2

PROCESSOS

Estamos prestes a iniciar um estudo detalhado de como os sistemas operacionais são projetados e construídos. Portanto, vamos apresentar neste capítulo o conceito mais importante da teoria dos sistemas operacionais, o conceito de *processo*. Tal conceito, em última análise, vem a ser uma abstração de um programa em execução. Tudo o mais gira em torno do conceito de processo, sendo então de fundamental importância que seu estudo seja o primeiro tópico da teoria dos sistemas operacionais. Neste capítulo usaremos eventualmente o exemplo do UNIX, uma vez que ele tem uma forma bastante simples, mas muito poderosa para gerenciar processos.

2.1 INTRODUÇÃO AO CONCEITO DE PROCESSO

Todos os computadores modernos podem fazer divesas coisas ao mesmo tempo. Enquanto estiver rodando um programa de usuário, o computador pode também ler de um disco e imprimir em uma impressora. Nos sistemas multiprogramados, o processador pode ser chaveado entre diversos programas, dando a cada um dezenas ou centenas de milissegundos de processamento. Deve-se observar que, apesar de só haver um programa executando em determinado instante de tempo, no curso de um segundo, o processador pode ter trabalhado para diversos programas, dando então aos usuários a ilusão do paralelismo de execução. Alguns autores referem-se ao **pseudoparalelismo (paralelismo aparente)** como forma de designar o rápido chaveamento do único processador disponível entre diversos programas, criando a ilusão da simultaneidade de execução, em contraste com o paralelismo real do hardware, onde o processador está trabalhando ao mesmo tempo que um ou mais dispositivos de ENTRADA/SAÍDA estão processando informações. O tratamento rigoroso de múltiplas atividades realizadas em paralelo não é tarefa fácil. Por isso, os estudiosos da teoria dos sistemas operacionais têm trabalhado no aperfeiçoamento de um modelo conceitual, cujo objetivo é o de tornar o mais simples possível o tratamento do paralelismo. Tal modelo é o assunto deste capítulo.

2.1.1. O Modelo do Processo

Neste modelo, todo software passível de rodar em um computador, muitas vezes incluindo o próprio sistema operacional, é organizado como um conjunto de **processos seqüenciais**, ou simplesmente **processos.** Um processo nada mais é que um programa em execução, incluindo os valores correntes de todos os registradores do hardware, e das variáveis, manipuladas por ele no curso de sua execução. Conceitualmente, cada processo tem o seu próprio processador virtual. Na realidade, o que ocorre é o chaveamento do processador entre cada um dos processos, mas para entender melhor o conceito é mais fácil imaginar um conjunto de processos rodando em paralelo (pseudoparalelismo), em vez de entrar nos detalhes de como o processador é chaveado de um processo para outro. O rápido chaveamento do processador entre diversos programas em execução (processos) é denominado **multiprogramação,** conforme mencionado anteriormente. Na Fig. 2.1(a), podemos observar como se realiza a multiprogramação de um processador entre quatro programas armazenados na memória do computador.

Na Fig. 2.1(b) aparece uma abstração da execução destes quatro processos, cada um tendo seu próprio fluxo de controle (contador de programa), e rodando de forma independente dos demais. Na Fig. 2.1(c), podemos observar que, se analisarmos um intervalo de tempo suficientemente grande, todos os processos terão progredido no decorrer deste intervalo; porém, num dado instante de tempo apenas um processo está efetivamente progredindo em sua execução.

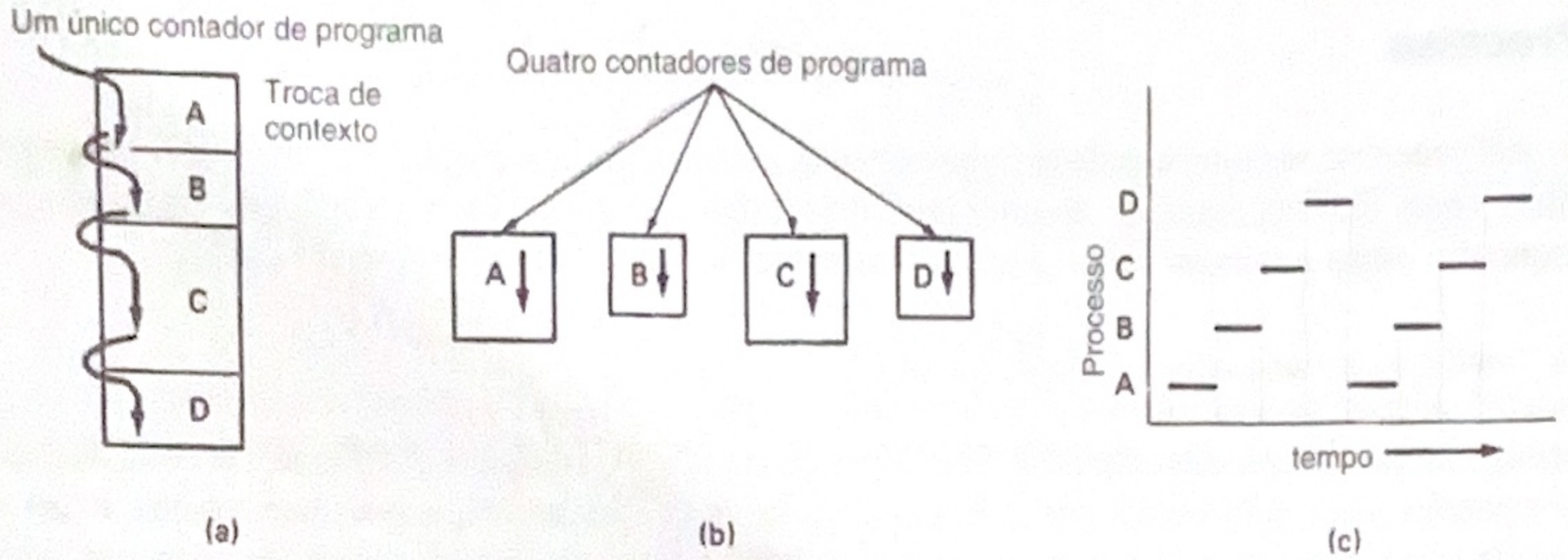

Fig. 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos seqüenciais independentes. (c) Somente um dos programas está ativo em um dado instante de tempo.

Com o processador sendo chaveado entre diversos processos, pode-se observar que a velocidade de execução de determinado processo não é constante, e provavelmente nunca conseguiremos que o mesmo processo seja executado por duas vezes, gastando em ambas o mesmo tempo de execução. Por este motivo, a programação dos processos não pode ter restrições de tempo. Considere, por exemplo, um processo ENTRADA/SAÍDA que esteja acessando uma unidade de fita magnética. Caso a fita esteja parada, o processo deve executar um loop de, digamos, 1.000 instruções nop (no-operation) para aguardar que a fita atinja sua velocidade nominal. Só então ele pode emitir um comando para leitura no primeiro registro. Imagine se o processador for forçado a trocar de contexto, durante o loop de nop's. Neste caso, o primeiro registro a ser lido pode passar pela cabeça de gravação quando o processo estiver inativo, ou seja, quando o processador estiver alocado a outro processo. Portanto, podemos concluir que quando os processos tiverem características de processamento em tempo real, alguns cuidados especiais devem ser tomados em sua programação. No entanto, o resultado final da maioria dos processos não é afetado pelos detalhes relativos à multiprogramação do processador, nem pelas velocidades relativas dos diversos processos do sistema.

A diferença entre um processo e um programa é sutil, porém crucial. Uma analogia pode ajudar bastante a clarear esta questão. Considere o caso de um cientista da computação que tenha acentuados dotes culinários, e que esteja preparando um bolo para o aniversário de sua filha. Ele tem à sua disposição uma receita de bolo, e uma dispensa com os ingredientes necessários à sua preparação: farinha, ovos, açúcar, baunilha etc. Neste exemplo, a receita é o programa, ou seja, um algoritmo expresso em alguma linguagem conhecida, o cientista da computação é o processador, e os ingredientes do bolo são os dados de entrada. Já o processo vem a ser a atividade resultante da preparação do bolo: leitura da receita, busca dos ingredientes, mistura dos ingredientes etc.

Agora imagine que a filha do nosso cientista apareça chorando, por haver sido picada por uma abelha, enquanto o pai está fazendo o bolo. Neste caso, o cientista deve guardar em que ponto da receita ele estava (o estado do processo corrente é salvo), apanhar um livro de primeiros socorros e seguir as instruções contidas no capítulo do livro que trata de picadas de abelha. Aqui vemos o processador, mais precisamente o cientista da computação, sendo chaveado de um processo (preparação do bolo), para outro de maior prioridade (administração de cuidados médicos), cada um deles constituído de um programa diferente (receita *versus* livro de primeiros socorros). Quando o ferrão da abelha tiver sido retirado da filha do cientista, este poderá voltar à preparação do bolo, continuando seu trabalho a partir do ponto onde foi interrompido.

A idéia central é que o processo constitui-se de um certo tipo de atividade. Ele possui um programa, uma entrada, uma saída e um estado. Um único processador pode ser compartilhado entre vários processos, com o uso de algum algoritmo de escalonamento para determinar quando o trabalho em um deles deve ser interrompido e qual o próximo processo a ser servido.

Hierarquias de Processos

Os sistemas operacionais que suportam o conceito de processo devem fornecer algum meio de criar todos os processos necessários. Em sistemas muito simples, ou naqueles projetados para rodar somente uma única aplicação, é possível ter todos os processos necessários criados quando o sistema é inicializado. Na grande maioria dos sistemas, é preciso haver um mecanismo que permita criar e destruir processos, quando necessário, durante a operação.

No UNIX, processos são criados pela chamada de sistema FORK, cuja execução cria um processo idêntico àquele que a chamou. Após o FORK, o processo-pai continua rodando em paralelo com o filho. O pai pode executar mais chamadas FORK, podendo portanto ter diversos filhos executando em determinado instante de tempo. Os filhos também podem executar chamadas FORK, formando uma árvore de processos, cuja profundidade é arbitrária.

No MS-DOS existe uma chamada de sistema para carregar um arquivo binário específico da memória, e executá-lo como um processo-filho. Em contraste com o UNIX, esta chamada do MS-DOS suspende a execução do processo-pai até que o filho venha a terminar sua execução, fazendo com que pai e filho não possam ser executados em paralelo.

Estados do Processo

Apesar de cada processo ser uma entidade independente, com seu próprio fluxo de controle e com seu próprio estado interno, eles muitas vezes têm necessidade de interagir com outros processos. Por exemplo, determinado processo pode gerar uma saída que um outro processo venha a usar como entrada. No comando abaixo do UNIX,

```
cat capítulo1 capítulo2 capítulo3 | grep árvore
```

o primeiro processo, *cat*, concatena três arquivos, gerando uma saída com o arquivo resultante da concatenação. O segundo processo em execução, *grep*, seleciona todas as linhas do arquivo concatenado que possuam a palavra "árvore". Dependendo da velocidade relativa entre os dois processos, que é função tanto da complexidade de cada um deles quanto do tempo que o processador vier a alocar a cada um, pode ocorrer que grep esteja pronto para rodar, mas que não haja ainda nenhuma entrada disponível para permitir sua execução. Neste caso, ele precisa ser **bloqueado** até que uma entrada esteja disponível.

O bloqueio de um processo ocorre normalmente pelo fato de ele não ter condições lógicas para prosseguir com sua execução, tipicamente por estar aguardando uma entrada ainda não disponível. É possível também efetuar o bloqueio de um processo que tenha condições de rodar. Tal bloqueio pode ocorrer por força da decisão do sistema operacional de entregar o processador a um outro processo, durante certo intervalo de tempo. As duas condições de bloqueio acima mencionadas são completamente diversas. No primeiro caso, o motivo da suspensão é óbvio, como por exemplo o fato de o sistema operacional não poder processar um comando até que ele tenha sido completamente teclado pelo usuário. No segundo caso, a decisão de bloquear o prosseguimento do processo é uma decisão técnica, tomada pelo sistema operacional, normalmente baseada no fato de só haver um processador e muitos usuários. Na Fig. 2.2 vemos um diagrama de estados mostrando os três estados pelos quais um processo pode passar no decorrer de seu processamento:

1. Rodando, ou seja, usando o processador neste instante.
2. Pronto, ou seja, em condições de rodar, mas bloqueado temporariamente para dar vez a um outro processo.
3. Bloqueado, ou seja, impedido de rodar até que ocorra um determinado evento externo ao processo.

Logicamente, os dois primeiros estados são similares. Em ambos, o processo está apto a rodar, só que no segundo caso não há processador disponível para ele. O terceiro estado é diferente dos dois primeiros, pois neste caso o processo não pode rodar mesmo que o processador esteja disponível.

São possíveis quatro transições entre os três estados dos processos, como se pode observar na Fig. 2.2. A transição 1 ocorre quando o processo descobre que não tem mais condições lógicas de prosseguir em seu processamento. Em alguns sistemas, tal processo executa uma chamada de sistema, BLOCK, indo para o estado bloqueado. Mais comumente, quando o processo precisa ler de um pipe ou de um arquivo especial, um terminal por exemplo, e não há entrada disponível, ele é automaticamente bloqueado.

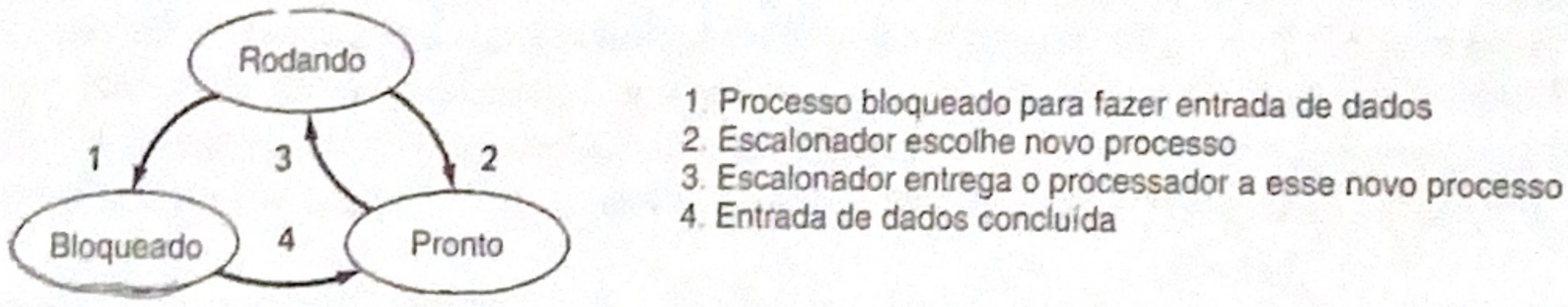

Fig. 2.2 Um processo pode estar rodando, bloqueado ou pronto para rodar. As transições entre tais estados ocorrem conforme mostrado.

As transições 2 e 3 são causadas pelo escalonador de processos, uma rotina do sistema operacional, sem que o processo tome conhecimento delas. A transição 2 acontece quando o escalonador decide que o processo corrente já ocupou o processador por tempo suficiente, sendo portanto o momento de deixar que outro processo execute. A transição 3 ocorre quando todos os demais processos já tiveram oportunidade de rodar, sendo então a vez de o primeiro processo pronto rodar novamente. A função do escalonador de processos — decidir qual processo vai rodar e por quanto tempo — é importantíssima, sendo objeto de estudo mais tarde, ainda neste capítulo. Muitos algoritmos foram elaborados na tentativa de balancear a alocação justa do processador a processos de acordo com a demanda gerada por eles, e a eficácia global do sistema.

A transição 4 ocorre quando acontece um evento externo pelo qual o processo estava aguardando, por exemplo, a conclusão de uma operação de entrada/saída. Se nenhum outro processo estiver rodando neste momento, a transição 3 é acionada imediatamente, e o processo que aguardava a condição vai rodar. Caso contrário, ele vai para o estado de *pronto* até que o processador fique disponível e o escalonador o escolha para rodar.

Usando o modelo do processo, fica mais fácil visualizar o que ocorre dentro do sistema. Alguns dos processos rodam programas que executam comandos de usuário. Outros são parte do sistema operacional e tratam de tarefas como a manipulação de arquivos ou a gerência da transferência de informações de um disco para uma fita. Quando ocorre a inter-

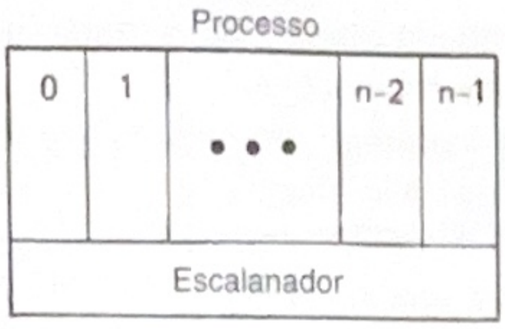

Fig. 2.3 A camada mais baixa de um sistema operacional estruturado no modelo do processo manipula interrupções e realiza funções de escalonamento do processador. O resto do sistema operacional é constituído de processos seqüenciais.

rupção do disco, o sistema operacional interrompe o processo corrente e coloca para rodar o processo responsável pela gerência do disco, que estava bloqueado, no aguardo da interrupção. Desta forma, em vez de pensar em interrupções, podemos tratar com processos de usuário, processos de terminal, e assim por diante, que estarão bloqueados quando no aguardo de eventos a ocorrer. Quando o bloco de informações do disco estiver lido ou quando o comando tiver sido teclado no terminal, o processo correspondente é ativado e colocado para rodar.

Esta visão dá origem ao modelo mostrado na Fig. 2.3. Nele, o nível mais baixo do sistema operacional é o escalonador de processos, com uma grande variedade de processos acima dele. Todo o tratamento das interrupções e os detalhes de ativar e bloquear processos são escondidos dentro do escalonador, o qual é programado em código de montagem. O resto do sistema operacional é estruturado confortavelmente na forma de processos.

2.1.2. Implementação de Processos

Para implementar o modelo de processo, o sistema operacional deve manter uma tabela, chamada **tabela de processos**, com uma entrada por processo. Esta entrada contém informações sobre o estado do processo, sobre a memória alocada, os valores de seu contador de programa e de seu apontador de pilha, o estado de seus arquivos abertos, sua contabilidade no uso de recursos, sua prioridade, e tudo o mais que for necessário guardar quando o processo passar do estado *rodando* pa-ra o *estado pronto,* de maneira que seja possível reiniciar seu processamento mais tarde, como se nada tivesse acontecido.

Apesar de as informações contidas na tabela de processos variarem de sistema operacional para sistema operacional, sempre algumas dizem respeito à gerência do processo, outras à gerência da memória, e outras ao sistema de arquivos. A Fig. 2.4 mostra alguns dos campos mais conhecidos da tabela de processos do UNIX.

Agora que tomamos conhecimento de alguns detalhes sobre a tabela de processos, é possível explicar um pouco mais sobre a questão de se manter vários processos seqüenciais ativos, em uma máquina com um só processador, e diversos dispositivos de entrada/saída. Associado a cada classe de dispositivos de entrada/saída (discos flexíveis, discos rígidos, terminais, temporizadores), existe uma informação armazenada em um endereço normalmente situado na parte baixa da memória, chamada **vetor de interrupção,** que contém o endereço da rotina de serviço da interrupção referente a uma particular classe de dispositivos. Como exemplo, suponha que o processo de usuário número 3 está rodando quando ocorre uma interrupção de disco. Os valores do contador de programa, da PSW e de alguns outros registradores são colocados na pilha pelo hardware responsável pela interrupção. A seguir, este hardware carrega no contador de programa o valor contido no endereço especificado pelo vetor de interrupção de disco. Isto é tudo que o hardware faz. Daqui para a frente é por conta do software.

A rotina de serviço da interrupção começa seu processamento salvando o valor de todos os registradores na entrada da tabela de processos, correspondente ao processo corrente. O número do processo corrente e um ponteiro para sua entrada na tabela de processos são mantidos como variáveis globais do sistema, de modo a acessá-los rapidamente. Então as informações colocadas na pilha pela interrupção são removidas, e o apontador de pilha é ajustado para apontar para uma pilha temporária, utilizada pelo processo que serve a interrupção. Ações como o salvamento de registradores e a colocação de valores no apontador de pilha não podem ser expressas em C (nem em nenhuma outra linguagem de alto nível), de

Gerência do processo	Gerência da mamária	Gerência de arquivos
Valor dos registradores	Ponteiro para o segmento de texto	Máscara UMASK
Valor do contador de programa	Ponteiro para o segmento de dados	Diretório raiz
Valor da palavra de estado (PSW)	Ponteiro para o segmento bss	Diretório de trabalho
Valor do apontador de pilha	Estado da saída	Descritores de arquivo
Estado do processo	Estado do sinal	Identificação efetiva use
Instante de início do processo	Identificação do processo	Identificação efetiva grp
Tempo de processador utilizado	Processo-pai	Parâmetros de chamadas
Tempo de processador do processo-filho	Grupo de processos	Diversos bits de flag
Tempo de ocorrência do próximo alarme	Identificação real do usuário	
Ponteiros para as filas de mensagem	Identificação efetiva do usuário	
Bits de sinal pendentes	Identificação real do grupo de usuários	
Identificação do processo	Identificação efetiva do grupo de usuários	
Diversos bits de flag	Mapas de bits para sinais	
	Diversos bits de flag	

Fig. 2.4 Alguns campos da tabela de processos do UNIX.

forma que são realizadas por um pequeno código em linguagem de montagem. Quando esta rotina termina, ela chama um procedimento em C para realizar o tratamento da interrupção.

O próximo passo é a identificação do processo que requisitou trabalho do disco e que conseqüentemente gerou a interrupção. Via de regra, este processo foi posto para dormir após solicitar o serviço, de forma que agora ele deve ser acordado. O estado deste processo é trocado de *bloqueado* para *pronto*, e o escalonador é chamado.

A identidade do processo da fila de prontos que vai ser escolhido para rodar depende do algoritmo implementado no escalonador. Sabemos, com certeza, que neste instante existem pelo menos dois processos prontos: o processo que solicitou serviço ao disco e o processo que foi interrompido pelo disco. É tarefa do escalonador escolher o mais adequado para rodar. No UNIX, é dada uma prioridade maior àqueles processos que fazem muita entrada/saída (I/O-bound), em detrimento daqueles que consomem muito processador (CPU-bound), pois os primeiros poderão rapidamente dar início a novas operações de entrada/saída, deixando livre o processador. Tal estratégia maximiza o grau de paralelismo do sistema, pois procura manter tanto o processador quando os dispositivos de entrada/saída ocupados a maior parte do tempo. Vale observar que outros sistemas operacionais usam esquemas diferentes para escalonamento de processos, enfatizando por exemplo a igual oportunidade entre os processos, em detrimento da eficiência do sistema.

Para entregar o processador ao processo que acabou de ser selecionado, o procedimento em C chamado pela rotina de interrupção, escrita em linguagem de montagem, retorna a esta última rotina, que restaura os valores dos registradores e da memória para que o novo processo possa reiniciar seu processamento. O trabalho do manipulador de interrupção e do escalonador são resumidos a seguir, na Fig. 2.5.

1. O hardware empilha o contador de programa etc.
2. O hardware obtém o novo valor do contador de programa a partir do vetor de interrupções.
3. Um procedimento em linguagem de montagem salva o valor dos registradores.
4. Este mesmo procedimento monta uma nova pilha.
5. Um procedimento em C executa o serviço solicitado pela interrupção.
6. O escalonador decide qual processo vai rodar em seguida.
7. O procedimento em C retorna ao procedimento em código de montagem.
8. Este procedimento executa as funções necessárias ao reinício do processo escolhido pelo escalonador.

Fig. 2.5 Esboço do que o sistema operacional faz em presença de uma interrupção.

2.2. COMUNICAÇÃO ENTRE PROCESSOS

Os processos precisam freqüentemente se comunciar com outros processos. Por exemplo, num pipe do shell, a saída do primeiro processo precisa ser passada à entrada do segundo, e assim por diante, ao longo de todo o pipe. Existe, portanto, a necessidade de haver comunicação entre tais processos, preferencialmente de uma forma estruturada, que não seja baseada em interrupções. Nas seções seguintes, vamos examinar alguns aspectos relativos à comunicação entre processos, conhecida pela sigla IPC (**InterProcess Communication**).

2.2.1. Condições de Corrida

Em alguns sistemas operacionais, processos que estão trabalhando em conjunto muitas vezes utilizam uma memória comum, onde cada processo pode ler ou escrever. A memória compartilhada pode ser a principal ou pode ser um arquivo em disco, sendo que a natureza da memória compartilhada não influi de forma alguma na naturea da comunicação nem nos problemas que são criados. Para se ter idéia de como a comunicação entre processos funciona na prática, vamos considerar um exemplo bem simples e bastante comum, um spool de impressão. Quando um processo deseja imprimir um arquivo, ele entra com o nome do arquivo em um diretório especial de spool. Outro processo, o **impressor,** verifica periodicamente este diretório, para ver se há arquivos a serem impressos. Havendo, eles são impressos e seus nomes são retirados do diretório.

Imagine que nosso diretório de spool tenha um grande número (potencialmente infinito) de entradas, numeradas como 0, 1, 2, 3, ..., cada uma delas capaz de armazenar um nome de arquivo. Considere que existam duas variáveis compartilhadas: *out*, que aponta para o nome do próximo arquivo a ser impresso, e *in*, que aponta para a próxima entrada livre do diretório. Tais variáveis devem ser mantidas em duas palavras de memória, acessíveis a todos os processos. Suponha que em determinado instante, as entradas de 0 a 3 estejam vazias (ou seja, os arquivos apontados por elas já foram impressos), e as entradas de 4 a 6 estejam ocupadas (contendo os nomes dos arquivos na fila de impressão). Mais ou menos simultaneamente, os processos *A* e *B* decidem colocar na fila um arquivo para impressão. Tal situação é ilustrada na Fig. 2.6.

Nos assuntos onde a Lei de Murphy pode ser aplicada, pode e deve ocorrer o seguinte: o processo *A* lê *in* e armazena o valor lido 7, numa variável local *próxima_entrada_livre*. Exatamente após esta operação ter sido concluída, ocorre uma interrupção de tempo, significando que o processo *A* já rodou o suficiente, devendo portanto ceder o processador para o processo *B*. Este lê o valor de *in*, obtendo o valor 7, mesmo valor que *A* obteve quando realizou a leitura. Desta forma, *B* guarda o nome do seu arquivo de impressão na entrada 7 do diretório, e atualiza o valor de *in* para 8. Após isto, ele continua seu processamento normalmente.

Já sabemos que em algum momento seguinte, o processo *A* vai reiniciar seu processamento no exato ponto onde houve a interrupção. Ao examinar o valor de sua variável local *próxima_entrada_livre*, enconra 7 armazenado nela, e es-

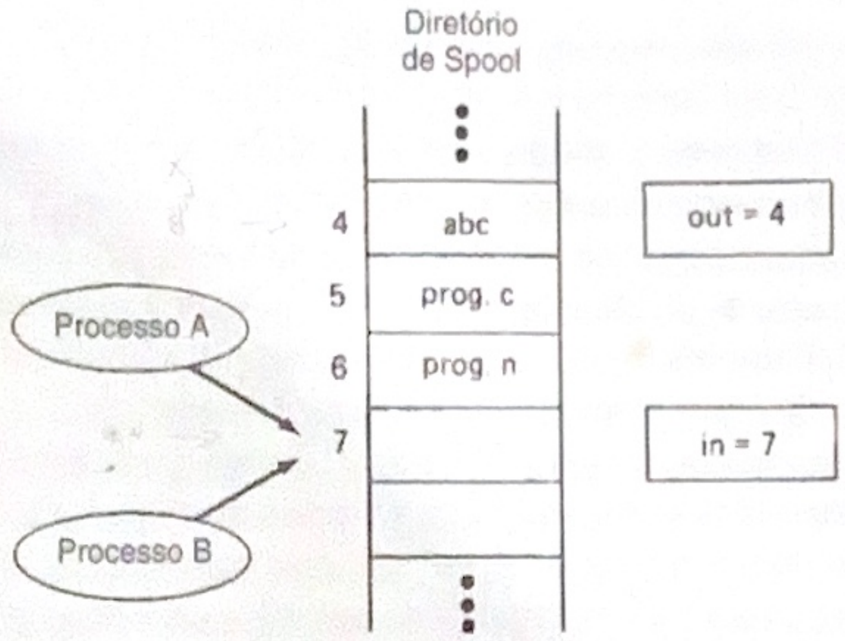

Fig. 2.6 Dois processos acessam a memória compartilhada ao mesmo tempo.

creve o nome de seu arquivo de impressão na entrada 7 do diretório, apagando o nome do arquivo escrito anteriormente por *B*, e ainda não impresso. Depois disso, o processo atualiza o valor de *in* para 8 e continua seu processamento. Observe que o diretório de spool está consistente, de forma que o processo impressor não vai notar nada de errado na estrutura do diretório. Ocorre que o processo *B* nunca vai ter sua saída impressa. Situações como esta, onde dois ou mais processos estão acessando dados compartilhados, e o resultado final do processamento depende de quem roda quando, são chamadas de **condições de corrida.** A análise de programas em que ocorrem condições de corrida não é uma tarefa fácil. Os resultados da maioria dos testes são consistentes com a estrutura do programa, mas de vez em quando ocorre algo imprevisto e inexplicável, em função do não-determinismo potencial, causado pelas condições de corrida.

2.2.2. Regiões Críticas

Como evitar a ocorrência de condições de corrida? A questão-chave para evitar problemas em casos envolvendo qualquer tipo de compartilhamento de recursos (memória, arquivo, e tudo o mais) é encontrar alguma forma de proibir que mais de um processo acesse o dado compartilhado ao mesmo tempo (pseudoparalelismo ou paralelismo real). Em outras palavras, precisamos encontrar um meio de implementar a **exclusão mútua** de execução — uma forma de se ter certeza de que se um processo estiver usando uma variável ou um arquivo compartilhados, os demais serão impedidos de fazer a mesma coisa. A dificuldade identificada no exemplo acima vem do fato de o processo *B* ter começado a usar uma das variáveis compartilhadas antes que o processo *A* tenha terminado de usar esta mesma variável. A escolha das operações primitivas adequadas à implementação da exclusão mútua de execução é um dos grandes desafios no projeto de qualquer sistema operacional. O assunto será examinado em detalhes nas seções seguintes.

O problema de se evitarem as condições de corrida pode também ser formulado de modo abstrato. Parte do tempo, um processo está ocupado realizando um processamento que não resultará em condição de corrida, por não estar manipulando dados ou arquivos compartilhados. No entanto, em outros momentos, o processo pode estar acessando uma parte da memória ou um arquivo compartilhado com outros processos ou realizando algum outro tipo de tarefa crítica que possa vir a levar a ocorrência de condições de corrida. Esta parte do programa, cujo processamento pode levar à ocorrência de condições de corrida, é denominada **região crítica** ou **seção crítica**. Se pudermos arranjar as coisas de forma que não seja nunca permitido que dois ou mais processos estejam processando suas seções críticas correspondentes* ao mesmo tempo, poderemos evitar a ocorrência das condições de corrida.

Apesar do esquema descrito acima evitar a ocorrência das condições de corrida, ele não é suficiente para permitir que processos paralelos cooperem correta e eficientemente no uso de recursos compartilhados. É necessário que quatro condições sejam atendidas para que tenhamos uma boa solução para o problema:

1. Dois ou mais processos não podem estar simultaneamente dentro de suas regiões críticas correspondentes.
2. Nenhuma consideração pode ser feita a respeito da velocidade relativa dos processos, ou a respeito do número de processadores disponíveis no sistema.
3. Nenhum processo que esteja rodando fora de sua região crítica pode bloquear a execução de outro processo.
4. Nenhum processo pode ser obrigado a esperar indefinidamente para entrar em sua região crítica.

2.2.3. Exclusão Mútua com Espera Ocupada

Nesta seção vamos examinar várias propostas para obtenção da exclusão mútua, de forma que quando um processo estiver acessando a memória compartilhada dentro de sua região crítica, nenhum outro processo poderá entrar em sua seção crítica correspondente.

*Regiões críticas referentes à mesma variável compartilhada. (N. do T.)

Inibição das Interrupções

A solução mais simples para tal problema é obtida fazendo com que cada processo iniba as interrupções logo após seu ingresso em uma região crítica, habilitando-as outra vez imediatamente antes de deixá-la. Com as interrupções inibidas, o processo não poderá ser interrompido por exceder o tempo de processamento concedido a ele. Como o processador só pode ser chaveado entre processos através de interrupções de tempo, com as interrupções inibidas não haverá chaveamento. Então, uma vez inibidas as interrupções, o processo pode examinar e atualizar a memória compartilhada, sem se preocupar com a eventual intervenção de outro processo em seu processamento.

Tal solução não se revela atrativa, pois não é boa prática dar a processos de usuários o poder de desabilitar interrupções. O que ocorrerá se um de tais processos inibir as interrupções do sistema e não habilitá-las novamente? Isto pode causar um problema muito sério ao sistema. Além do mais, em sistemas com mais de um processador, a inibição das interrupções afeta apenas o processador que executou a instrução de inibição. Os outros processadores porventura existentes vão continuar rodando normalmente, com seus processos podendo acessar a memória compartilhada.

Por outro lado, é muito conveniente que o kernel possa inibir as interrupções do sistema enquanto algumas instruções que realizam tarefas críticas estejam sendo executadas. Se uma interrupção vier a ocorrer enquanto o kernel estiver atualizando a fila de processos prontos, esta poderá ficar inconsistente, tornando possível a ocorrência de condições de corrida. A conclusão então é: inibir interrupção é uma função útil, se só puder ser realizada dentro do kernel, não sendo apropriada para servir como mecanismo para implementação de exclusão mútua de execução entre processos de usuários.

Variáveis de Travamento

Como segunda tentativa de implementação de exclusão mútua de execução, vamos examinar uma solução implementada por software. Considere a existência de uma única variável compartilhada (variável de travamento), cujo valor seja inicialmente 0. Quando um processo desejar entrar em sua região crítica, ele deve primeiro testar a variável de travamento. Se ela for 0, o processo muda seu valor para 1 e entra na região crítica. Se o valor da variável de travamento já for 1, o processo deve esperar que ele volte a 0, antes de entrar na região crítica. Nesta solução, o valor da variável de travamento em 0 significa que não há nenhum processo executando a sua região crítica, e o valor em 1 significa que algum processo está executando sua região crítica.

Infelizmente esta idéia incorre na mesma falha do diretório de spool. Suponha que um processo leia a variável de travamento e verifique que seu valor é 0. Antes que ele possa atualizá-lo para 1, ocorre uma interrupção, e outro processo é posto para rodar. Tal processo encontra a variável de travamento em 0, atualiza seu valor para 1 e entra na sua região crítica. Ocorre que o processo interrompido guardou em seu contexto o valor 0 para a variável de travamento, de modo que, quando ele for reativado, atualizará seu valor para 1 e executará sua região crítica. Observe então que dois processos poderão estar executando suas regiões críticas correspondentes ao mesmo tempo.

Poder-se-ia pensar em contornar o problema, lendo o valor da variável de travamento e verificando-o mais uma vez imediatamente antes de armazená-lo. Isto não funciona, pois uma condição de corrida pode ocorrer se o segundo processo alterar o valor da variável logo após o primeiro processo ter realizado sua segunda verificação do valor da variável.

Estrita Alternância

Uma terceira forma de abordar a questão da exclusão mútua é mostrada na Fig. 2.7. O trecho de programa que aparece na figura está escrito em C, como a grande maioria dos demais que aparecerão ao longo deste livro. Poderíamos ter escolhido Modula 2 ou Pascal em vez de C. Cada uma destas linguagens tem seus defensores e seus detratores. Escolhemos C, pois existem diversos sistemas operacionais escritos nesta linguagem, poucos escritos em Modula 2, e nenhum escrito em Pascal. Se você não estiver familiarizado com a linguagem C, é favor ler o Apêndice B, onde é apresentada uma introdução a ela.

Na Fig. 2.7, a variável inteira *turn*, inicialmente em 0, estabelece de quem é a vez de entrar na região crítica. Primeiramente, o processo 0 inspeciona *turn*, encontra seu valor em 0 e entra em sua região crítica. O processo 1 também encontra *turn* em 0, entrando em um loop, que testa continuamente o valor de *turn*, aguardando que ele se torne igual a 1. O teste contínuo do valor de uma variável, aguardando que ela assuma determinado valor é denominado de **espera ocupada**. A espera ocupada deve ser evitada, em razão de consumir tempo do processador. Somente nos casos em que houver uma razão muito forte para supor que a espera será muito curta é que devemos usar a espera ocupada.

```
while (TRUE) {
  while (turn != 0) /* espera */;
  regiao critica ( );
  turn = 1;
  regiao nao-critica ( );
}
```

(a)

```
while (TRUE) {
  while (turn != 1) /* espera */;
  regiao critica ( );
  turn = 0;
  regiao nao-critica ( );
}
```

(b)

Fig. 2.7 Uma proposta de solução para o problema da exclusão mútua.

Quando o processo 0 deixar sua região crítica, ele faz com que *turn* assuma o valor 1, para permitir que o processo 1 entre em sua região crítica. Suponha que o processo 1 termine rapidamente a execução de sua região crítica, fazendo com que ambos os processos estejam fora de suas regiões críticas, com *turn* igual a 0. O processo 0 executa seu loop rapidamente, retornando à sua seção não-crítica fazendo *turn* igual a 1, e voltando para o início do loop. Infelizmente ele não pode executar sua região crítica, mesmo com o processo 1 em seu trecho não-crítico, pois *turn* vale 1. Podemos concluir que a estrita alternância de processos executando suas regiões críticas não é uma boa idéia quando um dos processos é muito mais lento que o outro.

Esta situação viola a condição 3 estabelecida acima, pois o processo 0 está sendo bloqueado por um processo que não está em sua região crítica. Voltando ao assunto do diretório de spool, discutido anteriormente, se nós associássemos a região crítica com as operações de leitura e escrita do diretório, o processo 0 não iria poder imprimir outro arquivo, pois o processo 1 estava fazendo alguma outra coisa.

De fato, esta solução requer a entrada estritamente alternada de dois processos em suas regiões críticas. Apesar de este algoritmo representar uma solução para o problema da exclusão mútua, tal solução não é viável na prática.

Solução de Peterson

Combinando a idéia de estabelecer a vez dos processos entrarem em suas regiões críticas, com a da variável de travamento e com a de uma variável de tensão, um matemático holandês, T. Dekker, foi o primeiro a criar uma solução em software para o problema da exclusão mútua, que não se baseava na estrita alternância de execução da região crítica. Para uma discussão do algoritmo de Dekker, veja Dijkstra (1965).

Em 1981, G. L. Petersen descobriu uma forma muito mais simples que a de Dekker de se obter exclusão mútua, relegando a solução de Dekker à obsolescência. O algoritmo de Peterson aparece na Fig. 2.8, consistindo em dois procedimentos escritos em C ANSI, o que permite o fornecimento de protótipos para todas as funções definidas e usadas. Convencionalmente, tais protótipos são agrupados juntos em cabeçalhos de arquivos. A primeira linha da Fig. 2.8 inclui todos estes protótipos. Os detalhes não são de muita importância, não sendo, portanto, listados neste livro.

Antes de usar as variáveis compartilhadas (ou seja, antes de entrar na sua região crítica), cada processo deve chamar *enter_region*, com seu próprio número, 0 ou 1, como parâmetro. Esta chamada vai fazer com que o processo espere, se necessário, até que seja seguro entrar na região crítica. Quando ele terminar a execução da região crítica, deve chamar *leave_region*, permitindo a entrada do outro processo na região crítica, se for o caso.

Vamos ver como esta solução funciona. Inicialmente, nenhum processo está executando sua região crítica. Em seguida, o processo 0 chama *enter_region*. Ele indica seu interesse em executar a região crítica, sentando seu elemento do vetor *interested*, e fazendo *turn* igual a 0. Uma vez que o processo 1 não está interessado na execução de sua região crítica, ele sai imediatamente do procedimento enter_region. Se o processo 1 chamar *enter_region*, ele vai ficar preso nesta rotina até que interested [0] seja *FALSO*, o que só ocorrerá quando o processo 0 chamar *leave_region*.

Agora considere o caso de ambos os processos chamarem *enter_region* quase simultaneamente. Ambos vão armazenar seus números em *turn*. O armazenamento que for feito por último será o que vai valer, sendo o primeiro perdido. Suponha que o processo 1 armazene por último, de maneira que *turn* será 1. Quando ambos os processos executarem o comando while, o processo 0 vai executá-lo zero vezes, e entrar em sua região crítica. O processo 1 fica em loop no while, não chegando a executar sua região crítica.

```
# include "prototypes.h"
# define FALSE      0
# define TRUE       1
# define N          2                       /* numero de processos */

int turn;                                   /* de quem e a vez? */
int interested (N);                         /* todos os valores inicialmente 0 (FALSE) */

void enter_region (int process)             /* processo: quem esta entrando (0 ou 1) */
{
   int other;                               /* numero dos outros processos */

   other = 1 - process;                     /* o oposto do processo */
   interested [process] = TRUE;             /* mostra que voce esta interessado */
   turn = process;                          /* seta o flag */
   while (turn == process && interested[other] == TRUE)  /* comando sem efeito */;
}
void leave_region(int process)              /* processo: quem esta saindo (0 ou 1) */
{
   interested[process] = FALSE              /* indica saida da regiao critica */
}
```

Fig. 2.8 Solução de Peterson para obtenção de exclusão mútua de execução.

A Instrução TSL

Agora vamos examinar uma proposta que precisa de uma pequena ajuda do hardware. Muitos computadores, especialmente aqueles projetados com o propósito de suportar múltiplos processadores, possuem uma instrução básica, denominada TEST AND SET LOCKED (TSL) que funciona como descrito a seguir. Ela transfere o conteúdo de um endereço de memória para um registrador, e depois armazena um valor não nulo em tal endereço. As operações de transferência e de armazenamento são indivisíveis — nenhum outro processador ou processo pode acessar a palavra antes que a execução da instrução tenha chegado ao fim. O processador que estiver executando a instrução TSL bloqueia o barramento de memória, de forma a não permitir que nenhum outro processador tenha acesso à memória até que termine a execução da instrução.

Para executar a instrução TSL, é necessário o uso de uma variável compartilhada, *flag*, para coordenar o acesso à memória compartilhada. Quando *flag* for igual a 0, qualquer processo pode fazê-la igual a 1, através da execução da instrução TSL, e então ler ou escrever na memória compartilhada. Ao terminar o acesso à memória, o próprio processo faz *flag* de volta a 0, usando uma instrução básica de MOVE.

Como pode a instrução TSL evitar que dois processos entrem simultaneamente em suas regiões críticas? A solução é dada na Fig. 2.9, através de uma sub-rotina escrita em uma linguagem de montagem fictícia, composta de quatro instruções básicas. A primeira instrução copia o valor antigo de *flag* para um registrador e faz *flag* igual a 1. Na instrução seguinte o valor armazenado no registrador é comparado com 0. Se tal valor for não nulo, o processo que está executando esta sub-rotina deve ser impedido de entrar em sua região crítica. Há um desvio de volta para o início da rotina que será executada de novo pelo processo que deseja acesso à sua região crítica. Mais cedo ou mais tarde o valor de *flag* vai ser feito igual a 0, quando o processo que estiver executando sua região crítica terminar a execução da mesma, permitindo então que o processo que está executando a sub-rotina entre em sua região crítica. Para marcar o fim da execução de uma região crítica, basta que o processo que terminou a execução dela faça *flag* igual a 0. Não há necessidade de nenhuma instrução especial para se fazer isto.

```
enter_region:
        tsl register, flag      | copia o valor de flag no registrador e seta o flag em 1
        cmp register, #0        | o flag é 0?
        jnz enter_region        | se for não-nulo, lock  e setado, então o processo entra em loop
        ret                     | retorna a quem chamou; efetua entrada na regiãa crítica

leave_region:
        mov flag, #0            | guarda um 0 em flag
        ret                     | retorna a quem chamou
```

Fig. 2.9 Acesso a regiões críticas, usando a instrução básica TSL.

Enfim, encontramos uma solução simples e correta para o problema do acesso a regiões críticas. Antes de entrar em sua região crítica, o processo chama *enter_region*, a qual coloca o processo em espera ocupada até que seu ingresso nela seja permitido. Quando isto ocorre, a sub-rotina termina sua execução, permitindo que o processo que a chamou entre em sua região crítica. Ao terminar a execução desta, o processo chama *leave_region*, que armazena o valor 0 em *flag*. A exemplo de todas as soluções para o problema da região crítica, os processos devem chamar *enter_region* e *leave_region* nos momentos apropriados. Se um dos processos trapacear nas chamadas, deixando, por exemplo, de executar *leave_region*, a exclusão mútua de execução das regiões críticas estará comprometida.

2.2.4. Bloqueio e Desbloqueio de Processos: Primitivas SLEEP/WAKEUP

Tanto a solução de Peterson quanto a baseada na instrução TSL são corretas, mas ambas têm o defeito de usar a espera ocupada em suas implementações. Em essência, o que estas soluções fazem é o seguinte: quando um processo deseja entrar em sua região crítica, ele verifica uma condição para saber se sua entrada é ou não possível. Se não houver possibilidade de executar a região crítica naquele momento para salvaguardar a exclusão mútua de execução, o processo é colocado para executar um loop, até que lhe seja permitido entrar na região crítica.

Tal solução, além de gastar tempo de processamento, pode ter efeitos colaterais indesejáveis. Considere um computador que esteja executando dois processos, um *H* com alta prioridade, e outro *L*, com baixa prioridade. As regras de escalonamento são definidas de maneira que *H* rode sempre que estiver no estado de pronto. Em determinado instante, com *L* executando sua região crítica, vai para o estado de pronto quando, por exemplo, uma operação de entrada/saída solicitada por ele terminar. O processo *H* inicia então a execução de um loop de espera ocupado, mas uma vez que *L* nunca é escolhido para rodar enquanto *H* estiver rodando, ele nunca vai ter chance de sair de sua região crítica, de forma que *H* vai permanecer eternamente no loop de espera ocupada. Esta situação é conhecida na literatura como **problema da prioridade invertida.**

Vamos então procurar uma primitiva que bloqueie a continuação da execução de determinado processo, em vez de gastar tempo de processador quando sua entrada na região crítica não for permitida. Um dos pares de primitivas mais simples para resolver este problema é o denominado SLEEP e WAKEUP. A primitiva SLEEP é uma chamada de sistema que blo-

queia o processo que a chamou, isso é, suspende a execução de tal processo, até que outro processo o acorde. A primitiva WAKEUP tem um parâmetro, o processo a ser acordado. Alternativamente, tanto SLEEP quanto WAKEUP podem ter um outro parâmetro, um endereço de memória usado para casar uma primitiva SLEEP com sua correspondente WAKEUP.

O Problema do Produtor-Consumidor

O chamado problema do **produtor-consumidor**, também conhecido como problema do **buffer limitado (bounded buffer)**, é um bom exemplo do uso das primitivas acima descritas. Neste problema, dois processos compartilham um buffer de tamanho fixo. Um dos processos, o produtor, coloca informação no buffer, e o outro, o consumidor, retira informação do buffer.

Os problemas acontecem quando o produtor deseja colocar informação no buffer, mas este está cheio. A solução para isto é colocar o processo produtor para dormir, acordando-o quando o processo consumidor tiver retirado um ou mais itens do buffer. Da mesma forma, o processo consumidor pode desejar remover um item do buffer, encontrando-o vazio. Neste caso, o consumidor é posto para dormir, sendo acordado quando o produtor colocar algo no buffer.

Apesar de este problema parecer simples demais, ele pode levar a alguns tipos de condições de corrida já vistos na análise da questão do diretório de spool. Para controlar o número de itens no buffer, devemos usar uma variável, denominada *count*. Se o número máximo de itens que o buffer pode armazenar é *N*, o código do processo produtor deve, em primeiro lugar, verificar se *count* é igual a *N*. Se for, o produtor deve ser posto para dormir, se não, ele deve depositar um item no buffer e incrementar *count*.

O código do processo consumidor é semelhante ao do produtor: primeiro ele deve testar *count* para verificar se seu valor é 0. Se for, o processo consumidor deve ser posto para dormir; se não for, ele deve remover um item do buffer e decrementar *count*. Cada processo, ao executar, deve verificar se o outro está dormindo, para acordá-lo se for o caso. O código para os processos produtor e consumidor aparece na Fig. 2.10.

```
# include "prototypes.h"

# define N 100                                        /* numero de posicoes do buffer */
int count = 0;                                        /* numero de itens no buffer */

void producer(void)
{
   int item;

   while (TRUE)  {                                    /* loop */
         produce_item(&item);                         /* geracao do proximo item */
         if (count == N) sleep ( );                   /* se buffer cheio, processo vai dormir */
         enter_item(item);                            /* colocaçao de um item no buffer */
         count = count + 1;                           /* incrementa a contagem de itens no buffer */
         if (count == 1) wakeup (consumer);           /* o buffer esta vazio? */
   }
}

void consumer(void)
{
   int item;

   while (TRUE) {                                     /* loop */
         if (count == 0) sleep( );                    /* se buffer vazio, processo vai dormir */
         remove_item(&item);                          /* retirada de um item do buffer */
         count = count - 1;                           /* decrementa a contagem de itens no buffer */
         if (count == N-1) wakeup(producer);          /* o buffer esta cheio? */
         consume_item(item);                          /* imprime item */
   }
}
```

Fig. 2.10 O código de uma solução do problema do produtor-consumidor, com a ocorrência de condição de corrida que inviabiliza a solução apresentada.

Para expressar em C chamadas de sistema com SLEEP e WAKEUP, vamos considerá-las pertencentes a uma biblioteca de rotinas, apesar de elas não fazerem parte da biblioteca-padrão da linguagem C. Os procedimentos *enter_item* e *remove_item*, cujos códigos não são mostrados, simplesmente representam o ato de colocar e de retirar os itens do buffer.

Agora vamos retornar à questão da condição de corrida. Ela pode ocorrer por causa do acesso a *count* ser irrestrito. Observe que, por isso, a seguinte situação pode vir a ocorrer: O buffer está vazio e o processo consumidor acabou de ler *count* para verificar se seu valor é 0. Neste momento, ocorreu uma interrupção e o processo produtor começa a rodar, colocando um item no buffer e incrementando *count*, cujo valor passa a ser 1. Baseado no fato de *count* estar em 0 antes do

início de sua execução, o processo produtor infere que o processo consumidor deve estar dormindo, e então chama a primitiva WAKEUP para acordá-lo.

Lamentavelmente, o processo consumidor não está dormindo, apenas interrompido. Como tal, o processo consumidor está na fila de prontos e não na dos processos bloqueados aguardando que informações sejam colocados no buffer. Desta forma, o sinal emitido pela primitiva WAKEUP, irá se perder, pois não há nenhum processo dormindo. Quando o processo consumidor voltar a rodar, escolhido pelo escalonador, ele vai testar o valor de *count* lido anteriormente, e ao verificar que tal valor é 0, ele vai dormir. Mais cedo ou mais tarde o processo produtor vai encher o buffer e também vai dormir. Ambos dormirão eternamente.

A essência deste problema é que um sinal enviado para acordar um processo que não está dormindo é perdido. Se este sinal não se perdesse tudo estaria bem. Podemos então fazer uma pequena modificação no esquema de SLEEP/WAKEUP, acrescentando à definição das primitivas um bit que fosse setado sempre que um sinal fosse enviado para acordar um processo que já estivesse acordado. Mais tarde, quando o processo tentar dormir, estando este bit setado, ele não conseguirá fazê-lo. O efeito da primitiva SLEEP sobre tal processo será o de apenas resetar o bit citado anteriormente.

Apesar da introdução deste bit de controle resolver nosso problema, no caso de um único par de processos produtor-consumidor, podemos ter exemplos com três ou mais processos, nos quais um único bit não é suficiente. Devemos então adicionar tantos bits de controle quantos forem necessários, complicando um pouco a definição das primitivas.

2.2.5. Semáforos

Esta era a situação em 1965, quando E. W. Dijkstra sugeriu o uso de uma variável inteira para contar o número de sinais armazenados para uso futuro. De acordo com esta proposta, um novo tipo de variável foi criado, o tipo **semáforo**. Um semáforo pode ter valor 0, indicando que não há nenhum sinal armazenado, ou pode ter um valor positivo, indicando o número de sinais armazenados.

Dijkstra estruturou sua ferramenta com duas operações, DOWN e UP (generalizações de SLEEP e WAKEUP, respectivamente). A operação DOWN executada sobre um semáforo verifica se o valor do semáforo é maior que 0. Se for, seu valor é decrementado (ou seja, um sinal armazenado é gasto) e o processo simplesmente continua sua progressão. Se o valor do semáforo for 0, o processo que executou a operação DOWN é posto para dormir. A verificação do valor do semáforo, a modificação de seu valor, e eventualmente a colocação do processo para dormir, são ações únicas, indivisíveis, denominadas **ações atômicas**. A implementação da ferramenta deve garantir que, uma vez que um processo inicie uma operação sobre um semáforo, nenhum outro processo terá acesso ao semáforo até que a operação se conclua. A atomicidade é absolutamente essencial para resolver os problemas de sincronização entre processos, evitando as condições de corrida.

A operacão UP incrementa o valor do semáforo. Se um ou mais processos estiverem dormindo neste semáforo, impedidos de completar uma operação DOWN, um deles vai ser escolhido pelo sistema (randomicamente, por exemplo), sendo-lhe então permitido completar a operação DOWN. Então, após um UP sobre um semáforo (binário) com processos dormindo, o semáforo permanecerá em 0, porém haverá menos um processo dormindo associado a tal semáforo. A operação de incrementar o semáforo e acordar um processo é também indivisível. Nenhum processo pode ser bloqueado ao executar um UP, assim como nenhum processo era posto para dormir ao executar um WAKEUP definido anteriormente.

Vale observar que Dijkstra usou em seu artigo que introduziu o conceito de semáforo os nomes P e V em vez de UP e DOWN, respectivamente. Porém, como estas letras não têm significado para as pessoas que não falam holandês, continuaremos a usar os termos UP e DOWN, termos estes introduzidos na linguagem Algol 68.

Solução do Problema do Produtor-Consumidor Usando Semáforos

Os semáforos resolvem o problema da perda de sinais, ilustrado na Fig. 2.11. Voltamos a mencionar que é fundamental que as operações sobre um semáforo sejam definidas de maneira indivisível. O caminho mais comum para isto é o de implementar UP e DOWN como chamadas de sistema, com o sistema operacional inibindo as interrupções por um período de tempo muito pequeno, enquanto o semáforo estiver sendo testado, seu valor estiver sendo atualizado, quando o processo que executou a operação estiver sendo acordado ou posto para dormir. Todas estas ações são feitas com a execução de pouquíssimas instruções básicas, não havendo nenhum problema em inibir as interrupções por um intervalo de tempo tão curto. Se o sistema possuir mais de um processador, cada semáforo deverá ser protegido por uma variável de travamento, com a instrução TSL sendo usada para se ter certeza de que somente um dos processadores poderá acessar o semáforo em determinado instante de tempo. O leitor deve certificar-se de ter entendido que o uso da instrução TSL para impedir que mais de um processador acesse o semáforo ao mesmo tempo difere fundamentalmente de se usar a espera ocupada a que deve submeter-se o produtor ou o consumidor, esperando que o outro processo esvazie ou preencha o buffer. A operação sobre o semáforo gasta uns poucos microssegundos, enquanto que a operação do produtor e do consumidor pode ser arbitrariamente longa, fazendo com que a espera ocupada também venha a ser longa.

Nossa solução usa três semáforos: um chamado *full*, para contar o numero de posições do buffer que já foram preenchidas; um chamado *empty*, para contar o número de posições ainda vazias, e um chamado *mutex* para assegurar que mais de um processo acesse o buffer ao mesmo tempo. O valor inicial de *full* é 0, o de *empty* é igual ao número de posições disponíveis no buffer, e *mutex* vale inicialmente 1. Os semáforos inicializados em 1, e que são usados por dois ou mais processos para garantir que somente um deles possa entrar em sua região crítica, são chamados de **semáforos binários.** Se

cada processo executar um DOWN sobre um semáforo binário imediatamente antes de entrar em sua região crítica, e um UP imediatamente antes de deixá-la, a exclusão mútua de execução destas regiões críticas estará garantida.

Agora que já temos uma boa ferramenta para a comunicação entre processos, vamos voltar atrás e examinar novamente a seqüência de processamento da interrupção discutida na Fig. 2.5. Em um sistema que utiliza semáforos, a forma natural de inibir interrupções é associar um semáforo, com valor inicial 0, a cada um dos dispositivos de entrada/saída existentes. Imediatamente antes de autorizar a operação de um dispositivo, o processo gerente executa um DOWN no semáforo associado ao dispositivo. Esta operação tem como conseqüência o imediato bloqueio de tal processo. Quando o sinal de interrupção for emitido, significando que o dispositivo já terminou sua tarefa, a rotina de manipulação da interrupção executa um UP sobre o semáforo, fazendo com que o processo gerente do dispositivo assuma o estado de pronto, tornando-se novamente candidato ao uso do processador. Neste modelo, o paso 5 da Fig. 2.5 consiste na execução de um UP sobre o semáforo associado ao dispositivo, de forma que, no passo 6, o escalonador poderá novamente selecionar para rodar o processo gerente do dispositivo. Naturalmente, se vários processos estão prontos para rodar, o escalonador pode escolher o mais importante deles para entregar o processador. Estudaremos a questão do escalonamento ainda neste capítulo.

No exemplo da Fig. 2.11, os semáforos foram utilizados para resolver dois problemas distintos. A diferença entre eles é suficientemente importante para nos motivar a torná-la explícita. O semáforo *mutex* é utilizado para resolver a questão da exclusão mútua de execução, sendo projetado para garantir que somente um processo por vez possa estar lendo ou escrevendo no buffer, e manipulando as variáveis associadas ao mesmo. Já vimos que a exclusão mútua evita o não-determinismo na execução do programa produtor-consumidor.

```
#include "prototypes.h"

#define N 100                                   /* numero de posições do buffer */

typedef int semaphore;                          /* semaforo e um tipo especial representado por inteiros */

semaphore mutex = 1;                            /* controla o acesso a regiao critica */
semaphore empty = N;                            /* conta as posicoes vazias do buffer */
semaphore full = 0;                             /* conta as posicoes ocupadas do buffer */

void producer (void)
{
   int item;
   while (TRUE) {                               /* TRUE e a constante 1 */
       produce_item(&item);                     /* gera algo para ser colocado no buffer */
       down(&empty);                            /* decrementa o contador de posicoes vazias */
       down(&mutex);                            /* entra na regiao critica */
       enter_item(item);                        /* coloca um novo item no buffer */
       up(&mutex);                              /* deixa a regiao critica */
       up(&full);                               /* incrementa o contador de posições ocupadas */
   }
}

void consumer(void)
{
   int item;

   while (TRUE) {                               /* loop */
       down(&full);                             /* decrementa o contador de posicoes ocupadas */
       down(&mutex);                            /* entra na regiao critica */
       remove_item(&item);                      /* retira um item do buffer */
       up(&mutex);                              /* deixa a regiao critica */
       up(&empty);                              /* incrementa o contador de posicoes vazias */
       consume_item(item);                      /* faz alguma coisa com o item retirado do buffer */
   }
}
```

Fig. 2.11 A solução do problema do produtor-consumidor usando semáforos.

Os semáforos *full* e *empty* são utilizados para resolver o problema da **sincronização** entre os processos. Tais semáforos são necessários para garantir que uma certa seqüência de eventos possa vir ou não a ocorrer. Neste caso, eles asseguram que o produtor pare de rodar se o buffer estiver cheio, ou que o consumidor também pare quando o buffer estiver vazio. Observe que os semáforos são usados na solução de duas questões bem diferentes: exclusão mútua de execução e sincronização entre processos.

2.2.6. Contadores de Evento

A solução do problema do produtor-consumidor usando semáforos baseou-se na exclusão mútua para evitar a ocorrência de condições de corrida. Deve ser observado, porém, que é possível programar uma solução que não necessite da

exclusão mútua. Nesta seção vamos descrever esta solução, que é baseada numa variável especial, denominada **contadora de evento** (Reed e Kanodia, 1979).

São possíveis somente três operações sobre um contador de evento:

1. *Read (Ed):* Retorna o valor corrente de *E*.
2. *Advance (E):* Incrementa *E* de 1 unidade atomicamente.
3. *Await (E,v):* Espera até que *E* tenha um valor maior ou igual a *v*.

A operação de READ não é empregada na solução do problema do produtor-consumidor, mas é usada na solução de outros problemas envolvendo a questão do sincronismo entre processos.

Observe que o valor dos contadores de evento só pode crescer, nunca decrescer. Tal valor geralmente começa em 0. A Fig. 2.12 mostra mais uma vez uma solução do problema do produtor-consumidor, agora usando contadores de evento

```
#include "prototypes.h"

#define N 100                                   /* numero de posicoes no buffer */

typedef int event_counter;                      /* contadores de evento sao um tipo especial de inteiros */

event_counter in = 0;                           /* conta os itens inseridos no buffer */
event_counter out = 0;                          /* conta os itens removidos do buffer */

void producer(void)
{
   int item, sequence = 0;

   while (TRUE)  {                              /* loop */
         produce_item(&item);                   /* gera algo para ser colocado no buffer */
         sequence = sequence + 1;               /* conta os itens produzidos */
         await(out, sequence - N);              /* espera ate que haja espaco no buffer */
         enter_item(item);                      /* coloca item na posicao (sequencia - 1) % N */
         advance(&in);                          /* deixa o consumidor saber que novo item foi colocado */
   }
}

void consume (void)
{
   int item, sequence = 0;

   while (TRUE)  {                              /* loop */
         sequence = sequence + 1;               /* numero de itens a ser removidos do buffer */
         await(in, sequence);                   /* aguarda ate que o item desejado esteja presente */
         remove_item(&item);                    /* retira o item da posicao (sequencia - 1) % N */
         advance(&out);                         /* deixa o produtor saber que um item foi retirado */
         consume_item(item);                    /* faz algo com o item retirado */
   }
}
```

Fig. 2.12 A solução do problema do produtor-consumidor, usando contadores de evento.

Nesta solução, dois contadores de evento são usados. O primeiro deles, *in*, conta cumulativamente o número de itens que o produtor colocou no buffer desde que o programa iniciou sua execução. O outro, *out*, conta cumulativamente o número de itens que o consumidor removeu do buffer desde o início da execução. Fica claro que o valor de *in* deve ser maior ou no máximo igual ao de *out*, e nunca maior que o tamanho do buffer.

Quando o produtor computar um novo item, ele deve verificar para ver se há lugar no buffer, usando a chamada de sistema AWAIT. Inicialmente, o valor de *out* será 0, e o de *sequence-N* será negativo, de forma que o produtor não será bloqueado. Se o produtor gerar $N + 1$ itens antes que o consumidor possa consumir alguma, a execução do AWAIT o fará esperar até que *out* seja igual a 1, condição esta que só se efetivará quando o consumidor remover um item do buffer.

A lógica do consumidor é ainda mais simples. Antes de tentar remover o *k*-ésimo item, ele simplesmente espera até que *in* seja igual a *k*, ou seja, que o produtor tenha posto *k* itens no buffer.

2.2.7. Monitores

Com o uso de semáforos e de contadores de evento, a comunicação entre processos parece extremamente simples. No entanto, vamos olhar mais de perto a ordem de execução das primitivas DOWN antes de se ter acesso ao buffer da Fig. 2.11, e mostrar que a questão não é tão simples quanto parece. Suponha que os dois DOWNs no código do consumidor tenham tido sua ordem trocada, de forma que *mutex* foi decrementado antes de *empty* e não depois. Se o buffer estiver

completo, o produtor será bloqueado com *mutex* igual a 0. Conseqüentemente, a próxima vez que o consumidor tentar acesso ao buffer, ele deverá executar um DOWN sobre *mutex*, cujo valor é 0, ficando, portanto, bloqueado. Então ambos os processos ficarão bloqueados indefinidamente. Tal situação é denominada **deadlock,** e será estudada com detalhes no Cap. 6.

O problema acima foi mostrado para explicitar o cuidado que deve ser tomado quando estivermos tratando com semáforos. Um pequeno erro e tudo pode vir abaixo de repente. É um pouquinho pior do que programar em linguagem de montagem, pois os eros levam a condições de corrida, a deadlocks, e a outras situações imprevisíveis e de reprodução virtualmente impossível.

Para facilitar a escrita de programas paralelos, Hoare (1974) e Brinch Hansen (1975) propuseram uma primitiva de alto nível para sincronização de processos, denominada **monitor.** As propostas de Hoare e de Brinch Hansen são muito semelhantes, diferindo apenas em pequenos detalhes, conforme descrito a seguir. Um monitor é um conjunto de procedimentos, variáveis e estruturas de dados, todas agrupadas juntas em um módulo especial. Os processos podem chamar os procedimentos do monitor sempre que desejarem, mas nunca poderão acessar diretamente as estruturas de dados e as variáveis internas do monitor a partir de procedimentos declarados fora do monitor. A Fig. 2.13 ilustra um monitor escrito em uma linguagem imaginária, parecida com Pascal.

```
monitor exemplo
    integer i;
    condition c;

    procedure produtor (x);
    .
    .
    .
    end;

    procedure consumidor(x);
    .
    .
    .
    end;
end monitor;
```

Fig. 2.13 A estrutura de um monitor.

Os monitores têm uma propriedade muito importante que os torna úteis na implementação da exclusão mútua: somente um processo pode estar ativo dentro do monitor em um dado instante de tempo. Os monitores são uma construção de linguagem de alto nível, portanto o compilador sabe de sua natureza especial, e manipula as chamadas aos procedimentos do monitor de forma diferente que as chamadas aos procedimentos comuns. Tipicamente, quando um processo chama um procedimento de um monitor, as primeiras instruções do procedimento verificam se algum outro processo está ativo dentro dele. Se isto for verdade, o processo que chamou o procedimento do monitor será suspenso até que o outro processo deixe o monitor. Se não houver nenhum processo ativo dentro do monitor, o processo que chamou o procedimento do monitor poderá entrar e executar o procedimento chamado.

É tarefa do compilador inplementar a exclusão mútua de execução sobre o monitor, sendo o caminho mais usual utilizar os semáforos binários. Pelo fato de o compilador, e não do programador, estar envolvido com a exclusão mútua, é muito pouco provável que algo venha a dar errado. Desta forma, o programador do monitor não precisa preocupar-se com a forma do compilador implementar a exclusão mútua. Basta saber que, colocando todas as regiões críticas dentro do monitor, não haverá a possibilidae de mais de um processo executar sua região crítica ao mesmo tempo.

Apesar de os monitores fornecerem uma forma muito simples de se obter a exclusão mútua, conforme vimos anteriormente, isto não é suficiente. Devemos procurar por formas de bloqueio de processos quando não houver condições lógicas para que eles continuem processando. No problema do produtor-consumidor, é muito fácil colocar todos os testes de buffer cheio e de buffer vazio no código dos procedimentos do monitor, mas como vamos fazer para bloquear o produtor quando o buffer estiver cheio?

A solução encontrada foi a introdução de **variáveis de condição**, com duas operações possíveis sobre elas, WAIT e SIGNAL. Quando um procedimento do monitor descobre que sua execução não pode prosseguir (o produtor encontra o buffer cheio), ele executa um WAIT em alguma variável de condicão, por exemplo a variável *full*. Esta ação faz com que o processo que executou WAIT seja bloqueado na fila associada à variável de condição, permitindo também que um outro processo que tenha sido anteriormente impedido de entrar no monitor o faça agora.

Este outro processo, por exemplo o consumidor, pode acordar um produtor que esteja dormindo, através da execução de um SIGNAL, sobre a variável de condição na qual o produtor está bloqueado. Para evitar que haja dois processos ativos no monitor ao mesmo temo, há necessidade de se estabelecer uma regra que defina o que vai ocorrer após um SIGNAL.

Hoare propôs que se deixasse que o processo acordado ganhasse o direito de executar, suspendendo o processo que o acordou. Brinch Hansen propôs que o processo que executasse um SIGNAL *fosse obrigado* a deixar o monitor em seguida à execução da primitiva. Em outras palavras, a execução do SIGNAL só poderia aparecer como a última declaração do procedimento do monitor. Utilizaremos a proposta de Brinch Hansen, por ser conceitualmente mais simples e mais fácil de implementar. Se um SIGNAL for executado sobre uma variável de condição na qual vários processos estiverem esperando, somente um deles, determinado pelo escalonador do sistema, é reativado.

Conceitualmente, as variáveis de condição não se assemelham aos contadores de evento. Elas acumulam sinais para uso posterior, à imagem dos semáforos. Então, se um SIGNAL for executado sobre uma variável de condição sem nenhum processo bloqueado sobre ela, o SIGNAL é perdido. O WAIT precisa vir antes do SIGNAL. Tal regra torna a implementação bastante simples. Na prática isto não chega a constituir um problema, uma vez que é fácil controlar o estado de cada processo, se for necessário. Um processo que precise emitir um SIGNAL pode verificar se isto é necessário, observando as variáveis de condição do sistema.

Um esboço da solução do problema do produtor-consumidor usando monitores é dado na Fig. 2.14, numa linguagem Pascal-like.

Você pode estar pensando que as operações de WAIT e SIGNAL são semelhantes às de SLEEP e WAKEUP, que vimos anteriormente serem sujeitas à ocorrência de condições de corrida. Na verdade, elas são semelhantes, porém com uma diferença fundamental: SLEEP e WAKEUP falham porque, enquanto um processo está tentando dormir, um outro pode estar tentando acordá-lo. Com o monitor, isto não pode ocorrer. A exclusão mútua automática dos procedimentos do monitor garante que se, digamos, o produtor executando um procedimento do monitor descobre que o buffer está cheio, ele poderá

```
monitor ProdutorConsumidor
  condition full, empty;
  integer count;

procedure enter;
begin
    if count = N then wait(full);
    enter_item;
    count := count + 1;
    if count = 1 then signal(empty);
end;

procedure remove;
begin
    if count = 0 then wait(empty);
    remove_item;
    count := count - 1;
    if count = N - 1 then signal(full);
end;

  count := 0;
end monitor;

procedure produtor;
begin
  while true do
  begin
      produce_item;
      ProdutorConsumidor.enter;
  end
end;

procedure consumidor;
begin
  while true do
  begin
      ProdutorConsumidor.remove;
      consume_item;
  end
end;
```

Fig. 2.14 Uma solução do problema do produtor-consumidor, usando monitor.

completar a execução do WAIT, sem correr o risco do escalonador comutar a execução para o consumidor antes que a execução do WAIT venha a completar-se. O consumidor jamais conseguirá entrar no monitor antes do término do WAIT e do bloqueio do produtor.

Fazendo com que a exclusão mútua sobre as suas regiões críticas seja automática, os monitores tornam a programação paralela muito menos suscetível a erros do que com o uso de semáforos. Não é à toa que o código da Fig. 2.14 está escrito numa linguagem estranha, semelhante à Pascal, em vez de C, como os demais exemplos do livro. Como afirmamos anteriormente, o monitor é um conceito diretamente associado à linguagem de programação. O compilador precisa reconhecer sua estrutura e implementar a exclusão mútua de alguma forma. As linguagens C, Pascal e muitas outras não suportam o conceito de monitor, de forma que não é razoável esperar que seus compiladores sigam qualquer regra para fornecer exclusão mútua. De fato, como pode o compilador saber quais dos procedimentos pertencem ao monitor e quais os que não pertencem?

Estas mesmas linguagens também não suportam o conceito de semáforo, mas adicionar-lhes semáforos é relativamente fácil: tudo o que precisa ser feito é programar duas rotinas na linguagem de montagem, adicionando-as à biblioteca de chamadas de sistema, com os nomes UP e DOWN. Os compiladores não precisam sequer tomar conhecimento da existência de tais rotinas. Com o uso destas duas rotinas, é possível escrever programas paralelos em C, Pascal, e até em BASIC, se você gostar muito de sofrer. Já para utilizar monitores, há necessidade de uma linguagem específica que suporte este conceito. Existem hoje poucas linguagens com tal característica, como, por exemplo, a Concurrent Euclid (Holt, 1983).

Outro problema com os monitores e com os semáforos é que eles foram projetados para prover exclusão mútua em um ou mais processadores que tenham acesso a uma memória comum. Colocando os semáforos ou os contadores de evento em uma memória comum, e protegendo-os com a instrução TSL, não haverá a ocorrência de condições de corrida. Quando passamos a tratar com sistemas distribuídos, compostos por diversos processadores, cada um acessando sua própria memória, e conectados através de uma rede de comunicação, tais primitivas tornam-se não-adequadas. A conclusão é que os semáforos são ferramentas de muito baixo nível e que os monitores precisam de um linguagem específica para serem utilizados. Além do mais, nenhum deles permite a troca de informações entre máquinas diferentes. Para que isto se torne possível, algo mais é necessário.

2.2.8. Troca de Mensagens

Este algo mais é a **troca de mensagens.** Este método para comunicação entre processos usa duas primitivas: SEND e RECEIVE, as quais, tal como os semáforos e diferente dos monitores, são chamadas de sistema, e não construções de linguagem. Como tal, podem ser facilmente acrescentadas à biblioteca de procedimentos, com a seguinte formulação

```
send (destino, &mensagem);
```

e

```
receive (fonte, &mensagem).
```

A primeira primitiva envia uma mensagem para um dado destino e a outra recebe uma mensagem de determinada fonte (ou de qualquer fonte). Se não houver nenhuma mensagem a ser recebida, o receptor pode ser bloqueado até que uma mensagem chegue.

Aspectos do Projeto de Sistema Baseados em Troca de Mensagens

Os sistemas baseados em troca de mensagens têm muitos problemas que não ocorrem nos sistemas com semáforos ou com monitores, especialmente se os processos que se comunicam estiverem em máquinas diferentes, conectadas por uma rede. Por exemplo, as mensagens podem perder-se na rede. Para prevenir a perda de mensagens, o transmissor e o receptor devem ser programados de forma que, tão logo tenha recebido uma mensagem, o receptor envie de volta ao transmissor uma mensagem especial de **reconhecimento (acknowledgement)**. Se o transmissor não receber esta mensagem num certo intervalo de tempo, ele deve retransmitir a mensagem.

Consideremos agora o que ocorre se a mensagem for recebida corretamente, mas o reconhecimento for perdido. Neste caso, o transmissor transmite de novo a mensagem, e o receptor a recebe duas vezes. É fundamental que o receptor possa distinguir uma nova mensagem de uma velha que tenha sido transmitida novamente. Normalmente este problema é resolvido pela colocação de números seqüenciais em cada mensagem original. Se o receptor receber uma mensagem com o mesmo número da anterior, ele sabe que tal mensagem é duplicada e que pode ser ignorada.

Os sistemas baseados em troca de mensagens devem tratar também da questão da identificação dos processos, de forma a que os processos especificados no SEND e no RECEIVE não sejam confundidos. Freqüentemente o esquema de identificação usado nomeia os processos como *processo@máquina* ou *máquina@processo*.

Se o número de máquinas na rede for muito grande, e não houver um esquema central de alocação de nomes de máquinas, pode ocorrer que duas organizações venham a dar o mesmo nome às suas máquinas. O problema dos conflitos de nomes pode ser consideravelmente reduzido agrupando as máquinas em **domínios,** e então endereçando os processos

com processo@máquina.domínio. Neste esquema não haverá problema se duas máquinas tiverem o mesmo nome, contanto que estejam em domínios diferentes. O nome do domínio deve evidentemente ser único.

Outro problema dos sistemas com troca de mensagens é a questão envolvendo a **autenticação** das mensagens: como um processo sabe que está se comunicando com um servidor de arquivos real e não com um impostor? Como um servidor de arquivos sabe qual o processo que requisitou acesso a um arquivo? A criptografia das mensagens, com uma chave conhecida somente por usuário autorizado, pode ser muito útil neste caso.

No outro extremo do espectro deve haver questões que são importantes quando o transmissor e o receptor estão rodando na mesma máquina. Uma de tais questões diz respeito à performance. Copiar mensagens de um processo para outro demora muito mais do que fazer uma operação sobre um semáforo, ou uma chamada a um monitor. Muito trabalho de pesquisa tem sido despendido para tornar eficiente o esquema de troca de mensagens. Cheriton (1984) sugeriu, por exemplo, limitar o tamanho das mensagens de forma que elas possam ser armazenadas nos registradores da máquina, realizando a troca de mensagens usando os registradores.

A Solução do Problema do Produtor-Consumidor por Troca de Mensagens

Vamos examinar agora como o problema do produtor-consumidor pode ser resolvido por troca de mensagens, e não por compartilhamento de informações na memória. Uma solução é dada na Fig. 2.15. Vamos assumir que todas as mensagens têm o mesmo tamanho, e que mensagens enviadas e não recebidas ainda são guardadas automaticamente pelo sistema operacional. Nesta solução, consisderaremos um número máximo N de mensagens, tal como no caso das N posições do buffer do problema do produtor-consumidor, resolvido com compartilhamento de memória. O consumidor inicia o processamento enviando N mensagens vazias para o produtor. Sempre que o produtor tiver uma mensagem para enviar ao consumidor, ele usa uma mensagem vazia e envia uma cheia. Neste esquema, o número total de mensagens no sistema permanece constante no tempo, de maneira que elas podem ser armazenadas num dado espaço de memória.

Se o produtor trabalhar mais rápido que o consumidor, todas as mensagens vão acabar ficando cheias, aguardando a ação do consumidor, fazendo com que o produtor fique bloqueado, esperando que retorne uma mensagem vazia. Se o consumidor trabalhar mais depressa que o produtor, ocorrerá o inverso: todas as mensagens tornar-se-ão vazias, esperando que o produtor as preencha, provocando o bloqueio do consumidor, aguardando pela ação do produtor, ou seja, por uma mensagem cheia.

Muitas variantes são possíveis para um sistema de troca de mensagens. Uma alternativa ao endereçamento de processos acima discutido é o emprego de uma nova estrutura, denominada **caixa postal.** Uma caixa postal é um lugar para

```
#include "prototypes.h"

#define N 100                                   /* número de posicoes do buffer */
#define MSIZE 4                                 /* tamanho da mensagem */

typedef int message(MSIZE);

void producer(void)
{
   int item;
   message m:                                   /* buffer de mensagem */

   while (TRUE)  {
         produce_item(&item);                   /* gera algo para ser colocado no buffer */
         receive(consumer, &m);                 /* aguarda a chegada de mensagem vazia */
         build_message (&m, item);              /* constroi uma mensagem para enviar */
         send(consumer, &m);                    /* envia um item ao consumidor */
   }
}

void consumer(void)
{
   int item, i;
   message m;

   for (i = 0; i < N; i++) send(producer, &m);      /* envia N mensagens vazias */
   while (TRUE)  {
         receive(producer, &m);                 /* recebe mensagem contendo um item */
         extract_item(&m, &item);               /* retira item da mensagem */
         send(producer, &m);                    /* envia de volta resposta com mensagem vazia */
         consumer_item(item);                   /* faz algo com o item recebido */
   }
}
```

Fig. 2.15 A solução do problema do produtor-consumidor usando troca de mensagens, com um máximo de N mensagens podendo ser trocadas.

armazenamento de um certo número de mensagens, especificado quando da criação da caixa postal. Quando usamos caixa postal, os parâmetros de endereçamento das chamadas SEND e RECEIVE são caixas postais, não processos. Quando um processo tenta enviar uma mensagem para uma caixa postal que está cheia, ele é suspenso até que pelo menos uma mensagem seja removida da caixa postal.

No caso do problema do produtor-consumidor, tanto o produtor quanto o consumidor devem criar caixas postais grandes o suficiente para armazenar *N* mensagens. O produtor deve enviar mensagens contendo dados para a caixa postal do consumidor, e este deve enviar mensagens vazias para a caixa postal do produtor. Quando usamos caixas postais, o mecanismo de armazenamento é muito claro: a caixa postal destino armazena as mensagens que foram enviadas ao processo destino mas que até o momento não tenham sido aceitas.

No outro extremo do uso das caixas postais está a completa eliminação de qualquer tipo de armazenamento de mensagens. Quando optamos por este caminho, se um SEND é executado antes do RECEIVE, o processo transmissor é bloqueado até que o processo receptor execute um RECEIVE. Neste momento, a mensagem pode ser copiada diretamente do transmissor para o receptor, sem armazenamento intermediário. Da mesma forma, se o RECEIVE for executado primeiro, o receptor deve ficar bloqueado até que o transmissor execute um SEND. Esta estratégia é denominada **rendezvous.** É mais fácil de implementar que o esquema com armzenamento, mas é menos flexível, uma vez que tanto o transmissor quanto o receptor são forçados a rodar em sintonia.

A comunicação entre processos de usuário no UNIX é feita via pipes, que são, em última análise, caixas postais. A única diferenca entre um sistema de mensagens com caixa postal e um com pipes é que este último não preserva fronteiras entre mensagens. Em outras palavras, se um processo escreve 10 mensagens de 100 bytes num pipe, e um outro processo lê 1.000 bytes deste pipe, o leitor vai receber todas as 10 mensagens de uma só vez. Num sistema que implemente realmente a troca de mensagens, cada LEITOR deve receber sempre uma única mensagem de cada vez. Naturalmente, se os processos estiverem programados para sempre ler uma mensagem de tamanho fixo de um pipe, ou para terminar cada mensagem com um caracter especial (por exemplo um line-feed), nenhum problema surgirá para que os processos leiam sempre uma mensagem por vez.

2.2.9. Equivalência entre as Primitivas

Reed e Kanodia (1979) descreveram um método diferente para a comunicação entre processos, denominado **seqüenciadores.** Campbel e Habermann (1974) discutiram um método denominado **expressões de caminho.** Atkinson e Hewitt (1979) introduziram os **serializadores.** Apesar da lista de métodos não ser infinita, ela é muito grande, com novas metodologias sendo idealizadas constantemente. A limitação de tempo e espaço nos impede de falar sobre todos eles. Vale observar também que inúmeros esquemas propostos são muito similares a outros.

Nas seções anteriores, estudamos quatro primitivas diferentes para comunicação entre processos. Ao longo dos anos, cada uma delas vem acumulando seguidores, que afirmam ser esta a melhor solução de todas para o problema. A verdade, porém, é que todos estes métodos são semanticamente equivalentes, pelo menos quando usamos um único processador. Com a utilização de qualquer um de tais métodos você pode construir todos os demais.

Passaremos agora a examinar a equivalência entre semáforos, monitores e troca de mensagens. Esta análise não é apenas interessante por si só, mas também vai permitir um maior entendimento sobre o funcionamento das primitivas e de sua implementação. A falta de tempo e de espaço não nos permite trabalhar com os contadores de evento, mas a idéia geral ficará clara a partir dos demais exemplos.

Implementação de Monitores e de Troca de Mensagens com o Uso do Semáforo

Vamos analisar como podemos construir monitores usando semáforos. Se o sistema operacional fornecer a estrutura do semáforo como ferramenta básica, qualquer projetista de compiladores poderá facilmente implementar o conceito de monitor na linguagem, conforme descrito a seguir. Em primeiro lugar, deve-se programar um pequeno conjunto de procedimentos para gerenciar o monitor e anexá-los à biblioteca. Tais procedimentos têm sua programação mostrada na Fig. 2.16. Então, sempre que o compilador gerar código que envolva monitores, devem ser feitas as chamadas aos procedimentos apropriados que realizarão as funções necessárias à implementação do conceito de monitor.

Associado a cada monitor, existe um semáforo binário, *mutex*, inicializado em 1, para controlar a entrada no monitor, e um semáforo adicional, inicialmente 0, para cada uma das variáveis de condição definidas dentro do monitor. Quando um processo entra no monitor, o compilador gera uma chamada ao procedimento *enter-monitor*, o qual executa um DOWN sobre o semáforo *mutex* associado ao monitor em questão. Se tal monitor estiver em uso, o processo será bloqueado.

Apesar de parecer lógico que o código a ser executado quando um processo deixa o monitor resuma-se simplesmente a um UP no semáforo *mutex*, tal solução nem sempre funciona. Quando um processo não foi sinalizado por nenhum outro processo, ele pode simplesmente executar um UP sobre *mutex* e sair do monitor. Este caso é mostrado na Fig. 2.16, no procedimento *leave-normally*.

A complicação vem das variáveis de condição. A operação de WAIT sobre determinada variável de condição *c* é efetivada como um seqüência de duas operações sobre semáforos. Primeiro deve vir um UP sobre *mutex*, para permitir que outros processos entrem no monitor. A seguir vem um DOWN em *c*, bloqueando o processo corrente na fila da variável de condição.

```
#include "prototypes.h"

typedef int semaphore;

semaphore mutex = 1;                    /* para controlar acesso ao monitor */

void enter-monitor(void)                /* codigo para executar quando desejar entrar no monitor */
{
   down(mutex);                         /* por favor so entrar um por vez */
}

void leave_normally(void)               /* sai do monitor sem sinalizar */
{
   up(mutex);                           /* permite que outros processos entrem */
}

void leave_with_signal(c)               /* sinaliza para c e deixa o monitor */
semaphore c;                            /* variavel de condicao a ser sinalizada */
{
   up(c);                               /* libera um processo na fila de c */
}

void wait(c)                            /* vai dormir na fila da condição c */
semaphore c:                            /* variavel de condicao */
{
   up(mutex);                           /* permite que outro processo entre */
   down(c);                             /* vai dormir na fila da condicao c */
}
```

Fig. 2.16 Biblioteca de procedimentos necessários à implementação de monitores com semáforos.

Lembre-se de que um SIGNAL deve ser sempre a última operação realizada antes de se deixar um monitor. Esta regra é necessária para permitir que tanto a sinalização quanto a saída sejam realizadas por um único procedimento, denominado *leave_with_signal*. Tudo o que este procedimento faz é um UP na variável de condição. A seguir, ele abandona o monitor sem liberar *mutex*. O problema aqui é que, sendo *mutex* igual a 0, nenhum outro processo poderá entrar no monitor. Entretanto, o processo que acabou de ser acordado está ativo dentro do monitor, e irá executar um UP sobre *mutex* quando vier a deixar o monitor. Desta forma, o processo que sinalizou passa efetivamente o controle da exclusão mútua para aquele que foi sinalizado para assegurar que nenhum outro processo venha a competir com este último pelo acesso ao monitor.

Vamos agora analisar como este mecanismo funciona na solução do problema do produtor-consumidor. O semáforo *mutex* garante que cada processo tem acesso exclusivo ao monitor através de sua região crítica. Suponha que o consumidor comece primeiro, e verifique que não há nenhum trabalho a ser feito no buffer. Ele então executa um WAIT *empty* e vai dormir, permitindo que o produtor entre no monitor tão logo deseje. Quando o produtor descobre que *count* (veja Fig. 2.14) é 1, ele executa um SIGNAL *empty*, acordando o consumidor. Neste momento, ambos os processos, produtor e consumidor, estão ativos dentro do monitor, mas como nossas regras para programar monitores asseguram que, após a execução de um SIGNAL, o processo deve deixar o monitor imediatamente, não haverá nenhum problema.

Note que se o produtor executar um UP sobre *mutex* e deixar o monitor antes que o consumidor tenha dado um DOWN sobre este mesmo *mutex*, é possível ao produtor tentar mais uma vez entrar no monitor rapidamente. No entanto, esta tentativa vai falhar, uma vez que *mutex* é 0. Só quando o consumidor deixar o monitor é que o produtor poderá entrar de novo.

Agora vamos analisar como se implementa a troca de mensagens com o uso de semáforos. Associado a cada processo, deve haver um semáforo, inicialmente em 0, no qual tal processo poderá ser bloqueado quando um SEND ou um RECEIVE precisar aguardar complementação. Uma área de armazenamento compartilhada deverá ser usada para armazenamento de caixas postais, cada uma contendo uma matriz de posições de mensagens. As posições em cada caixa postal são encadeadas juntas através de uma lista ligada, de forma que as mensagens são despachadas na ordem em que foram recebidas. Cada caixa postal possui variáveis inteiras informando o número de posições cheias e o número de posições vazias. Finalmente, cada caixa postal contém também a indicação do início de duas filas, a primeira contendo os processos que não podem enviar mensagens para a caixa postal, é a última contendo os processos que não podem receber mensagens desta caixa postal. Tais filas só precisam ter o número de cada processo que estiver esperando, de modo que um UP pode ser executado sobre o semáforo associado a tais processos. O buffer é protegido por um semáforo binário, *mutex*, para assegurar que apenas um processo por vez pode inspecionar e atualizar a estrutura de dados compartilhados. Este esquema é mostrado na Fig. 2.17.

Quando um SEND ou um RECEIVE é emitido para uma caixa postal contendo no mínimo uma posição vazia ou uma cheia, respectivamente, a operação insere ou remove uma mensagem, atualiza os contadores e as ligações das filas, e termina normalmente. O uso de *mutex* no início e no final da região crítica assegura que somente um processo por vez pode usar os contadores e ponteiros, evitando a ocorrência de condições de corrida.

Quando um RECEIVE é executado sobre uma caixa postal vazia, o processo que estiver tentando receber uma mensagem entra primeiro na fila de recepção da caixa postal e então dá um UP em *mutex* e um DOWN em seu próprio semáforo, colocando a si próprio para dormir. Mais tarde, quando ele for acordado, dará imediatamente um DOWN em *mutex*, de maneira análoga ao caso do uso de semáforos para construir monitores.

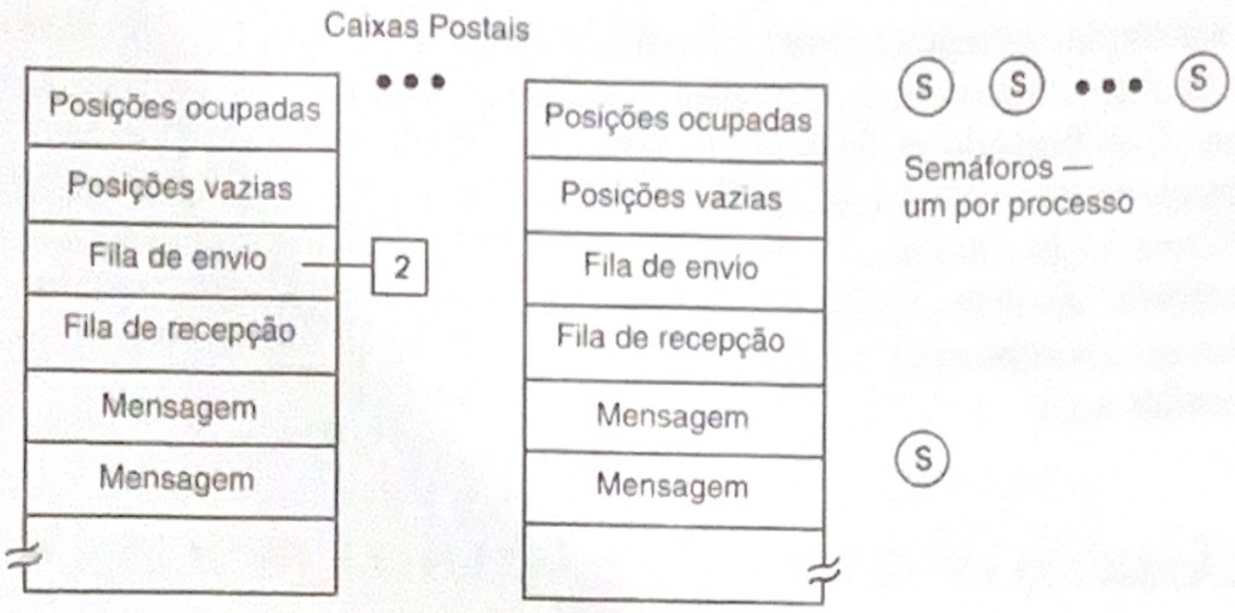

Fig. 2.17 O buffer compartilhado para implementação de troca de mensagens com semáforos.

Quando um SEND é executado, e existe lugar na caixa postal de destino, a mensagem é colocada lá, e o transmissor verifica se a fila desta caixa postal tem algum processo aguardando. Se houver, o primeiro deles é removido da fila, e o transmissor dá um UP em seu semáforo. O transmissor então deixa a região crítica, e o processo que foi acordado pode continuar sua execução. Os respectivos UP e DOWN emitidos sobre *mutex* se cancelam, independentemente da ordem em que venham a ocorrer, e não acontecerá nenhum problema, uma vez que, a exemplo do caso do monitor, um processo que acorda outro processo sempre executa um WAKEUP como última operação antes de deixar a região crítica.

Quando um SEND não puder terminar, devido ao fato de a caixa postal estar cheia, o transmissor primeiro se coloca na fila da caixa postal de destino, depois dá um UP em *mutex*, e um DOWN em seu próprio semáforo. Mais tarde, quando um receptor remover uma mensagem da caixa postal outrora cheia, e verificar que há um processo na fila tentando enviar uma mensagem, o transmissor será acordado.

Uso do Monitor para Implementar Semáforos e Troca de Mensagens

A implementação de semáforos e troca de mensagens usando monitor segue basicamente o mesmo esquema que acabou de ser descrito, sendo, contudo, um pouco mais simples, pois os monitores são construções de nível bem mais alto que os semáforos. Para implementar semáforos, precisamos de um contador e de uma lista ligada para cada semáforo a ser implementado, além de uma variável de condição por processo. Quando um DOWN for executado, o processo que executou verifica dentro do monitor se o contador para este semáforo é maior do que 0. Se for, o contador é incrementado, e o processo simplesmente sai do monitor. Se o contador for zero, o processo adiciona seu próprio número à lista ligada deste semáforo e executa um WAIT sobre sua variável de condição.

Quando um UP for executado sobre um semáforo, o processo executor incrementa o contador (dentro do monitor naturalmente) e então verifica se a lista ligada para este semáforo tem alguma entrada. Se houver, o processo remove um dos processos da lista e executa um SIGNAL sobre a variável de condição deste processo. Observe que o processo que executou o UP não precisa escolher o primeiro processo da fila. Em implementação mais sofisticada, cada processo pode colocar sua própria prioridade na lista junto a seu número, de forma que aquele de maior prioridade seja sempre acordado em primeiro lugar.

A implementação de troca de mensagens usando monitores é essencialmente a mesma que com semáforos, exceto que, em vez de um semáforo por processo, teremos uma variável de condição por processo. A estrutura das caixas postais é a mesma para ambas as implementações.

O Uso da Troca de Mensagens para Implementar Semáforos e Monitores

Se um sistema de troca de mensagens estiver disponível, é possível implementar semáforos e monitores usando um pequeno artifício. Tal artifício consiste na introdução de um novo tipo de processo, o *processo sincronizador*. Vamos primeiro examinar como este processo pode ser usado na implementação de semáforos. O sincronizador deve manter um contador e uma lista ligada de processos para cada semáforo. Para executar um UP ou um DOWN, um processo deve chamar o procedimento correspondente, presente na biblioteca, denominados *up* ou *down*, os quais enviam uma mensagem para o sincronizador, especificando a operação desejada e o semáforo a ser usado. O procedimento da biblioteca executa então um RECEIVE para obter resposta do sincronizador.

Quando a mensagem chega, o sincronizador verifica o contador para ver se a operação requisitada pode ser completada. Os UPs sempre poderão ser completados, mas os DOWNs poderão ser bloqueados se o valor de seu semáforo for 0. Se a operação puder ser realizada, o sincronizador envia de volta uma mensagem vazia, desbloqueando o processo que chamou. Se, no entanto, a operação for um DOWN, e o semáforo estiver em 0, o sincronizador coloca o processo que chamou na fila, e não manda resposta. O resultado é o bloqueio do processo que executou o DOWN. Mais tarde, quando for executado um UP, o sincronizador pega um dos processos bloqueados no semáforo, usando a política FIFO, ou a ordem de prioridades, ou qualquer outra política, e envia uma resposta para ele. As condições de corrida são evitadas porque o processo sincronizador manipula somente uma requisição por vez.

Os monitores podem ser implementados usando troca de mensagens através do mesmo artifício. Já mostramos como os monitores podem ser construídos com o uso dos semáforos, além de como os semáforos podem ser implementados com o uso de troca de mensagens. Combinando as duas implementações, obtemos a forma de construir monitores através de troca de mensagens. Um dos caminhos para atingir tal objetivo é fazer com que o compilador implemente os procedimentos do monitor através de chamadas às rotinas *up* e *down* de sua biblioteca para o semáforo *mutex* e para os semáforos de cada um dos processos, conforme descrito no início desta seção. Tais rotinas devem, portanto, ser implementadas pelo envio de mensagens ao processo sincronizador. Existem outras formas de implementar monitores através da troca de mensagens, que não serão analisadas aqui.

2.3 PROBLEMAS CLÁSSICOS DE COMUNICAÇÃO ENTRE PROCESSOS

A literatura dos sistemas operacionais é pródiga de problemas muito interessantes sobre a questão da comunicação entre processos e que têm sido amplamento discutidos e analisados. Na seção seguinte, iremos examinar três dos mais conhecidos destes problemas.

2.3.1. O Problema dos Filósofos Glutões

Em 1965, Dijkstra propôs e resolveu um problema de sincronização entre processos denominados de **problema dos filósofos glutões**. Desde então, todo mundo que inventa uma nova primitiva de sincronização tenta demonstrar o quão maravilhosa esta nova primitiva é, mostrando a forma elegante com que o problema dos filósofos é resolvido com o uso desta nova primitiva. A formulação deste interessante e simpático problema é dada a seguir. Cinco filósofos estão sentados em torno de uma mesa redonda. Cada filósofo tem à sua frente um prato de macarrão. O macarrão foi preparado com tanta manteiga que ficou bastante escorregadio, de forma que um filósofo que deseje comer precisa contar com o auxílio de dois garfos. Entre cada prato existe um garfo. A mesa dos filósofos é mostrada na Fig. 2.18.

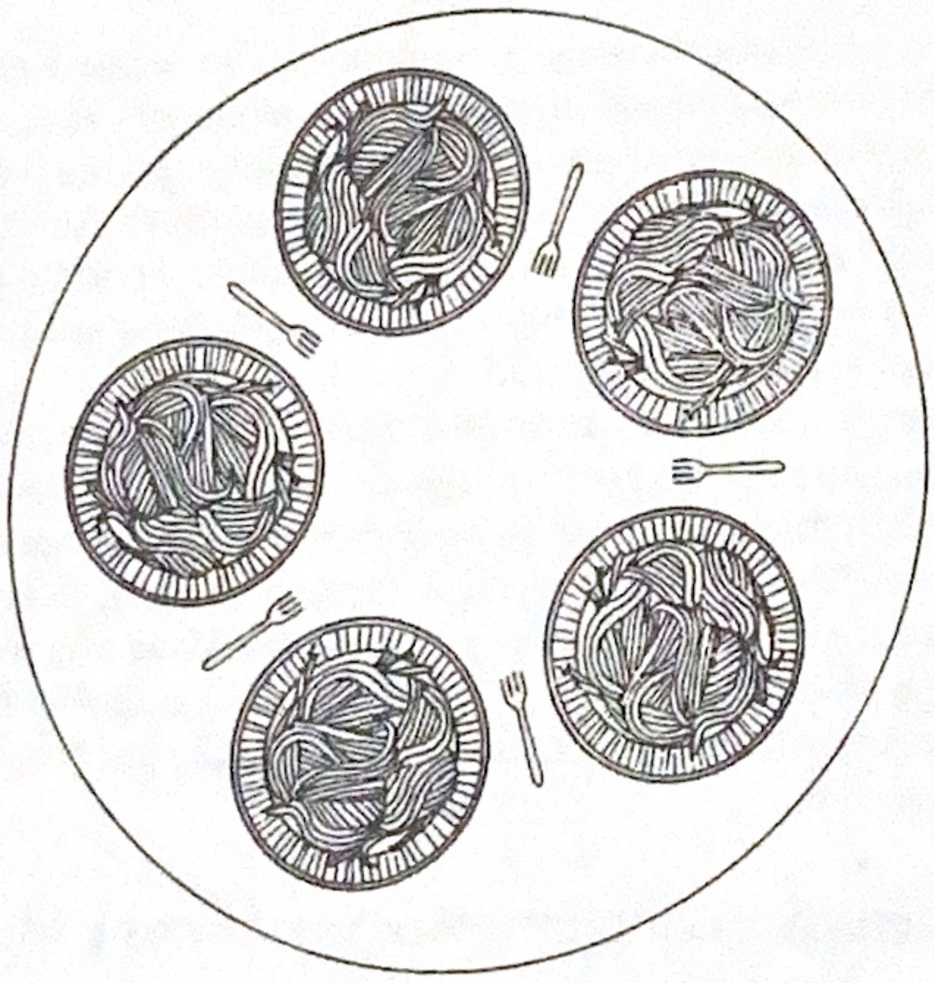

Fig. 2.18 Hora do almoço no Departamento de Filosofia.

A vida de um filósofo consiste em períodos nos quais ele alternadamente come e pensa. (É claro que isto é apenas uma abstração, na medida em que as demais atividadades dos filósofos não são relevantes no contexto do problema.) Quando um filósofo sente fome, ele tenta tomar posse dos garfos à sua direita e à sua esquerda, um por vez, em qualquer ordem. Se ele tiver sucesso nesta empreitada, come um pouco do macarrão, libera os garfos e continua a pensar. A questão a ser respondida é: podemos escrever um programa para simular a ação de cada filósofo que nunca leve a situações de deadlock? (Deve-se observar que a necessidade de dois garfos para comer macarrão não é essencial. Talvez Dijkstra devesse reformular o problema, substituindo o macarrão por comida japonesa, e os dois garfos pelos chopsticks.)

A Fig. 2.19 mostra um solução óbvia. O procedimento *take-fork* espera até que o garfo especificado esteja livre, e então aloca-o ao filósofo que executou o procedimento. Infelizmente, esta solução está errada. Suponha que todos os cinco filósofos resolvam comer ao mesmo tempo e que consigam acesso aos garfos à esquerda de cada um. A partir daí, nenhum deles vai ter acesso ao garfo da direita, configurando-se desta forma um situação de deadlock, na qual todos morrerão de fome.

```
#include "prototypes.h"

#define N 5                                     /* numero de filosofos */

void philosopher(int i)                         /* i: identificacao do filosofo (0 to N-1) */
{
  while (TRUE)  {
        think( );                               /* filosofo pensando */
        take_fork(i);                           /* apanha o garfo a esquerda */
        take_fork( (i+1) % N);                  /* apanha o garfo a direita - % modulo operator */
        eat ( );                                /* come o macarrao
        put_fork(i);                            /* devolve o garfo a esquerda */
        put_fork( (i+1) % N);                   /* devolve o garfo a direita */
  }
}
```

Fig. 2.19 Uma solução errada para o problema dos filósofos.

Podemos modificar o programa de maneira a fazer com que após pegar o garfo da esquerda, o filósofo verifique se o da direita está livre. Se não estiver, ele deve liberar o garfo da esquerda, esperar algum tempo e repetir todo o processo. Esta solução também falha, mas por um motivo diferente daquele que inviabilizou a solução anterior. Com uma pequena dose de azar, é possível que todos os filósofos comecem simultaneamente a executar o algoritmo, pegando os garfos à esquerda. Ao observar que os garfos à direita estão ocupados, todos devolvem ao mesmo tempo o garfo já adquirido, esperam, tornam a pegar os garfos à esquerda, constatam que os da direita não estão disponíveis, devolvem simultaneamente os garfos da esquerda, e assim sucessivamente, para sempre. Uma situação como esta, em que os programas rodam indefinidamente e não fazem nenhum progresso em seu processamento, é denominada **starvation.**

Um observador atento pode constatar que se fizermos com que os filósofos aguardem um tempo randômico em vez de aguardar o mesmo tempo para fazer uma nova tentativa, após constatar que o garfo à sua direita não está disponível, a chance de ocorrer a situação acima é praticamente nula. Tal observação é verdadeira, mas em algumas aplicações devemos procurar por uma solução que funcione sempre, e que não possa vir a falhar por conta de uma série improvável de números randômicos. (Pense no controle da segurança de uma usina nuclear.)

Podemos melhorar a solução da Fig. 2.19 tornando-a livre do deadlock e do starvation, com a utilização de um semáforo binário, *mutex*, para proteger os cinco comandos que se seguem ao comando de *think*. Antes de iniciar a aquisição de garfos, o filósofo precisa executar um DOWN sobre *mutex*. Após a devolução de um garfo, o filósofo deve executar um UP sobre *mutex*. Do ponto de vista teórico, esta solução é adequada. Na prática, porém, ela contém um bug: somente um filósofo pode comer em cada instante. Com quatro garfos disponíveis, devemos permitir que dois filósofos comam ao mesmo tempo.

A solução da Fig. 2.20 é correta, e permite o máximo de paralelismo para um número arbitrário de filósofos. Ela utiliza uma matriz, *state*, para controlar se um filósofo está comendo, pensando ou com fome (tentanto adquirir garfos). Um filósofo só pode passar para o estado *comendo* se nenhum vizinho estiver comendo. Os vizinhos do filósofo *i* são definidos pelas macros *LEFT* e *RIGHT*. Em outras palavras, se *i* vale 2, *LEFT* vale 1 e *RIGHT* vale 3.

O programa usa arrays de semáforos, um para cada filósofo, de forma que os filósofos com fome possam ser bloqueados se os garfos necessários para que ele coma estiverem ocupados. Observe que cada processo roda o procedimento *philosopher* como seu código principal. Os demais procedimentos, *take_forks*, *put_forks*, e *test* são procedimentos comuns, e não processos separados.

2.3.2 O Problema dos Leitores e Escritores

O problema dos filósofos é adequado para modelar a situação em que processos estão competindo por acesso exclusivo a um número limitado de recursos, tais como unidades de fita ou outros dispositivos de entrada/saída. Um outro problema muito famoso é o dos leitores e escritores (Courtouis *et al.*, 1971), que modela o acesso a bases de dados. Imagine uma imensa base de dados, tal como um sistema de passagens de uma companhia de aviação, com muitos processos competindo pelo direito de ler e escrever nela. É aceitável que haja mais de um processo lendo a base de dados ao mesmo tempo, mas se um processo estiver escrevendo na base (ou seja, modificando-a), nenhum outro processo, nem mesmo os leitores, poderão ter acesso a ela enquanto o escritor não terminar. A questão é: como podemos programar o problema dos leitores e dos escritores? Uma solução é mostrada da Fig. 2.21.

Nela, o primeiro leitor a ter acesso à base de dados executa um DOWN no semáforo *db*. Subseqüentemente, os leitores só incrementam um contador, *rc*, antes de acessar a base. Cada leitor que deixar a base de dados decrementa *rc*, e o último a sair executa um UP sobre *db*, permitindo que um escritor bloqueado, se houver algum, tenha acesso à base.

Implicitamente, tal solução prioriza os leitores sobre os escritores. Se um escritor aparecer quando vários leitores estiverem na base, o escritor deve esperar. Se diversos outros leitores forem surgindo, enquanto no mínimo um leitor estiver na base, o escritor vai permanecer esperando até que nenhum leitor esteja interessado em acessá-la. Courtois *et al.* também apresentaram a solução que dá prioridade aos escritores. Para mais informações sobre as soluções para este problema consulte o artigo de Courtois *et al.*, citado na bibliografia.

```
#include "prototypes.h"

#define N                                   5       /* numero de filosofos */
#define LEFT                          (i-1) %N      /* número de vizinhos a esquerda de i */
#define RIGHT                         (i+1) %N      /* numero de vizinhos a direita de i */
#define THINKING                            0       /* filosofo pensando */
#define HUNGRY                              1       /* filosofo tentando pegar garfos */
#define EATING                              2       /* filosofo comendo */

typedef int semaphore;                              /* semaforos sao tipos especiais representados por inteiro */
int state[N];                                       /* vetor para controlar estado dos filosofos */
semaphore mutex = 1;                                /* exclusao mutua para regiao critica */
semaphore s[N];                                     /* um semaforo por filosofo */

void philosopher (int i)                            /* i: numero do filosofo (de 1 ate N) */
{
   while (TRUE) {                                   /* loop */
         think( );                                  /* filosofo pensando */
         take_forks(i);                             /* adquire dois garfos ou fica bloqueado */
         eat( );                                    /* come o macarrao */
         put_forks(i);                              /* coloca ambos os garfos de volta na mesa */
   }
}

void take_forks(int i)                              /* i: numero do filosofo (de 1 até N) */
{
   down(&mutex);                                    /* entra na regiao critica */
   state[i] = HUNGRY;                               /* lembra do fato de que o filosofo i esta com fome */
   test(i);                                         /* tenta adquirir dois garfos */
   up(&mutex);                                      /* sai da regiao critica */
   down(&s[i]);                                     /* fica bloqueado se nao conseguir pegar dois garfos */
}

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

void test(int i)                                    /* i: numero do filósofo (de 1 ate N) */
{
   if (state[i]== HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
         state[i] = EATING;
         up(&s[i]);
   }
}
```

Fig. 2.20 Uma solução para o problema dos filósofos.

2.3.3. O Problema do Barbeiro Dorminhoco

Outro problema clássico de comunicação entre processos é o do barbeiro dorminhoco, cujo cenário é uma barbearia. Nela existem um barbeiro, uma cadeira de barbeiros e diversos lugares onde os clientes que estiverem esperando, se houver algum, poderão sentar. Se não houver nenhum cliente presente, o barbeiro simplesmente senta em sua cadeira e cai no sono, conforme ilustrado na Fig. 2.22. Quando chegarem fregueses, enquanto o barbeiro estiver cortando o cabelo de outro, estes devem ou sentar, se houver cadeira vazia, ou ir embora, se não houver nenhuma cadeira disponível. O problema é programar tanto o barbeiro quanto os clientes, sem que seja permitida a ocorrência de condições de corrida.

Nossa solução usa três semáforos: *customers*, que conta os clientes que estão esperando (excluindo aquele que está na cadeira de barbeiro, que, na verdade, não está esperando), barbers, o número de barbeiros livres esperando por clientes (0 ou 1) e *mutex*, que é usado para garantir a exclusão mútua. Precisamos também de uma variável, *waiting*, que também conta os clientes que estão aguardando o serviço do barbeiro. Tal variável é essencialmente uma cópia de *customers*. A razão para a existência de *waiting* é o fato de não ser permitida a leitura do valor corrente de um semáforo, e, nesta solução, um cliente que entra na barbearia deve contar o número daqueles que estão esperando. Se tal número for menor que o de cadeiras, o novo cliente senta e aguarda, se for maior ou igual, ele vai embora sem cortar o cabelo.

A solução proposta é mostrada na Fig. 2.23. Quando o barbeiro chega de manhã para trabalhar, ele executa o procedimento *Barber*, causando o bloqueio de tal procedimento no semáforo *customers*, significando que o barbeiro vai dormir até que chegue algum cliente, como mostra a Fig. 2.22.

```
#include "prototypes.h"

typedef int semaphore;                  /* use sua imaginacao */

semaphore mutex = 1;                    /* controla o acesso a regiao critica */
semaphore db = 1;                       /* controla o acesso a base de dados */
int rc = 0;                             /* nº de processos lendo ou tentando ler */

void reader(void)
{
  while (TRUE) {                        /* loop */
        down(&mutex);                   /* obtem acesso a regiao critica */
        rc = rc + 1;                    /* mais um leitor */
        if (rc == 1) down(&db);         /* se este for o primeiro leitor ... */
        up(&mutex);                     /* libera acesso a regiao critica */
        read_data_base( );              /* acesso ao dado */
        down(&mutex);                   /* obtem acesso a regiao critica */
        rc = rc - 1;                    /* menos um leitor */
        if (rc == 0) up(&db);           /* se este for o ultimo leitor ... */
        up(&mutex);                     /* libera acesso a regiao critica */
        use_data_read( );               /* seçao nao-critica */
  }
}

void writer(void)
{
  while (TRUE) {                        /* loop */
        think_up_data( );               /* secao nao-critica */
        down(&db);                      /* obtém acesso a regiao critica */
        write_data_base( );             /* atualiza o dado */
        up(&db);                        /* libera acesso a regiao critica */
  }
}
```

Fig. 2.21 Uma solução para o problema dos leitores e escritores.

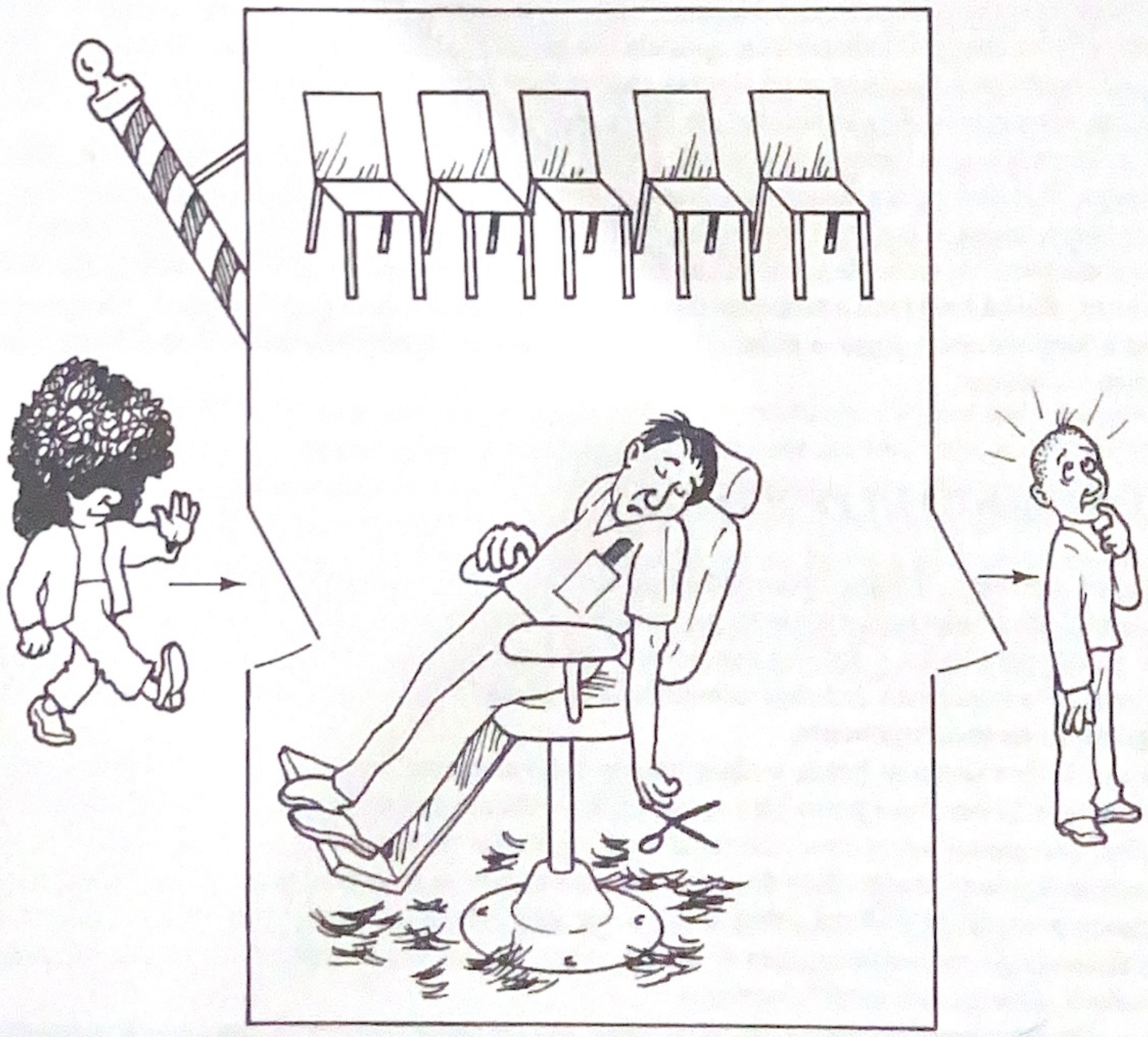

Fig. 2.22 O barbeiro dorminhoco.

```
#include "prototypes.h"

#define CHAIRS 5                        /* # de cadeiras para clientes esperando */

typedef int semanhore;                  /* use sua imaginacao */

semaphore customers = 0;                /* # de clientes esperando por servico */
semaphore barbers = 0;                  /* # de barbeiros esperando por clientes */
semaphore mutex = 1;                    /* para exclusao mutua */
int waiting = 0;                        /* clientes estao esperando */

void Barber(void)
{
   while (TRUE) {
        down(customers);                /* vai dormir se # de clientes é 0 */
        down(mutex);                    /* adquire acesso a 'waiting' */
        waiting = waiting - 1;          /* decrementa contador de clientes esperando */
        up(barbers);                    /* um barbeiro esta pronto para cortar cabelo */
        up(mutex);                      /* libera acesso a 'waiting' */
        cut_hair( );                    /* corta o cabelo (fora da regiao critica) */
   }
}

void Customer(void)
{
   down(mutex);                         /* entra na regiao critica */
   if (waiting < CHAIRS) {              /* vai embora se nao houver nenhuma cadeira livre */
        waiting = waiting + 1;          /* incrementa contador de clientes que estao esperando */
        up(customers);                  /* acorda o barbeiro se necessario */
        up(mutex);                      /* libera acesso a 'waiting' */
        down(barbers);                  /* vai dormir se # de barbeiros livres e 0 */
        get_haircut( );                 /* sente-se na cadeira do barbeiro e corte seu cabelo */
   } else {
        up(mutex);                      /* barbearia muito cheia, nao espere */
   }
}
```

Fig. 2.23 Uma solução para o problema do barbeiro dorminhoco.

Quando chega o primeiro freguês, este executa *Customer,* iniciando tal execução com um DOWN sobre *mutex,* garantindo que se outro cliente chegar imediatamente após ele não estará habilitado a fazer nada, até que o primeiro execute um UP sobre *mutex*. O cliente então verifica se há alguma cadeira livre, comparando o número de clientes que esperam com o número de cadeiras disponíveis. Se não houver, ele libera *mutex* e vai embora.

Havendo pelo menos uma cadeira disponível, o cliente incrementa a variável inteira *waiting*. Então dá um UP no semáforo *customers,* ficando à espera do barbeiro. Neste momento, tanto o cliente quanto o barbeiro estão acordados. Então, quando o cliente libera *mutex,* o barbeiro apodera-se dele, dá uma arrumadinha no salão e inicia o corte.

Quando o corte termina, o cliente completa a execução do procedimento e deixa a barbearia. Em contraposição aos exemplos anteriores, não há loop para o processo cliente, porque cortar o cabelo é uma operação idempotente. Já o barbeiro deve estar num loop tentando pegar o próximo cliente. Se houver algum presente, um novo corte é iniciado. Se não houver, o barbeiro vai dormir.

2.4. ESCALONAMENTO DE PROCESSOS

Nos exemplos anteriores, tivemos oportunidade de observar diversas situações nas quais dois ou mais processos (por exemplo, produtor e consumidor) estavam logicamente em situação de rodar. Quando mais de um processo está nesta situação, ou seja, pronto para rodar, o sistema operacional deve decidir qual deles vai rodar primeiro. A parte do sistema operacional a quem cabe tomar esta decisão é denominada de escalonador, e o algoritmo utilizado em sua programação é chamado de **algoritmo de escalonamento.**

Na época dos velhos sistemas batch, o algoritmo de escalonamento era extremamente simples: resumia-se a fazer com que o próximo job da fita fosse posto para rodar. Com o advento dos sistemas de compartilhamento de tempo, com múltiplos usuários, que muitas vezes deve combinar a execução das tarefas destes usuários com a execução de jobs em batch rodando em background, o algoritmo de escalonamento tornou-se mais complexo. Invariavelmente, existem vários usuários aguardando por serviço, podendo, além disso, haver jobs em batch e em background aguardando vez para executar. Mesmo nos sistemas de compartilhamento de tempo puros, muitas vezes existem tarefas que são executadas em background, tal como o software do correio eletrônico.

Antes de analisar os algoritmos específicos, devemos pensar naquilo que o escalonador deve buscar. Em primeiro lugar, cabe ao escalonador decidir baseado numa política, e não fornecer o mecanismo. Vários critérios nos vêm à mente