

## Exercícios Resolvidos (Problemas Clássicos e Outros)

### 1) Produtor-consumidor com buffer limitado

Este problema pode ser enunciado como segue. Um par de processos compartilha um buffer de N posições. O primeiro processo, denominado produtor, passa a vida a produzir mensagens e a colocá-las no buffer. O segundo processo, denominado consumidor, passa a vida a retirar mensagens do buffer (na mesma ordem em que elas foram colocadas) e a consumí-las.

A relação produtor-consumidor ocorre comumente em sistemas concorrentes e o problema se resume em administrar o buffer que tem tamanho limitado. Se o buffer está cheio, o produtor deve se bloquear, se o buffer está vazio, o consumidor deve se bloquear. A programação desse sistema com buffer de 5 posições e supondo que as mensagens sejam números inteiros, é mostrada a seguir.

Variáveis globais:       buffer: array[5] of integer;  
                              cheios: semaphore initial 0;  
                              vazios: semaphore initial 5;

Processo produtor:

```
msg, in : integer;
loop
  % produz mensagem msg
  P(vazios);
  in:= (in mod 5)+1;
  buffer[in]:= msg;
  V(cheios)
endloop
```

Processo consumidor:

```
msg, out : integer;
loop
  P(cheios);
  out:= (out mod 5)+1;
  msg:= buffer[out];
  V(vazios);
  % consome a mensagem
endloop
```

O semáforo cheios conta o número de buffers cheios e o semáforo vazios conta número de buffers vazios. Conforme já foi referido, as variáveis inteiras que não são inicializadas tem seu valor inicial igual a zero.

Observe que a solução não se preocupou em garantir exclusão mútua no acesso ao buffer. Isto porque os dois processos trabalham com variáveis locais in, out e msg e, certamente, irão acessar sempre posições diferentes do vetor global buffer.

### 2) Jantar dos Filósofos

Este problema ilustra as situações de deadlock e de postergação indefinida que podem ocorrer em sistemas nos quais processos adquirem e liberam recursos continuamente. Existem N filósofos que passam suas vidas pensando e comendo. Cada um possui seu lugar numa mesa circular, em cujo centro há um grande prato de spaghetti. A figura ilustra a situação para 5 filósofos. Como a massa é muito escorregadia, ela requer dois garfos para ser comida. Na mesa existem N garfos, um entre cada dois filósofos, e os únicos garfos que um filósofo pode usar são os dois que lhe correspondem (o da sua esquerda e o da sua direita). O problema consiste em simular o comportamento dos filósofos procurando evitar situações de deadlock (bloqueio permanente) e de postergação indefinida (bloqueio por tempo indefinido).



```

void take_forks(int i)          /* i: philosopher number, from 0 to N-1 */
{ down(&mutex);                /* enter critical region */
  state[i] = HUNGRY;           /* record fact that philosopher i is hungry */
  test(i);                     /* try to acquire 2 forks */
  up(&mutex);                  /* exit critical region */
  down(&s[i]);                 /* block if forks were not acquired */
}

void put_forks(i)             /* i: philosopher number, from 0 to N-1 */
{ down(&mutex);                /* enter critical region */
  state[i] = THINKING;         /* philosopher has finished eating */
  test(LEFT);                  /* see if left neighbor can now eat */
  test(RIGHT);                 /* see if right neighbor can now eat */
  up(&mutex);                  /* exit critical region */
}

void test(i)                  /* i: philosopher number, from 0 to N-1 */
{ if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
  state[i] = EATING; up(&s[i]); }
}

```

### 3) Barbeiro dorminhoco

O problema consiste em simular o funcionamento de uma barbearia com as seguintes características. A barbearia tem uma sala de espera com N cadeiras e uma cadeira de barbear. Se não tem clientes à espera, o barbeiro senta numa cadeira e dorme. Quando chega um cliente, ele acorda o barbeiro. Se chega outro cliente enquanto o barbeiro está trabalhando, ele ocupa uma cadeira e espera (se tem alguma cadeira disponível) ou vai embora (se todas as cadeiras estão ocupadas).

A solução a seguir usa 3 semáforos: clientes, fila e mutex. O semáforo clientes tranca o barbeiro, sendo suas “bolitas” produzidas pelos clientes que chegam. O valor desse semáforo indica o número de clientes à espera (excluindo o cliente na cadeira do barbeiro, que não está à espera). O semáforo fila tranca os clientes e implementa a fila de espera. O semáforo mutex garante exclusão mútua. Também é usada uma variável inteira, count, que conta o número de clientes à espera. O valor desta variável é sempre igual ao “número de bolitas” do semáforo clientes.

Um cliente que chega na barbearia verifica o número de clientes à espera. Se esse número é menor que o número de cadeiras, o cliente espera, caso contrário, ele vai embora. A solução é apresentada a seguir, considerando o número de cadeiras na sala de espera igual a 3.

Inicialmente, o barbeiro executa a operação P(clientes), onde fica bloqueado (dormindo) até a chegada de algum cliente. Quando chega um cliente, ele começa adquirindo a exclusão mútua. Outro cliente que chegar imediatamente após, irá se bloquear até que o primeiro libere a exclusão mútua. Dentro da região crítica, o cliente verifica se o número de pessoas à espera é menor ou igual ao número de cadeiras. Se não é, ele libera mutex e vai embora sem cortar o cabelo.

Se tem alguma cadeira disponível, o cliente incrementa a variável count e executa a operação V no semáforo clientes. Se o barbeiro está dormindo, ele é acordado; caso contrário, é adicionada uma “bolita” no semáforo clientes. A seguir, o cliente libera a exclusão mútua e entra na fila de espera. O barbeiro adquire a exclusão mútua, decrementa o número de clientes, pega o primeiro da fila de espera e vai fazer o corte.

Variáveis globais:    clientes, fila: semaphore init 0;  
                          mutex: semaphore init 1;

```
count : integer initial 0;
```

Processo barbeiro:

Processo cliente:

```
loop
P(mutex);
P(clientes); /*dorme, se for o caso*/
if count < 3
P(mutex);
then { count:= count+1;
count:= count -1;
V(clientes); /*acorda o barbeiro*/
V(fila); /*pega próximo cliente*/
V(mutex);
V(mutex);
P(fila); /*espera o barbeiro*/
/*corta o cabelo*/
/*corta o cabelo*/
endloop
}

else V(mutex)
```

Quando termina o corte de cabelo, o cliente deixa a barbearia e o barbeiro repete o seu loop onde tenta pegar um próximo cliente. Se tem cliente, o barbeiro faz outro corte. Se não tem, o barbeiro dorme.

#### 4) Leitores e escritores

O problema dos *readers and writers* ilustra outra situação comum em sistemas de processos concorrentes. Este problema surge quando processos executam operações de leitura e de atualização sobre um arquivo global (ou sobre uma estrutura de dados global). A sincronização deve ser tal que vários readers (isto é, processos leitores, que não alteram a informação) possam utilizar o arquivo simultaneamente. Entretanto, qualquer processo writer deve ter acesso exclusivo ao arquivo.

Na solução a seguir é dada prioridade para os processos *readers*. São utilizadas duas variáveis semáforas, *mutex* e *w*, para exclusão mútua, e uma variável inteira *nr*, para contar o número de processos leitores ativos. Note que o primeiro reader bloqueia o progresso dos *writers* que chegam após ele, através do semáforo *w*. Enquanto houver reader ativo, os *writers* ficarão bloqueados.

Variáveis globais: *mutex*, *w* : semaphore initial 1; *nr* : integer initial 0;

Processo leitor:

```
...
P(mutex);
nr:=nr+1;
if nr=1 then P(w);
V(mutex);
READ
P(mutex);
nr:=nr-1;
if nr=0 then V(w);
V(mutex);
...
```

Processo escritor:

```
...
P(w);
WRITE
V(w);
...
```

## 5) Problema da Montanha Russa

Existem  $n$  passageiros, que repetidamente aguardam para entrar em um carrinho da montanha russa, fazem o passeio, e voltam a aguardar. Vários passageiros podem entrar no carrinho ao mesmo tempo, pois este tem várias portas. A montanha russa tem somente um carrinho, onde cabem  $C$  passageiros ( $C < n$ ). O carrinho só começa seu percurso se estiver lotado. Sincronize as ações dos processos Passageiro e Carrinho usando semáforos:.

Uma possível solução é mostrada abaixo:

```
semaphore    passageiro = C
semaphore    carrinho = 0
semaphore    andando = 0
semaphore    mutex = 1
int          Npass = 0
```

```
Passageiro() {
```

```
    while (true) {
        DOWN(passageiro)
        entra_no_carrinho() /* vários passageiros podem entrar "ao mesmo tempo" */
        DOWN(mutex)
        Npass++
        if (Npass == C) {           /* carrinho lotou */
            UP(carrinho)           /* autoriza carrinho a andar */
            DOWN(andando)         /* espera carrinho parar */
            UP(mutex)
        }
        else {
            UP(mutex)
            DOWN(andando)         /* espera carrinho lotar, passear e voltar */
        }
    }
}
```

```
Carrinho() {
```

```
    while (true){
        DOWN(carrinho)           /* espera autorização para andar */
        passeia()                /* faz o passeio e volta */
        Npass := 0               /* esvazia carrinho */
        for (int i=0; i<C; i++){
            UP(andando);         /* libera passageiro que andou de volta à fila */
        }
        UP(passageiro);         /* libera entrada no carrinho */
    }
}
```

## 6) Problema do Supermercado

Considere um supermercado com  $N$  caixas de pagamento com um empregado em cada caixa. Enquanto houver clientes na sua fila o empregado atende-os. Se não tiver nenhum cliente para ser atendido na sua fila, o empregado pode atender um cliente de outra fila. Se não estiver ninguém para atender em nenhuma das filas, o empregado bloqueia-se à espera de clientes. O cliente quando chega, escolhe a fila (ou uma das filas) que tiver menos clientes. Mas uma vez escolhida, o cliente não pode trocar de fila, excepto para ser atendido conforme descrito

anteriormente. O número de clientes por fila é ilimitado.

Implemente, usando semáforos (e, posteriormente, Monitor), em pseudo-código C, as rotinas `Empregado(int fila)` e `Cliente()`, que correspondem respectivamente às funções de empregado e cliente. Considere que ainda existem 2 rotinas - `Atender()` e `serAtendido()` - que são chamadas respectivamente pelas rotinas `Empregado()` e `Cliente()` quando o empregado está a atender o cliente e, por sua vez, o cliente está a ser atendido pelo empregado. A implementação das rotinas `Atender()` e `serAtendido()` não faz parte do exercício (considere-as uma caixa preta). Uma possível solução usando Semáforos é dada abaixo:

```
Semaphore filas [0..N-1] //inicializados em 0
Semaphore emp [0..N-1] //inicializados em 0
Semaphore mutex = 1
int cfilas [0..N-1] //inicializados em 0

Cliente (){
    down (mutex)
    int mf = 0;
    int count = cfilas[0];
    for (int i=1; i<N;i++)
        if(count==0) break; //otimização, pois não haverá fila menor que 0
        if (cfilas[i]<cfilas[mf]){
            mf=i;
            count=cfilas[i];
        }
    }
    up(emp[mf]);
    cfilas[mf]++;
    up(mutex);
    down(filas[mf]);
    serAtendido();
}
```

```
Empregado (int fila){
    for(;;){
        while (true){
            down(mutex);
            if (cfilas[fila]>0){
                cfilas[fila]--;
                up(filas[fila]);
                up(mutex);
                down(emp[fila]);
                ATENDER();
            } else {
                up(mutex);
                break;
            }
        }
        int nf, n-vazias=0;
        for (int j=0; j<N-1;j++){
            nf=next(fila,j);//pega a pos. da prox. fila
            down(mutex);
            if(cfila[nf]>0)
                n-vazias=0;
            cfila[nf]--;
        }
    }
}
```

```

        up(mutex);
        up(fila[nf]);
        down(emp[nf]);
        ATENDE();
    } else{
        up(mutex);
        n-vazias++
    }
} //for(int j...)
down(mutex)
if (n-vazias==N && cfilas(fila)==0){
    up(mutex);
    down(emp[filas]);
} else
    up(mutex);
}
}
}

```

## 7) Vida de Hoare

Em um determinado stand de uma feira, um demonstrador apresenta um filme sobre a vida de Hoare. Quando 10 pessoas chegam, o demonstrador fecha o pequeno auditório que não comporta mais do que essa platéia. Novos candidatos a assistirem o filme devem esperar a próxima exibição. Esse filme faz muito sucesso com um grupo grande de fãs (de bem mais de 10 pessoas), que permanecem na feira só assistindo o filme seguidas vezes. Cada vez que um desses fãs consegue assistir uma vez o filme, ele vai telefonar para casa para contar alguns detalhes novos para sua mãe. Depois de telefonar ele volta mais uma vez ao stand para assistir o filme outra vez.

Usando semáforos, modele o processo fã e o processo demonstrador, lembrando que existem muitos fãs e apenas um demonstrador. Como cada fã é muito ardoroso, uma vez que ele chega ao stand ele não sai dali até assistir o filme. Suponha que haja muitos telefones disponíveis na feira e, portanto, que a tarefa de telefonar para casa não impõe nenhuma necessidade de sincronização.” OBS: Observe que o demonstrador só pode começar a exibir o filme quando há 10 pessoas no stand, e que as pessoas que chegam durante uma exibição têm que esperar a próxima. Importante: observe que um fã só pode ir telefonar para a mãe depois que acaba a exibição do filme! Isso tem que estar modelado na sincronização entre os processos demonstrador e fãs.”

```

#define N 10

int nFans=0;
semaphore mutex = 1;
semaphore dem = 0;
semaphore fila = 0;

fan(){
    while(true){
        P(mutex);
        nFans++;
        V(mutex);
        V(dem);
        P(fila);
        assisteFilme();
        telefona();
    }
}

```

```

demonstrator (){
    while(true){
        while (nFans<N)
            P(dem);
        P(mutex);
        nFans=nFans-N;
        V(mutex);
        for (i=0;i<N; i++)
            V(filas);
        exhibeFilme();
    }
}

```

## 8) Jantar dos Canibais

Suponha que um grupo de N canibais come jantares a partir de uma grande travessa que comporta M porções. Quando alguém quer comer, ele(ela) se serve da travessa, a menos que ela esteja vazia. Se a travessa está vazia, o canibal acorda o cozinheiro e espera até que o cozinheiro coloque mais M porções na travessa.

Desenvolva o código para as ações dos canibais e do cozinheiro. A solução deve evitar deadlock e deve acordar o cozinheiro apenas quando a travessa estiver vazia. Suponha um longo jantar, onde cada canibal continuamente se serve e come, sem se preocupar com as demais coisas na vida de um canibal...

```

semaphorecozinha = 0
semaphorecomida = M+1
semaphoremutex = 1
semaphoreenchendo = 0
int count= 0

```

### Canibal

```

While (1) {
    P(comida)
    P(mutex)
    count++
    if (count > M) then
        V(cozinha)
        P(enchendo)
        count=1
    come(); V(mutex);
}

```

### Cozinheiro

```

While (1) {
    P(cozinha)
    enche_travessa()
    for (int i=1; i ≤ M; i++)
        V(comida);
    V(enchendo) count=1
}

```

#### Problemas:

- 1) acesso serial se fosse come() e depois V(mutex)
- 2) Se M canibais perdem a posse antes do come() o M\_ésimo+1 acorda o cozinheiro para colocar mais porções na travessa que ainda está cheia.

## 9) Problema do Pombo

Considere a seguinte situação. Um pombo correio leva mensagens entre os sites A e B, mas só quando o número de mensagens acumuladas chega a 20. Inicialmente, o pombo fica em A, esperando que existam 20 mensagens para carregar, e dormindo enquanto não houver. Quando as



mensagens chegam a 20, o pombo deve levar exatamente (nenhuma a mais nem a menos) 20 mensagens de A para B, e em seguida voltar para A.

Caso existam outras 20 mensagens, ele parte imediatamente; caso contrário, ele dorme de novo até que existam as 20 mensagens. As mensagens são escritas em um post-it pelos usuários; cada usuário, quando tem uma mensagem pronta, cola sua mensagem na mochila do pombo. Caso o pombo tenha partido, ele deve esperar o seu retorno p/ colar a mensagem na mochila. O vigésimo usuário deve acordar o pombo caso ele esteja dormindo. Cada usuário tem seu bloquinho inesgotável de post-it e continuamente prepara uma mensagem e a leva ao pombo.

Usando semáforos, modele o processo pombo e o processo usuário, lembrando que existem muitos usuários e apenas um pombo. Identifique regiões críticas na vida do usuário e do pombo.

```
#define N=20
```

```
int contaPostIt=0;
semaforo mutex=1;           //controlar acesso à variável contaPostIt
semaforo cheia=0;          //usado para fazer o pombo dormir enquanto ã há 20 msg
semaforo enchendo=N;       //usado p/ fazer usuários dormirem enquanto pombo faz o transporte
```

```
usuario() {
    while(true){
        down(enchendo);
        down(mutex);
        colaPostIt_na_mochila();
        contaPostIt++;
        if (contaPostIt == N)
            up(cheia);
        up(mutex);
    }
}
```

```
pombo() {
    while(true){
        down(cheia);
        down(mutex);
        leva_mochila_ate_B_e_volta();
        contaPostIt=0;
        for (i=0; i<N; i++)
            up(enchendo);
        up(mutex);
    }
}
```