com três threads, o modelo de programação é muito mais simples: o primeiro thread apenas interage com o usuário, o segundo reformata o documento quando solicitado, o terceiro escreve os conteúdos da RAM para o disco periodicamente.

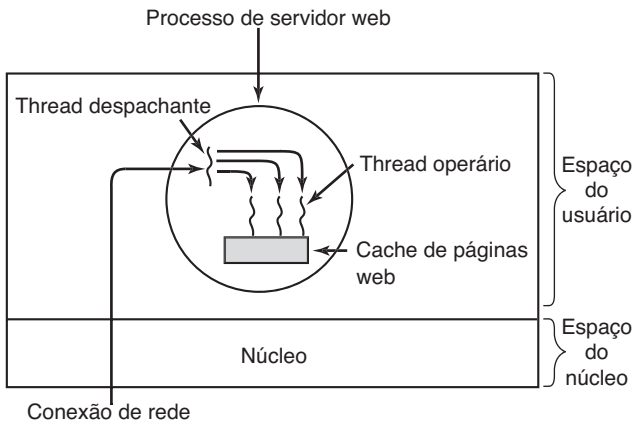
Deve ficar claro que ter três processos em separado não funcionaria aqui, pois todos os três threads precisam operar no documento. Ao existirem três threads em vez de três processos, eles compartilham de uma memória comum e desse modo têm acesso ao documento que está sendo editado. Com três processos isso seria impossível.

Uma situação análoga existe com muitos outros programas interativos. Por exemplo, uma planilha eletrônica é um programa que permite a um usuário manter uma matriz, na qual alguns elementos são dados fornecidos pelo usuário e outros são calculados com base nos dados de entrada usando fórmulas potencialmente complexas. Quando um usuário muda um elemento, muitos outros precisam ser recalculados. Ao ter um thread de segundo plano para o recálculo, o thread interativo pode permitir ao usuário fazer mudanças adicionais enquanto o cálculo está sendo realizado. De modo similar, um terceiro thread pode cuidar sozinho dos backups periódicos para o disco.

Agora considere mais um exemplo onde os threads são úteis: um servidor para um website. Solicitações para páginas chegam e a página solicitada é enviada de volta para o cliente. Na maioria dos websites, algumas páginas são mais acessadas do que outras. Por exemplo, a página principal da Sony é acessada muito mais do que uma página mais profunda na árvore contendo as especificações técnicas de alguma câmera em particular. Servidores da web usam esse fato para melhorar o desempenho mantendo uma coleção de páginas intensamente usadas na memória principal para eliminar a necessidade de ir até o disco para buscá-las. Essa coleção é chamada de **cache** e é usada em muitos outros contextos também. Vimos caches de CPU no Capítulo 1, por exemplo.

Uma maneira de organizar o servidor da web é mostrada na Figura 2.8(a). Aqui, um thread, o **despachante**, lê as requisições de trabalho que chegam da rede. Após examinar a solicitação, ele escolhe um **thread operário** ocioso (isto é, bloqueado) e passa para ele a solicitação, possivelmente escrevendo um ponteiro para a mensagem em uma palavra especial associada com cada thread. O despachante então acorda o operário adormecido, movendo-o do estado bloqueado para o estado pronto.

FIGURA 2.8 Um servidor web multithread.



Quando o operário desperta, ele verifica se a solicitação pode ser satisfeita a partir do cache da página da web, ao qual todos os threads têm acesso. Se não puder, ele começa uma operação read para conseguir a página do disco e é bloqueado até a operação de disco ser concluída. Quando o thread é bloqueado na operação de disco, outro thread é escolhido para ser executado, talvez o despachante, a fim de adquirir mais trabalho, ou possivelmente outro operário esteja pronto para ser executado agora.

Esse modelo permite que o servidor seja escrito como uma coleção de threads sequenciais. O programa do despachante consiste em um laço infinito para obter requisições de trabalho e entregá-las a um operário. Cada código de operário consiste em um laço infinito que aceita uma solicitação de um despachante e confere a cache da web para ver se a página está presente. Se estiver, ela é devolvida ao cliente, e o operário é bloqueado aguardando por uma nova solicitação. Se não estiver, ele pega a página do disco, retorna-a ao cliente e é bloqueado esperando por uma nova solicitação.

Um esquema aproximado do código é dado na Figura 2.9. Aqui, como no resto deste livro, *TRUE* é presumido que seja a constante 1. Do mesmo modo, *buf* e *page* são estruturas apropriadas para manter uma solicitação de trabalho e uma página da web, respectivamente.

Considere como o servidor web teria de ser escrito na ausência de threads. Uma possibilidade é fazê-lo operar um único thread. O laço principal do servidor web recebe uma solicitação, examina-a e a conduz até sua conclusão antes de receber a próxima. Enquanto espera pelo disco, o servidor está ocioso e não processa nenhum outro pedido chegando. Se o servidor web estiver sendo executado em uma máquina dedicada, como é o caso no geral, a CPU estará simplesmente ociosa

FIGURA 2.9 Um esquema aproximado do código para a Figura 2.8. (a) Thread despachante. (b) Thread operário.

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
     if (page_not_in_cache(&page))
         read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

enquanto o servidor estiver esperando pelo disco. O resultado final é que muito menos solicitações por segundo poderão ser processadas. Assim, threads ganham um desempenho considerável, mas cada thread é programado sequencialmente, como de costume.

Até o momento, vimos dois projetos possíveis: um servidor web multithread e um servidor web com um único thread. Suponha que múltiplos threads não estejam disponíveis, mas que os projetistas de sistemas consideram inaceitável a perda de desempenho decorrente do único thread. Se uma versão da chamada de sistema read sem bloqueios estiver disponível, uma terceira abordagem é possível. Quando uma solicitação chegar, o único thread a examina. Se ela puder ser satisfeita a partir da cache, ótimo, se não, uma operação de disco sem bloqueios é inicializada.

O servidor registra o estado da solicitação atual em uma tabela e então lida com o próximo evento. O próximo evento pode ser uma solicitação para um novo trabalho ou uma resposta do disco sobre uma operação anterior. Se for um novo trabalho, esse trabalho é iniciado. Se for uma resposta do disco, a informação relevante é buscada da tabela e a resposta processada. Com um sistema de E/S de disco sem bloqueios, uma resposta provavelmente terá de assumir a forma de um sinal ou interrupção.

Nesse projeto, o modelo de "processo sequencial" que tínhamos nos primeiros dois casos é perdido. O estado da computação deve ser explicitamente salvo e restaurado na tabela toda vez que o servidor chaveia do trabalho de uma solicitação para outra. Na realidade, estamos simulando os threads e suas pilhas do jeito mais difícil. Um projeto como esse, no qual cada computação tem um estado salvo e existe algum conjunto de eventos que pode ocorrer para mudar o estado, é chamado de **máquina de estados finitos**. Esse conceito é amplamente usado na ciência de computação.

Deve estar claro agora o que os threads têm a oferecer. Eles tornam possível reter a ideia de processos sequenciais que fazem chamadas bloqueantes (por exemplo, para E/S de disco) e ainda assim alcançar o paralelismo. Chamadas de sistema bloqueantes tornam a programação mais fácil, e o paralelismo melhora o desempenho. O servidor de thread único retém a simplicidade das chamadas de sistema bloqueantes, mas abre mão do desempenho. A terceira abordagem alcança um alto desempenho por meio do paralelismo, mas usa chamadas não bloqueantes e interrupções, e assim é difícil de programar. Esses modelos são resumidos na Figura 2.10.

Um terceiro exemplo em que threads são úteis encontra-se nas aplicações que precisam processar grandes quantidades de dados. Uma abordagem normal é ler em um bloco de dados, processá-lo e então escrevê-lo de novo. O problema aqui é que se houver apenas a disponibilidade de chamadas de sistema bloqueantes, o processo é bloqueado enquanto os dados estão chegando e saindo. Ter uma CPU ociosa quando há muita computação a ser feita é um claro desperdício e deve ser evitado se possível.

Threads oferecem uma solução: o processo poderia ser estruturado com um thread de entrada, um de processamento e um de saída. O thread de entrada lê dados para um buffer de entrada; o thread de processamento pega os dados do buffer de entrada, processa-os e coloca os resultados no buffer de saída; e o thread de saída escreve esses resultados de volta para o disco. Dessa maneira, entrada, saída e processamento podem estar todos acontecendo ao mesmo tempo. É claro que esse modelo funciona somente se uma chamada de sistema bloqueia apenas o thread de chamada, não o processo inteiro.

FIGURA 2.10 Três maneiras de construir um servidor.

Modelo	Características
Threads	Paralelismo, chamadas de sistema bloqueantes
Processo monothread	Não paralelismo, chamadas de sistema bloqueantes
Máquina de estados finitos	Paralelismo, chamadas não bloqueantes, interrupções

2.2.2 O modelo de thread clássico

Agora que vimos por que os threads podem ser úteis e como eles podem ser usados, vamos investigar a ideia um pouco mais de perto. O modelo de processo é baseado em dois conceitos independentes: agrupamento de recursos e execução. Às vezes é útil separá-los; é onde os threads entram. Primeiro, examinaremos o modelo de thread clássico; depois disso veremos o modelo de thread Linux, que torna indistintas as diferenças entre processos e threads.

Uma maneira de se ver um processo é que ele é um modo para agrupar recursos relacionados. Um processo tem um espaço de endereçamento contendo o código e os dados do programa, assim como outros recursos. Esses recursos podem incluir arquivos abertos, processos filhos, alarmes pendentes, tratadores de sinais, informação sobre contabilidade e mais. Ao colocá-los juntos na forma de um processo, eles podem ser gerenciados com mais facilidade.

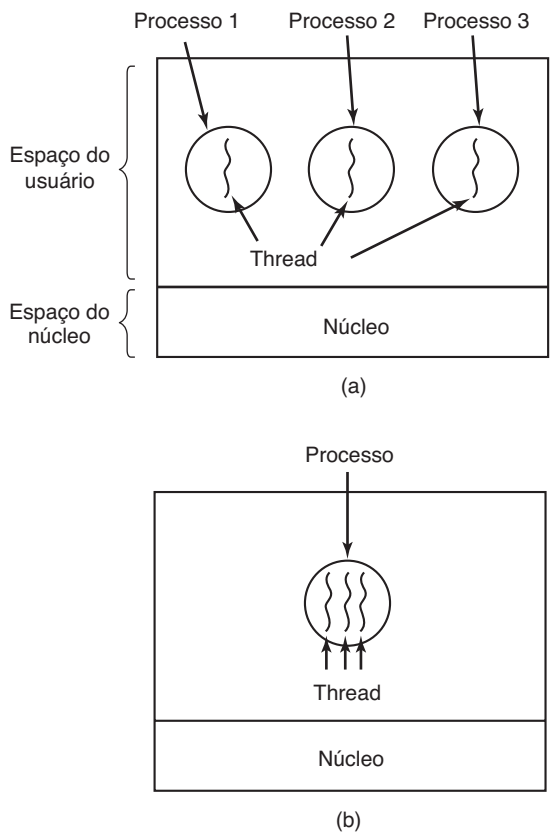
O outro conceito que um processo tem é de uma linha (*thread*) de execução, normalmente abreviado para apenas **thread**. O thread tem um contador de programa que controla qual instrução deve ser executada em seguida. Ele tem registradores, que armazenam suas variáveis de trabalho atuais. Tem uma pilha, que contém o histórico de execução, com uma estrutura para cada rotina chamada, mas ainda não retornada. Embora um thread deva executar em algum processo, o thread e seu processo são conceitos diferentes e podem ser tratados separadamente. Processos são usados para agrupar recursos; threads são as entidades escalonadas para execução na CPU.

O que os threads acrescentam para o modelo de processo é permitir que ocorram múltiplas execuções no mesmo ambiente, com um alto grau de independência uma da outra. Ter múltiplos threads executando em paralelo em um processo equivale a ter múltiplos processos executando em paralelo em um computador. No primeiro caso, os threads compartilham um espaço de endereçamento e outros recursos. No segundo caso, os processos compartilham memórias físicas, discos, impressoras e outros recursos. Como threads têm algumas das propriedades dos processos, às vezes eles são chamados de **processos leves**. O termo **multithread** também é usado para descrever a situação de permitir múltiplos threads no mesmo processo. Como vimos no Capítulo 1, algumas CPUs têm suporte de hardware direto para multithread e permitem que chaveamentos de threads aconteçam em uma escala de tempo de nanossegundos.

Na Figura 2.11(a) vemos três processos tradicionais. Cada processo tem seu próprio espaço de endereçamento e um único thread de controle. Em comparação, na Figura 2.11(b) vemos um único processo com três threads de controle. Embora em ambos os casos tenhamos três threads, na Figura 2.11(a) cada um deles opera em um espaço de endereçamento diferente, enquanto na Figura 2.11(b) todos os três compartilham o mesmo espaço de endereçamento.

Quando um processo multithread é executado em um sistema de CPU única, os threads se revezam executando. Na Figura 2.1, vimos como funciona a multiprogramação de processos. Ao chavear entre múltiplos processos, o sistema passa a ilusão de processos sequenciais executando em paralelo. O multithread funciona da mesma maneira. A CPU chaveia rapidamente entre os threads, dando a ilusão de que eles estão executando em paralelo, embora em uma CPU mais lenta do que a real. Em um processo limitado pela CPU com três threads, eles pareceriam executar em paralelo, cada um em uma CPU com um terço da velocidade da CPU real.

FIGURA 2.11 (a) Três processos, cada um com um thread. (b) Um processo com três threads.



Threads diferentes em um processo não são tão independentes quanto processos diferentes. Todos os threads têm exatamente o mesmo espaço de endereçamento, o que significa que eles também compartilham as mesmas variáveis globais. Tendo em vista que todo thread pode acessar todo espaço de endereçamento de memória dentro do espaço de endereçamento do processo, um thread pode ler, escrever, ou mesmo apagar a pilha de outro thread. Não há proteção entre threads, porque (1) é impossível e (2) não seria necessário. Ao contrário de processos distintos, que podem ser de usuários diferentes e que podem ser hostis uns com os outros, um processo é sempre propriedade de um único usuário, que presumivelmente criou múltiplos threads de maneira que eles possam cooperar, não lutar. Além de compartilhar um espaço de endereçamento, todos os threads podem compartilhar o mesmo conjunto de arquivos abertos, processos filhos, alarmes e sinais, e assim por diante, como mostrado na Figura 2.12. Assim, a organização da Figura 2.11(a) seria usada quando os três processos forem essencialmente não relacionados, enquanto a Figura 2.11(b) seria apropriada quando os três threads fizerem na realidade parte do mesmo trabalho e estiverem cooperando uns com os outros de maneira ativa e próxima.

Na Figura 2.12, os itens na primeira coluna são propriedades de processos, não threads de propriedades. Por exemplo, se um thread abre um arquivo, esse arquivo fica visível aos outros threads no processo e eles podem ler e escrever nele. Isso é lógico, já que o processo, e não o thread, é a unidade de gerenciamento de recursos. Se cada thread tivesse o seu próprio espaço de endereçamento, arquivos abertos, alarmes pendentes e assim por diante, seria um processo em separado. O que estamos tentando alcançar com o conceito de thread é a capacidade para múltiplos threads de execução de compartilhar um conjunto de recursos de maneira que possam trabalhar juntos intimamente para desempenhar alguma tarefa.

Como um processo tradicional (isto é, um processo com apenas um thread), um thread pode estar em qualquer um de vários estados: em execução, bloqueado, pronto, ou concluído. Um thread em execução tem a CPU naquele momento e está ativo. Em comparação, um thread bloqueado está esperando por algum evento para desbloqueá-lo. Por exemplo, quando um thread realiza uma chamada de sistema para ler do teclado, ele está bloqueado até que uma entrada seja digitada. Um thread pode bloquear esperando por algum evento externo acontecer ou por algum outro thread para desbloqueá-lo. Um thread pronto está programado para ser executado e o será tão logo chegue a sua vez. As transições entre estados de thread são as mesmas que aquelas entre estados de processos e estão ilustradas na Figura 2.2.

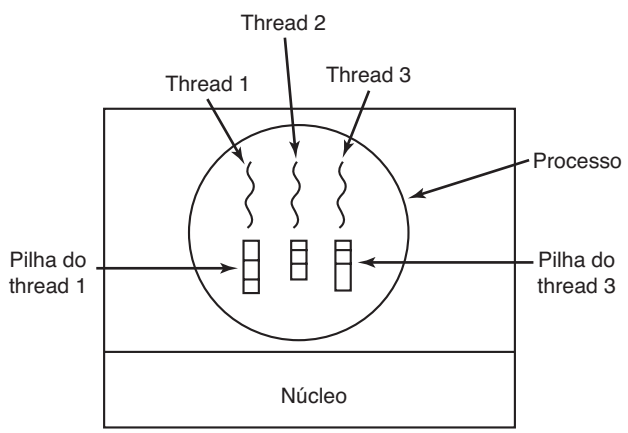
É importante perceber que cada thread tem a sua própria pilha, como ilustrado na Figura 2.13. Cada pilha do thread contém uma estrutura para cada rotina chamada, mas ainda não retornada. Essa estrutura contém as variáveis locais da rotina e o endereço de retorno para usar quando a chamada de rotina for encerrada. Por exemplo, se a rotina *X* chama a rotina *Y* e *Y* chama a rotina *Z*, então enquanto *Z* está executando, as estruturas para *X, Y* e *Z* estarão todas na pilha. Cada thread geralmente chamará rotinas diferentes e desse modo terá uma história de execução diferente. Essa é a razão pela qual cada thread precisa da sua própria pilha.

Quando o multithreading está presente, os processos normalmente começam com um único thread presente. Esse thread tem a capacidade de criar novos, chamando

FIGURA 2.12 A primeira coluna lista alguns itens compartilhados por todos os threads em um processo. A segunda lista alguns itens específicos a cada thread.

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e tratadores de sinais	
Informação de contabilidade	

FIGURA 2.13 Cada thread tem a sua própria pilha.



uma rotina de biblioteca como *thread_create*. Um parâmetro para *thread_create* especifica o nome de uma rotina para o novo thread executar. Não é necessário (ou mesmo possível) especificar algo sobre o espaço de endereçamento do novo thread, tendo em vista que ele automaticamente é executado no espaço de endereçamento do thread em criação. Às vezes, threads são hierárquicos, com uma relação pai-filho, mas muitas vezes não existe uma relação dessa natureza, e todos os threads são iguais. Com ou sem uma relação hierárquica, normalmente é devolvido ao thread em criação um identificador de thread que nomeia o novo thread.

Quando um thread tiver terminado o trabalho, pode concluir sua execução chamando uma rotina de biblioteca, como *thread_exit*. Ele então desaparece e não é mais escalonável. Em alguns sistemas, um thread pode esperar pela saída de um thread (específico) chamando uma rotina, por exemplo, *thread_join*. Essa rotina bloqueia o thread que executou a chamada até que um thread (específico) tenha terminado. Nesse sentido, a criação e a conclusão de threads é muito semelhante à criação e ao término de processos, com mais ou menos as mesmas opções.

Outra chamada de thread comum é *thread_yield*, que permite que um thread abra mão voluntariamente da CPU para deixar outro thread ser executado. Uma chamada dessas é importante porque não há uma interrupção de relógio para realmente forçar a multiprogramação como há com os processos. Desse modo, é importante que os threads sejam educados e voluntariamente entreguem a CPU de tempos em tempos para dar aos outros threads uma chance de serem executados. Outras chamadas permitem que um thread espere por outro thread para concluir algum trabalho, para um thread anunciar que terminou alguma tarefa e assim por diante.

Embora threads sejam úteis na maioria das vezes, eles também introduzem uma série de complicações no modelo de programação. Para começo de conversa, considere os efeitos da chamada fork de sistema UNIX. Se o processo pai tem múltiplos threads, o filho não deveria tê-los também? Do contrário, é possível que o processo não funcione adequadamente, tendo em vista que todos eles talvez sejam essenciais.

No entanto, se o processo filho possuir tantos threads quanto o pai, o que acontece se um thread no pai estava bloqueado em uma chamada read de um teclado? Dois threads estão agora bloqueados no teclado, um no pai e outro no filho? Quando uma linha é digitada, ambos os threads recebem uma cópia? Apenas o pai? Apenas o filho? O mesmo problema existe com conexões de rede abertas.

Outra classe de problemas está relacionada ao fato de que threads compartilham muitas estruturas de dados. O que acontece se um thread fecha um arquivo enquanto outro ainda está lendo dele? Suponha que um thread observe que há pouca memória e comece a alocar mais memória. No meio do caminho há um chaveamento de threads, e o novo também observa que há pouca memória e também começa a alocar mais memória. A memória provavelmente será alocada duas vezes. Esses problemas podem ser solucionados com algum esforço, mas os programas de multithread devem ser pensados e projetados com cuidado para funcionarem corretamente.

2.2.3 Threads POSIX

Para possibilitar que se escrevam programas com threads portáteis, o IEEE definiu um padrão para threads no padrão IEEE 1003.1c. O pacote de threads que ele define é chamado **Pthreads**. A maioria dos sistemas UNIX dá suporte a ele. O padrão define mais de 60 chamadas de função, um número grande demais para ser visto aqui. Em vez disso, descreveremos apenas algumas das principais para dar uma ideia de como funcionam. As chamadas que descreveremos a seguir estão listadas na Figura 2.14.

Todos os threads têm determinadas propriedades. Cada um tem um identificador, um conjunto de registradores (incluindo o contador de programa), e um conjunto de atributos, que são armazenados em uma estrutura. Os atributos incluem tamanho da pilha, parâmetros de escalonamento e outros itens necessários para usar o thread.

FIGURA 2.14 Algumas das chamadas de função do Pthreads.

Chamada de thread	Descrição
Pthread_create	Cria um novo thread
Pthread_exit	Conclui a chamada de thread
Pthread_join	Espera que um thread específico seja abandonado
Pthread_yield	Libera a CPU para que outro thread seja executado
Pthread_attr_init	Cria e inicializa uma estrutura de atributos do thread
Pthread_attr_destroy	Remove uma estrutura de atributos do thread

Um novo thread é criado usando a chamada *pthread_create*. O identificador de um thread recentemente criado é retornado como o valor da função. Essa chamada é intencionalmente muito parecida com a chamada de sistema fork (exceto pelos parâmetros), com o identificador de thread desempenhando o papel do PID (número de processo), em especial para identificar threads referenciados em outras chamadas.

Quando um thread tiver acabado o trabalho para o qual foi designado, ele pode terminar chamando *pthread_exit*. Essa chamada para o thread e libera sua pilha.

Muitas vezes, um thread precisa esperar outro terminar seu trabalho e sair antes de continuar. O que está esperando chama *pthread_join* para esperar outro thread específico terminar. O identificador do thread pelo qual se espera é dado como parâmetro.

Às vezes acontece de um thread não estar logicamente bloqueado, mas sente que já foi executado tempo suficiente e quer dar a outro thread a chance de ser executado. Ele pode atingir essa meta chamando *pthread_yield*. Essa chamada não existe para processos, pois o pressuposto aqui é que os processos são altamente competitivos e cada um quer o tempo de CPU que conseguir obter. No entanto, já que os threads de um processo estão trabalhando juntos e seu código é invariavelmente escrito pelo mesmo programador, às vezes o programador quer que eles se deem outra chance.

As duas chamadas seguintes de thread lidam com atributos. *Pthread_attr_init* cria a estrutura de atributos associada com um thread e o inicializa com os valores padrão. Esses valores (como a prioridade) podem ser modificados manipulando campos na estrutura de atributos.

Por fim, *pthread_attr_destroy* remove a estrutura de atributos de um thread, liberando a sua memória. Essa chamada não afeta os threads que a usam; eles continuam a existir.

FIGURA 2.15 Um exemplo de programa usando threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* Esta funcao imprime o identificador do thread e sai. */
        printf("Ola mundo. Boas vindas do thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* O programa principal cria 10 threads e sai. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Metodo Main. Criando thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d\n", status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Para ter uma ideia melhor de como o Pthreads funciona, considere o exemplo simples da Figura 2.15. Aqui o principal programa realiza *NUMBER_OF_THREADS* iterações, criando um novo thread em cada iteração, após anunciar sua intenção. Se a criação do thread fracassa, ele imprime uma mensagem de erro e então termina. Após criar todos os threads, o programa principal termina.

Quando um thread é criado, ele imprime uma mensagem de uma linha anunciando a si mesmo e então termina. A ordem na qual as várias mensagens são intercaladas é indeterminada e pode variar em execuções consecutivas do programa.

As chamadas Pthreads descritas anteriormente não são as únicas. Examinaremos algumas das outras após discutir a sincronização de processos e threads.

2.2.4 Implementando threads no espaço do usuário

Há dois lugares principais para implementar threads: no espaço do usuário e no núcleo. A escolha é um pouco controversa, e uma implementação híbrida também é possível. Descreveremos agora esses métodos, junto com suas vantagens e desvantagens.

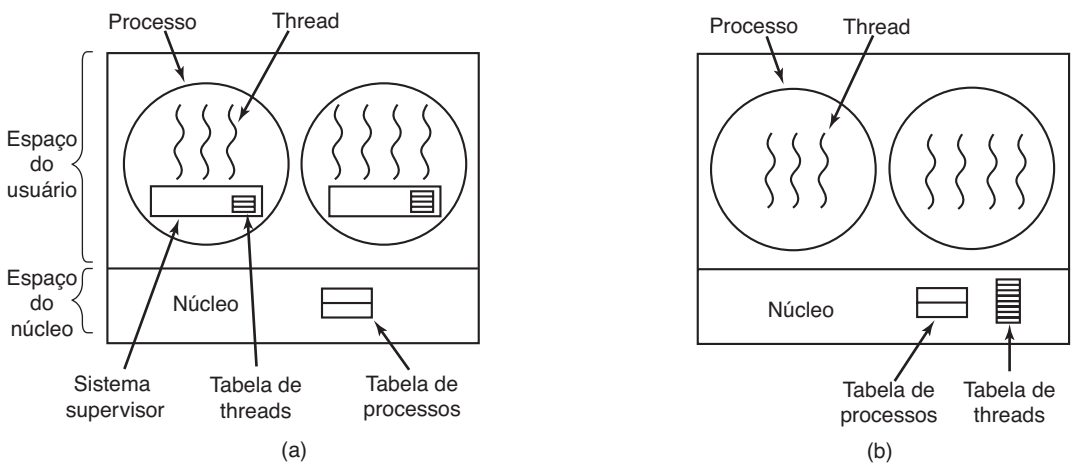
O primeiro método é colocar o pacote de threads inteiramente no espaço do usuário. O núcleo não sabe nada a respeito deles. Até onde o núcleo sabe, ele está gerenciando processos comuns de um único thread. A primeira vantagem, e mais óbvia, é que o pacote de threads no nível do usuário pode ser implementado em um sistema operacional que não dá suporte aos threads. Todos os sistemas operacionais costumavam cair nessa categoria, e mesmo agora alguns ainda caem. Com essa abordagem, threads são implementados por uma biblioteca.

Todas essas implementações têm a mesma estrutura geral, ilustrada na Figura 2.16(a). Os threads executam em cima de um sistema de tempo de execução, que é uma coleção de rotinas que gerencia os threads. Já vimos quatro desses: *pthread_create*, *pthread_exit*, *pthread_join* e *pthread_yield*, mas geralmente há mais.

Quando os threads são gerenciados no espaço do usuário, cada processo precisa da sua própria **tabela de threads** privada para controlá-los naquele processo. Ela é análoga à tabela de processo do núcleo, exceto por controlar apenas as propriedades de cada thread, como o contador de programa, o ponteiro de pilha, registradores, estado e assim por diante. A tabela de threads é gerenciada pelo sistema de tempo de execução. Quando um thread vai para o estado pronto ou bloqueado, a informação necessária para reiniciá-lo é armazenada na tabela de threads, exatamente da mesma maneira que o núcleo armazena informações sobre processos na tabela de processos.

Quando um thread faz algo que possa bloqueá-lo localmente, por exemplo, esperar que outro thread em seu processo termine um trabalho, ele chama uma rotina do sistema de tempo de execução. Essa rotina verifica se o thread precisa ser colocado no estado bloqueado. Caso isso seja necessário, ela armazena os registradores do thread (isto é, os seus próprios) na tabela de threads, procura na tabela por um thread pronto para ser executado, e recarrega os registradores da máquina com os valores salvos do novo thread. Tão logo o ponteiro de pilha e o contador de programa tenham sido trocados, o novo thread ressurge para a vida novamente de maneira automática. Se a máquina porventura tiver uma instrução para

FIGURA 2.16 (a) Um pacote de threads no espaço do usuário. (b) Um pacote de threads gerenciado pelo núcleo.



armazenar todos os registradores e outra para carregá-los, todo o chaveamento do thread poderá ser feito com apenas um punhado de instruções. Realizar um chaveamento de thread como esse é pelo menos uma ordem de magnitude — talvez mais — mais rápida do que desviar o controle para o núcleo, além de ser um forte argumento a favor de pacotes de threads de usuário.

No entanto, há uma diferença fundamental com os processos. Quando um thread decide parar de executar — por exemplo, quando ele chama *thread_yield* — o código de *thread_yield* pode salvar as informações do thread na própria tabela de thread. Além disso, ele pode então chamar o escalonador de threads para pegar outro thread para executar. A rotina que salva o estado do thread e o escalonador são apenas rotinas locais, de maneira que invocá-las é algo muito mais eficiente do que fazer uma chamada de núcleo. Entre outras coisas, nenhuma armadilha (*trap*) é necessária, nenhum chaveamento de contexto é necessário, a cache de memória não precisa ser esvaziada e assim por diante. Isso torna o escalonamento de thread muito rápido.

Threads de usuário também têm outras vantagens. Eles permitem que cada processo tenha seu próprio algoritmo de escalonamento customizado. Para algumas aplicações, por exemplo, aquelas com um thread coletor de lixo, é uma vantagem não ter de se preocupar com um thread ser parado em um momento inconveniente. Eles também escalam melhor, já que threads de núcleo sempre exigem algum espaço de tabela e de pilha no núcleo, o que pode ser um problema se houver um número muito grande de threads.

Apesar do seu melhor desempenho, pacotes de threads de usuário têm alguns problemas importantes. Primeiro, o problema de como chamadas de sistema bloqueantes são implementadas. Suponha que um thread leia de um teclado antes que quaisquer teclas tenham sido acionadas. Deixar que o thread realmente faça a chamada de sistema é algo inaceitável, visto que isso parará todos os threads. Uma das principais razões para ter threads era permitir que cada um utilizasse chamadas com bloqueio, enquanto evitaria que um thread bloqueado afetasse os outros. Com chamadas de sistema bloqueantes, é difícil ver como essa meta pode ser alcançada prontamente.

As chamadas de sistema poderiam ser todas modificadas para que não bloqueassem (por exemplo, um read no teclado retornaria apenas 0 byte se nenhum caractere já estivesse no buffer), mas exigir mudanças para o sistema operacional não é algo atraente. Além disso, um argumento para threads de usuário era precisamente que eles podiam ser executados com sistemas operacionais *existentes*. Ainda, mudar a semântica de read exigirá mudanças para muitos programas de usuários.

Mais uma alternativa encontra-se disponível no caso de ser possível dizer antecipadamente se uma chamada será bloqueada. Na maioria das versões de UNIX, existe uma chamada de sistema, select, que permite a quem chama saber se um read futuro será bloqueado. Quando essa chamada está presente, a rotina de biblioteca *read* pode ser substituída por uma nova que primeiro faz uma chamada select e, então, faz a chamada read somente se ela for segura (isto é, não for bloqueada). Se a chamada read for bloqueada, a chamada não é feita. Em vez disso, outro thread é executado. Da próxima vez que o sistema de tempo de execução assumir o controle, ele pode conferir de novo se a read está então segura. Essa abordagem exige reescrever partes da biblioteca de chamadas de sistema, e é algo ineficiente e deselegante, mas há pouca escolha. O código colocado em torno da chamada de sistema para fazer a verificação é chamado de **jacket** ou **wrapper**.

Um problema de certa maneira análogo às chamadas de sistema bloqueantes é o problema das faltas de páginas. Nós as estudaremos no Capítulo 3. Por ora, basta dizer que os computadores podem ser configurados de tal maneira que nem todo o programa está na memória principal ao mesmo tempo. Se o programa chama ou salta para uma instrução que não esteja na memória, ocorre uma falta de página e o sistema operacional buscará a instrução perdida (e suas vizinhas) do disco. Isso é chamado de uma falta de página. O processo é bloqueado enquanto as instruções necessárias estão sendo localizadas e lidas. Se um thread causa uma falta de página, o núcleo, desconhecendo até a existência dos threads, naturalmente bloqueia o processo inteiro até que o disco de E/S esteja completo, embora outros threads possam ser executados.

Outro problema com pacotes de threads de usuário é que se um thread começa a ser executado, nenhum outro naquele processo será executado a não ser que o primeiro thread voluntariamente abra mão da CPU. Dentro de um único processo, não há interrupções de relógio, impossibilitando escalonar processos pelo esquema de escalonamento circular (dando a vez ao outro). A menos que um thread entre voluntariamente no sistema de tempo de execução, o escalonador jamais terá uma chance.

Uma solução possível para o problema de threads sendo executados infinitamente é obrigar o sistema de tempo de execução a solicitar um sinal de relógio

(interrupção) a cada segundo para dar a ele o controle, mas isso, também, é algo grosseiro e confuso para o programa. Interrupções periódicas de relógio em uma frequência mais alta nem sempre são possíveis, e mesmo que fossem, a sobrecarga total poderia ser substancial. Além disso, um thread talvez precise também de uma interrupção de relógio, interferindo com o uso do relógio pelo sistema de tempo de execução.

Outro — e o mais devastador — argumento contra threads de usuário é que os programadores geralmente desejam threads precisamente em aplicações nas quais eles são bloqueados com frequência, por exemplo, em um servidor web com múltiplos threads. Esses threads estão constantemente fazendo chamadas de sistema. Uma vez que tenha ocorrido uma armadilha para o núcleo a fim de executar uma chamada de sistema, não daria muito mais trabalho para o núcleo trocar o thread sendo executado, se ele estiver bloqueado, o núcleo fazendo isso elimina a necessidade de sempre realizar chamadas de sistema select que verificam se as chamadas de sistema read são seguras. Para aplicações que são em sua essência inteiramente limitadas pela CPU e raramente são bloqueadas, qual o sentido de usar threads? Ninguém proporia seriamente computar os primeiros *n* números primos ou jogar xadrez usando threads, porque não há nada a ser ganho com isso.

2.2.5 Implementando threads no núcleo

Agora vamos considerar que o núcleo saiba sobre os threads e os gerencie. Não é necessário um sistema de tempo de execução em cada um, como mostrado na Figura 2.16(b). Também não há uma tabela de thread em cada processo. Em vez disso, o núcleo tem uma tabela que controla todos os threads no sistema. Quando um thread quer criar um novo ou destruir um existente, ele faz uma chamada de núcleo, que então faz a criação ou a destruição atualizando a tabela de threads do núcleo.

A tabela de threads do núcleo contém os registradores, estado e outras informações de cada thread. A informação é a mesma com os threads de usuário, mas agora mantidas no núcleo em vez de no espaço do usuário (dentro do sistema de tempo de execução). Essa informação é um subconjunto das informações que os núcleos tradicionais mantêm a respeito dos seus processos de thread único, isto é, o estado de processo. Além disso, o núcleo também mantém a tabela de processos tradicional para controlar os processos.

Todas as chamadas que poderiam bloquear um thread são implementadas como chamadas de sistema, a um custo consideravelmente maior do que uma chamada para uma rotina de sistema de tempo de execução. Quando um thread é bloqueado, o núcleo tem a opção de executar outro thread do mesmo processo (se um estiver pronto) ou algum de um processo diferente. Com threads de usuário, o sistema de tempo de execução segue executando threads a partir do seu próprio processo até o núcleo assumir a CPU dele (ou não houver mais threads prontos para serem executados).

Em decorrência do custo relativamente maior de se criar e destruir threads no núcleo, alguns sistemas assumem uma abordagem ambientalmente correta e reciclam seus threads. Quando um thread é destruído, ele é marcado como não executável, mas suas estruturas de dados de núcleo não são afetadas de outra maneira. Depois, quando um novo thread precisa ser criado, um antigo é reativado, evitando parte da sobrecarga. A reciclagem de threads também é possível para threads de usuário, mas tendo em vista que o overhead de gerenciamento de threads é muito menor, há menos incentivo para fazer isso.

Threads de núcleo não exigem quaisquer chamadas de sistema novas e não bloqueantes. Além disso, se um thread em um processo provoca uma falta de página, o núcleo pode facilmente conferir para ver se o processo tem quaisquer outros threads executáveis e, se assim for, executar um deles enquanto espera que a página exigida seja trazida do disco. Sua principal desvantagem é que o custo de uma chamada de sistema é substancial, então se as operações de thread (criação, término etc.) forem frequentes, ocorrerá uma sobrecarga muito maior.

Embora threads de núcleo solucionem alguns problemas, eles não resolvem todos eles. Por exemplo, o que acontece quando um processo com múltiplos threads é bifurcado? O novo processo tem tantos threads quanto o antigo, ou possui apenas um? Em muitos casos, a melhor escolha depende do que o processo está planejando fazer em seguida. Se ele for chamar exec para começar um novo programa, provavelmente um thread é a escolha correta, mas se ele continuar a executar, reproduzir todos os threads talvez seja o melhor.

Outra questão são os sinais. Lembre-se de que os sinais são enviados para os processos, não para os threads, pelo menos no modelo clássico. Quando um sinal chega, qual thread deve cuidar dele? Threads poderiam talvez registrar seu interesse em determinados sinais, de maneira que, ao chegar um sinal, ele seria dado para o thread que disse querê-lo. Mas o que acontece se dois ou mais threads registraram interesse para o mesmo sinal? Esses são apenas dois dos problemas que os threads introduzem, mas há outros.

2.2.6 Implementações híbridas

Várias maneiras foram investigadas para tentar combinar as vantagens de threads de usuário com threads de núcleo. Uma maneira é usar threads de núcleo e então multiplexar os de usuário em alguns ou todos eles, como mostrado na Figura 2.17. Quando essa abordagem é usada, o programador pode determinar quantos threads de núcleo usar e quantos threads de usuário multiplexar para cada um. Esse modelo proporciona o máximo em flexibilidade.

Com essa abordagem, o núcleo está consciente *apenas* dos threads de núcleo e os escalona. Alguns desses threads podem ter, em cima deles, múltiplos threads de usuário multiplexados, os quais são criados, destruídos e escalonados exatamente como threads de usuário em um processo executado em um sistema operacional sem capacidade de múltiplos threads. Nesse modelo, cada thread de núcleo tem algum conjunto de threads de usuário que se revezam para usá-lo.

2.2.7 Ativações pelo escalonador

Embora threads de núcleo sejam melhores do que threads de usuário em certos aspectos-chave, eles são também indiscutivelmente mais lentos. Como consequência, pesquisadores procuraram maneiras de melhorar a situação sem abrir mão de suas boas propriedades. A seguir descreveremos uma abordagem desenvolvida por Anderson et al. (1992), chamada **ativações pelo escalonador**. Um trabalho relacionado é discutido por Edler et al. (1988) e Scott et al. (1990).

A meta do trabalho da ativação pelo escalonador é imitar a funcionalidade dos threads de núcleo, mas com melhor desempenho e maior flexibilidade normalmente associados aos pacotes de threads implementados no espaço do usuário. Em particular, threads de usuário não deveriam ter de fazer chamadas de sistema especiais sem bloqueio ou conferir antecipadamente se é seguro fazer determinadas chamadas de sistema. Mesmo assim, quando um thread é bloqueado em uma chamada de sistema ou uma página falha, deve ser possível executar outros threads dentro do mesmo processo, se algum estiver pronto.

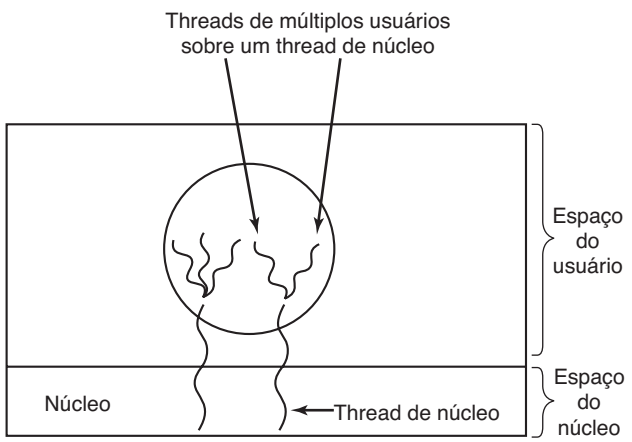
A eficiência é alcançada evitando-se transições desnecessárias entre espaço do usuário e do núcleo. Se um thread é bloqueado esperando por outro para fazer algo, por exemplo, não há razão para envolver o núcleo, poupando assim a sobrecarga da transição núcleo-usuário. O sistema de tempo de execução pode bloquear o thread de sincronização e escalonar sozinho um novo.

Quando ativações pelo escalonador são usadas, o núcleo designa um determinado número de processadores virtuais para cada processo e deixa o sistema de tempo de execução (no espaço do usuário) alocar threads para eles. Esse mecanismo também pode ser usado em um multiprocessador onde os processadores virtuais podem ser CPUs reais. O número de processadores virtuais alocados para um processo é de início um, mas o processo pode pedir mais e também pode devolver processadores que não precisa mais. O núcleo também pode retomar processadores virtuais já alocados a fim de colocá-los em processos mais necessitados.

A ideia básica para o funcionamento desse esquema é a seguinte: quando sabe que um thread foi bloqueado (por exemplo, tendo executado uma chamada de sistema bloqueante ou causado uma falta de página), o núcleo notifica o sistema de tempo de execução do processo, passando como parâmetros na pilha o número do thread em questão e uma descrição do evento que ocorreu. A notificação acontece quando o núcleo ativa o sistema de tempo de execução em um endereço inicial conhecido, de maneira mais ou menos análoga a um sinal em UNIX. Esse mecanismo é chamado **upcall**.

Uma vez ativado, o sistema de tempo de execução pode reescalonar os seus threads, tipicamente marcando o thread atual como bloqueado e tomando outro da lista pronta, configurando seus registradores e reinicializando-o. Depois, quando o núcleo fica sabendo que o thread original pode executar de novo (por exemplo, o pipe do qual ele estava tentando ler agora contém dados, ou a página sobre a qual ocorreu uma falta foi trazida do disco), ele faz outro upcall para informar o sistema de tempo de execução. O sistema de tempo de execução pode então

FIGURA 2.17 Multiplexando threads de usuário em threads de núcleo.



reiniciar o thread bloqueado imediatamente ou colocá-lo na lista de prontos para executar mais tarde.

Quando ocorre uma interrupção de hardware enquanto um thread de usuário estiver executando, a CPU interrompida troca para o modo núcleo. Se a interrupção for causada por um evento que não é de interesse do processo interrompido, como a conclusão da E/S de outro processo, quando o tratador da interrupção terminar, ele coloca o thread de interrupção de volta no estado de antes da interrupção. Se, no entanto, o processo está interessado na interrupção, como a chegada de uma página necessitada por um dos threads do processo, o thread interrompido não é reinicializado. Em vez disso, ele é suspenso, e o sistema de tempo de execução é inicializado naquela CPU virtual, com o estado do thread interrompido na pilha. Então cabe ao sistema de tempo de execução decidir qual thread escalonar naquela CPU: o thread interrompido, o recentemente pronto ou uma terceira escolha.

Uma objeção às ativações pelo escalonador é a confiança fundamental nos upcalls, um conceito que viola a estrutura inerente em qualquer sistema de camadas. Em geral, a camada n oferece determinados serviços que a camada $n + 1$ pode chamar, mas a camada n pode não chamar rotinas na camada $n + 1$. Upcalls não seguem esse princípio fundamental.

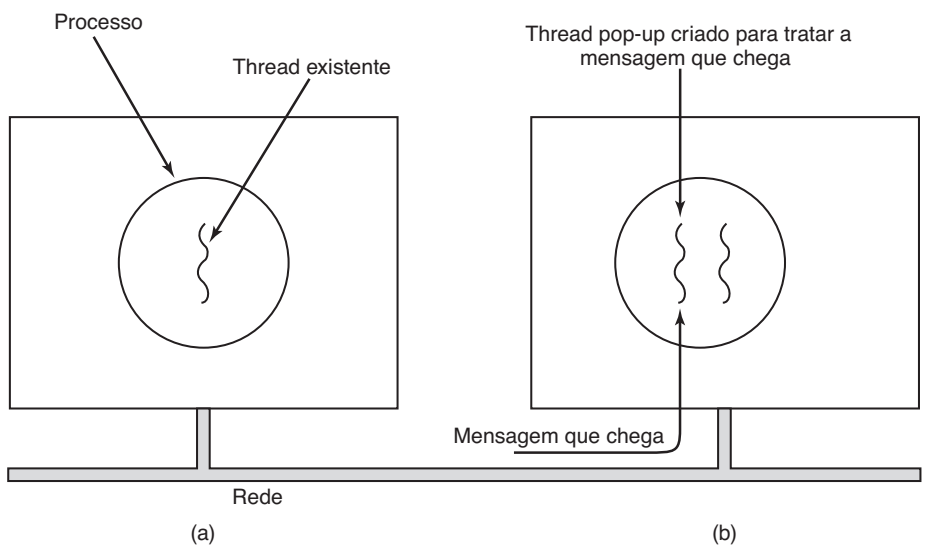
2.2.8 Threads pop-up

Threads costumam ser úteis em sistemas distribuídos. Um exemplo importante é como mensagens que chegam — por exemplo, requisições de serviços — são tratadas. A abordagem tradicional é ter um processo ou thread que esteja bloqueado em uma chamada de sistema receive esperando pela mensagem que chega. Quando uma mensagem chega, ela é aceita, aberta, seu conteúdo examinado e processada.

No entanto, uma abordagem completamente diferente também é possível, na qual a chegada de uma mensagem faz o sistema criar um novo thread para lidar com a mensagem. Esse thread é chamado de **thread pop-up** e está ilustrado na Figura 2.18. Uma vantagem fundamental de threads pop-up é que como são novos, eles não têm história alguma — registradores, pilha, o que quer que seja — que devem ser restaurados. Cada um começa fresco e cada um é idêntico a todos os outros. Isso possibilita a criação de tais threads rapidamente. O thread novo recebe a mensagem que chega para processar. O resultado da utilização de threads pop-up é que a latência entre a chegada da mensagem e o começo do processamento pode ser encurtada.

Algum planejamento prévio é necessário quando threads pop-up são usados. Por exemplo, em qual processo o thread é executado? Se o sistema dá suporte a threads sendo executados no contexto núcleo, o thread pode ser executado ali (razão pela qual não mostramos o núcleo na Figura 2.18). Executar um thread pop-up no espaço núcleo normalmente é mais fácil e mais rápido do que colocá-lo no espaço do usuário. Também, um thread pop-up no espaço núcleo consegue facilmente acessar todas as tabelas do núcleo e os dispositivos de E/S, que podem ser necessários para o processamento de interrupções. Por outro lado, um thread de núcleo

FIGURA 2.18 Criação de um thread novo quando a mensagem chega. (a) Antes da chegada da mensagem. (b) Depois da chegada da mensagem.



com erros pode causar mais danos que um de usuário com erros. Por exemplo, se ele for executado por tempo demais e não liberar a CPU, dados que chegam podem ser perdidos para sempre.

2.2.9 Convertendo código de um thread em código multithread

Muitos programas existentes foram escritos para processos monothread. Convertê-los para multithreading é muito mais complicado do que pode parecer em um primeiro momento. A seguir examinaremos apenas algumas das armadilhas.

Para começo de conversa, o código de um thread em geral consiste em múltiplas rotinas, exatamente como um processo. Essas rotinas podem ter variáveis locais, variáveis globais e parâmetros. Variáveis locais e de parâmetros não causam problema algum, mas variáveis que são globais para um thread, mas não globais para o programa inteiro, são um problema. Essas são variáveis que são globais no sentido de que muitos procedimentos dentro do thread as usam (como poderiam usar qualquer variável global), mas outros threads devem logicamente deixá-las sozinhas.

Como exemplo, considere a variável *errno* mantida pelo UNIX. Quando um processo (ou thread) faz uma chamada de sistema que falha, o código de erro é colocado em *errno*. Na Figura 2.19, o thread 1 executa a chamada de sistema access para descobrir se ela tem permissão para acessar determinado arquivo. O sistema operacional retorna a resposta na variável global *errno*. Após o controle ter retornado para o thread 1, mas antes de ele ter uma chance de ler *errno*, o escalonador decide que o thread 1 teve tempo de CPU suficiente para o momento e decide trocar para o thread 2. O thread 2 executa uma chamada open que falha, o que faz que *errno* seja sobrescrito e o código de acesso do thread 1 seja perdido para sempre. Quando o thread 1 inicia posteriormente, ele terá o valor errado e comportar-se-á incorretamente.

Várias soluções para esse problema são possíveis. Uma é proibir completamente as variáveis globais. Por mais válido que esse ideal possa ser, ele entra em conflito com grande parte dos softwares existentes. Outra é designar a cada thread as suas próprias variáveis globais privadas, como mostrado na Figura 2.20. Dessa maneira, cada thread tem a sua própria cópia privada de *errno* e outras variáveis globais, de modo que os conflitos são evitados. Na realidade, essa decisão cria um novo nível de escopo, variáveis visíveis a todas as rotinas de um thread (mas não para outros threads), além dos níveis de escopo existentes de variáveis visíveis apenas para uma rotina e variáveis visíveis em toda parte no programa.

No entanto, acessar as variáveis globais privadas é um pouco complicado já que a maioria das linguagens de programação tem uma maneira de expressar variáveis locais e variáveis globais, mas não formas intermediárias. É possível alocar um pedaço da memória para as globais e passá-lo para cada rotina no thread como um parâmetro extra. Embora dificilmente você possa considerá-la uma solução elegante, ela funciona.

Alternativamente, novas rotinas de biblioteca podem ser introduzidas para criar, alterar e ler essas variáveis globais restritas ao thread. A primeira chamada pode parecer assim:

```
create_global("bufptr");
```

FIGURA 2.19 Conflitos entre threads sobre o uso de uma variável global.

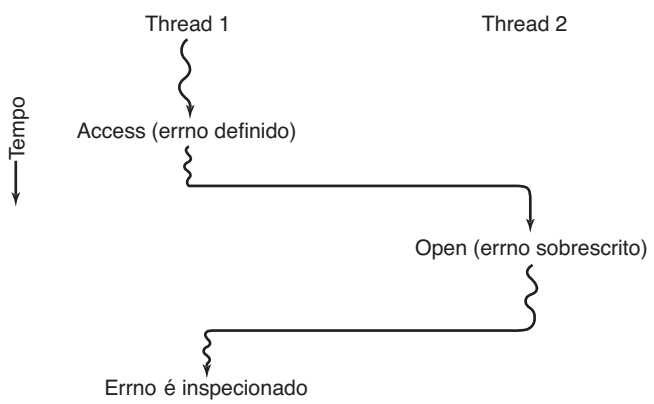
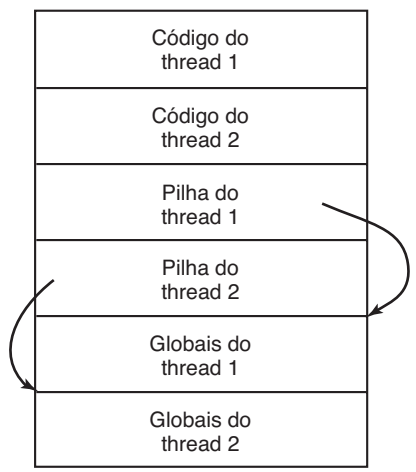


FIGURA 2.20 Threads podem ter variáveis globais individuais.



Ela aloca memória para um ponteiro chamado *bufptr* no heap ou em uma área de armazenamento especial reservada para o thread que emitiu a chamada. Não importa onde a memória esteja alocada, apenas o thread que emitiu a chamada tem acesso à variável global. Se outro thread criar uma variável global com o mesmo nome, ele obterá uma porção de memória que não entrará em conflito com a existente.

Duas chamadas são necessárias para acessar variáveis globais: uma para escrevê-las e a outra para lê-las. Para escrever, algo como

```
set_global("bufptr", &buf)
```

bastará. Ela armazena o valor de um ponteiro na porção de memória previamente criada pela chamada para *create_global*. Para ler uma variável global, a chamada pode ser algo como

```
bufptr = read_global("bufptr").
```

Ela retorna o endereço armazenado na variável global, de maneira que os seus dados possam ser acessados.

O próximo problema em transformar um programa de um único thread em um programa com múltiplos threads é que muitas rotinas de biblioteca não são reentrantes. Isto é, elas não foram projetadas para ter uma segunda chamada feita para uma rotina enquanto uma anterior ainda não tiver sido concluída. Por exemplo, o envio de uma mensagem através de uma rede pode ser programado com a montagem da mensagem em um buffer fixo dentro da biblioteca, seguido de um chaveamento para enviá-la. O que acontece se um thread montou a sua mensagem no buffer, então, uma interrupção de relógio força um chaveamento para um segundo thread que imediatamente sobrescreve o buffer com sua própria mensagem?

Similarmente, rotinas de alocação de memória como *malloc* no UNIX, mantêm tabelas cruciais sobre o uso de memória, por exemplo, uma lista encadeada de pedaços de memória disponíveis. Enquanto *malloc* está ocupada atualizando essas listas, elas podem temporariamente estar em um estado inconsistente, com ponteiros que apontam para lugar nenhum. Se um chaveamento de threads ocorrer enquanto as tabelas forem inconsistentes e uma nova chamada entrar de um thread diferente, um ponteiro inválido poderá ser usado, levando à queda do programa. Consertar todos esses problemas efetivamente significa reescrever toda a biblioteca. Fazê-lo é uma atividade não trivial com uma possibilidade real de ocorrer a introdução de erros sutis.

Uma solução diferente é fornecer a cada rotina uma proteção que altera um bit para indicar que a biblioteca está sendo usada. Qualquer tentativa de outro thread usar uma rotina de biblioteca enquanto a chamada anterior ainda não tiver sido concluída é bloqueada. Embora essa abordagem possa ser colocada em funcionamento, ela elimina muito o paralelismo em potencial.

Em seguida, considere os sinais. Alguns são logicamente específicos a threads, enquanto outros não. Por exemplo, se um thread chama alarm, faz sentido para o sinal resultante ir até o thread que fez a chamada. No entanto, quando threads são implementados inteiramente no espaço de usuário, o núcleo não tem nem ideia a respeito dos threads e mal pode dirigir o sinal para o thread certo. Uma complicação adicional ocorre se um processo puder ter apenas um alarme pendente por vez e vários threads chamam alarm independentemente.

Outros sinais, como uma interrupção de teclado, não são específicos aos threads. Quem deveria pegá-los? Um thread designado? Todos os threads? Um thread pop-up recentemente criado? Além disso, o que acontece se um thread mudar os tratadores de sinal sem contar para os outros threads? E o que acontece se um thread quiser pegar um sinal em particular (digamos, o usuário digitando CTRL-C), e outro thread quiser esse sinal para concluir o processo? Essa situação pode surgir se um ou mais threads executarem rotinas de biblioteca-padrão e outros forem escritos por usuários. Claramente, esses desejos são incompatíveis. Em geral, sinais são suficientemente difíceis para gerenciar em um ambiente de um thread único. Ir para um ambiente de múltiplos threads não torna a situação mais fácil de lidar.

Um último problema introduzido pelos threads é o gerenciamento de pilha. Em muitos sistemas, quando a pilha de um processo transborda, o núcleo apenas fornece àquele processo mais pilha automaticamente. Quando um processo tem múltiplos threads, ele também tem múltiplas pilhas. Se o núcleo não tem ciência de todas essas pilhas, ele não pode fazê-las crescer automaticamente por causa de uma falha de pilha. Na realidade, ele pode nem se dar conta de que uma falha de memória está relacionada com o crescimento da pilha de algum thread.

Esses problemas certamente não são insuperáveis, mas mostram que apenas introduzir threads em um sistema existente sem uma alteração bastante substancial do sistema não vai funcionar mesmo. No mínimo, as semânticas das chamadas de sistema talvez precisem ser redefinidas e as bibliotecas reescritas. E todas essas coisas devem ser feitas de maneira a permanecerem compatíveis com programas já existentes para o caso limitante de um processo com apenas um thread. Para informações adicionais sobre threads, ver Hauser et al. (1993), Marsh et al. (1991) e Rodrigues et al. (2010).