



Sincronização e Comunicação entre Processos

► 7.1 Introdução

Na década de 1960, com o surgimento dos sistemas multiprogramáveis, passou a ser possível estruturar aplicações de maneira que partes diferentes do código do programa pudessem executar concurrentemente. Este tipo de aplicação, denominado *aplicação concorrente*, tem como base a execução cooperativa de múltiplos processos ou threads, que trabalham em uma mesma tarefa na busca de um resultado comum.

Em um sistema multiprogramável com único processador, os processos alternam sua execução segundo critérios de escalonamento estabelecidos pelo sistema operacional. Mesmo não havendo neste tipo de sistema um paralelismo na execução de instruções, uma aplicação concorrente pode obter melhorias no seu desempenho. Em sistemas com múltiplos processadores, a possibilidade do paralelismo na execução de instruções somente estende as vantagens que a programação concorrente proporciona.

É natural que processos de uma aplicação concorrente compartilhem recursos do sistema, como arquivos, registros, dispositivos de E/S e áreas de memória. O compartilhamento de recursos entre processos pode ocasionar situações indesejáveis, capazes até de comprometer a execução das aplicações. Para evitar esse tipo de problema, os processos concorrentes devem ter suas execuções sincronizadas, a partir de mecanismos oferecidos pelo sistema operacional, com o objetivo de garantir o processamento correto dos programas.

Este capítulo apresenta como a concorrência de processos pode ser implementada, os problemas do compartilhamento de recursos, soluções e mecanismos do sistema operacional para sincronizar processos como semáforos e monitores. No final do capítulo é também apresentado o problema do deadlock, suas consequências e análise de soluções.

► 7.2 Aplicações Concorrentes

Muitas vezes, em uma aplicação concorrente, é necessário que processos se comuniquem entre si. Esta comunicação pode ser implementada através de diversos mecanismos, como variáveis compartilhadas na memória principal ou trocas de mensagens. Nesta situação, é necessário que os processos concorrentes tenham sua execução sincronizada através de mecanismos do sistema operacional.

A Fig. 7.1 apresenta um exemplo onde dois processos concorrentes compartilham um buffer para trocar informações através de operações de gravação e leitura. Neste exemplo, um processo só poderá gravar dados no buffer caso este não esteja cheio. Da mesma forma, um processo só poderá ler dados armazenados do buffer caso exista algum dado para ser lido. Em ambas as situações, os processos deverão aguardar até que o buffer esteja pronto para as operações, seja de gravação, seja de leitura.

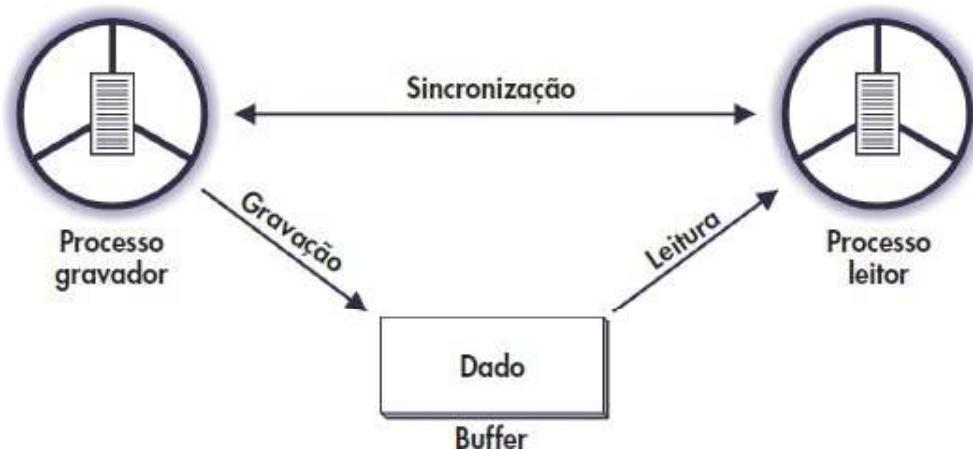


Fig. 7.1 Sincronização e comunicação entre processos.

Os mecanismos que garantem a comunicação entre processos concorrentes e o acesso a recursos compartilhados são chamados *mecanismos de sincronização*. No projeto de sistemas operacionais multiprogramáveis, é fundamental a implementação destes mecanismos para garantir a integridade e a confiabilidade na execução de aplicações concorrentes.

Apesar de o termo processo ser utilizado na exemplificação de aplicações concorrentes, os termos subprocesso e thread têm o mesmo significado nesta abordagem.

► 7.3 Especificação de Concorrência em Programas

Existem várias notações utilizadas para especificar a concorrência em programas, ou seja, as partes de um programa que devem ser executadas concorrentemente. As técnicas mais recentes tentam expressar a concorrência no código dos programas de uma forma mais clara e estruturada.

A primeira notação para a especificação da concorrência em um programa foram os comandos FORK e JOIN, introduzidos por Conway (1963) e Dennis e Van Horn (1966). O exemplo a seguir apresenta a implementação da concorrência em um programa com uma sintaxe simplificada:

```
PROGRAM A; PROGRAM B;  
..  
..
```

```

FORK B; .
...
JOIN B; END.
.
.
END.

```

O programa A começa a ser executado e, ao encontrar o comando FORK, faz com que seja criado um outro processo para a execução do programa B, concorrentemente ao programa A. O comando JOIN permite que o programa A sincronize-se com B, ou seja, quando o programa A encontrar o comando JOIN só continuará a ser processado após o término da execução do programa B. Os comandos FORK e JOIN são bastante poderosos e práticos, sendo utilizados de forma semelhante no sistema operacional Unix.

Uma das implementações mais claras e simples de expressar concorrência em um programa é a utilização dos comandos PARBEGIN e PAREND (Dijkstra, 1965a), que, posteriormente, foram chamados de COBEGIN e COEND. No decorrer deste capítulo, os comandos PARBEGIN e PAREND serão utilizados nos algoritmos apresentados.

O comando PARBEGIN especifica que a sequência de comandos seja executada concorrentemente em uma ordem imprevisível, através da criação de um processo (Processo_1, Processo_2, Processo_n) para cada comando (Comando_1, Comando_2, Comando_n). O comando PAREND define um ponto de sincronização, onde o processamento só continuará quando todos os processos ou threads criados já tiverem terminado suas execuções. Os comandos delimitados pelos comandos PARBEGIN e PAREND podem ser comandos simples, como atribuições ou chamadas a procedimentos (Fig. 7.2).

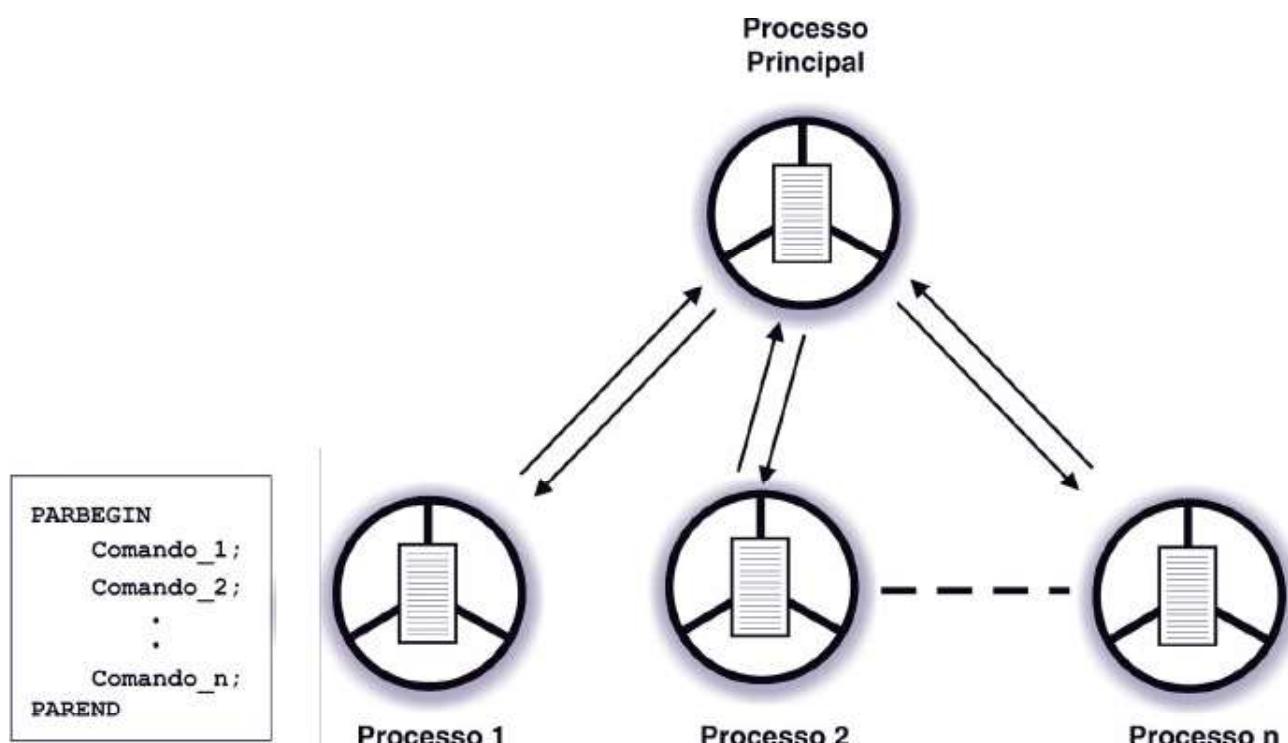


Fig. 7.2 Concorrência em programas.

Para exemplificar o funcionamento dos comandos PARBEGIN e PAREND em uma aplicação

concorrente, o programa Expressao realiza o cálculo do valor da expressão aritmética descrita a seguir:

```
X := SQRT (1024) + (35.4 * 0.23) - (302 / 7)
```

Os comandos de atribuição situados entre PARBEGIN e PARENDA são executados concorrentemente. O cálculo final de X só pode ser realizado quando todas as variáveis dentro da estrutura tiverem sido calculadas.

```
PROGRAM Expressao;
VAR X, Temp1, Temp2, Temp3 : REAL;
BEGIN
PARBEGIN
Temp1 := SQRT (1024);
Temp2 := 35.4 * 0.23;
Temp3 := 302 / 7;
PARENDA;
X := Temp1 + Temp2 - Temp3;
WRITELN ('x = ', X);
END.
```

► 7.4 Problemas de Compartilhamento de Recursos

Para a compreensão de como a sincronização entre processos concorrentes é fundamental para a confiabilidade dos sistemas multiprogramáveis, são apresentados alguns problemas de compartilhamento de recursos. A primeira situação envolve o compartilhamento de um arquivo em disco; a segunda apresenta uma variável na memória principal sendo compartilhada por dois processos.

O primeiro problema é analisado a partir do programa Conta_Corrente, que atualiza o saldo bancário de um cliente após um lançamento de débito ou crédito no arquivo de contas-correntes Arq_Contas. Neste arquivo são armazenados os saldos de todos os correntistas do banco. O programa lê o registro do cliente no arquivo (Reg_Cliente), lê o valor a ser depositado ou retirado (Valor_Dep_Ret) e, em seguida, atualiza o saldo no arquivo de contas.

```
PROGRAM Conta_Corrente;
.

READ (Arq_Contas, Reg_Cliente);
READLN (Valor_Dep_Ret);
Reg_Cliente.Saldo := Reg_Cliente.Saldo + Valor_Dep_Ret;
WRITE (Arq_Contas, Reg_Cliente);
.

.

END.
```

Considerando processos concorrentes pertencentes a dois funcionários do banco que atualizam o saldo de um mesmo cliente simultaneamente, a situação de compartilhamento do recurso pode ser analisada no exemplo da Tabela 7.1. O processo do primeiro funcionário (Caixa 1) lê o registro do cliente e soma ao campo Saldo o valor do lançamento de débito. Antes de gravar o novo saldo no arquivo, o processo do

segundo funcionário (Caixa 2) lê o registro do mesmo cliente, que está sendo atualizado, para realizar outro lançamento, desta vez de crédito. Independentemente de qual dos processos atualize primeiro o saldo no arquivo, o dado gravado estará inconsistente.

Tabela 7.1 Problema de Concorrência I

Caixa	Instrução	Saldo arquivo	Valor dep/ret	Saldo memória
1	READ	1.000	*	1.000
1	READLN	1.000	-200	1.000
1	:=	1.000	-200	800
2	READ	1.000	*	1.000
2	READLN	1.000	+300	1.000
2	:=	1.000	+300	1.300
1	WRITE	800	-200	800
2	WRITE	1.300	+300	1.300

Outro exemplo, ainda mais simples, onde o problema da concorrência pode levar a resultados inesperados é a situação onde dois processos (A e B) executam um comando de atribuição. O Processo A soma 1 à variável X e o Processo B diminui 1 da mesma variável que está sendo compartilhada. Inicialmente, a variável X possui o valor 2.

Processo A	Processo B
X := X + 1;	X := X - 1;

Seria razoável pensar que o resultado final da variável X, após a execução dos Processos A e B, continuasse 2, porém isto nem sempre será verdade. Os comandos de atribuição, em uma linguagem de alto nível, podem ser decompostos em comandos mais elementares, como visto a seguir:

Processo A	Processo B
LOAD x, R _a	LOAD x, R _b
ADD1, R _a	SUB1, R _b
STORE R _a , x	STORE R _b , x

Utilizando o exemplo da Tabela 7.2, considere que o Processo A carregue o valor de X no registrador R_a, some 1 e, no momento em que vai armazenar o valor em X, seja interrompido. Nesse instante, o Processo B inicia sua execução, carrega o valor de X em R_b e subtrai 1. Dessa vez, o Processo B é interrompido e o Processo A volta a ser processado, atribuindo o valor 3 à variável X e concluindo sua execução.

Finalmente, o Processo B retorna a execução, atribui o valor 1 a X, e sobrepõe o valor anteriormente gravado pelo Processo A. O valor final da variável X é inconsistente em função da forma concorrente com que os dois processos executaram.

Tabela 7.2 Problema de Concorrência II

Processo	Instrução	X	R _a	R _b
A	LOAD X, R _a	2	2	*
A	ADD 1, R _a	2	3	*
B	LOAD X, R _b	2	*	2
B	SUB 1, R _b	2	*	1
A	STORE R _a , X	3	3	*
B	STORE R _b , X	1	*	1

Analizando os dois exemplos apresentados, é possível concluir que em qualquer situação, onde dois ou mais processos compartilham um mesmo recurso, devem existir mecanismos de controle para evitar esses tipos de problemas, conhecidos como *condições de corrida* (*race conditions*).

► 7.5 Exclusão Mútua

A solução mais simples para evitar os problemas de compartilhamento apresentados no item anterior é impedir que dois ou mais processos acessem um mesmo recurso simultaneamente. Para isso, enquanto um processo estiver acessando determinado recurso, todos os demais processos que queiram acessá-lo deverão esperar pelo término da utilização do recurso. Essa ideia de exclusividade de acesso é chamada *exclusão mútua* (*mutual exclusion*).

A exclusão mútua deve afetar apenas os processos concorrentes somente quando um deles estiver fazendo acesso ao recurso compartilhado. A parte do código do programa onde é feito o acesso ao recurso compartilhado é denominada *região crítica* (*critical region*). Se for possível evitar que dois processos entrem em suas regiões críticas ao mesmo tempo, ou seja, se for garantida a execução mutuamente exclusiva das regiões críticas, os problemas decorrentes do compartilhamento serão evitados.

Os mecanismos que implementam a exclusão mútua utilizam protocolos de acesso à região crítica. Toda vez que um processo desejar executar instruções de sua região crítica, obrigatoriamente deverá executar antes um protocolo de entrada nessa região. Da mesma forma, ao sair da região crítica um protocolo de saída deverá ser executado. Os protocolos de entrada e saída garantem a exclusão mútua da região crítica de um programa.

BEGIN

.

.

```

Entra_Regiao_Critica; (* Protocolo de Entrada *)
Regiao_Critica;
Sai_Regiao_Critica; (* Protocolo de Saída *)
.
.
END.

```

Utilizando o programa Conta_Corrente apresentado no item anterior, a aplicação dos protocolos para dois processos (A e B) pode ser implementada no compartilhamento do arquivo de contas-correntes. Sempre que o Processo A for atualizar o saldo de um cliente, antes de ler o registro o acesso exclusivo ao arquivo deve ser garantido através do protocolo de entrada da sua região crítica. O protocolo indica se já existe ou não algum processo acessando o recurso. Caso o recurso esteja livre, o Processo A pode entrar em sua região crítica para realizar a atualização. Durante este período, caso o Processo B tente acessar o arquivo, o protocolo de entrada faz com que esse processo permaneça aguardando até que o Processo A termine o acesso ao recurso. Quando o Processo A terminar a execução de sua região crítica, deve sinalizar aos demais processos concorrentes que o acesso ao recurso foi concluído. Isso é realizado através do protocolo de saída, que informa aos outros processos que o recurso já está livre e pode ser utilizado de maneira exclusiva por um outro processo.

Como é possível, então, observar, para garantir a implementação da exclusão mútua os processos envolvidos devem fazer acesso aos recursos de forma sincronizada. Diversas soluções foram desenvolvidas com esse propósito; porém, além da garantia da exclusão mútua, duas situações indesejadas também devem ser evitadas.

A primeira situação indesejada é conhecida como *starvation* ou *espera indefinida*. Starvation é a situação em que um processo nunca consegue executar sua região crítica e, consequentemente, acessar o recurso compartilhado. No momento em que um recurso alocado é liberado, o sistema operacional possui um critério para selecionar, dentre os processos que aguardam pelo uso do recurso, qual será o escolhido. Em função do critério de escolha, o problema do starvation pode ocorrer. Os critérios de escolha aleatória ou com base em prioridades são exemplos de situações que podem gerar o problema. No primeiro caso, não é garantido que um processo em algum momento fará o uso do recurso, pois a seleção é randômica. No segundo caso, a baixa prioridade de um processo em relação aos demais pode impedir o processo de acessar o recurso compartilhado. Uma solução bastante simples para o problema é a criação de filas de pedidos de alocação para cada recurso, utilizando o esquema FIFO. Sempre que um processo solicita um recurso, o pedido é colocado no final da fila associada ao recurso. Quando o recurso é liberado, o sistema seleciona o primeiro processo da fila.

Outra situação indesejada na implementação da exclusão mútua é aquela em que um processo fora da sua região crítica impede que outros processos entrem nas suas próprias regiões críticas. No caso de esta situação ocorrer, um recurso estaria livre, porém alocado a um processo. Com isso, vários processos estariam sendo impedidos de utilizar o recurso, reduzindo o grau de compartilhamento.

Diversas soluções foram propostas para garantir a exclusão mútua de processos concorrentes. A seguir, são apresentadas algumas soluções de hardware e de software com discussões sobre benefícios e problemas de cada proposta.

7.5.1 Soluções de Hardware

A exclusão mútua pode ser implementada através de mecanismos de hardware. Neste item são apresentadas as soluções de desabilitação de interrupções e instruções test-and-set.

■ Desabilitação de interrupções

A solução mais simples para o problema da exclusão mútua é fazer com que o processo desabilite todas as interrupções antes de entrar em sua região crítica, e as reabilite após deixar a região crítica. Como a mudança de contexto de processos só pode ser realizada através de interrupções, o processo que as desabilitou terá acesso exclusivo garantido.

```
BEGIN  
.  
. .  
Desabilita_Interrupcoes;  
Regiao_Critica;  
Habilita_Interrupcoes;  
. .  
END .
```

Esta solução, apesar de simples, apresenta algumas limitações. Primeiro, a multiprogramação pode ficar seriamente comprometida, já que a concorrência entre processos tem como base o uso de interrupções. Um caso mais grave poderia ocorrer caso um processo desabilitasse as interrupções e não tornasse a habilitá-las. Neste caso, é provável que o sistema tivesse seu funcionamento seriamente comprometido.

Em sistemas com múltiplos processadores, essa solução torna-se ineficiente devido ao tempo de propagação quando um processador sinaliza aos demais que as interrupções devem ser habilitadas ou desabilitadas. Outra consideração é que o mecanismo de clock do sistema é implementado através de interrupções, devendo esta solução ser utilizada com bastante critério.

Apesar das limitações apresentadas, essa solução pode ser útil quando se deseja que a execução de parte do núcleo do sistema operacional ocorra sem que haja interrupção. Dessa forma, o sistema pode garantir que não ocorrerão problemas de inconsistência em suas estruturas de dados durante a execução de algumas rotinas.

■ Instrução test-and-set

Muitos processadores possuem uma instrução de máquina especial que permite ler uma variável, armazenar seu conteúdo em uma outra área e atribuir um novo valor à mesma variável. Essa instrução especial é chamada *instrução test-and-set* e tem como característica ser executada sem interrupção, ou seja, trata-se de uma instrução indivisível. Dessa forma, é garantido que dois processos não manipulem uma variável compartilhada ao mesmo tempo, possibilitando a implementação da exclusão mútua.

A instrução test-and-set possui o formato a seguir, e quando executada o valor lógico da variável Y é

copiado para X, sendo atribuído à variável Y o valor lógico verdadeiro.

```
Test-and-Set (X, Y);
```

Para coordenar o acesso concorrente a um recurso, a instrução test-and-set utiliza uma variável lógica global, que no programa Test_and_Set é denominada Bloqueio. Quando a variável Bloqueio for falsa, qualquer processo poderá alterar seu valor para verdadeiro através da instrução test-and-set e, assim, acessar o recurso de forma exclusiva. Ao terminar o acesso, o processo deve simplesmente retornar o valor da variável para falso, liberando o acesso ao recurso.

```
PROGRAM Test_and_Set;
VAR Bloqueio : BOOLEAN;

PROCEDURE Processo_A;
VAR Pode_A : BOOLEAN;
BEGIN
REPEAT
Pode_A := true;
WHILE (Pode_A) DO
Test_and_Set (Pode_A, Bloqueio);
Regiao_Critica_A;
Bloqueio := false;
UNTIL false;
END;

PROCEDURE Processo_B;
VAR Pode_B : BOOLEAN;
BEGIN
REPEAT
Pode_B := true;
WHILE (Pode_B) DO
Test_and_Set (Pode_B, Bloqueio);
Regiao_Critica_B;
Bloqueio := false;
UNTIL false;
END;

BEGIN
Bloqueio := false;
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.
```

No exemplo da Tabela 7.3, o Processo B executa a instrução Test_and_Set antes do processo A, alterando o valor da variável Bloqueio para true e executando a Regiao_Critica_B. Enquanto o Processo B não alterar o valor da variável Bloqueio para false, o Processo A ficará bloqueado. Quando o Processo B desbloquear o acesso ao recurso, o Processo A poderá executar a Regiao_Critica_A.

Tabela 7.3 Execução do Programa Test_and_Set

Processo	Instrução	Pode_A	Pode_B	Bloqueio
A	REPEAT	*	*	false
B	REPEAT	*	*	false
A	Pode_A := true	true	*	false
B	Pode_B := true	*	true	false
B	WHILE	*	true	false
B	Test_and_Set	*	false	true
A	WHILE	true	*	true
A	Test_and_Set	true	*	true
B	Regiao_Critica_B	*	false	true
A	WHILE	true	*	true
A	Test_and_Set	true	*	true
B	Bloqueio := false	*	false	false
B	UNTIL false	*	false	false
B	REPEAT	*	false	false
A	WHILE	true	*	false
A	Test_and_Set	false	*	true
A	Regiao_Critica_A	false	*	true

O uso de uma instrução especial de máquina oferece algumas vantagens, como a simplicidade de implementação da exclusão mútua em múltiplas regiões críticas e o uso da solução em arquiteturas com múltiplos processadores. A principal desvantagem é a possibilidade do starvation, pois a seleção do processo para acesso ao recurso é arbitrária.

7.5.2 Soluções de Software

Diversos algoritmos foram propostos na tentativa de implementar a exclusão mútua através de soluções de software. As primeiras soluções tratavam apenas da exclusão mútua para dois processos e, inicialmente, apresentavam alguns problemas. A seguir apresenta-se de forma evolutiva como foi o desenvolvimento de uma solução definitiva para a exclusão mútua entre N processos.

■ Primeiro algoritmo

Este algoritmo apresenta uma solução para exclusão mútua entre dois processos, onde um mecanismo de controle alterna a execução das regiões críticas. Cada processo (A e B) é representado por um procedimento que possui um loop infinito (REPEAT/UNTIL), onde é feito o acesso a um recurso por diversas vezes. A sequência de comandos, dentro do loop, é formada por um protocolo de entrada, uma região crítica e um protocolo de saída. A região crítica é representada por uma rotina, onde o acesso ao recurso realmente acontece. Após o acesso à região crítica, tanto o Processo A quanto o Processo B realizam processamentos individuais.

O mecanismo utiliza uma variável de bloqueio, indicando qual processo pode ter acesso à região crítica. Inicialmente, a variável global Vez é igual a ‘A’, indicando que o Processo A pode executar sua região crítica. O Processo B, por sua vez, fica esperando enquanto Vez for igual a ‘A’. O Processo B só executará sua região crítica quando o Processo A atribuir o valor ‘B’ à variável de bloqueio Vez. Desta forma, estará garantida a exclusão mútua entre os dois processos.

```
PROGRAM Algoritmo_1;
VAR Vez : CHAR;

PROCEDURE Processo_A;
BEGIN
REPEAT
WHILE (Vez = 'B') DO (* Nao faz nada *);
Regiao_Critica_A;
Vez := 'B';
Processamento_A;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
WHILE (Vez = 'A') DO (* Nao faz nada *);
Regiao_Critica_B;
Vez := 'A';
Processamento_B;
UNTIL false;
END;

BEGIN
Vez := 'A';
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.
```

Este algoritmo, apesar de implementar a exclusão mútua, apresenta duas limitações, das quais a primeira surge do próprio mecanismo de controle utilizado. O acesso ao recurso compartilhado só pode ser realizado por dois processos e sempre de maneira alternada; com isso, um processo que necessite utilizar o recurso mais vezes do que outro permanecerá grande parte do tempo bloqueado.

No exemplo apresentado na Tabela 7.4, caso o Processo A permaneça muito tempo na rotina Processamento_A, é possível que o Processo B queira executar sua região crítica e não consiga, mesmo que o Processo A não esteja utilizando o recurso. Como o Processo B pode executar seu loop mais rapidamente que o Processo A, a possibilidade de executar sua região crítica fica limitada pela velocidade de processamento do Processo A.

Tabela 7.4 Execução do Programa Algoritmo_1

Processo	Instrução	Vez
B	WHILE (Vez = 'A')	A
A	WHILE (Vez = 'B')	A
A	Regiao_Critica_A	A
A	Vez := 'B'	B
A	Processamento_A	B
B	Regiao_Critica_B	B
B	Vez := 'A'	A
B	Processamento_B	A
B	WHILE (Vez = 'A')	A

Outro problema existente nesta solução é que, no caso da ocorrência de algum problema com um dos processos, de forma que a variável de bloqueio não seja alterada, o outro processo permanecerá indefinidamente bloqueado, aguardando o acesso ao recurso.

■ Segundo algoritmo

O problema principal do primeiro algoritmo é que ambos os processos trabalham com uma mesma variável global, cujo conteúdo indica qual processo tem o direito de entrar na região crítica. Para evitar esta situação, o segundo algoritmo introduz uma variável para cada processo (CA e CB) que indica se o processo está ou não em sua região crítica. Neste caso, toda vez que um processo desejar entrar em sua região crítica a variável do outro processo é testada para verificar se o recurso está livre para uso.

```

PROGRAM Algoritmo_2;
VAR CA, CB : BOOLEAN;

PROCEDURE Processo_A;
BEGIN
REPEAT
WHILE (CB) DO (* Nao faz nada *);
CA := true;
Regiao_Critica_A;

```

```

CA := false;
Processamento_A;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
WHILE (CA) DO (* Nao faz nada *);
CB := true;
Regiao_Critica_B;
CB := false;
Processamento_B;
UNTIL false;
END;

BEGIN
CA := false;
CB := false;
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.

```

Neste segundo algoritmo, o uso do recurso não é realizado necessariamente alternado. Caso ocorra algum problema com um dos processos fora da região crítica, o outro processo não ficará bloqueado, o que, porém, não resolve por completo o problema. Caso um processo tenha um problema dentro da sua região crítica ou antes de alterar a variável, o outro processo permanecerá indefinidamente bloqueado. Mais grave que o problema do bloqueio é que esta solução, na prática, é pior do que o primeiro algoritmo apresentado, pois nem sempre a exclusão mútua é garantida. O exemplo apresentado na Tabela 7.5 ilustra o problema, considerando que cada processo executa cada instrução alternadamente.

Tabela 7.5 Execução do Programa Algoritmo_2

Processo	Instrução	CA	CB
A	WHILE (CB)	false	false
B	WHILE (CA)	false	false
A	CA := true	true	false
B	CB := true	true	true
A	Regiao_Critica_A	true	true
B	Regiao_Critica_B	true	true

■ Terceiro algoritmo

O terceiro algoritmo tenta solucionar o problema apresentado no segundo, colocando a instrução de atribuição das variáveis CA e CB antes do loop de teste.

```
PROGRAM Algoritmo_3;
VAR CA, CB : BOOLEAN;

PROCEDURE Processo_A;
BEGIN
REPEAT
CA := true;
WHILE (CB) DO (* Nao faz nada *);
Regiao_Critica_A;
CA := false;
Processamento_A;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
CB := true;
WHILE (CA) DO (* Nao faz nada *);
Regiao_Critica_B;
CB := false;
Processamento_B;
UNTIL false;
END;
```

Esta alteração resulta na garantia da exclusão mútua, porém introduz um novo problema, que é a possibilidade de bloqueio indefinido de ambos os processos. Caso os dois processos alterem as variáveis CA e CB antes da execução da instrução WHILE, nenhum dos dois processos poderá entrar em suas regiões críticas, como se o recurso já estivesse alocado.

■ Quarto algoritmo

No terceiro algoritmo, cada processo altera o estado da sua variável indicando que entrará na região crítica sem conhecer o estado do outro processo, o que acaba resultando no problema apresentado. O quarto algoritmo apresenta uma implementação onde o processo, da mesma forma, altera o estado da variável antes de entrar na sua região crítica, porém existe a possibilidade de esta alteração ser revertida.

```
PROGRAM Algoritmo_4;
VAR CA,CB : BOOLEAN;

PROCEDURE Processo_A;
BEGIN
REPEAT
CA:= true;
```

```

WHILE (CB) DO
BEGIN
CA:= false;
{ pequeno intervalo de tempo aleatório }
CA:= true
END;
Regiao_Critica_A;
CA := false;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
CB := true;
WHILE (CA) DO
BEGIN
CB := false;
{ pequeno intervalo de tempo aleatório }
CB := true;
END;
Regiao_Critica_B;
CB := false;
UNTIL false;
END;

BEGIN
CA := false;
CB := false;
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.

```

Apesar de esta solução garantir a exclusão mútua e não gerar o bloqueio simultâneo dos processos, uma nova situação indesejada pode ocorrer eventualmente. No caso de os tempos aleatórios serem próximos e a concorrência gerar uma situação onde os dois processos alterem as variáveis CA e CB para falso antes do término do loop, nenhum dos dois processos conseguirá executar sua região crítica. Mesmo esta situação não sendo permanente, pode gerar alguns problemas.

■ Algoritmo de Dekker

A primeira solução de software que garantiu a exclusão mútua entre dois processos sem a incorreção de outros problemas foi proposta pelo matemático holandês T. Dekker, com base no primeiro e no quarto algoritmos. O algoritmo de Dekker possui uma lógica bastante complexa e pode ser encontrado em Ben-Ari (1990) e Stallings (1997). Posteriormente, G. L. Peterson propôs outra solução mais simples para o mesmo problema, conforme apresentado a seguir.

■ Algoritmo de Peterson

O algoritmo proposto por G. L. Peterson apresenta uma solução para o problema da exclusão mútua entre dois processos que pode ser facilmente generalizada para o caso de N processos. Similar ao terceiro algoritmo, esta solução, além das variáveis de condição (CA e CB) que indicam o desejo de cada processo entrar em sua região crítica, introduz a variável Vez para resolver os conflitos gerados pela concorrência.

Antes de acessar a região crítica, o processo sinaliza esse desejo através da variável de condição, porém o processo cede o uso do recurso ao outro processo, indicado pela variável Vez. Em seguida, o processo executa o comando WHILE como protocolo de entrada da região crítica. Neste caso, além da garantia da exclusão mútua, o bloqueio indefinido de um dos processos no loop nunca ocorrerá, já que a variável Vez sempre permitirá a continuidade da execução de um dos processos.

```
PROGRAM Algoritmo_Peterson;
VAR CA,CB: BOOLEAN;
Vez : CHAR;

PROCEDURE Processo_A;
BEGIN
REPEAT
CA := true;
Vez := 'B';
WHILE (CB and Vez = 'B') DO (* Nao faz nada *);
Regiao_Critica_A;
CA := false;
Processamento_A;
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
CB := true;
Vez := 'A';
WHILE (CA and Vez = 'A') DO (* Nao faz nada *);
Regiao_Critica_B;
CB := false;
Processamento_B;
UNTIL false;
END;

BEGIN
CA := false;
CB := false;
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.
```

■ Algoritmo para exclusão mútua entre N processos

O algoritmo de Dekker e a versão inicial do algoritmo de Peterson garantem a exclusão mútua de dois processos. Posteriormente, o algoritmo de Peterson foi generalizado para o caso de N processos Hofri (1990). O algoritmo do padeiro (bakery algorithm), proposto por Lamport (1974), é uma solução clássica para o problema da exclusão mútua entre N processos e pode ser encontrado em Ben-Ari (1990) e Silberschatz, Galvin e Gagne (2001). Outros algoritmos que também foram apresentados para exclusão mútua de N processos podem ser consultados em Peterson (1983) e Lamport (1987).

Apesar de todas as soluções até então apresentadas implementarem a exclusão mútua, todas possuíam uma deficiência conhecida como *espera ocupada* (*busy wait*). Na espera ocupada, toda vez que um processo não consegue entrar em sua região crítica, por já existir outro processo acessando o recurso, o processo permanece em looping, testando uma condição, até que lhe seja permitido o acesso. Dessa forma, o processo em looping consome tempo do processador desnecessariamente, podendo ocasionar problemas ao desempenho do sistema.

A solução para o problema da espera ocupada foi a introdução de mecanismos de sincronização que permitiram que um processo, quando não pudesse entrar em sua região crítica, fosse colocado no estado de espera. Esses mecanismos, conhecidos como semáforos e monitores, são apresentados posteriormente.

► 7.6 Sincronização Condicional

Sincronização condicional é uma situação em que o acesso ao recurso compartilhado exige a sincronização de processos vinculada a uma condição de acesso. Um recurso pode não se encontrar pronto para uso devido a uma condição específica. Nesse caso, o processo que deseja acessá-lo deverá permanecer bloqueado até que o recurso fique disponível.

Um exemplo clássico desse tipo de sincronização é a comunicação entre dois processos por meio de operações de gravação e leitura em um buffer, como foi visto na Fig. 7.1, em que processos geram informações (*processos produtores*) utilizadas por outros processos (*processos consumidores*). Nessa comunicação, enquanto um processo grava dados em um buffer, o outro lê os dados, concorrentemente. Os processos envolvidos devem estar sincronizados a uma variável de condição, de forma que um processo não tente gravar dados em um buffer cheio ou realizar uma leitura em um buffer vazio.

O programa a seguir exemplifica o problema da sincronização condicional, também conhecido como *problema do produtor/consumidor* ou do *buffer limitado*. O recurso compartilhado é um buffer, definido no algoritmo com o tamanho TamBuf e sendo controlado pela variável Cont. Sempre que a variável Cont for igual a 0, significa que o buffer está vazio e o processo Consumidor deve permanecer aguardando até que se grave um dado. Da mesma forma, quando a variável Cont for igual a TamBuf, significa que o buffer está cheio e o processo Produtor deve aguardar a leitura de um novo dado. Nessa solução, a tarefa de colocar e retirar os dados do buffer é realizada, respectivamente, pelos procedimentos Grava_Buffer e Le_Buffer, executados de forma mutuamente exclusiva. Não está sendo considerada, neste algoritmo, a implementação da exclusão mútua na variável compartilhada Cont.

```
PROGRAM Produtor_Consumidor_1;
```

```

CONST TamBuf= (* Tamanho qualquer *);
TYPE Tipo_Dado = (* Tipo qualquer *);
VAR Buffer : ARRAY [1..TamBuf] OF Tipo_Dado;
Dado_1 : Tipo_Dado;
Dado_2 : Tipo_Dado;
Cont: 0..TamBuf;

PROCEDURE Produtor;
BEGIN
REPEAT
Produz_Dado (Dado_1);
WHILE (Cont = TamBuf) DO (* Nao faz nada *);
Grava_Buffer (Dado_1, Buffer);
Cont := Cont + 1;
UNTIL false;
END;

PROCEDURE Consumidor;
BEGIN
REPEAT
WHILE (Cont = 0) DO (* Nao faz nada *);
Le_Buffer (Dado_2, Buffer);
Consome_Dado (Dado_2);
Cont := Cont - 1;
UNTIL false;
END;

BEGIN
Cont := 0;
PARBEGIN
Produtor;
Consumidor;
PAREND;
END.

```

Apesar de o algoritmo apresentado resolver a questão da sincronização condicional, o problema da espera ocupada também se faz presente, sendo somente solucionado pelos mecanismos de sincronização semáforos e monitores.

► 7.7 Semáforos

O conceito de *semáforos* foi proposto por E. W. Dijkstra em 1965, sendo apresentado como um mecanismo de sincronização que permitia implementar, de forma simples, a exclusão mútua e a sincronização condicional entre processos. De fato, o uso de semáforos tornou-se um dos principais mecanismos utilizados em projetos de sistemas operacionais e em aplicações concorrentes. Atualmente, a maioria das linguagens de programação disponibiliza rotinas para uso de semáforos.

Um semáforo é uma variável inteira, não negativa, que só pode ser manipulada por duas instruções: DOWN e UP, também chamadas originalmente por Dijkstra de instruções P (*proberen*, teste em holandês)

e V (*verhogen*, incremento em holandês). As instruções DOWN e UP são indivisíveis, ou seja, a execução destas instruções não pode ser interrompida. A instrução UP incrementa uma unidade ao valor do semáforo, enquanto a DOWN decrementa a variável. Como, por definição, valores negativos não podem ser atribuídos a um semáforo, a instrução DOWN executada em um semáforo com valor 0 faz com que o processo entre no estado de espera. Em geral, essas instruções são implementadas no processador, que deve garantir todas essas condições.

Os semáforos podem ser classificados como binários ou contadores. Os *semáforos binários*, também chamados *mutexes (mutual exclusion semaphores)*, só podem assumir os valores 0 e 1, enquanto os *semáforos contadores* podem assumir qualquer valor inteiro positivo, além do 0.

A seguir, será apresentado o uso de semáforos na implementação da exclusão mútua e da sincronização condicional. Ao final deste item, alguns problemas clássicos de sincronização são também apresentados.

7.7.1 Exclusão Mútua Utilizando Semáforos

A exclusão mútua pode ser implementada através de um semáforo binário associado ao recurso compartilhado. A principal vantagem desta solução em relação aos algoritmos anteriormente apresentados é a não ocorrência da espera ocupada.

As instruções DOWN e UP funcionam como protocolos de entrada e saída, respectivamente, para que um processo possa entrar e sair de sua região crítica. O semáforo fica associado a um recurso compartilhado, indicando quando o recurso está sendo acessado por um dos processos concorrentes. O valor do semáforo igual a 1 indica que nenhum processo está utilizando o recurso, enquanto o valor 0 indica que o recurso está em uso.

Sempre que deseja entrar na sua região crítica, um processo executa uma instrução DOWN. Se o semáforo for igual a 1, este valor é decrementado, e o processo que solicitou a operação pode executar as instruções da sua região crítica. De outra forma, se uma instrução DOWN é executada em um semáforo com valor igual a 0, o processo fica impedido do acesso, permanecendo em estado de espera e, consequentemente, não gerando overhead no processador.

O processo que está acessando o recurso, ao sair de sua região crítica, executa uma instrução UP, incrementando o valor do semáforo e liberando o acesso ao recurso. Se um ou mais processos estiverem esperando pelo uso do recurso (operações DOWN pendentes), o sistema selecionará um processo na fila de espera associada ao recurso e alterará o seu estado para pronto (Fig. 7.3).

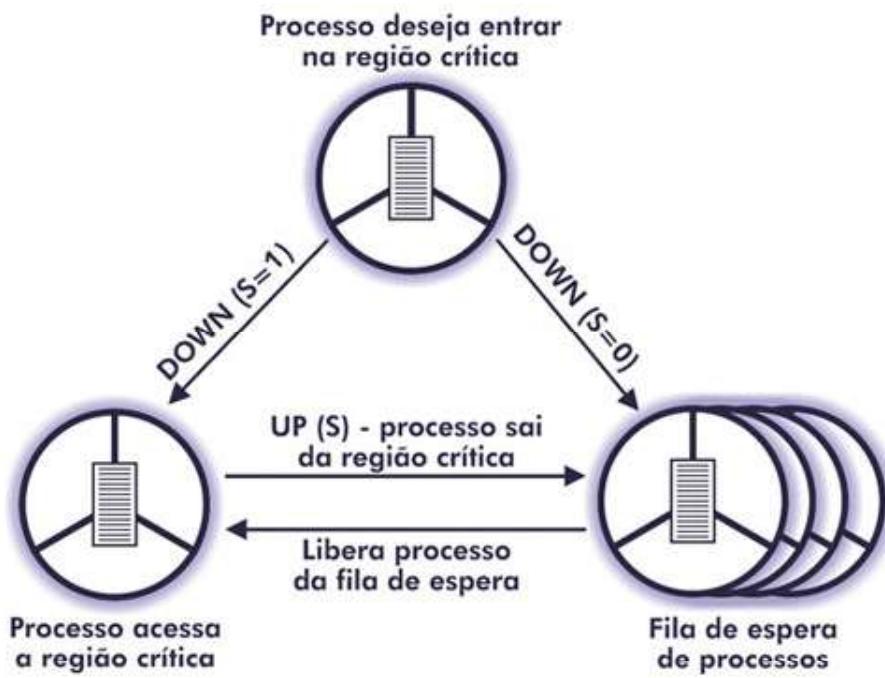


Fig. 7.3 Utilização do semáforo binário na exclusão mútua.

O programa Semaforo_1 apresenta uma solução para o problema da exclusão mútua entre dois processos utilizando semáforos. O semáforo é inicializado com o valor 1, indicando que nenhum processo está executando sua região crítica.

```

PROGRAM Semaforo_1;
VAR s : Semaforo := 1; (* inicializacao do semaforo *)

PROCEDURE Processo_A;
BEGIN
REPEAT
DOWN (s);
Regiao_Critica_A;
UP (s);
UNTIL false;
END;

PROCEDURE Processo_B;
BEGIN
REPEAT
DOWN (s);
Regiao_Critica_B;
UP (s);
UNTIL false;
END;

BEGIN
PARBEGIN
Processo_A;
Processo_B;
PAREND;
END.

```

O exemplo apresentado na Tabela 7.6 ilustra a execução do programa Semaforo_1, onde o Processo A executa a instrução DOWN, fazendo com que o semáforo seja decrementado de 1 e passe a ter o valor 0. Em seguida, o Processo A ganha o acesso à sua região crítica. O Processo B também executa a instrução DOWN no semáforo, mas como seu valor é igual a 0 ficará aguardando até que o Processo A execute a instrução UP, ou seja, volte o valor do semáforo para 1.

Tabela 7.6 Execução do Programa Semaforo_1

Processo	Instrução	S	Espera
A	REPEAT	1	*
B	REPEAT	1	*
A	DOWN (s)	0	*
B	REPEAT	0	*
A	Regiao_Critica_A	0	*
B	DOWN (s)	0	Processo_B
A	UP (s)	1	Processo_B
B	DOWN (s)	1	Processo_B
A	REPEAT	1	Processo_B
B	Regiao_Critica_B	0	*

7.7.2 Sincronização Condicional Utilizando Semáforos

Além de permitirem a implementação da exclusão mútua, os semáforos podem ser utilizados nos casos onde a sincronização condicional é exigida. Um exemplo desse tipo de sincronização ocorre quando um processo solicita uma operação de E/S. O pedido faz com que o processo execute uma instrução DOWN no semáforo associado ao evento e fique no estado de espera, até que a operação seja completada. Quando a operação termina, a rotina de tratamento da interrupção executa um UP no semáforo, liberando o processo do estado de espera.

O problema do produtor/consumidor já apresentado é um exemplo de como a exclusão mútua e a sincronização condicional podem ser implementadas com o uso de semáforos. O programa Produtor_Consumidor_2 apresenta uma solução para esse problema, utilizando um buffer de apenas duas posições. O programa utiliza três semáforos, sendo um do tipo binário, empregado para implementar a exclusão mútua, e dois semáforos contadores para a sincronização condicional. O semáforo binário Mutex permite a execução das regiões críticas Grava_Buffer e Le_Buffer de forma mutuamente exclusiva. Os semáforos contadores Vazio e Cheio representam, respectivamente, se há posições livres no buffer para serem gravadas e posições ocupadas a serem lidas. Quando o semáforo Vazio for igual a 0, significa que o buffer está cheio e o processo produtor deverá aguardar até que o consumidor leia algum dado. Da

mesma forma, quando o semáforo Cheio for igual a 0, significa que o buffer está vazio e o consumidor deverá aguardar até que o produtor grave algum dado. Em ambas as situações, o semáforo sinaliza a impossibilidade de acesso ao recurso.

```

PROGRAM Produtor_Consumidor_2;
CONST TamBuf= 2;
TYPE Tipo_Dado = (* Tipo qualquer *);
VAR Vazio : Semaforo := TamBuf;
Cheio : Semaforo := 0;
Mutex : Semaforo := 1;
Buffer : ARRAY [1..TamBuf] OF Tipo_Dado;
Dado_1 : Tipo_Dado;
Dado_2 : Tipo_Dado;
PROCEDURE Produtor;
BEGIN
REPEAT
Produz_Dado (Dado_1);
DOWN (Vazio);
DOWN (Mutex);
Grava_Buffer (Dado_1, Buffer);
UP (Mutex);
UP (Cheio);
UNTIL false;
END;

PROCEDURE Consumidor;
BEGIN
REPEAT
DOWN (Cheio);
DOWN (Mutex);
Le_Buffer (Dado_2, Buffer);
UP (Mutex);
UP (Vazio);
Consome_Dado (Dado_2);
UNTIL false;
END;

BEGIN
PARBEGIN
Produtor;
Consumidor;
PAREND;
END.

```

O exemplo apresentado na Tabela 7.7 considera que o processo Consumidor é o primeiro a ser executado. Como o buffer está vazio ($Cheio = 0$), a operação DOWN ($Cheio$) faz com que o processo Consumidor permaneça no estado de espera. Em seguida, o processo Produtor grava um dado no buffer e, após executar o UP ($Cheio$), libera o processo Consumidor, que estará apto para ler o dado do buffer.

Tabela 7.7 Execução do Programa Produtor_Consumidor_2

Processo	Instrução	Vazio	Cheio	Mutex	Espera
Produtor	Produz_Dado	2	0	1	*
Consumidor	DOWN (Cheio)	2	0	1	Consumidor
Produtor	DOWN (Vazio)	1	0	1	Consumidor
Produtor	DOWN (Mutex)	1	0	0	Consumidor
Produtor	Grava_Buffer	1	0	0	Consumidor
Produtor	UP (Mutex)	1	0	1	Consumidor
Produtor	UP (Cheio)	1	1	1	Consumidor
Consumidor	DOWN (Cheio)	1	0	1	*
Produtor	Produz_Dado	1	0	1	Produtor
Consumidor	DOWN (Mutex)	1	0	0	Produtor
Consumidor	Le_Buffer	1	0	0	Produtor
Consumidor	UP (Mutex)	1	0	1	Produtor
Consumidor	UP (Vazio)	2	0	1	Produtor

Semáforos contadores são bastante úteis quando aplicados em problemas de sincronização condicional onde existem processos concorrentes alocando recursos do mesmo tipo (pool de recursos). O semáforo é inicializado com o número total de recursos do pool, e, sempre que um processo deseja alocar um recurso, executa um DOWN, subtraindo 1 do número de recursos disponíveis. Da mesma forma, sempre que o processo libera um recurso para o pool, executa um UP. Se o semáforo contador ficar com o valor igual a 0, significa que não existem mais recursos a serem utilizados, e o processo que solicita um recurso permanece em estado de espera até que outro processo libere algum recurso para o pool.

7.7.3 Problema dos Filósofos

O problema dos filósofos é um exemplo clássico de sincronização de processos proposto por Dijkstra. Nesse problema, há uma mesa com cinco pratos e cinco garfos onde os filósofos podem sentar, comer e pensar. Toda vez que um filósofo para de pensar e deseja comer é necessário que ele utilize dois garfos, posicionados à sua direita e à sua esquerda.

O algoritmo Filosofo_1 apresenta uma solução que não resolve o problema totalmente, pois se todos os filósofos estiverem segurando apenas um garfo cada um, nenhum filósofo conseguirá comer. Esta situação é conhecida como deadlock.

```
PROGRAM Filosofo_1;
VAR Garfos:ARRAY [0..4] of Semaforo := 1;
I : INTEGER;
```

```

PROCEDURE Filosofo (I : INTEGER);
BEGIN
REPEAT
Pensando;
DOWN (Garfos[I]);
DOWN (Garfos[(I+1) MOD 5]);
Comendo;
UP (Garfos[I]);
UP (Garfos[(I+1) MOD 5]);
UNTIL false;
END;

BEGIN
PARBEGIN
FOR I := 0 TO 4 DO
Filosofo (I);
PAREND;
END.

```

Existem várias soluções para resolver o problema dos filósofos sem a ocorrência do deadlock, como as apresentadas a seguir. A solução (a) é descrita no algoritmo Filosofo_2.

- (a) Permitir que apenas quatro filósofos sentem à mesa simultaneamente;
- (b) Permitir que um filósofo pegue um garfo apenas se o outro estiver disponível;
- (c) Permitir que um filósofo ímpar pegue primeiro o seu garfo da esquerda e depois o da direita, enquanto um filósofo par pegue o garfo da direita e, em seguida, o da esquerda.

```

PROGRAMFilosofo_2;
VARGarfos :ARRAY [0..4] of Semaforo := 1;
Lugares:Semaforo := 4;
I:INTEGER;

```

```

PROCEDURE Filosofo (I : INTEGER);
BEGIN
REPEAT
Pensando;
DOWN (Lugares);
DOWN (Garfos[I]);
DOWN (Garfos[(I+1) MOD 5]);
Comendo;
UP (Garfos[I]);
UP (Garfos[(I+1) MOD 5]);
UP (Lugares);
UNTIL false;
END;

```

```

BEGIN
PARBEGIN

```

```

FOR I := 0 TO 4 DO
Filosofo (I);
PAREND;
END.

```

7.7.4 Problema do Barbeiro

O problema do barbeiro é outro exemplo clássico de sincronização de processos. Neste problema, um barbeiro recebe clientes para cortar o cabelo. Na barbearia há uma cadeira de barbeiro e apenas cinco cadeiras para clientes esperarem. Quando um cliente chega, caso o barbeiro esteja trabalhando ele se senta, se houver cadeira vazia, ou vai embora, se todas as cadeiras estiverem ocupadas. No caso de o barbeiro não ter nenhum cliente para atender, ele se senta na cadeira e dorme até que um novo cliente apareça.

```

PROGRAM Barbeiro;
CONST Cadeiras = 5;
VAR Clientes: Semaforo:= 0;
Barbeiro: Semaforo:= 0;
Mutex: Semaforo:= 1;
Espera: integer:= 0;

PROCEDURE Barbeiro;
BEGIN
REPEAT
DOWN (Clientes);
DOWN (Mutex);
Espera := Espera - 1;
UP (Barbeiro);
UP (Mutex);
Corta_Cabelo;
UNTIL false;
END;

PROCEDURE Cliente;
BEGIN
DOWN (Mutex);
IF (Espera < Cadeiras)
BEGIN
Espera := Espera + 1;
UP (Clientes);
UP (Mutex);
DOWN (Barbeiro);
Ter_Cabelo_Cortado;
END
ELSE UP (Mutex);
END;

```

A solução utiliza o semáforo contador Clientes e os semáforos binários Barbeiro e Mutex. No processo Barbeiro é executado um DOWN no semáforo Clientes com o objetivo de selecionar um cliente para o

corte. Caso não exista nenhum cliente aguardando, o processo Barbeiro permanece no estado de espera. No caso de existir algum cliente, a variável Espera é decrementada, e para isso é utilizado o semáforo Mutex visando garantir o uso exclusivo da variável. Uma instrução UP é executada no semáforo Barbeiro para indicar que o recurso barbeiro está trabalhando.

No processo Cliente, o semáforo Mutex garante a exclusão mútua da variável Espera, que permite verificar se todas as cadeiras já estão ocupadas. Nesse caso, o cliente vai embora, realizando um UP no Mutex. Caso contrário, a variável Espera é incrementada e o processo solicita o uso do recurso barbeiro através da instrução DOWN no semáforo Barbeiro.

► 7.8 Monitores

Monitores são mecanismos de sincronização de alto nível que tornam mais simples o desenvolvimento de aplicações concorrentes. O conceito de monitores foi proposto por Brinch Hansen, em 1972, e desenvolvido por C.A.R. Hoare em 1974, como um mecanismo de sincronização estruturado, ao contrário dos semáforos, que são considerados não estruturados.

O uso de semáforos exige do desenvolvedor bastante cuidado, pois qualquer engano pode levar a problemas de sincronização imprevisíveis e difíceis de serem reproduzidos devido à execução concorrente dos processos. Monitores são considerados mecanismos de alto nível e estruturados em função de serem implementados pelo compilador. Assim, o desenvolvimento de programas concorrentes fica mais fácil e as chances de erro são menores. Inicialmente, algumas poucas linguagens de programação, como Concurrent Pascal, Modula-2 e Modula-3, ofereciam suporte ao uso de monitores. Atualmente, a maioria das linguagens de programação disponibiliza rotinas para uso de monitores.

O monitor é formado por procedimentos e variáveis encapsulados dentro de um módulo. Sua característica mais importante é a implementação automática da exclusão mútua entre os procedimentos declarados, ou seja, somente um processo pode estar executando um dos procedimentos do monitor em um determinado instante. Toda vez que algum processo faz uma chamada a um desses procedimentos, o monitor verifica se já existe outro processo executando algum procedimento do monitor. Caso exista, o processo ficará aguardando a sua vez em uma fila de entrada. As variáveis globais de um monitor são visíveis apenas aos procedimentos da sua estrutura, sendo inacessíveis fora do contexto do monitor. Toda a inicialização das variáveis é realizada por um bloco de comandos do monitor, sendo executado apenas uma vez, na ativação do programa onde está declarado o monitor (Fig. 7.4).

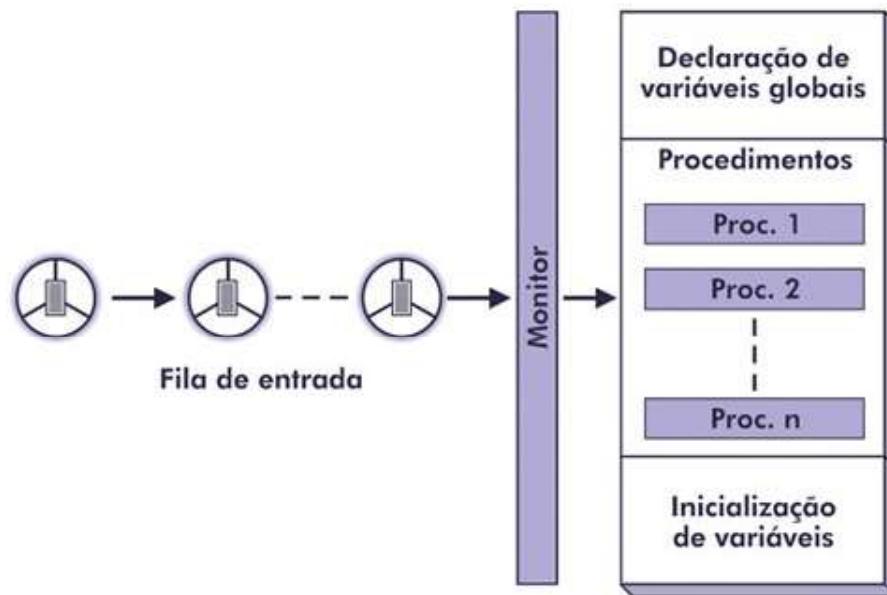


Fig. 7.4 Estrutura do monitor.

Um monitor é definido especificando-se um nome, declarando-se variáveis locais, procedimentos e um código de inicialização. A estrutura de um monitor é mostrada a seguir, utilizando uma sintaxe Pascal não convencional:

```

MONITOR Exclusao_Mutua;
(* Declaracao das variaveis do monitor *)
PROCEDURE Regiao_Critica_1;

BEGIN
.
.
.
END;

PROCEDURE Regiao_Critica_2;
BEGIN
.
.
.
END;

PROCEDURE Regiao_Critica_3;
BEGIN
.
.
.
END;

BEGIN
(* Codigo de inicializacao *)
END;

```

7.8.1 Exclusão Mútua Utilizando Monitores

A implementação da exclusão mútua utilizando monitores não é realizada diretamente pelo programador, como no caso do uso de semáforos. As regiões críticas devem ser definidas como procedimentos no monitor, e o compilador se encarregará de garantir a exclusão mútua entre esses procedimentos.

A comunicação do processo com o monitor é feita unicamente através de chamadas a seus procedimentos e dos parâmetros passados. O programa Monitor_1 apresenta a solução para o problema anteriormente apresentado, onde dois processos somam e diminuem, concorrentemente, o valor 1 da variável compartilhada X.

```
PROGRAM Monitor_1;
MONITOR Regiao_Critica;
VAR X : INTEGER;

PROCEDURE Soma;
BEGIN
  X := X + 1;
END;

PROCEDURE Diminui;
BEGIN
  X := X - 1;
END;

BEGIN
  X := 0;
END;

BEGIN
  PARBEGIN
    Regiao_Critica.Soma;
    Regiao_Critica.Diminui;
  PAREND;
END.
```

A inicialização da variável X com o valor zero só acontecerá uma vez, no momento da inicialização do monitor Regiao_Critica. Neste exemplo, podemos garantir que o valor de X ao final da execução concorrente de Soma e Diminui será igual a zero, porque, como os procedimentos estão definidos dentro do monitor, estará garantida a execução mutuamente exclusiva.

7.8.2 Sincronização Condicional Utilizando Monitores

Monitores também podem ser utilizados na implementação da sincronização condicional. Por meio de *variáveis especiais de condição*, é possível associar a execução de um procedimento que faz parte do monitor a uma determinada condição, garantindo a sincronização condicional.

As variáveis especiais de condição são manipuladas por intermédio de duas instruções, conhecidas como WAIT e SIGNAL. A instrução WAIT faz com que o processo seja colocado no estado de espera, até que algum outro processo sinalize com a instrução SIGNAL que a condição de espera foi satisfeita. Caso a

instrução SIGNAL seja executada e não haja processo aguardando a condição, nenhum efeito surtirá.

É possível que vários processos estejam com suas execuções suspensas, aguardando a sinalização de diversas condições. Para isso, o monitor organiza os processos em espera utilizando filas associadas às condições de sincronização. A execução da instrução SIGNAL libera apenas um único processo da fila de espera da condição associada. Um processo pode executar um procedimento de um monitor, mesmo quando um ou mais processos estão na fila de espera de condições (Fig. 7.5).

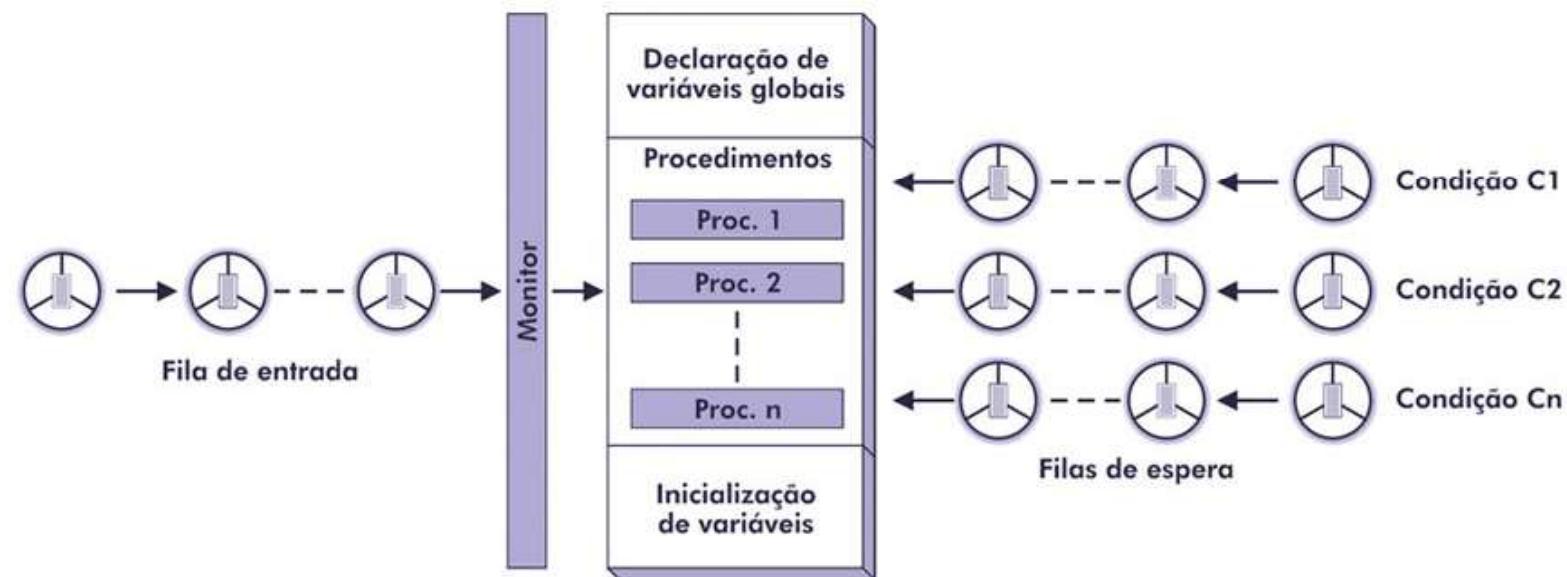


Fig. 7.5 Estrutura do monitor com variáveis de condição.

O problema do produtor/consumidor é mais uma vez apresentado, desta vez com o uso de um monitor para a solução. O monitor Condisional é estruturado com duas variáveis especiais de condição (Cheio e Vazio) e dois procedimentos (Produz e Consome).

```

MONITOR Condisional;
VAR Cheio, Vazio : (* Variaveis especiais de condicao *);

PROCEDURE Produz;
BEGIN
IF (Cont = TamBuf) THEN WAIT (Cheio);
.
.
IF (Cont = 1) THEN SIGNAL (Vazio);
END;

PROCEDURE Consome;
BEGIN
IF (Cont = 0) THEN WAIT (Vazio);
.
.
IF (Cont = TamBuf - 1) THEN SIGNAL (Cheio);
END;

```

```
BEGIN  
END;
```

Sempre que o processo produtor desejar gravar um dado no buffer, deverá fazê-lo através de uma chamada ao procedimento Produz do monitor (Condisional.Produz). Neste procedimento deve ser testado se o número de posições ocupadas no buffer é igual ao seu tamanho, ou seja, se o buffer está cheio. Caso o teste seja verdadeiro, o processo deve executar um WAIT em Cheio, indicando que a continuidade da sua execução depende dessa condição, e permanecer aguardando em uma fila de espera associada à variável Cheio. O processo bloqueado só poderá prosseguir sua execução quando um processo consumidor executar um SIGNAL na variável Cheio, indicando que um elemento do buffer foi consumido.

No caso do processo consumidor, sempre que desejar ler um dado do buffer deverá fazê-lo através de uma chamada ao procedimento Consome do monitor (Condisional.Consome). Sua condição de execução é existir ao menos um elemento no buffer. Caso o buffer esteja vazio, o processo deve executar um WAIT em Vazio, indicando que a continuidade da sua execução depende dessa condição, e permanecer aguardando em uma fila de espera associada à variável Vazio. O processo produtor que inserir o primeiro elemento no buffer deverá sinalizar esta condição através do comando SIGNAL na variável Vazio, liberando o consumidor.

O programa Produtor_Consumidor_3 descreve a solução completa com o uso do monitor Condisional apresentado. Os procedimentos Produz_Dado e Consome_Dado servem, respectivamente, como entrada e saída de dados para o programa. Os procedimentos Grava_Dado e Le_Dado são responsáveis, respectivamente, pela transferência e pela recuperação do dado no buffer.

```
PROGRAM Produtor_Consumidor_3;  
CONST TamBuf= (* Tamanho qualquer *);  
TYPE Tipo_Dado = (* Tipo qualquer*);  
VAR Buffer : ARRAY [1..TamBuf] OF Tipo_Dado;  
Dado: Tipo_Dado;  
  
MONITOR Condisional;  
VAR Vazio, Cheio : (* Variaveis de condicao *);  
Cont : INTEGER;  
  
PROCEDURE Produz;  
BEGIN  
IF (Cont = TamBuf) THEN WAIT (Cheio);  
Grava_Dado (Dado, Buffer);  
Cont = Cont + 1;  
IF (Cont = 1) THEN SIGNAL (Vazio);  
END;  
  
PROCEDURE Consome;  
BEGIN  
IF (Cont = 0) THEN WAIT (Vazio);  
Le_Dado (Dado, Buffer);  
Cont := Cont - 1;  
IF (Cont = TamBuf - 1) THEN SIGNAL(Cheio);  
END;
```

```

BEGIN
Cont := 0;
END;

PROCEDURE Produtor;
BEGIN
REPEAT
Produz_Dado (Dado);
Condicional.Produz;
UNTIL false;
END;

PROCEDURE Consumidor;
BEGIN
REPEAT
Condicional.Consome;
Consome_Dado (Dado);
UNTIL false;
END;

BEGIN
PARBEGIN
Produtor;
Consumidor;
PAREND;
END.

```

► 7.9 Troca de Mensagens

Troca de mensagens é um mecanismo de comunicação e sincronização entre processos. O sistema operacional possui um subsistema de mensagem que suporta esse mecanismo sem que haja necessidade do uso de variáveis compartilhadas. Para que haja a comunicação entre os processos deve existir um canal de comunicação, podendo esse meio ser um buffer ou um link de uma rede de computadores.

Os processos cooperativos podem fazer uso de um buffer para trocar mensagens através de duas rotinas: SEND (receptor, mensagem) e RECEIVE (transmissor, mensagem). A rotina SEND permite o envio de uma mensagem para um *processo receptor*, enquanto a rotina RECEIVE possibilita o recebimento de mensagem enviada por um *processo transmissor* (Fig. 7.6).



Fig. 7.6 Transmissão de mensagem.

O mecanismo de troca de mensagens exige que os processos envolvidos na comunicação tenham suas execuções sincronizadas. A sincronização é necessária, já que uma mensagem somente pode ser tratada por um processo após ter sido recebida, ou o envio de mensagem pode ser uma condição para a continuidade de execução de um processo.

A troca de mensagens entre processos pode ser implementada de duas maneiras distintas, comunicação direta e comunicação indireta. A *comunicação direta* entre dois processos exige que, ao enviar ou receber uma mensagem, o processo enderece explicitamente o nome do processo receptor ou transmissor. Uma característica deste tipo de comunicação é só permitir a troca de mensagem entre dois processos. O principal problema da comunicação direta é a necessidade da especificação do nome dos processos envolvidos na troca de mensagens. No caso de mudança na identificação dos processos, o código do programa deve ser alterado e recompilado (Fig. 7.7).

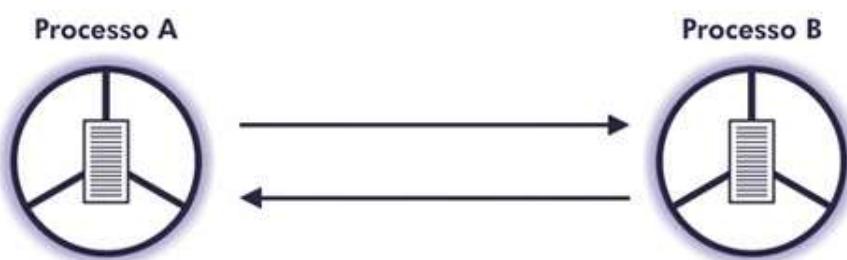


Fig. 7.7 Comunicação direta.

A *comunicação indireta* entre processos utiliza uma área compartilhada, onde as mensagens podem ser colocadas pelo processo transmissor e retiradas pelo receptor. Esse tipo de buffer é conhecido como *mailbox* ou *port*, e suas características, como identificação e capacidade de armazenamento de mensagens, são definidas no momento de criação. Na comunicação indireta, vários processos podem estar associados a mailbox, e os parâmetros dos procedimentos SEND e RECEIVE passam a ser nomes de mailboxes, e não mais nomes de processos (Fig. 7.8).

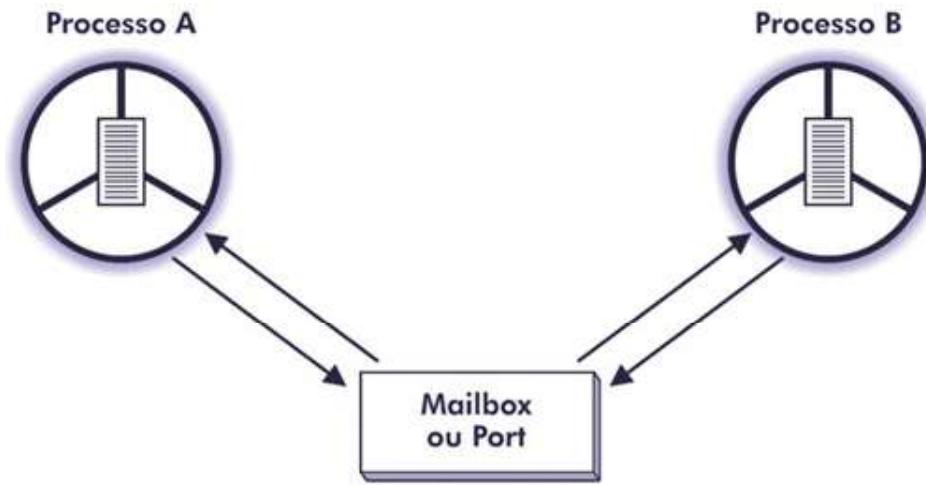


Fig. 7.8 Comunicação indireta.

Independentemente do mecanismo de comunicação utilizado, processos que estão trocando mensagens devem ter suas execuções sincronizadas em função do fluxo de mensagens. Um processo não pode tratar uma mensagem até que esta tenha sido enviada por outro processo, ou um processo não pode receber uma mesma mensagem mais de uma vez. Existem três diferentes esquemas para implementar a sincronização entre processos que trocam mensagens.

O primeiro esquema de sincronização é garantir que um processo, ao enviar uma mensagem, permaneça esperando até que o processo receptor a leia. Na mesma condição, um processo, ao tentar receber uma mensagem ainda não enviada, deve permanecer aguardando até que o processo transmissor faça o envio. Esse tipo de comunicação implementa uma forma síncrona de comunicação entre os processos conhecida como *rendezvous*. O problema desta implementação é que a execução dos processos fica limitada ao tempo de processamento no tratamento das mensagens.

Uma variação mais eficiente do *rendezvous* é permitir que o processo transmissor não permaneça bloqueado aguardando a leitura da mensagem pelo processo receptor. Neste caso, um processo enviará mensagens para diversos destinatários tão logo seja possível.

O terceiro esquema implementa uma forma assíncrona de comunicação, onde nem o processo receptor nem o processo transmissor permanecem aguardando o envio e o recebimento de mensagens. Nesse caso, além da necessidade de buffers para armazenar as mensagens deve haver outros mecanismos de sincronização que permitam ao processo identificar se uma mensagem já foi enviada ou recebida. A grande vantagem deste mecanismo é aumentar a eficiência de aplicações concorrentes.

Uma solução para o problema do produtor/consumidor é apresentada utilizando a troca de mensagens.

```

PROGRAM Produtor_Consumidor_4;
PROCEDURE Produtor;
VAR Msg : Tipo_Msg;
BEGIN
REPEAT
Produz_Mensagem (Msg);
SEND (Msg);
UNTIL false;
END;
```

```

PROCEDURE Consumidor;
VAR Msg : Tipo_Msg;
BEGIN
REPEAT
RECEIVE (Msg) ;
Consome_Mensagem (Msg) ;
UNTIL false;
END;

BEGIN
PARBEGIN
Produtor;
Consumidor;
PAREND;
END.

```

► 7.10 Deadlock

Deadlock é a situação em que um processo aguarda por um recurso que nunca estará disponível ou um evento que não ocorrerá. Essa situação é consequência, na maioria das vezes, do compartilhamento de recursos, como dispositivos, arquivos e registros, entre processos concorrentes em que a exclusão mútua é exigida. O problema do deadlock torna-se cada vez mais frequente e crítico na medida em que os sistemas operacionais evoluem no sentido de implementar o paralelismo de forma intensiva e permitir a alocação dinâmica de um número ainda maior de recursos.

A Fig. 7.9 ilustra graficamente o problema do deadlock entre os processos PA e PB, quando utilizam os recursos R1 e R2. Inicialmente, PA obtém acesso exclusivo de R1, da mesma forma que PB obtém de R2. Durante o processamento, PA necessita de R2 para poder prosseguir. Como R2 está alocado a PB, PA ficará aguardando que o recurso seja liberado. Em seguida, PB necessita utilizar R1 e, da mesma forma, ficará aguardando até que PA libere o recurso. Como cada processo está esperando que o outro libere o recurso alocado é estabelecida uma condição conhecida por *espera circular*, caracterizando uma situação de deadlock.

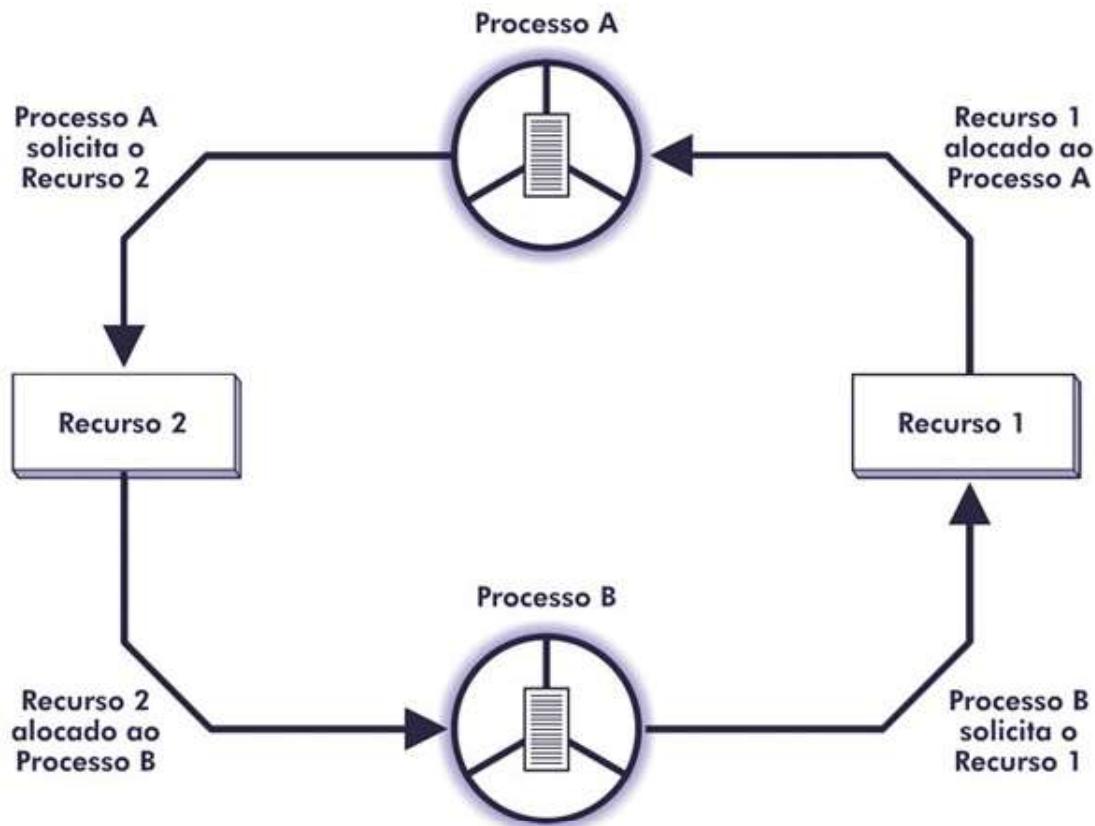


Fig. 7.9 Espera circular.

Para que ocorra a situação de deadlock, quatro condições são necessárias simultaneamente (Coffman, Elphick e Shoshani, 1971):

1. Exclusão mútua: cada recurso só pode estar alocado a um único processo em um determinado instante;
2. Espera por recurso: um processo, além dos recursos já alocados, pode estar esperando por outros recursos;
3. Não preempção: um recurso não pode ser liberado de um processo só porque outros processos desejam o mesmo recurso;
4. Espera circular: um processo pode ter de esperar por um recurso alocado a outro processo, e vice-versa.

O problema do deadlock existe em qualquer sistema multiprogramável; no entanto as soluções implementadas devem considerar o tipo do sistema e o impacto em seu desempenho. Por exemplo, um deadlock em um sistema de tempo real, que controla uma usina nuclear, deve ser tratado com mecanismos voltados para esse tipo de aplicação, diferentes dos adotados por um sistema de tempo compartilhado comum.

A seguir, são examinados os principais mecanismos de prevenção, detecção e correção de deadlocks.

7.10.1 Prevenção de Deadlock

Para prevenir a ocorrência de deadlocks, é preciso garantir que uma das quatro condições apresentadas, necessárias para sua existência, nunca se satisfaça.

A ausência da primeira condição (exclusão mútua) certamente acaba com o problema do deadlock, pois nenhum processo terá que esperar para ter acesso a um recurso, mesmo que já esteja sendo utilizado por outro processo. Os problemas decorrentes do compartilhamento de recursos entre processos concorrentes sem a exclusão mútua já foram apresentados, inclusive com a exemplificação das sérias inconsistências geradas.

Para evitar a segunda condição (espera por recurso), processos que já possuam recursos garantidos não devem requisitar novos recursos. Uma maneira de implementar esse mecanismo de prevenção é que, antes do início da execução, um processo deve pré-alocar todos os recursos necessários. Nesse caso, todos os recursos necessários ao processo devem estar disponíveis para o início da execução, caso contrário nenhum recurso será alocado e o processo permanecerá aguardando. Esse mecanismo pode produzir um grande desperdício na utilização dos recursos do sistema, pois um recurso pode permanecer alocado por um grande período de tempo, sendo utilizado apenas por um breve momento. Outra dificuldade, decorrente desse mecanismo, é a determinação do número de recursos que um processo deve alocar antes da sua execução. O mais grave, no entanto, é a possibilidade de um processo sofrer starvation caso os recursos necessários à sua execução nunca estejam disponíveis ao mesmo tempo.

A terceira condição (não preempção) pode ser evitada quando é permitido que um recurso seja retirado de um processo no caso de outro processo necessitar do mesmo recurso. A liberação de recursos já garantidos por um processo pode ocasionar sérios problemas, podendo até impedir a continuidade de execução do processo. Outro problema desse mecanismo é a possibilidade de um processo sofrer starvation, pois, após garantir os recursos necessários à sua execução, um processo pode ter que liberá-los sem concluir seu processamento.

A última maneira de evitar um deadlock é excluir a possibilidade da quarta condição (espera circular). Uma forma de implementar esse mecanismo é forçar o processo a ter apenas um recurso por vez. Caso o processo necessite de outro recurso, o recurso já alocado deve ser liberado. Esta condição restringiria muito o grau de compartilhamento e o processamento de programas.

A prevenção de deadlocks evitando-se a ocorrência de qualquer uma das quatro condições é bastante limitada e, por isso, não é utilizada na prática. É possível evitar o deadlock mesmo se todas as condições necessárias à sua ocorrência estejam presentes. A solução mais conhecida para essa situação é o algoritmo do banqueiro (Banker's Algorithm) proposto por Dijkstra (1965). Basicamente, este algoritmo exige que os processos informem o número máximo de cada tipo de recurso necessário à sua execução. Com essas informações, é possível definir o estado de alocação de um recurso, que é a relação entre o número de recursos alocados e disponíveis e o número máximo de processos que necessitam desses recursos. Com base nessa relação, o sistema pode fazer a alocação dos recursos de forma segura e evitar a ocorrência de deadlocks. Muitos livros abordam esse algoritmo com detalhes, e ele pode ser encontrado em Tanenbaum (2001).

Apesar de evitar o problema do deadlock, essa solução possui também várias limitações. A maior delas é a necessidade de um número fixo de processos ativos e de recursos disponíveis no sistema. Essa limitação impede que a solução seja implementada na prática, pois é muito difícil prever o número de usuários no sistema e o número de recursos disponíveis.

7.10.2 Detecção do Deadlock

Em sistemas que não possuam mecanismos que previnam a ocorrência de deadlocks, é necessário um esquema de detecção e correção do problema. A *detecção do deadlock* é o mecanismo que determina, realmente, a existência da situação de deadlock, permitindo identificar os recursos e processos envolvidos no problema.

Para detectar deadlocks, os sistemas operacionais devem manter estruturas de dados capazes de identificar cada recurso do sistema, o processo que o está alocando e os processos que estão à espera da liberação do recurso. Toda vez que um recurso é alocado ou liberado por um processo, a estrutura deve ser atualizada. Geralmente, os algoritmos que implementam esse mecanismo verificam a existência da espera circular, percorrendo toda a estrutura sempre que um processo solicita um recurso e ele não pode ser imediatamente garantido.

Dependendo do tipo de sistema, o ciclo de busca por um deadlock pode variar. Em sistemas de tempo compartilhado, o tempo de busca pode ser maior, sem comprometer o desempenho e a confiabilidade do sistema. Sistemas de tempo real, por sua vez, devem constantemente certificar-se da ocorrência de deadlocks, porém a maior segurança gera maior overhead.

7.10.3 Correção do Deadlock

Após a detecção do deadlock, o sistema operacional deverá de alguma forma corrigir o problema. Uma solução bastante utilizada pela maioria dos sistemas é, simplesmente, eliminar um ou mais processos envolvidos no deadlock e desalocar os recursos já garantidos por eles, quebrando, assim, a espera circular.

A eliminação dos processos envolvidos no deadlock e, consequentemente, a liberação de seus recursos podem não ser simples, dependendo do tipo do recurso envolvido. Se um processo estiver atualizando um arquivo ou imprimindo uma listagem, o sistema deve garantir que esses recursos sejam liberados sem problemas. Os processos eliminados não têm como ser recuperados, porém outros processos, que antes estavam em deadlock, poderão prosseguir a execução.

A escolha do processo a ser eliminado é feita, normalmente, de forma aleatória ou, então, com base em algum tipo de prioridade. Este esquema, porém, pode consumir considerável tempo do processador, ou seja, gerar elevado overhead ao sistema.

Uma solução menos drástica envolve a liberação de apenas alguns recursos alocados aos processos para outros processos, até que o ciclo de espera termine. Para que esta solução seja realmente eficiente, é necessário que o sistema possa suspender um processo, liberar seus recursos e, após a solução do problema, retornar à execução do processo, sem perder o processamento já realizado. Este mecanismo é conhecido como *rollback* e, além do overhead gerado, é muito difícil de ser implementado, por ser bastante dependente da aplicação que está sendo processada.

► 7.11 Exercícios

1. Defina o que é uma aplicação concorrente e dê um exemplo de sua utilização.
2. Considere uma aplicação que utilize uma matriz na memória principal para a comunicação entre vários processos concorrentes. Que tipo de problema pode ocorrer quando dois ou mais processos acessam uma mesma posição da matriz?
3. O que é exclusão mútua e como é implementada?
4. Como seria possível resolver os problemas decorrentes do compartilhamento da matriz, apresentado anteriormente, utilizando o conceito de exclusão mútua?
5. O que é starvation e como podemos solucionar esse problema?
6. Qual o problema com a solução que desabilita as interrupções para implementar a exclusão mútua?
7. O que é espera ocupada e qual o seu problema?
8. Explique o que é sincronização condicional e dê um exemplo de sua utilização.
9. Explique o que são semáforos e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para a sincronização condicional.
10. Apresente uma solução para o problema dos filósofos que permita que os cinco pensadores sentem à mesa, porém evite a ocorrência de starvation e deadlock.
11. Explique o que são monitores e dê dois exemplos de sua utilização: um para a solução da exclusão mútua e outro para a sincronização condicional.
12. Qual a vantagem da forma assíncrona de comunicação entre processos e como esta pode ser implementada?
13. O que é deadlock, quais as condições para obtê-lo e quais as soluções possíveis?
14. Em uma aplicação concorrente que controla saldo bancário em contas-correntes, dois processos compartilham uma região de memória onde estão armazenados os saldos dos clientes A e B. Os processos executam concorrentemente os seguintes passos:

Processo 1 (Cliente A)

```
/* saque em A */
1a. x := saldo_do_cliente_A;
1b. x := x - 200;
1c. saldo_do_cliente_A := x;

/* deposito em B */
1d. x := saldo_do_cliente_B;
1e. x := x + 100;
1f. saldo_do_cliente_B := x;
```

Processo 2 (Cliente B)

```
/*saque em A */
2a. y := saldo_do_cliente_A;
2b. y := y - 100;
2c. saldo_do_cliente_A := y;

/* deposito em B */
2d. y := saldo_do_cliente_B;
2e. y := y + 200;
2f. saldo_do_cliente_B := y;
```

Supondo que os valores dos saldos de A e B sejam, respectivamente, 500 e 900, antes de os

processos executarem, pede-se:

- a) Quais os valores corretos esperados para os saldos dos clientes A e B após o término da execução dos processos?
- b) Quais os valores finais dos saldos dos clientes se a sequência temporal de execução das operações for: 1a, 2a, 1b, 2b, 1c, 2c, 1d, 2d, 1e, 2e, 1f, 2f?
- c) Utilizando semáforos, proponha uma solução que garanta a integridade dos saldos e permita o maior compartilhamento possível dos recursos entre os processos, não esquecendo a especificação da inicialização dos semáforos.

15. O problema dos leitores/escritores, apresentado a seguir, consiste em sincronizar processos que consultam/atualizam dados em uma base comum. Pode haver mais de um leitor lendo ao mesmo tempo; no entanto, enquanto um escritor está atualizando a base, nenhum outro processo pode ter acesso a ela (nem mesmo leitores).

```
VAR Acesso: Semaforo := 1;
Exclusao: Semaforo := 1;
Nleitores: integer := 0;

PROCEDURE Escritor;
BEGIN
ProduzDado;
DOWN (Acesso);
Escreve;
UP (Acesso);
END;

PROCEDURE Leitor;
BEGIN
DOWN (Exclusao);
Nleitores := Nleitores + 1;
IF ( Nleitores = 1 ) THEN DOWN (Acesso);
UP (Exclusao);
Leitura;
DOWN (Exclusao);
Nleitores := Nleitores - 1;
IF ( Nleitores = 0 ) THEN UP (Acesso);
UP (Exclusao);
ProcessaDado;
END;
```

- a) Suponha que exista apenas um leitor fazendo acesso à base. Enquanto este processo realiza a leitura, quais os valores das três variáveis?
- b) Chega um escritor enquanto o leitor ainda está lendo. Quais os valores das três variáveis após o bloqueio do escritor? Sobre qual(is) semáforo(s) se dá o bloqueio?
- c) Chega mais um leitor enquanto o primeiro ainda não acabou de ler e o escritor está bloqueado. Descreva os valores das três variáveis quando o segundo leitor inicia a leitura.
- d) Os dois leitores terminam simultaneamente a leitura. É possível haver problemas quanto à integridade do valor da variável Nleitores? Justifique.

- e) Descreva o que acontece com o escritor quando os dois leitores terminam suas leituras. Descreva os valores das três variáveis quando o escritor inicia a escrita.
- f) Enquanto o escritor está atualizando a base, chegam mais um escritor e mais um leitor. Sobre qual(is) semáforo(s) eles ficam bloqueados? Descreva os valores das três variáveis após o bloqueio dos recém-chegados.
- g) Quando o escritor tiver terminado a atualização, é possível prever qual dos processos bloqueados (leitor ou escritor) terá acesso primeiro à base?
- h) Descreva uma situação onde os escritores sofram starvation (adiamento indefinido).