Capítulo 4

O PROBLEMA DA EXCLUSÃO MÚTUA

Martín Fierro:
Y el consejo del prudente
no hace falta en la partida;
siempre ha de ser comedida
la palabra de un cantor:
y aura quiero que me digas
de dónde nace el amor.

El moreno:
A pregunta tan escura
trataré de responder,
aunque es mucho pretender
de un pobre negro de estancia;
mas conocer su inorancia
es principio del saber.

Ama el pájaro en los aires
que cruza por donde quiera,
y si al fin de su carrera
se asienta en alguna rama,
con su alegre canto llama
a su amante compañera.

Este capítulo discute o problema de garantir exclusividade de execução para um processo, em um sistema de processos concorrentes. Os protocolos que implementam exclusão mútua entre processos são algoritmos clássicos e interessantes por si só. Nos algoritmos aqui apresentados não são usados recursos especiais de programação. Na verdade, eles podem ser expressos em qualquer linguagem de programação.

4.1 Caracterização do problema

Quando processos concorrentes compartilham dados (variáveis, estruturas de dados, arquivos), é necessário controlar o acesso a esses dados, para obter *determinância de execução*. Operações de atualização não podem ser feitas simultaneamente por diferentes processos. Tampouco operações

de leitura podem ocorrer simultaneamente com atualizações, pois os dados lidos podem estar temporariamente inconsistentes.

O seguinte exemplo ilustra os problemas que podem ocorrer quando não se garante exclusão mútua no acesso aos dados compartilhados. Supor que um vetor global R de 5 elementos seja utilizado para controlar a alocação de 5 unidades de um recurso (o recurso poderia ser, por exemplo, 5 blocos de memória de igual tamanho). A variável global T é usada como contador de unidades disponíveis e cada elemento do vetor identifica uma unidade. O valor R[T] (isto é, o elemento de R cujo índice é T) indica a próxima unidade a ser alocada. Os valores iniciais de R e T (com todas as unidades livres) são:

```
R = [5, 4, 3, 2, 1]
T = 5
```

Os procedimentos para requisitar e liberar uma unidade do recurso são:

```
requisita(U):          libera(U):
    . . .                  . . .
    U:=R[T];               T:=T+1;
    T:=T-1;                R[T]:=U;
    . . .                  . . .
```

Vamos supor que as 5 unidades tenham sido requisitadas (e alocadas) sem ocorrência de erro e que, em seguida, 3 delas tenham sido liberadas. Supondo que as unidades liberadas sejam as de número 3, 1 e 4, nesta ordem, a estrutura de dados que descreve o recurso ficaria na seguinte situação:

```
R = [3, 1, 4, 2, 1]
T = 3
```

Isto significa que, no momento, estão disponíveis as unidades 3, 1 e 4 (portanto, estão alocadas as unidades 5 e 2). Supor que, agora, dois processos executem as operações “libera(5)” e “requisita(K)” e que as 4 operações envolvidas sejam executadas na seguinte ordem:

```
T:=T+1      (passo 1 de libera(5): T=4)
K:=R[T]     (passo 1 de requisita(K): K=R[4]=2)
T:=T-1      (passo 2 de requisita(K): T=3)
R[T]:=5     (passo 2 de libera(5): R[3]=5)
```

O resultado neste caso será:

```
R = [3, 1, 5, 2, 1]
T = 3
```

E terá acontecido o seguinte: a unidade 2, que estava sendo usada por um processo, terá sido alocada para outro processo (o que não pode acontecer) e a unidade 5, liberada, estará

ocupando a 3ª posição do vetor R (lugar onde estava a unidade 4). O fato da unidade 5 ocupar o lugar onde estava a unidade 4, faz com que a unidade 4 desapareça do conjunto de unidades disponíveis. Em resumo, a unidade 2 estará alocada para dois processos ao mesmo tempo e estarão disponíveis as unidades 3, 1 e 5 (a unidade 4 sumiu do sistema).

O resultado correto da execução das operações "libera(5) || requisita(K)" poderia ser qualquer um dos dois seguintes:

R = [3, 1, 4, 5, 1]	ou	R = [3, 1, 5, 2, 1]
T = 3		T = 3

No primeiro caso teria sido alocada a unidade K=5, ficando disponíveis as unidades 3, 1 e 4. No segundo caso teria sido alocada a unidade K=4, ficando disponíveis as unidades 3, 1 e 5.

Este exemplo ilustra o conceito de condição de corrida (*race condition* – seção 2.2), pois o resultado da execução é função das velocidades relativas dos processos (ou da ordem com que os processos acessam os dados compartilhados). Na história da computação existem diversos exemplos de desastres devidos a esse tipo de erro. O caso da máquina de radioterapia Therac-25 é um exemplo clássico [LEV 93]. Uma condição de corrida fazia com que, às vezes, essa máquina administrasse dosagens milhares de vezes maior que a normal, resultando na morte ou em sérios danos para os pacientes.

O conceito de região crítica

Os trechos dos processos onde os dados compartilhados são acessados são denominados *trechos críticos, regiões críticas* ou *seções críticas.* A maneira de eliminar as condições de corrida de um programa concorrente é simples: basta garantir a exclusão mútua na execução dos trechos críticos dos processos. A seguir serão apresentados algoritmos que implementam exclusão mútua na execução de regiões críticas de sistemas concorrentes.

4.2 Exclusão mútua para 2 processos

Os algoritmos apresentados a seguir são válidos para o caso de 2 processos. Soluções para n processos são apresentadas na seção 4.3. Alguns dos algoritmos apresentam erros, os quais são devidamente apontados. Tanto faz considerar os processos sendo executados por um único processador (execução logicamente paralela) ou considerar cada processo executado por um processador próprio (execução fisicamente paralela); se há algum problema com o algoritmo, esse problema se manifesta em ambos os casos.

Supõe-se que os processos sejam cíclicos, isto é, que eles entrem e saiam várias vezes de suas seções críticas. Todas as soluções usam a técnica de *busy loop*[29] (ou *busy wait*) para fazer

[29] Um busy loop é uma situação na qual um processo fica utilizando o processador de forma contínua (testando uma condição), sem realizar trabalho útil.

um processo esperar pelo outro. São utilizadas *variáveis globais* (compartilhadas) para decidir se é possível entrar na região crítica ou não, e, em toda solução, ambos os processos executam o mesmo código (ou protocolo de acesso).

Os algoritmos são especificados utilizando a linguagem Vale 4. Nessa linguagem, o símbolo "¬" representa o operador lógico "not" (na verdade, o compilador aceita ambas as representações "¬" e "not").

4.2.1 Cinco tentativas de solução

Todas as cinco tentativas de solução desta seção apresentam algum tipo de erro. Como o erro é apontado no parágrafo que segue à apresentação do algoritmo, é aconselhável que o leitor tente encontrar o erro antes de ler esse parágrafo. Estes algoritmos foram originalmente apresentados no artigo clássico de Dijkstra [DIJ 65a].

Tentativa 1

Variável global: *em_uso*: boolean initial false;

A variável global indica se alguma região crítica está em uso ou não.

Código de um processo:

```
. . .
loop
    exit when ¬em_uso
endloop;
em_uso:=true;
REGIÃO CRÍTICA;
em_uso:= false;
. . .
```

A solução não é correta, pois os processos podem chegar à conclusão "simultaneamente" que a entrada está livre (*em_uso*=false). Isto é, os dois processos podem ler (testar) o valor de *em_uso* antes que essa variável seja feita igual a *true* por um deles.

OBSERVAÇÃO:
Para os próximos algoritmos, supõe-se que os processos tenham duas variáveis locais, *eu* e *outro*. Para o processo 1, *eu*=1 e *outro*=2; para o processo 2, *eu*=2 e *outro*=1. Os códigos dos processos são iguais, mas cada processo usa suas variáveis privativas *eu* e *outro*.

Tentativa 2

Variável global: *vez*: integer initial 1;

Esta variável indica de quem é a vez, na hora de entrar na região crítica.

Código de um processo:

```
. . .
loop
    exit when vez=eu
endloop;
REGIÃO CRÍTICA;
vez:= outro;
. . .
```

Este algoritmo garante exclusão mútua, mas obriga a alternância na execução das regiões críticas (isto é, primeiro entra P1, depois P2, depois P1, etc). Não é possível um mesmo processo entrar duas vezes consecutivamente. Se P2 desejar entrar primeiro na sua região crítica, ele não poderá fazê-lo, pois P1 deve ser o primeiro. Se um processo terminar, o outro não poderá mais entrar na sua região crítica. Não é aceitável que um processo tranque a entrada de outro sem necessidade.

Tentativa 3

Variável global: *quer*: array[2] of boolean initial false;

Se *quer*[i] é *true*, isto indica que o processo Pi (i∈{1,2}) quer entrar na sua região crítica. Observe que o valor inicial especificado para um *array* (vetor ou matriz) se aplica a todos os elementos desse *array*. Esta convenção é adotada na linguagem V4.

Código de um processo:

```
. . .
loop
    exit when ¬quer[outro]
endloop;
quer[eu]:=true;
REGIÃO CRÍTICA;
quer[eu]:=false;
. . .
```

A solução não assegura exclusão mútua. Repete-se aqui o mesmo problema da tentativa 1, pois cada processo pode chegar à conclusão que o outro não quer entrar e assim entrarem

simultaneamente na região crítica. Isso acontece porque existe a possibilidade de cada processo testar se o outro não quer entrar antes de um deles marcar a sua intenção de entrar.

O próximo algoritmo (tentativa 4) soluciona esse problema fazendo com que cada processo marque sua intenção de entrar antes de testar a intenção do outro.

OBSERVAÇÃO IMPORTANTE:

Se cada processo marca sua intenção de entrar antes de testar a intenção do outro, então, certamente, se um processo chega à conclusão que o outro não quer entrar é porque esse outro ainda não testou a intenção do primeiro (pois o teste é realizado depois de marcar a intenção de entrar). Portanto, quando esse outro processo for realizar o teste, certamente, irá encontrar a indicação que o parceiro quer entrar. Isto impossibilita que os dois processos cheguem "simultaneamente" a conclusão que o outro não quer entrar.

Tentativa 4

Variável global: *quer*: array[2] of boolean initial false;

É o mesmo algoritmo anterior, porém marcando a intenção de entrar, antes de testar a intenção do outro processo.

Código de um processo:

```
. . .
quer[eu]:=true;
loop
    exit when ¬quer[outro]
endloop;
REGIÃO CRÍTICA;
quer[eu]:=false;
. . .
```

Com este algoritmo a exclusão mútua é garantida, mas, infelizmente, os processos podem entrar em um loop eterno. Isto porque ambos os processos podem marcar "simultaneamente" a intenção de entrar (antes que um deles consiga testar, dentro do loop, se o outro quer entrar ou não). Nesse caso, depois de entrarem no loop, os processos não vão sair mais de lá.

O próximo algoritmo melhora esta tentativa de solução, pois, dentro do loop, o processo verifica se o outro também quer entrar e, em caso afirmativo, dá a vez para o parceiro.

Tentativa 5

Variável global: *quer*: array[2] of boolean initial false;

É semelhante ao algoritmo anterior, porém o processo dá a vez para o outro no caso do outro querer entrar.

Código de um processo:

```
            . . .
início:  quer[eu]:=true;
         if quer[outro] then
             {  quer[eu]:=false;
                goto inicio
             };
         REGIÃO CRÍTICA;
         quer[eu]:=false;
         . . .
```

A exclusão mútua continua garantida, mas pode acontecer dos processos ficarem dando a vez um para o outro indefinidamente. Embora essa "sincronização" dos processos seja difícil de acontecer na prática, ela não deixa de ser teoricamente possível.

O algoritmo seguinte soluciona definitivamente o problema, usando uma variável adicional *vez* para resolver as situações de empate. Esta variável só é usada quando os dois processos querem entrar simultaneamente nas regiões críticas.

4.2.2 Solução de Dekker

Variáveis globais: *quer*: array[2] of boolean initial false;
vez: integer initial 1;

Esta solução foi proposta pelo matemático holandês T. Dekker e discutida por Dijkstra [DIJ 65a]. Trata-se da primeira solução completa para o problema. Conforme já referido, é similar ao algoritmo anterior. A diferença está no uso da variável *vez*, para realizar o desempate (*tie-break*). No caso em que os dois processos entram no bloco entre chaves, só um deles (decidido pelo valor da variável *vez*) dá a vez para o outro.

Código de um processo:

```
            . . .
início:    quer[eu]:=true;
denovo: if quer[outro] then
            {   if vez=eu then goto denovo;
                quer[eu]:=false;
```

```
    loop
        exit when vez=eu
    endloop;
    goto início
};
REGIÃO CRÍTICA;
quer[eu]:=false;
vez:=outro;
. . .
```

O algoritmo satisfaz todas as exigências para a solução ser considerada correta, quaisquer que sejam as velocidades relativas dos processos. Por exemplo, não acontece de um processo ficar eternamente tentando entrar na sua região crítica enquanto o outro, mais veloz, entra e sai repetidamente da sua região crítica.

4.2.3 Solução de Peterson

O algoritmo anterior foi apresentado por Dekker na década de 60. Na década de 80, G. Peterson tornou mais simples esta solução [PET 81]. O algoritmo de Peterson é apresentado a seguir em duas versões que diferem apenas nos nomes das variáveis utilizadas. O truque do algoritmo consiste no seguinte: ao marcar sua intenção de entrar, o processo já indica (para o caso de haver empate) que a vez é do outro.

Versão 1:

A exclusão mútua é garantida através do vetor *quer*. Se há empate, então é usada a variável *vez*, que dá a vez para o primeiro que chegou.

Variáveis globais: *quer*: array[2] of boolean initial false;
vez: integer;

Código de um processo:

```
. . .
quer[eu]:= true;
vez:= outro;
loop
    exit when ¬quer[outro] or vez=eu
endloop;
REGIÃO CRÍTICA;
quer[eu]:= false;
. . .
```

Versão 2:

É o mesmo algoritmo anterior, porém com outros nomes de variáveis, que irão facilitar o entendimento da generalização para *n* processos, a ser discutida na seção 4.3.4. Na generalização, interessa saber qual foi o último processo que chegou e não o primeiro. Na versão a seguir, em caso de empate, fica trancado o último que chegou (antes, nesse caso, entrava o primeiro que tinha chegado – o que é equivalente).

Variáveis globais: *quer*: array[2] of boolean initial false;
ultimo: integer;

Código de um processo:

```
. . .
quer[eu]:= true;
ultimo:= eu;
loop
  exit when ¬quer[outro] or ultimo≠eu
endloop;
REGIÃO CRÍTICA;
quer[eu]:= false;
. . .
```

4.2.4 Solução com instruções "test and set"

O que complica o problema da exclusão mútua é a possibilidade de um processo perder a UCP depois de testar o valor de uma variável global, mas antes de conseguir atribuir novo valor à mesma. O algoritmo da tentativa 1 (seção 4.2.1), o mais simples de todos, poderia funcionar corretamente se existisse uma instrução capaz de testar uma variável e atribuir novo valor à mesma sem possibilidade de interrupção. Neste caso as operações "testar" e "atribuir" seriam realizadas de forma atômica ou indivisível.

Hoje em dia todos os computadores possuem instruções do tipo "test and set" que, em uma única instrução de máquina (sem possibilidade de interrupção), testam o valor de uma variável e atribuem novo valor à mesma. No que segue, vamos supor a existência de uma função booleana TS(X), que executa o seguinte código, de forma indivisível:

```
function TS(X: boolean) returns boolean;
{   TS:= X;
    X:= false
}
```

A seguir é mostrada a solução do problema da exclusão mútua utilizando a função TS. O algoritmo utiliza uma variável global *is_free* para controlar o acesso à região crítica.

Variável global: *is_free*: boolean initial true;

Código de um processo:

```
. . .
loop
    exit when TS(is_free)
endloop;
REGIÃO CRÍTICA;
is_free:= true;
. . .
```

Deve ser observado que esta solução é válida para qualquer número de processos. As soluções anteriores valiam para 2 processos apenas.

4.3 Exclusão mútua com *n* processos

A seguir são apresentadas 6 soluções para o problema da exclusão mútua envolvendo múltiplos processos. Uma solução para *n* processos é correta se assegura o cumprimento dos seguintes requisitos:

1) garantia de exclusão mútua,

2) garantia de progresso para os processos,

3) garantia de tempo de espera limitado.

O primeiro requisito é óbvio, pois é o objetivo principal da solução. O segundo diz que se não há processo dentro de região crítica e existem processos desejando entrar, então apenas os processos que desejam entrar participam na decisão de quem entra e essa decisão não é postergada indefinidamente. O terceiro requisito diz que um processo que deseja entrar na região crítica não deve esperar indefinidamente por sua vez; isto é, quando existe algum processo esperando para entrar, deve haver um limite no número de vezes que outros processos entram e saem de suas regiões críticas.

4.3.1 Algoritmo de Dijkstra

Dijkstra apresentou este algoritmo em 1965 [DIJ 65b].

Variáveis globais: *quer*, *dentro*: array[*n*] of boolean;
vez: integer;

Os dois vetores são inicializados com *false*. Os elementos *quer*[*i*] e *dentro*[*i*] são alterados apenas pelo processo *i*, 1≤*i*≤*n*. A variável *vez* assume valores entre 1 e *n* e seu valor inicial é irrelevante. Cada processo utiliza uma variável local *k*.

O comando "for *k*:=1 to *n* st *k*≠*i* ... endfor" não faz parte da linguagem Vale 4. Nele, a sigla "st" deve ser entendida como *such that* (tal que). Em sua execução, a variável *k* vai assumir os valores 1, 2, ..., *n*, sendo deixada de fora a iteração correspondente a *k*=*i*.

Código do processo i:

```
        . . .
        quer[i]:=true;
inicio: loop
            dentro[i]:=false;
            if ¬quer[vez] then vez:=i;
            exit when vez=i
        endloop;
        dentro[i]:=true;
        for k:=1 to n st k≠i
            if dentro[k] then goto inicio
        endfor;
        REGIÃO CRÍTICA;
        quer[i]:=false;
        dentro[i]:=false;
        . . .
```

Um processo *i* só entra na região crítica se após fazer o seu *dentro*[*i*] igual a *true* ele encontra todos os demais processos com "*dentros*" iguais a *false*. Isto garante exclusão mútua.

O requisito de progresso é garantido pela seguinte propriedade: após um processo *i* fazer *vez*:=*i*, nenhum outro conseguirá atribuir seu número à *vez* antes da região crítica ser executada. Explicando melhor: se o processo *vez* não está entre os que desejam entrar, então os que desejam entrar vão fazer *vez*:="eu"; após todas as atribuições à variável *vez* serem realizadas, essa variável vai indicar um dos processos que deseja entrar e esse valor não vai ser mais alterado. Todos os processos que desejavam entrar, com exceção daquele indicado por *vez*, vão fazer seus "*dentros*" iguais a *false* e ficarão repetindo o comando loop.

O requisito "espera limitada" não é garantido, pois, teoricamente, um processo pode ficar indefinidamente tentando entrar na sua região crítica, enquanto os outros entram e saem (convença-se disso).

4.3.2 Algoritmo de Eisenberg e McGuire

Este algoritmo foi publicado em 1972 [EIS 72].

Variáveis globais: *quer*, *dentro*: array[*n*] of boolean;
vez: integer;

Os dois vetores são inicializados com *false*. Os valores de quer[*i*] e *dentro*[*i*] são alterados apenas pelo processo *i*, 1≤*i*≤*n*. A variável *vez* assume valores entre 1 e *n* e seu valor inicial é irrelevante. Cada processo utiliza uma variável local *k*.

Código do processo i:

```
            . . .
            quer[i]:=true;
inicio:     dentro[i]:=false;
            k:=vez;
            loop
                if ¬quer[k] then k:=(k mod n)+1
                else k:=vez;
                exit when k=i
            endloop;
            dentro[i]:=true;
            for k:=1 to n st k≠i
                if dentro[k] then goto inicio
            endfor;
            /* Aqui o processo  i  já garantiu a exclusão mútua,
               mas vai dar uma última chance ao processo vez.
            */
            if vez≠i and quer[vez] then goto inicio;
            vez:=i;
            REGIÃO CRÍTICA;
            k:=(vez mod n)+1;
            loop
                exit when quer[k];
                k:=(k mod n)+1
            endloop;
            vez:=k;
            quer[i]:=false;
            dentro[i]:=false;
            . . .
```

O algoritmo é similar ao de Dijkstra, porém corrigido em relação ao problema da espera limitada. A exclusão mútua é garantida, como antes, através do vetor *dentro*.

A garantia de progresso é conseguida da seguinte maneira: o valor de *vez* é alterado apenas quando um processo entra na região crítica (e, mais tarde, quando sai); então, se nenhum processo está entrando ou saindo de sua região crítica, o valor de *vez* permanece constante; dentre os processos que desejam entrar, o primeiro na ordenação cíclica *vez*, *vez*+1, ..., *n*, 1, 2, ..., *vez*-1 é o que vai conseguir.

A espera limitada é garantida porque quando um processo deixa a sua região crítica ele designa como seu único sucessor o primeiro processo, dentre os que desejam entrar, na ordem cíclica *vez*+1, ..., *n*, 1, 2, ..., *vez*-1, *vez*; isto garante que qualquer processo desejando entrar espere no máximo *n*-1 processos entrar antes dele. Convém observar que, se nenhum processo deseja entrar na sua região crítica, o processo que sai da região crítica faz *vez* igual ao seu próprio número.

Por motivos didáticos, o algoritmo acima usou os mesmos nomes de variáveis usados pelo algoritmo de Dijkstra. Isto facilita o seu atendimento e deixa claro o relacionamento existente entre os dois algoritmos. O algoritmo original de Eisenberg e McGuire [EIS 72] pode ser encontrado na maioria dos livros de programação concorrente e sistemas operacionais.

4.3.3 Algoritmo de Lamport

Este algoritmo é inspirado na técnica de escalonamento comumente usada em ferragens, onde cada cliente que chega recebe um ticket de atendimento. Originalmente, o algoritmo foi desenvolvido para um ambiente distribuído [LAM 74].

Diferentemente do que acontece no comércio, pode acontecer de dois ou mais processos receberem o mesmo ticket. No caso de empate, o processo de menor identificação (o mais velho) é atendido primeiro, isto é, se P_i e P_j recebem o mesmo ticket e i<j então P_i é atendido em primeiro lugar.

Variáveis globais: *pegando*: array[*n*] of boolean;
ticket: array[*n*] of integer;

Os vetores são inicializados com *false* e zero, respectivamente. Utiliza-se a seguinte notação:

- (a,b) < (c,d) significa que a<c ou, se a=c, então b<d;
- $\max(a_1,a_2,\ldots, a_n)$ é o menor número *k* tal que $k \geq a_i$ para $i = 1,\ldots,n$.

Código do processo i:

```
. . .
pegando[i]:= true;
ticket[i]:= max(ticket[1],ticket[2],...,ticket[n]) + 1;
pegando[i]:= false;
for j:= 1 to n st j≠i
    loop
        exit when ¬pegando[j]
    endloop;
    loop
        exit when ticket[j]=0  or  (ticket[i],i)<(ticket[j],j)
    endloop
endfor;
REGIÃO CRÍTICA;
ticket[i]:= 0;
. . .
```

Antes de entrar na região crítica, o processo *i* compara o seu ticket com o ticket de cada outro processo *j*; essa comparação é feita só após *i* adquirir a certeza (no primeiro loop) que ou (1) *j* já pegou o seu ticket (i.é, já foi atribuido um valor à *ticket*[*j*]), ou (2) *j* ainda não iniciou o processo de retirar um ticket (e nesse caso certamente *j* irá pegar um ticket maior que *ticket*[*i*]). No primeiro caso pode-se ter qualquer relação entre *ticket*[*i*] e *ticket*[*j*] (maior, menor ou igual), mas certamente o par (*ticket*[*id*], *id*) de um dos processos será estritamente menor que o par do outro. Nesse caso, o processo com o maior par ficará retido no segundo loop até que o outro entre e saia da sua região crítica, garantindo a exclusão mútua. Para mostrar que o "progresso" e a "espera limitada" são garantidos e que o algoritmo é justo (isto é, atende igualmente a todos), é suficiente observar que os processos entram nas suas regiões críticas na ordem FCFS ("First Come First Served").

Teoricamente, é possível que mais de um processo entre na região crítica. Isto pode ocorrer quando o limite de representação de inteiros é ultrapassado no comando que obtém o número do ticket, sendo atribuído a um processo um número menor que o do processo correntemente na região crítica.

4.3.4 Algoritmo de Peterson

A generalização de Peterson para *n* processos [PET 81] envolve alguns novos elementos em relação à solução para 2 processos. A idéia básica é que cada processo deve passar por *n*-1 estágios antes de poder entrar na sua região crítica. Um processo avança um estágio sempre que (a) ele está na frente de todos os demais (ninguém está em um estágio maior ou igual ao dele) ou

(b) chega outro processo no estágio em que ele está (ele deixa de ser o último no seu estágio atual).

Variáveis globais: *estagio*: array[*n*] of integer initial 0;
ultimo: array[*n*-1] of integer;

No vetor *estagio*, o elemento *estagio*[*i*], $1 \leq i \leq n$, indica o estágio atual do processo *i*. Portanto, cada elemento corresponde a um processo (inicialmente, todos os elementos estão zerados). No vetor *ultimo*, o elemento *ultimo*[*j*], $1 \leq j \leq n-1$, indica qual é o último processo que chegou ao estágio *j*, portanto cada elemento corresponde a um estágio.

Código do processo i:

```
. . .
for j:= 1 to n-1                    /*percorre os n-1 estágios*/
    estagio[i]:=j;                      /*estágio atual de i*/
    ultimo[j]:=i;                       /*o último sou eu*/
    for k:= 1 to n st k≠i
        loop
            exit when estagio[i]>estagio[k] or ultimo[j]≠i
        endloop
    endfor
endfor;
REGIÃO CRÍTICA;
estagio[i]:= 0;
. . .
```

Em cada estágio o processo se compara com todos os demais. O processo decide se avança nessas comparações através do algoritmo de 2 processos, mas com uma interpretação um pouco diferente. Aqui o processo avança quando ele está num estágio mais adiantado (em relação ao outro) ou quando ele não é o último no estágio atual. Só após ter sucesso na comparação com todos os demais é que o processo avança um estágio. O comportamento resultante é o seguinte: o processo *i* avança um estágio sempre que (a) todos os processos na sua frente saem de suas regiões críticas ou (b) outro processo entra no seu estágio atual (p_i deixa de ser o último).

Enquanto um processo está executando (ou tentando entrar) na região crítica, os estágios funcionam como barreiras (em cada barreira fica retido o último que lá chega): o estágio 1 deixa passar no máximo *n*-1 processos, o estágio 2 deixa passar no máximo *n*-2 e o último deixa passar apenas 1 processo. Isto garante a exclusão mútua. Para se convencer que os outros dois requisitos também são satisfeitos basta observar que um processo se bloqueia somente se algum outro está na sua frente; como um processo não fica indefinidamente na sua região crítica, isto garante que todos os processos possam avançar.

Deve ser observado que para entrar na região crítica todo processo executa $O(n^2)$ vezes o algoritmo para 2 processos, mesmo que só um esteja desejando entrar. O algoritmo a seguir reduz o número de operações para $O(n*m)$ quando a competição envolve apenas m dos n processos.

4.3.5 Algoritmo de Block e Woo

A seguinte constatação permite otimizar o algoritmo de Peterson [BLO 90]. Se o processo está na frente de todos os demais, então ele pode entrar diretamente na região crítica (sem passar pelos restantes estágios).

Variáveis globais: *quer*: array[*n*] of integer;
ultimo: array[*n*] of integer;

Cada processo possui uma variável local, denominada *estagio*, que indica o estágio atual do mesmo. Os elementos do vetor *quer* assumem apenas os valores 0 e 1. O somatório dos elementos desse vetor ($\sum quer$) indica o número de processos que estão tentando entrar na região crítica.

Código do processo i:

```
. . .
estagio:= 0;
quer[i]:= 1;
repeat
      estagio:= estagio+1;          /*estágio atual de i*/
      ultimo[estagio]:= i;          /*último sou eu*/
      loop
            exit when ultimo[estagio]≠i or estagio=∑quer
      endloop
until ultimo[estagio] = i;          /*estagio=∑quer*/
REGIÃO CRÍTICA;
quer[i]:= 0;
. . .
```

O processo avança um estágio (sem se comparar com os demais) quando ele deixa de ser o último no seu estágio atual. Mais do que isso, o processo entra diretamente na região crítica quando seu estágio é igual ao número de processos que estão tentando entrar na região crítica. Isso significa que todos os demais processos estão atrasados em relação a ele (lembre que cada estágio retém um processo).

Considerando que *m* processos estão competindo, a complexidade do algoritmo é O(*n***m*) visto que são percorridos apenas *m* estágios e, em cada estágio, são verificados os valores dos *n* elementos do vetor *quer*.

4.3.6 Algoritmo de Toscani

Nesta solução, um processo executa *n*-1 vezes o algoritmo de 2 processos antes de entrar na região crítica. Trata-se de uma generalização natural, pois não introduz novos elementos na solução original de 2 processos. Na realidade, o mesmo método pode ser usado para generalizar qualquer solução de 2 processos sem acrescentar novos elementos na generalização.

Variáveis globais: *quer*: array[*n*,*n*] of boolean initial false;
ultimo: array[*n*,*n*] of integer;

O array *quer* é inicializado com *false* em todas as suas posições. No algoritmo que segue, *ii* representa min(*i*,*j*) e *jj* representa max(*i*,*j*).

Código do processo i:

```
. . .
for j:= 1 to n st j≠i
    quer[i,j]:=true;
    ultimo[ii,jj]:=i;
    loop
        exit when ¬quer[j,i] or ultimo[ii,jj]≠i
    endloop
endfor;
REGIÃO CRÍTICA
for j:= 1 to n
    quer[i,j]:=false
endfor
. . .
```

Na competição dos processos *i* e *j*, o processo *i* avisa ao processo *j* que quer entrar, usando o elemento *quer*[*i*,*j*] e, reciprocamente, o processo *j* avisa o processo *i* usando o elemento *quer*[*j*,*i*]. Para o caso de empate nessa competição, é usado o elemento *ultimo*[*ii*,*jj*], onde *ii* representa min(*i*,*j*) e *jj* representa max(*i*,*j*). Portanto, o algoritmo usa apenas o triângulo superior da matriz *ultimo*.

O número de operações do algoritmo é O(*n*), pois um processo executa *n*-1 vezes o protocolo de entrada na região crítica para 2 processos. A idéia da generalização é simples: cada processo compete com todos os demais pelo “direito de entrar na região crítica”. Para competir,

cada par de processos <*i, j*>, com *i≠j*, usa o algoritmo de dois processos, com variáveis *únicas e exclusivas*[30] do par. Nessa competição é decidido quem prossegue e quem fica retido.

A prova de que o algoritmo satisfaz as condições de "progresso" e de "espera limitada" é deixada como exercício.

4.4 EXERCÍCIOS

Exercício 4.1 Um especialista em programação concorrente apresentou a seguinte solução para o problema da exclusão mútua entre dois processos:

Variáveis compartilhadas: *quer* : array[2] of boolean initial false;
vez : integer;

Processo 1:

```
. . .
quer[1] := true;
loop
    exit when vez=1;
    if ¬quer[2] then vez:=1
endloop;
REGIÃO CRÍTICA;
quer[1] := false;
. . .
```

Processo 2:

```
. . .
quer[2] := true;
loop
    exit when vez=2;
    if ¬quer[1] then vez:=2
endloop;
REGIÃO CRÍTICA;
quer[2] := false;
. . .
```

Responda (justificando a resposta):

(a) Pode haver bloqueio eterno dos dois processos?

(b) Os dois processos podem estar simultaneamente na região crítica?

Exercício 4.2 (Solução de Hyman [HYM 66].) Explique se a seguinte solução para 2 processos satisfaz as 3 condições para que um algoritmo de exclusão mútua seja considerado correto:

Variáveis globais: *quer* : array[2] of boolean initial false;
vez : integer initial 1

Código de um processo:

```
. . .
quer[eu]:= true;
loop
    exit when vez=eu;
    loop
```

[30] Mesmo os pares <*i, j*> e <*i, k*> usam conjuntos disjuntos de variáveis, se *j≠k*.

```
            exit when ¬quer[outro]
        endloop;
        vez:= eu
    endloop;
    REGIÃO CRÍTICA;
    quer[eu]:= false;
    . . .
```

Exercício 4.3 (Solução de Doran & Thomas [DOR 80]) Compare a seguinte solução com a solução de Dekker e diga se ela é correta ou não.

Código de um processo:

```
    . . .
    quer[eu]:=true;
    if quer[outro] then
        if vez=outro then
        {   quer[eu]:=false;
            loop
                exit when vez=eu
            endloop
            quer[eu]:=true;
        }
    REGIÃO CRÍTICA;
    quer[eu]:=false;
    vez:=outro
    . . .
```

Exercício 4.4 Considerando o algoritmo de Dijkstra, mostre que o mesmo não satisfaz o requisito de espera limitada, isto é, mostre que um processo pode ficar indefinidamente tentando entrar na sua região crítica enquanto os outros entram e saem.

Exercício 4.5 Responda se o algoritmo de Dijkstra permite que se tenha um processo *k* dentro da região crítica e ao mesmo tempo se tenha $vez \neq k$.

Exercício 4.6 A seguir é apresentada a solução original de Eisenberg e McGuire. Compare a solução com o algoritmo apresentado no texto.

Variáveis globais: *flag*: array[*n*] of (*out*, *quer*, *in*);
vez: integer;

Os elementos do vetor *flag* são inicializados com *out*. O valor de *flag*[*i*] é alterado apenas pelo processo *i*. A variável *vez* assume valores entre 1 e *n* e seu valor inicial é irrelevante. Cada processo utiliza uma variável local *k*.

Código do processo i:

```
        . . .
inicio: flag[i]:=quer;
        k:=vez;
        loop
            if flag[k]=out then k:=(k mod n)+1
            else k:=vez;
            exit when k=i
        endloop;
        flag[i]:=in;
        for k:=1 to n st k≠i
            if flag[k]=in then goto inicio
        endfor;
        if vez≠i and flag[vez]≠out then goto inicio;
        vez:=i;
        REGIÃO CRÍTICA;
        k:=(vez mod n)+1;
        loop
            exit when flag[k]≠out;
            k:=(k mod n)+1
        endloop;
        vez:=k;
        flag[i]:=out;
        . . .
```

Exercício 4.7 No algoritmo de Block&Woo por que o comando *repeat* termina por "until *ultimo*[*estagio*] = *i*" e não por "until *estagio*=∑*quer*" ?

Exercício 4.8 Prove que o algoritmo de Toscani satisfaz as condições de "progresso" e de "espera limitada".

Exercício 4.9 Usando a linguagem V4, programe os diversos algoritmos de exclusão mútua e teste o funcionamento dos mesmos.