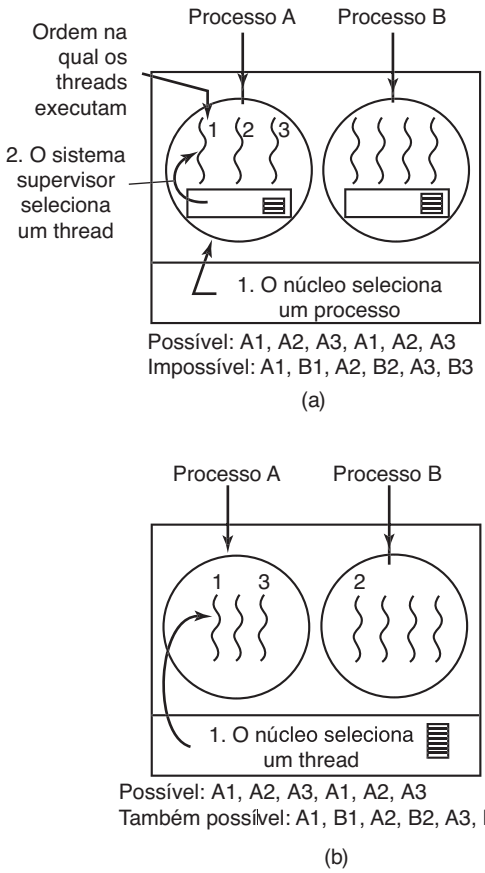
FIGURA 2.44 (a) Escalonamento possível de threads de usuário com quantum de processo de 50 ms e threads que executam 5 ms por surto de CPU. (b) Escalonamento possível de threads de núcleo com as mesmas características que (a).



bloqueados após 5 ms, a ordem do thread por algum período de 30 ms pode ser *A1*, *B1*, *A2*, *B2*, *A3*, *B3,* algo que não é possível com esses parâmetros e threads de usuário. Essa situação está parcialmente descrita na Figura 2.44(b).

Uma diferença importante entre threads de usuário e de núcleo é o desempenho. Realizar um chaveamento de thread com threads de usuário exige um punhado de instruções de máquina. Com threads de núcleo é necessário um chaveamento de contexto completo, mudar o mapa de memória e invalidar o cache, o que representa uma demora de magnitude várias ordens maior. Por outro lado, com threads de núcleo, ter um bloqueio de thread na E/S não suspende todo o processo como ocorre com threads de usuário.

Visto que o núcleo sabe que chavear de um thread no processo *A* para um thread no processo *B* é mais caro do que executar um segundo thread no processo *A* (por ter de mudar o mapa de memória e invalidar a memória de cache), ele pode levar essa informação em conta quando toma uma decisão. Por exemplo, dados dois threads que de outra forma são igualmente importantes, com um deles pertencendo ao mesmo processo que um thread que foi bloqueado há pouco e outro pertencendo a um processo diferente, a preferência poderia ser dada ao primeiro.

Outro fator importante é que os threads de usuário podem empregar um escalonador de thread específico de uma aplicação. Considere, por exemplo, o servidor na web da Figura 2.8. Suponha que um thread operário foi bloqueado há pouco e o thread despachante e dois threads operários estão prontos. Quem deve ser executado em seguida? O sistema de tempo de execução, sabendo o que todos os threads fazem, pode facilmente escolher o despachante para ser executado em seguida, de maneira que ele possa colocar outro operário para executar. Essa estratégia maximiza o montante de paralelismo em um ambiente onde operários frequentemente são bloqueados pela E/S de disco. Com threads de núcleo, o núcleo jamais saberia o que cada thread fez (embora a eles pudessem ser atribuídas prioridades diferentes). No geral, entretanto, escalonadores de threads específicos de aplicações são capazes de ajustar uma aplicação melhor do que o núcleo.

2.5 Problemas clássicos de IPC

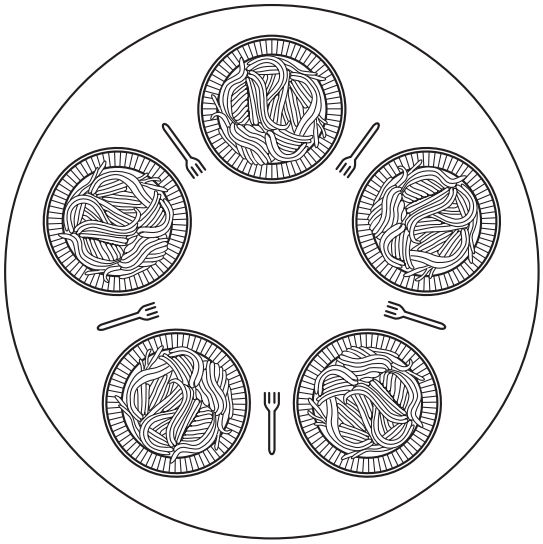
A literatura de sistemas operacionais está cheia de problemas interessantes que foram amplamente discutidos e analisados usando variados métodos de sincronização. Nas seções a seguir, examinaremos três dos problemas mais conhecidos.

2.5.1 O problema do jantar dos filósofos

Em 1965, Dijkstra formulou e então solucionou um problema de sincronização que ele chamou de **problema do jantar dos filósofos**. Desde então, todos os que inventaram mais uma primitiva de sincronização sentiram-se obrigados a demonstrar quão maravilhosa é a nova primitiva exibindo quão elegantemente ela soluciona o problema do jantar dos filósofos. O problema pode ser colocado de maneira bastante simples, como a seguir: cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete é tão escorregadio que um filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos há um garfo. O desenho da mesa está ilustrado na Figura 2.45.

A vida de um filósofo consiste em alternar períodos de alimentação e pensamento. (Trata-se de um tipo de abstração, mesmo para filósofos, mas as outras atividades são irrelevantes aqui.) Quando um filósofo fica suficientemente faminto, ele tenta pegar seus garfos à esquerda

FIGURA 2.45 Hora do almoço no departamento de filosofia.



e à direita, um de cada vez, não importa a ordem. Se for bem-sucedido em pegar dois garfos, ele come por um tempo, então larga os garfos e continua a pensar. A questão fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e jamais fique travado? (Já foi apontado que a necessidade de dois garfos é de certa maneira artificial; talvez devamos trocar de um prato italiano para um chinês, substituindo o espaguete por arroz e os garfos por pauzinhos.)

A Figura 2.46 mostra a solução óbvia. O procedimento *take_fork* espera até o garfo específico estar disponível e então o pega. Infelizmente, a solução óbvia está errada. Suponha que todos os cinco filósofos peguem seus garfos esquerdos simultaneamente. Nenhum será capaz de pegar seus garfos direitos, e haverá um impasse.

Poderíamos facilmente modificar o programa de maneira que após pegar o garfo esquerdo, o programa confere para ver se o garfo direito está disponível. Se não estiver, o filósofo coloca de volta o esquerdo sobre a mesa, espera por um tempo, e repete todo o processo. Essa proposta também fracassa, embora por uma razão diferente. Com um pouco de azar, todos os filósofos poderiam começar o algoritmo simultaneamente, pegando seus garfos esquerdos, vendo que seus garfos direitos não estavam disponíveis, colocando seus garfos esquerdos de volta sobre a mesa, esperando, pegando seus garfos esquerdos de novo ao mesmo tempo, assim por diante, para sempre. Uma situação como essa, na qual todos os programas continuam a executar indefinidamente, mas fracassam em realizar qualquer progresso, é chamada de **inanição** (*starvation*). (Ela é chamada de inanição mesmo quando o problema não ocorre em um restaurante italiano ou chinês.)

Agora você pode pensar que se os filósofos simplesmente esperassem um tempo aleatório em vez de ao mesmo tempo fracassarem em conseguir o garfo direito, a chance de tudo continuar em um impasse mesmo por uma hora é muito pequena. Essa observação é verdadeira, e em quase todas as aplicações tentar mais tarde não é um problema. Por exemplo, na popular rede de área local Ethernet, se dois computadores enviam um pacote ao mesmo tempo, cada um espera um tempo aleatório e tenta de novo; na prática essa solução funciona bem. No entanto, em algumas aplicações você preferiria uma solução que sempre funcionasse e não pudesse fracassar por uma série improvável de números aleatórios. Pense no controle de segurança em uma usina de energia nuclear.

Uma melhoria para a Figura 2.46 que não apresenta impasse nem inanição é proteger os cinco comandos seguindo a chamada *think* com um semáforo binário. Antes de começar a pegar garfos, um filósofo realizaria

FIGURA 2.46 Uma não solução para o problema do jantar dos filósofos.

```
#define N 5                                     /* numero de filosofos */

void philosopher(int i)                         /* i: numero do filosofo, de 0 a 4 */
{
    while (TRUE) {
        think( );                               /* o filosofo esta pensando */
        take_fork(i);                           /* pega o garfo esquerdo */
        take_fork((i+1) % N);                   /* pega o garfo direito; % e o operador modulo */
        eat( );                                 /* hummm, espaguete */
        put_fork(i);                            /* devolve o garfo esquerdo a mesa */
        put_fork((i+1) % N);                    /* devolve o garfo direito a mesa */
    }
}
```

um down em *mutex*. Após substituir os garfos, ele realizaria um up em *mutex*. Do ponto de vista teórico, essa solução é adequada. Do ponto de vista prático, ela tem um erro de desempenho: apenas um filósofo pode estar comendo a qualquer dado instante. Com cinco garfos disponíveis, deveríamos ser capazes de ter dois filósofos comendo ao mesmo tempo.

A solução apresentada na Figura 2.47 é livre de impasse e permite o máximo paralelismo para um número arbitrário de filósofos. Ela usa um arranjo, *estado*, para controlar se um filósofo está comendo, pensando, ou com fome (tentando conseguir garfos). Um filósofo pode passar para o estado comendo apenas se nenhum de seus vizinhos estiver comendo. Os vizinhos do

FIGURA 2-47 Uma solução para o problema do jantar dos filósofos.

```
#define N               5               /* numero de filosofos */
#define LEFT            (i+N–1)%N       /* numero do vizinho a esquerda de i */
#define RIGHT           (i+1)%N         /* numero do vizinho a direita de i */
#define THINKING        0               /* o filosofo esta pensando */
#define HUNGRY          1               /* o filosofo esta tentando pegar garfos */
#define EATING          2               /* o filosofo esta comendo */

typedef int semaphore;                  /* semaforos sao um tipo especial de int */
int state[N];                           /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;                    /* exclusao mutua para as regioes criticas */
semaphore s[N];                         /* um semaforo por filosofo */

void philosopher(int i)                 /* i: o numero do filosofo, de 0 a N–1 */
{
    while (TRUE) {                      /* repete para sempre */
        think( );                       /* o filosofo esta pensando */
        take_forks(i);                  /* pega dois garfos ou bloqueia */
        eat( );                         /* hummm, espaguete! */
        put_forks(i);                   /* devolve os dois garfos a mesa */
    }
}

void take_forks(int i)                  /* i: o numero do filosofo, de 0 a N–1 */
{
    down(&mutex);                       /* entra na regiao critica */
    state[i] = HUNGRY;                  /* registra que o filosofo esta faminto */
    test(i);                            /* tenta pegar dois garfos */
    up(&mutex);                         /* sai da regiao critica */
    down(&s[i]);                        /* bloqueia se os garfos nao foram pegos */
}

void put_forks(i)                       /* i: o numero do filosofo, de 0 a N–1 */
{
    down(&mutex);                       /* entra na regiao critica */
    state[i] = THINKING;                /* o filosofo acabou de comer */
    test(LEFT);                        /* ve se o vizinho da esquerda pode comer agora */
    test(RIGHT);                        /* ve se o vizinho da direita pode comer agora */
    up(&mutex);                         /* sai da regiao critica */
}

void test(i)/* i: o numero do filosofo, de 0 a N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

filósofo *i* são definidos pelas macros *LEFT* e *RIGHT*. Em outras palavras, se *i* é 2, *LEFT* é 1 e *RIGHT* é 3.

O programa usa um conjunto de semáforos, um por filósofo, portanto os filósofos com fome podem ser bloqueados se os garfos necessários estiverem ocupados. Observe que cada processo executa a rotina *philosopher* como seu código principal, mas as outras rotinas, *take_forks, put_forks* e *test*, são rotinas ordinárias e não processos separados.

2.5.2 O problema dos leitores e escritores

O problema do jantar dos filósofos é útil para modelar processos que estão competindo pelo acesso exclusivo a um número limitado de recursos, como em dispositivos de E/S. Outro problema famoso é o problema dos leitores e escritores (COURTOIS et al., 1971), que modela o acesso a um banco de dados. Imagine, por exemplo, um sistema de reservas de uma companhia aérea, com muitos processos competindo entre si desejando ler e escrever. É aceitável ter múltiplos processos lendo o banco de dados ao mesmo tempo, mas se um processo está atualizando (escrevendo) o banco de dados, nenhum outro pode ter acesso, nem mesmo os leitores. A questão é: como programar leitores e escritores? Uma solução é mostrada na Figura 2.48.

Nessa solução, para conseguir acesso ao banco de dados, o primeiro leitor realiza um down no semáforo *db*. Leitores subsequentes apenas incrementam um contador, *rc*. À medida que os leitores saem, eles decrementam o contador, e o último a deixar realiza um up no semáforo, permitindo que um escritor bloqueado, se houver, entre.

A solução apresentada aqui contém implicitamente uma decisão sutil que vale observar. Suponha que enquanto um leitor está usando o banco de dados, aparece outro leitor. Visto que ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Leitores adicionais também podem ser admitidos se aparecerem.

FIGURA 2.48 Uma solução para o problema dos leitores e escritores.

```
typedef int semaphore;                  /* use sua imaginacao */
semaphore mutex = 1;                    /* controla o acesso a 'rc' */
semaphore db = 1;                       /* controla o acesso a base de dados */
int rc = 0;                             /* numero de processos lendo ou querendo ler */

void reader(void)
{
     while (TRUE) {                     /* repete para sempre */
          down(&mutex);                 /* obtem acesso exclusivo a 'rc' */
          rc = rc + 1;                  /* um leitor a mais agora */
          if (rc == 1) down(&db);       /* se este for o primeiro leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          read_data_base( );            /* acesso aos dados */
          down(&mutex);                 /* obtem acesso exclusivo a 'rc' */
          rc = rc – 1;                  /* um leitor a menos agora */
          if (rc == 0) up(&db);         /* se este for o ultimo leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          use_data_read( );             /* regiao nao critica */
     }
}


void writer(void)
{
     while (TRUE) {                     /* repete para sempre */
          think_up_data( );             /* regiao nao critica */
          down(&db);                    /* obtem acesso exclusivo */
          write_data_base( );           /* atualiza os dados */
          up(&db);                      /* libera o acesso exclusivo */
     }
}
```

Agora suponha que um escritor apareça. O escritor pode não ser admitido ao banco de dados, já que escritores precisam ter acesso exclusivo, então ele é suspenso. Depois, leitores adicionais aparecem. Enquanto pelo menos um leitor ainda estiver ativo, leitores subsequentes serão admitidos. Como consequência dessa estratégia, enquanto houver uma oferta uniforme de leitores, todos eles entrarão assim que chegarem. O escritor será mantido suspenso até que nenhum leitor esteja presente. Se um novo leitor aparecer, digamos, a cada 2 s, e cada leitor levar 5 s para realizar o seu trabalho, o escritor jamais entrará.

Para evitar essa situação, o programa poderia ser escrito de maneira ligeiramente diferente: quando um leitor chega e um escritor está esperando, o leitor é suspenso atrás do escritor em vez de ser admitido imediatamente. Dessa maneira, um escritor precisa esperar por leitores que estavam ativos quando ele chegou, mas não precisa esperar por leitores que chegaram depois dele. A desvantagem dessa solução é que ela alcança uma concorrência menor e assim tem um desempenho mais baixo. Courtois et al. (1971) apresentam uma solução que dá prioridade aos escritores. Para detalhes, consulte o artigo.

2.6 Pesquisas sobre processos e threads

No Capítulo 1, estudamos algumas das pesquisas atuais sobre a estrutura dos sistemas operacionais. Neste capítulo e nos subsequentes, estudaremos pesquisas mais específicas, começando com processos. Como ficará claro com o tempo, alguns assuntos são muito menos controversos do que outros. A maioria das pesquisas tende a ser sobre os tópicos novos, em vez daqueles que estão por aí há décadas.

O conceito de um processo é um exemplo de algo que já está muito bem estabelecido. Quase todo sistema tem alguma noção de um processo como um contêiner para agrupar recursos relacionados como um espaço de endereçamento, threads, arquivos abertos, permissões de proteção e assim por diante. Sistemas diferentes realizam o agrupamento de maneira ligeiramente diferente, mas trata-se apenas de diferenças de engenharia. A ideia básica não é mais muito controversa, e há pouca pesquisa nova sobre processos.

O conceito de threads é mais recente do que o de processos, mas ele, também, já foi bastante estudado. Ainda assim, o estudo ocasional sobre threads aparece de tempos em tempos, como o estudo a respeito de aglomeração de threads em multiprocessadores (TAM et al., 2007), ou sobre quão bem os sistemas operacionais modernos como o Linux lidam com muitos threads e muitos núcleos (BOYD-WICKIZER, 2010).

Uma área de pesquisa particularmente ativa lida com a gravação e a reprodução da execução de um processo (VIENNOT et al., 2013). A reprodução ajuda os desenvolvedores a procurar erros difíceis de serem encontrados e especialistas em segurança a investigar incidentes.

De modo similar, muita pesquisa na comunidade de sistemas operacionais concentra-se hoje em questões de segurança. Muitos incidentes demonstraram que os usuários precisam de uma proteção melhor contra agressores (e, ocasionalmente, contra si mesmos). Uma abordagem é controlar e restringir com cuidado os fluxos de informação em um sistema operacional (GIFFIN et al., 2012).

O escalonamento (tanto de uniprocessadores quanto de multiprocessadores) ainda é um tópico que mora no coração de alguns pesquisadores. Alguns tópicos sendo pesquisados incluem o escalonamento de dispositivos móveis em busca da eficiência energética (YUAN e NAHRSTEDT, 2006), escalonamento com tecnologia hyperthreading (BULPIN e PRATT, 2005) e escalonamento *bias-aware* (KOUFATY, 2010). Com cada vez mais computação em smartphones com restrições de energia, alguns pesquisadores propõem migrar o processo para um servidor mais potente na nuvem, quando isso for útil (GORDON et al, 2012). No entanto, poucos projetistas de sistemas andam preocupados com a falta de um algoritmo de escalonamento de threads decente, então esse tipo de pesquisa parece ser mais um interesse de pesquisadores do que uma demanda de projetistas. Como um todo, processos, threads e escalonamento, não são mais os tópicos quentes de pesquisa que já foram um dia. A pesquisa seguiu para tópicos como gerenciamento de energia, virtualização, nuvens e segurança.

2.7 Resumo

A fim de esconder os efeitos de interrupções, os sistemas operacionais fornecem um modelo conceitual que consiste de processos sequenciais executando em paralelo. Processos podem ser criados e terminados dinamicamente. Cada processo tem seu próprio espaço de endereçamento.

Para algumas aplicações é útil ter múltiplos threads de controle dentro de um único processo. Esses threads são escalonados independentemente e cada um tem sua própria pilha, mas todos os threads em um processo compartilham de um espaço de endereçamento comum.