SISTEMAS OPERATIVOS

Sincronização entre processos

António Godinho

PROBLEMA DA SINCRONIZAÇÃO

Nesta parte da matéria:

- Conceitos sobre sincronização. Problemas de sincronização: como os identificar. Secção crítica
- Conceito de exclusão mútua. Implementação da exclusão mútua a baixo nível
- Conceito de semáforo (usado como mecanismo genérico para resolver problemas de sincronização)
- Conceito de Mutex

Nota: o API de semáforos em Unix não é abordado.

A conseguir nesta parte a matéria:

- Saber identificar problemas de sincronização (exclusão mútua)
- Saber resolver a nível abstracto com recurso a semáforos/mutexes
- Saber usar o API de mutexes em Unix

Importante: Este documento é um resumo. É importante ver a bibliogafia

SINCRONIZAÇÃO

Motivos para sincronização

1. Num sistema no qual existam em execução simultânea várias entidades (processos e threads) que utilizem recursos e dados partilhados entre si exige a coordenação do acesso a esses mesmos recursos ou dados
2. Qualquer sistema que seja composto por mais do que uma entidade activa em que uma das entidades seja dependente de acontecimentos originados por outra(s) dessas entidades exige a coordenação da sua execução em função da execução dessas outras entidades

Exemplos típicos da bibliografia

- Gestores de recursos (memória partilhada), gestão de produtores/consumidores Exemplos concretos da vida real

- Sistemas gestores de dados, sistemas de utilização simultânea por vários utilizadores (ex.: sistemas de vendas de bilhetes; maquinas ATM)

SINCRONIZAÇÃO

Situações típicas (exemplos) que envolvem sincronização

Cooperação

Diversas actividades concorrem para a conclusão de uma aplicação comum

Competição

Diversos processos competem pela obtenção de um recurso limitado

A competição deve ser resolvida de forma a que o recurso seja utilizado de forma coerente

Exclusão mútua

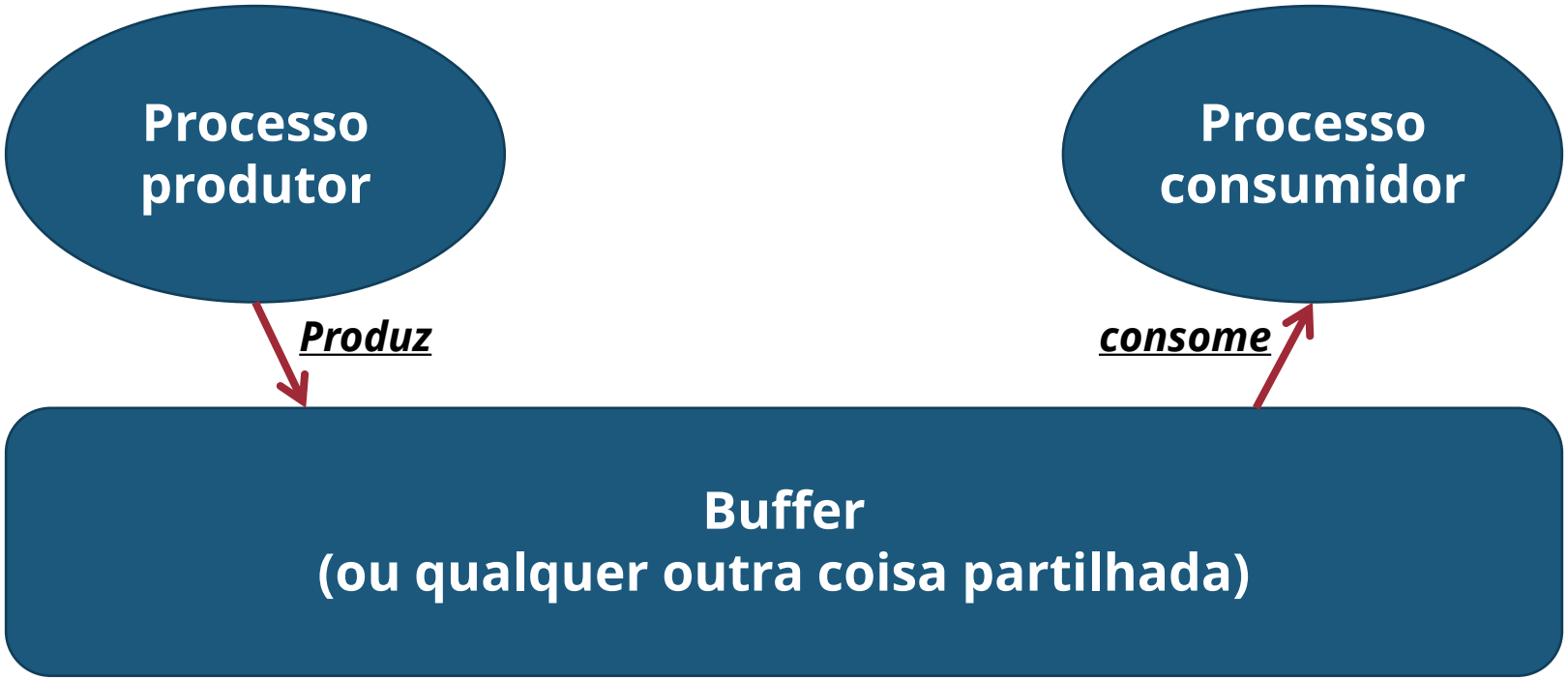
A utilização concorrente de uma zona de dados partilhada pode levar a que os dados fiquem inconsistentes

A utilização deve ser feita de uma forma exclusiva: apenas uma entidade activa utiliza o recurso.

A execução de uma secção de código que manipula dados partilhados constitui uma situação típica de acesso em exclusão mútua

PROBLEMA DO PRODUTOR-CONSUMIDOR

- **Passamos de um problema da"vida real“ para um problema clássico**
- **Representativo de muitas funções dos SOs:**
  - **um servidor Web produz (fornece) páginas que um cliente consome (lê)**



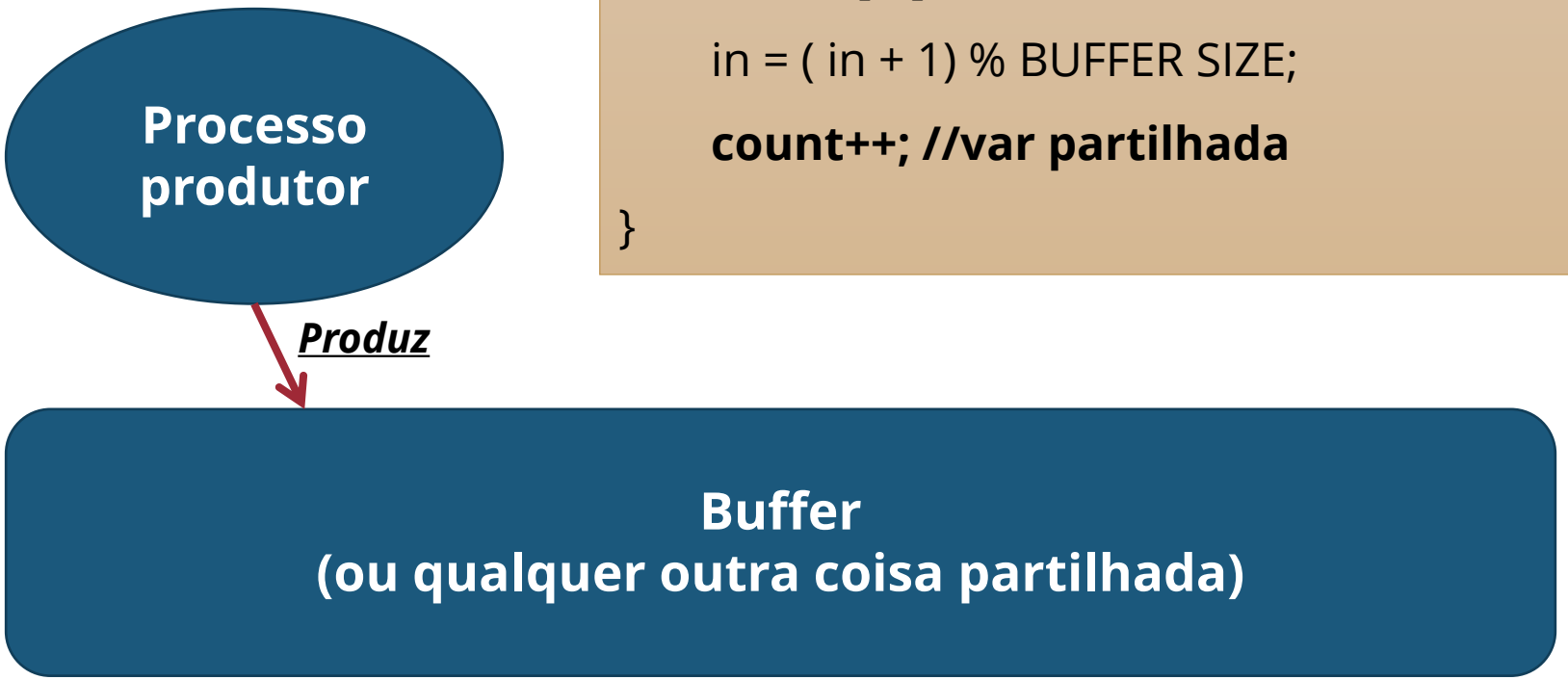
BUFFER PARTILHADO

```
#define BUFFER_SIZE 10


typedef struct {
        ...
} item;
item buffer[BUFFER_SIZE];
int in=0;
int out=0;
```

Buffer
(ou qualquer outra coisa partilhada)

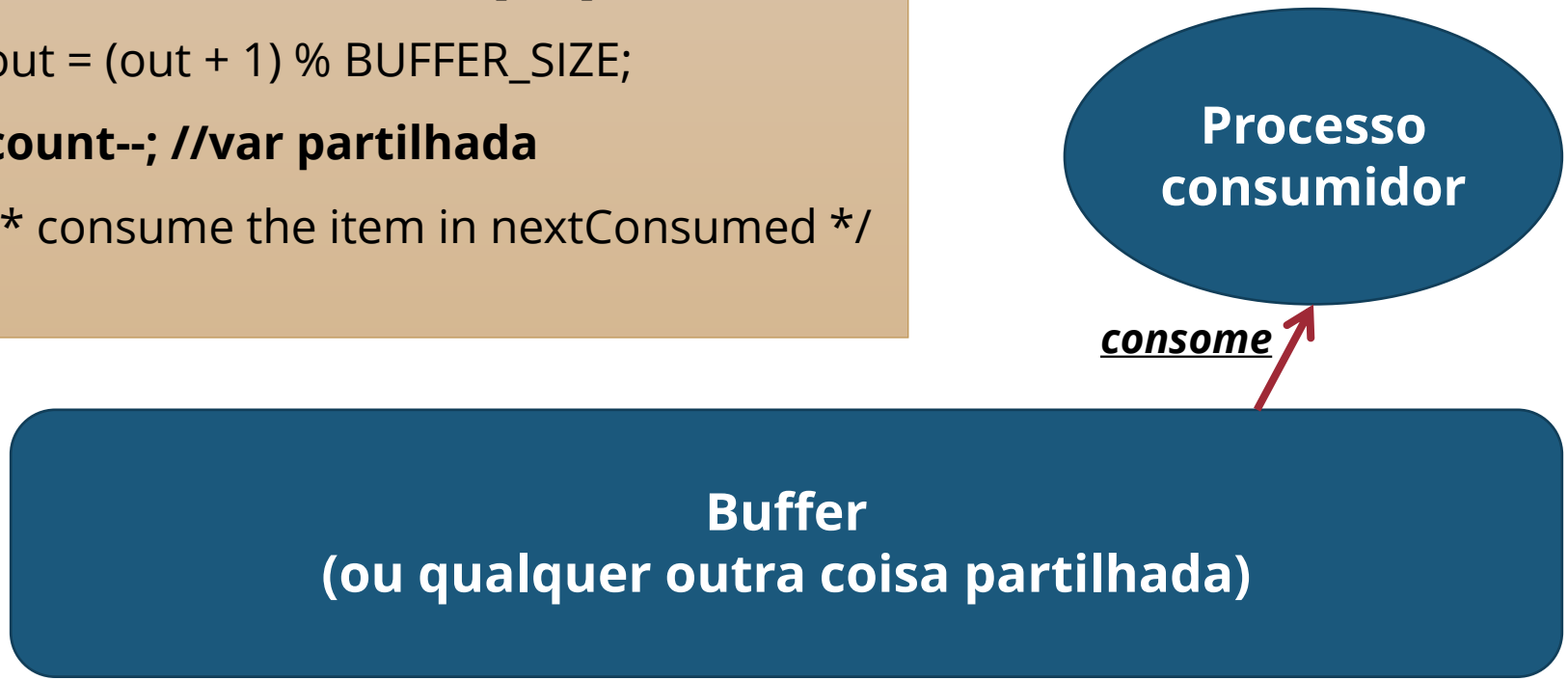
PROCESSO PRODUTOR

```
while (true) {
        /* produce an item and put in nextProduced */
        while (count == BUFFER_SIZE); // do nothing
        buffer [in] = nextProduced;
        in = ( in + 1) % BUFFER SIZE;
        count++; //var partilhada
}
```



PROCESSO CONSUMIDOR

```
while (true) {
        while (count == 0)
                ; // do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        count--; //var partilhada
        /* consume the item in nextConsumed */
}
```



PROBLEMA: RACE CONDITION

- **Imagine que** count = 5 **e que o produtor e o consumidor executam** count++ **e** count–- **de forma concorrente**
  - **o resultado final deveria ser** count = 5
  - **na prática, pode não ser!**
- **Porquê? Estes dois comandos podem ser implementados em linguagem máquina desta forma:**

count++

```
register1 = count
register1 = register1 + 1
count = register1
```

count--

```
register2 = count
register2 = register2 - 1
count = register2
```

- Podendo despoletar uma race condition:

t0 (produtor):	register1 = count	{register1 = 5}
t1 (produtor):	register1 = register1 + 1	{register1 = 6}
t2 (consumidor):	register2 = count	{register2 = 5}
t3 (consumidor):	register2 = register2 – 1	{register2 = 4}
t4 (produtor):	count = register1	{count = 6 }
t5 (consumidor):	count = register2	{ count = 4}

RAIZ DO PROBLEMA E SOLUÇÃO

- **Vários processos a manipular uma variável partilhada de forma concorrente**

- **Como resolver?**
  - **temos que garantir que apenas um processo de cada vez manipula a variável partilhada (**counter)
    - **i.e., só um de cada vez é que vai comprar leite!**

- **Situação muito comum em sistemas operativos**
  - **e com o surgimento de sistemas multicore e com cada vez mais aplicações a usar múltiples** threads **a situação tende a ser cada vez mais complexa**

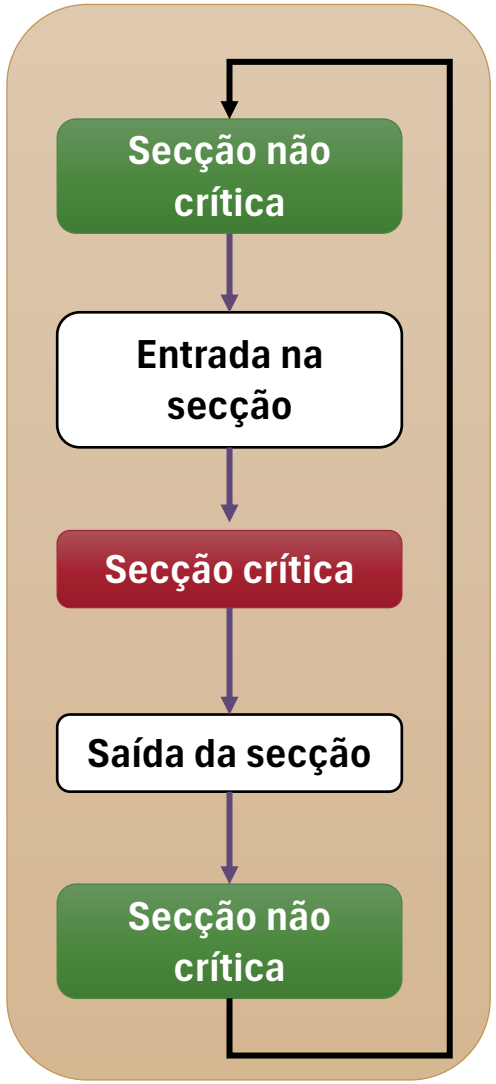
PROBLEMA DA SECÇÃO CRÍTICA

```
while (TRUE) {
        \\secção de entrada

        SECÇÃO CRÍTICA

        \\secção de saída

        CÓDIGO RESTANTE
}
```

- **Quando um processo está na secção crítica, mais nenhum processo pode estar na secção crítica**
- **Os processos têm de pedir permissão para poderem entrar na secção crítica.**

SECÇÃO CRÍTICA

- Na partilha de dados por vários processos, cada processo tem uma sequência de código, designado por secção crítica, onde são acedidos os dados partilhados.
- Deve ser assegurado que, enquanto um processo está a executar na secção crítica, a nenhum outro processo deve ser permitido executar na sua secção crítica. Para o efeito deve ser desenvolvido um mecanismo que permita a sincronização e coordenação entre processos cooperantes de modo a evitar as condições de corrida (race conditions), mantendo a consistência dos dados.
- Cada processo deve "solicitar" a entrada na secção crítica, executar a secção crítica em modo exclusivo e abandonar a secção crítica, garantindo que outros processos possam aceder à secção crítica.



SECÇÃO CRÍTICA: CONDIÇÕES DE COFFMAN

- **Exclusão mútua**
  - **se um processo está a executar na secção crítica, mais nenhum processo pode estar a executar nessa secção**
- **Manter e esperar ou manutenção de recursos**
  - **um processo está atualmente a manter pelo menos um recurso e a solicitar recursos adicionais que estão a ser mantidos por outros processos.**
- **Sem preempção**
  - **um recurso só pode ser libertado voluntariamente pelo processo que o detém.**
- **Espera circular**
  - **cada processo deve estar à espera de um recurso que está a ser retido por outro processo, o qual, por sua vez, está à espera que o primeiro processo liberte o recurso.**

SECÇÃO CRÍTICA: CONDIÇÕES DE COFFMAN

- **Tipo de soluções:**
  1. Software: mecanismos genéricos de uma linguagem de programação (soluções algorítmicas).
  2. Hardware: mecanismos disponibilizados pela CPU.
  3. Recursos do sistema operativo: semáforos, passagem de mensagens, etc.
  4. Monitores: mecanismo específico de uma linguagem de programação ou biblioteca

MUTEX LOCK

```
while (TRUE) {
    acquire lock
    SECÇÃO CRÍTICA
    release lock
    CÓDIGO RESTANTE
}
```

- **Solução simples (em software) para resolver o problema da secção crítica**
  - Mutex = **mut**ual **ex**clusion

- **Quando queremos aceder a uma região crítica (e assim prevenir** race conditions**) fazemos lock** (acquire**) do mutex**
- **Ao sair da região crítica fazemos unlock** (release**) do mutex**

IMPLEMENTAÇÃO

```
acquire_lock() {
        while (available == false); //espera ativa
        available = false;
}
```

```
release_lock() {
    available = true;
}
```

- **Se o** mutex
  - **está disponível, a chamada à função** acquire() **executa com sucesso e o** lock **fica indisponível**
  - **está indisponível, a chamada à função bloqueia até este ser liberto**
- **Estas funções têm de executar atomicamente**
  - **isto pode ser garantido por hardware**
- **Problema desta implementação: espera ativa**

  enquanto se espera pelo mutex **fica-se continuamente num** loop**, gastando recursos do CPU**

ALGORITMO DE DEKKER (V.1)

- Alternância estrita

```
int vez=1; /* dados partilhados */
```

Processo 1

```
...
while (vez == 2);      /*testa acesso*/
/*<secção crítica>*/
vez = 2;               /*autoriza acesso ao outro processo*/
...
```

Processo 2

```
...
while (vez == 1);      /*testa acesso*/
/*<secção crítica>*/
vez = 1;               /*autoriza acesso ao outro processo*/
...
```

- Análise da solução:
  - Apenas aplicável a dois processos.
  - Garante a exclusão mútua entre dois processos.
  - Não garante a progressão. Não permite que um processo execute a secção crítica duas vezes consecutivas.
  - Apenas garante a alternância entre os dois processos no acesso à secção crítica. O desempenho de um processo condiciona o desempenho do outro (lockstep synchronization).
  - Não garante a espera limitada. Se um dos processos falha, o outro fica permanentemente bloqueado.
  - O teste while (vez==..) não produz qualquer trabalho útil . Consome tempo do processador (espera activa).

ALGORITMO DE DEKKER (V.2)

- Alternância estrita

```
int P1Dentro=false; /* dados partilhados */
int P2Dentro=false;
```

Processo 1

```
while (P2Dentro); /*P2 na SC ?*/
P1Dentro = True;
/*<executa secção crítica>*/
P1Dentro = False; /*autoriza acesso ao outro processo*/
...
```

Processo 2

```
while (P1Dentro); /*P1 na SC ?*/
P2Dentro = True;
/*<executa secção crítica>*/
P2Dentro = False; /*autoriza acesso ao outro processo*/
...
```

- Análise da solução:
  - Apenas aplicável a dois processos.
  - Garante a progressão. Não obriga à alternância estrita no acesso à secção crítica.
  - Não garante a exclusão mútua.
  - Não garante a espera limitada.
  - Espera activa.

ALGORITMO DE DEKKER (V.3)

```
int P1QuerEntrar=false; /* dados partilhados */
int P2QuerEntrar=false;
```

Processo 1

```
P1QuerEntrar = True;
while (P2QuerEntrar);   /*P2 quer entrar na SC ?*/
/*<executa secção crítica>*/
P1QuerEntrar = False;   /*autoriza acesso ao outro processo*/
...
```

Processo 2

```
P2QuerEntrar = True;
while (P1QuerEntrar);   /*P1 quer entrar na SC ?*/
/*<executa secção crítica>*/
P2QuerEntrar = False;   /*autoriza acesso ao outro processo*/
...
```

- Análise da solução:
  - Apenas aplicável a dois processos.
  - Garante a exclusão mútua entre dois processos.
  - Garante a progressão. Não obriga à alternância estrita no acesso à secção crítica.
  - Não garante a espera limitada.
  - Introduz a interblocagem.
  - Espera activa.

ALGORITMO DE DEKKER (V. FINAL)

- Análise da solução:
  - Apenas aplicável a dois processos.
  - Garante a exclusão mútua entre dois processos.
  - Garante a progressão. Não obriga à alternância estrita no acesso à secção crítica.
  - Garante a espera limitada.
  - Espera activa.

```
int P1QuerEntrar=false; /* dados partilhados */
int P2QuerEntrar=false;
int vez = 1;
```

Processo 1

```
P1QuerEntrar = true;     /*P1 quer entrar na SC*/
while (P2QuerEntrar)     /*mas dá a vez a P2, se ele precisar*/
     if (vez == 2) then
     {P1QuerEntrar = false;
      while (vez == 2);        /*testa acesso*/
      P1QuerEntrar = true;};
/*<executa secção crítica>*/
vez = 2;                 /*autoriza acesso a P2*/
P1QuerEntrar = false;
...
```

Processo 2

```
P2QuerEntrar = true;     /*P2 quer entrar na SC*/
while (P1QuerEntrar)     /*mas dá a vez a P1, se ele precisar*/
     if (vez == 1) then
     {P2QuerEntrar = false;
      while (vez == 1);        /*testa acesso*/
      P2QuerEntrar = true;};
/*<executa secção crítica>*/
vez = 1;                 /*autoriza acesso a P1*/
P2QuerEntrar = false;
...
```

ALGORITMO DE PETERSON

```
int P1QuerEntrar=false; /* dados partilhados */
int P2QuerEntrar=false;
int vez = 1;
```

Processo 1

```
P1QuerEntrar = true;    /*P1 quer entrar na SC*/
vez=2;
while (P2QuerEntrar && vez==2); /*testa acesso*/
/*<executa secção crítica>*/
P1QuerEntrar = false;
...
```

Processo 2

```
P2QuerEntrar = true;    /*P2 quer entrar na SC*/
vez=1;
while (P1QuerEntrar && vez==1); /*testa acesso*/
/*<executa secção crítica>*/
P2QuerEntrar = false;
...
```

- Análise da solução:
  - Apenas aplicável a dois processos.
  - Garante a exclusão mútua entre dois processos.
  - Garante a progressão. Não obriga à alternância estrita no acesso à secção crítica.
  - Garante a espera limitada.
  - Espera activa.
  - Mais simples que a solução de Dekker

ALGORITMO DE LAMPORT-N PROCESSOS (1)

- Descrição:
  - Cria uma fila de processos à espera de entrar na secção crítica através do uso de senhas.
  - Antes de entrar na secção crítica, cada processo obtém uma senha.
  - Depois de atribuída a senha (número de ordem), o processo efectua um ciclo para determinar se o seu número de ordem é o menor de todos.
  - Quando o número de ordem (senha) for inferior ao de todos os outros, o processo entra na secção crítica.
  - Garante a exclusão mútua entre N processos.
  - Garante a progressão.
  - Garante a espera limitada.
  - Espera activa.

ALGORITMO DE LAMPORT-N PROCESSOS (2)

- Análise do algoritmo:
  - Apenas aplicável a dois processos.
  - Devido à execução concorrente, a função maxSenha() pode devolver o mesmo valor para dois processos. Em caso de empate, utiliza-se o número do processo para determinar quem terá acesso à secção crítica (processo com menor índice).
  - Garante a exclusão mútua entre N processos.
  - Garante a progressão.
  - Garante a espera limitada. Todos os processos à espera de entrar na secção crítica são admitidos pela ordem de chegada (FIFO).
  - Espera activa.

```
/* dados partilhados */
boolean escolha[N]; /*Processo i pediu senha*/
int senha[N]; /*Senha do processo i*/
```

Processo i

```
/*i = número do processo */
escolha[i]=true; /*pede senha*/
senha[i]=maxSenha()+1; /* devolve o valor máximo no array senha[]*/
escolha[i]=false; /*fim pede senha*/
for (j=0;j<N;j++){/*aguarda a sua vez*/
  if (j!=i) {
   while (escolha[j]!=false); /*espera que processo j tenha senha*/
   while (senha[j]!=0 && senha[j]<senha[i]);
   if (senha[j]==senha[i] && j < i)
      while (senha[j]!=0);
  }
}
/*<executa secção crítica>*/
senha[i]= 0;
...
```

EXCLUSÃO MÚTUA – SECÇÃO CRÍTICA - TRINCOS LÓGICOS

- As soluções anteriores são complexas para o programador que deseja escrever aplicações concorrentes e garantir a exclusão mútua.
- Uma solução possível passa pela utilização de uma variável que funciona como um trinco lógico de uma estrutura de dados partilhada. O trinco é fechado quando se acede à estrutura de dados e aberto à saída. O trinco lógico é manipulado por duas primitivas: lock e unlock. Ambas as primitivas são executadas de forma indivisível (atómicas), isto é, o processo não está sujeito à preempção aquando da execução destas funções.
- Um processo que tenta aceder a uma secção crítica protegida por um trinco ficará em ciclo até o trinco ser libertado (espera activa).

```
void lock(boolean *trinco){
 /* inst. indivisíveis */
   while (*trinco);
   *trinco = true;
}
void unlock(boolean *trinco){
   *trinco = false;
}
```

```
boolean trinco=false; /*variável partilhada
pelos processos concorrentes*/

 ...
lock(&trinco);
/*<acede a dados partilhados em modo
exclusivo>*/
unlock(&trinco); /* ou apenas trinco=false*/
...
```

- Garante a exclusão mútua para sistemas monoprocessador. O mesmo não acontece num sistema multiprocessador, pois 2 processos (em processadores diferentes) podem ver o trinco a FALSE e entrar na secção crítica.
- Não garante a espera limitada. Nada impede que o mesmo processo, por razões de escalonamento, consiga consecutivamente obter o trinco lógico, não permitindo que outros processos acedam à secção crítica.

EXCLUSÃO MÚTUA: TRINCOS LÓGICOS POR HARDWARE

- Para garantir a atomicidade das primitivas lock e unlock ou da própria secção crítica, é possível recorrer:
  - à inibição das interrupções no início da secção crítica. Esta solução não é muito utilizada por colocar em risco o normal funcionamento do SO.
  - à utilização de instruções especiais do processador que são executadas num único ciclo de instrução (indivisibilidade/atomicidade):
    - Instrução Test-And-Set, também designada por Read-Modify-Write (RMW) que lê, modifica e escreve o valor de uma variável numa única instrução.
    - Instrução Swap que troca o valor de duas variáveis numa única instrução.

EXCLUSÃO MÚTUA: INSTRUÇÃO TESTANDSET

```
boolean TestAndSet(boolean *trinco){
 /* inst. indivisíveis */
   boolean rv=*trinco;
   *trinco=true;
   return rv;
}
```

```
boolean trinco=false; /*variável
partilhada pelos processos concorrentes*/

 ...
While TestAndSet(&trinco);
/*<acede a dados partilhados em modo
exclusivo>*/
trinco=false;
...
```

- Análise da solução:
  - Aquando da execução da instrução TestAndSet, se a variável trinco é verdadeira, permanece verdadeira e o processo mantém-se no ciclo While (valor verdadeiro devolvido pela instrução). Se a variável trinco é falsa, esta é colocada a verdadeiro e o valor devolvido é falso, terminando o ciclo While e entrando na secção crítica.
  - Garante a exclusão mútua.
  - Garante a progressão.
  - **Não garante a espera limitada.**
  - Espera activa.

EXCLUSÃO MÚTUA: INSTRUÇÃO SWAP

```
void Swap(boolean *a, boolean *b){
 /* inst. indivisíveis */
   boolean temp=*a;
   *a=*b;
   *b=temp;;
}
```

```
boolean trinco=false; /*variável
partilhada pelos processos concorrentes*/
boolean chave; /*variável local a cada
processo*/
 ...
chave=true;
While (chave)
    Swap(&trinco,&chave);
/*<acede a dados partilhados em modo
exclusivo>*/
trinco=false;
```

- Análise da solução:
  - Aquando da execução da instrução Swap, são trocados os valores das variáveis chave e trinco, durante um único ciclo de instrução. O processo entra na secção crítica apenas quando consegue trocar o valor verdadeiro da chave com o valor falso de trinco (não existe processo a executar a secção crítica).
  - Garante a exclusão mútua.
  - Garante a progressão.
  - **Não garante a espera limitada.**
  - Espera activa.

EXCLUSÃO MÚTUA E ESPERA LIMITADA : TESTANDSET

- Análise da solução:
  - No fim de um processo aceder à secção crítica, verifica se existe outro processo à espera. Se sim, não liberta o trinco e permite que o referido processo entre na secção crítica. Se não existir nenhum processo à espera, o trinco é libertado.
  - Garante a exclusão mútua.
  - Garante a progressão.
  - Garante a espera limitada.
  - Espera activa

```
/*variáveis partilhadas pelos processos concorrentes*/
boolean trinco=false;
boolean espera[N];

/*variável local a cada processo*/
boolean chave;

…
/* processo i */
espera[i]=true;
chave=true;
While (espera[i] && chave)
    chave=TestAndSet(&trinco);
espera[i]=false;
/*<acede a dados partilhados em modo exclusivo>*/
j= (i+1) % N;
While ((j !=i) && espera[j])
   j=(j+1) % N;
If (j==i)
   trinco=false; /*nenhum processo à espera, liberta o trinco */
 else
   espera[j]=false; /*processo j à espera, não liberta o trinco */
...
```

SEMÁFOROS (RECURSO DO SO)

▪Espera activa: acontece quando um processo que está na secção crítica é retirado do processador (preempção). O trinco mantém-se fechado e os restantes processos são escalonados para utilizarem o processador, continuando a testar o trinco à espera da sua libertação. As soluções que recorrem a um mecanismo de espera activa, fazem uso do processador de uma forma ineficiente.

▪Semáforo: recurso do SO que disponibiliza um mecanismo de sincronização sem espera activa. Para evitar a espera activa, um processo que espera a libertação dum recurso (ex. trinco) é bloqueado. Apenas quando o recurso é libertado, o processo passa para a fila de processos prontos a executar.

▪Propostos em 1965 por Dijkstra.

▪Implementação:

```
typedef struct {
 int valor;     /* valor do semáforo */
 struct process *fila; /* fila dos processos bloqueados no semáforo */
} semaforo;
```

SINCRONIZAÇÃO – EXCLUSÃO MÚTUA - SEMÁFOROS

Solução ideal para garantir o acesso em exclusão mútua a uma secção crítica e dados

▪Manter a indivisibilidade do teste-decisão-acção e acrescentar:

Eliminação da espera activa:

O processo deve ficar bloqueado em vez de estar em ciclo de teste

Quando puder avançar é acordado e ter a garantia que pode avançar sem testar mais nada

Problema

O SO é a única entidade com poderes garantidos para bloquear ou acordar qualquer processo

O SO não sabe que um processo está À espera de poder avançar. Um clico de teste são apenas mais umas instruções como tantas outras

Solução:

▪Transformar a tentativa de acesso a uma zona de código (dados) numa espera de um recurso controlado pelo SO (para o SO saber que o processo está à espera e quando pode avançar)

▪O SO bloqueará o processo se esse recurso não estiver disponível e desbloqueá-lo-á quanto voltar a estar disponível

▪Este mecanismo deve ser muito simples de usar e genérico

▪Existem vários recursos para esse efeito: o mais genérico é o Semáforo

SEMÁFOROS (RECURSO DO SO)

Operações realizadas atomicamente (indivisíveis):

- esperar(S)/wait(S);
- assinalar(S)/signal(S).

```
espera(semaforo *S){
S->valor--;
if (S->valor < 0) {
 /* adiciona processo à fila S->fila */
 Bloquear(); }
}
```

```
assinala(semaforo *S){
S->valor++;
if (S->valor <=0) /*Processo à espera?*/
   { /* Retira processo P da fila*/
   Desbloquear(S->fila); }
}
```

- Bloquear(): significa retirar o processo em execução (running) e transferi-lo para a fila de processos bloqueados (waiting) do respectivo semáforo.
- Desbloquear(P): significa retirar um processo, de acordo com um algoritmo FIFO ou outro, da fila de processos bloqueados e transferi-lo para a fila de processos prontos a executar.

SINCRONIZAÇÃO – APLICAÇÕES POSSÍVEIS

Um semáforo S é criado com um valor inicial. Os casos mais comuns são:

- S.valor=0 : Cooperação entre processos por sincronização indirecta. Um dos processos deve assinalar o semáforo para que outro não fique bloqueado.

- S.valor=1 : Exclusão mútua no acesso a uma secção crítica. O primeiro processo a chegar ao semáforo tem acesso à secção crítica.

- S.valor >=1 : Competição por recurso de capacidade limitada (Nota: A exclusão mútua é um caso particular da competição por recurso capacidade =1).

SEMÁFOROS – EXCLUSÃO MÚTUA

- O valor do semáforo é inicializado a 1. O primeiro processo a executar a função espera() não bloqueia (semáforo=0) e pode entrar na secção crítica. Enquanto este estiver na secção crítica, outros processos que invoquem a função espera() ficam bloqueados (semáforo<0). Enquanto o semáforo tiver um valor negativo (<0) existem processos à espera. No final da secção crítica, é feito o assinalar() que incrementa o semáforo e verifica se existem processos à espera, desbloqueando um processo da fila.

- Garante exclusão mútua entre N processos.

- Garante a progressão.

- Garante espera limitada, embora dependa do algoritmo de gestão da fila de processos bloqueados no semáforo (ex. FIFO).

SEMÁFOROS – SINCRONIZAÇÃO ENTRE PROCESSOS

- Exemplo de cooperação entre processos: Execução da secção de código B no processo P_j apenas depois da secção A do processo P_i

```
semaforo flag; /*dados partilhados pelos dois processos*/
flag.valor=0; /* variável do semáforo inicializada a 0*/
```

```
/* processo Pi */
/* <secção A> */
...
assinala(&flag);
...
```

```
/* processo Pj */
...
espera(&flag);
/* <secção B> */
...
```

- O valor do semáforo é inicializado a 0. Quando o processo P_j executa a função espera(), este fica bloqueado até que seja executada a função assinala() no processo P_i

SEMÁFOROS – DEADLOCK E STARVATION

Interbloqueio (deadlock – será visto em maior detalhe noutro capítulo): dois ou mais processos estão indefinidamente bloqueados à espera de um evento que apenas pode ser causado por um dos processos que se encontra bloqueado.

```
semaforo S,Q; /*dados partilhados pelos dois processos*/
S.valor=1; /*variável dos semáforos inicializada a 1*/
Q.valor=1;
```

```
/* processo P0 */
...
espera(&S);
espera(&Q);
...
assinala(&S);
assinala(&Q);
```

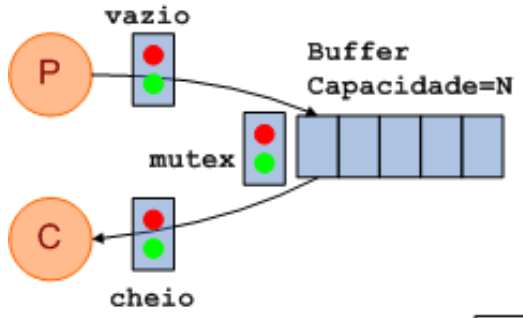
```
/* processo P1 */
...
espera(&Q);
espera(&S);
...
assinala(&Q);
assinala(&S);
```

- P0 executa espera(&S) e P1 executa espera(&Q) Quando P0 executa espera(&Q), este fica bloqueado até que P 1 execute assinala(&Q) Quando P1 executa espera(&S), este fica bloqueado à espera que P 1 execute assinala(&S) Ambos os processos ficam bloqueados indefinidamente
- Fome (starvation)

  dependendo da forma como os processos são desbloqueados da fila de espera de um semáforo, pode dar se o caso de privação, ou seja, o processo fica indefinidamente bloqueado (ex Algoritmo Last In First Out ou baseado em prioridades)

SEMÁFOROS – PRODUTOR-CONSUMIDOR

- Exemplo clássico de sincronização entre processos. Um processo produtor coloca itens num buffer de capacidade limitada. Um consumidor retira os itens do buffer.
- Requisitos:
  - Controlo de acesso a recurso com capacidade limitada;
  - Acesso em modo exclusivo ao buffer por este ser partilhado pelos dois processos;
  - O processo produtor deve atender à capacidade máxima do buffer;
  - O processo consumidor deve atender à existência de itens no buffer;



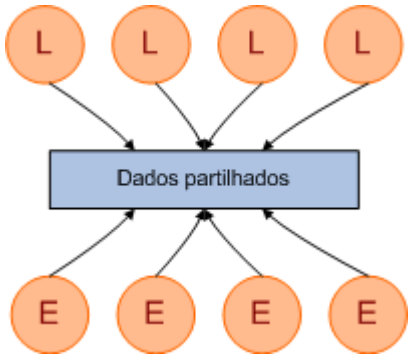

```
/*semáforos*/
semaforo mutex; /*exclusão mútua*/
semaforo cheio; /*nº de posições ocupadas no buffer*/
semaforo vazio; /*nº de posições livres no buffer*/

cheio.valor=0;
vazio.valor=N; /*nº de posições do buffer*/
mutex.valor=1;
```

```
/*Produtor*/
...
espera(&vazio);
espera(&mutex);
...
/* coloca item no buffer */
...
assinala(&mutex);
assinala(&cheio);
...
```

```
/*Consumidor*/
...
espera(&cheio);
espera(&mutex);
...
/* retira item do buffer */
...
assinala(&mutex);
assinala(&vazio);
...
```

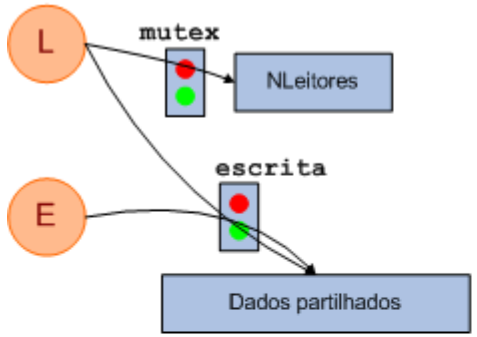
SEMÁFOROS – LEITORES-ESCRITORES (1)

- Exemplo clássico de sincronização entre processos:
  - Os dados são partilhados por vários processos;
  - Qualquer número de processos leitores pode aceder aos dados partilhados de forma concorrente;
  - Apenas um processo escritor pode aceder aos dados (exclusão mútua). Se um processo escritor aceder à secção crítica, nenhum outro processo, escritor ou leitor, poderá aceder;
- Soluções possíveis:
  - Solução 1: acesso preferencial aos leitores. Se um leitor está na secção crítica, outros processos leitores podem aceder, ainda que existam processos escritores à espera.
  - Solução 2: acesso preferencial aos escritores. Se um processo escritor pretende aceder, assim que a secção crítica esteja livre, o processo escritor tem acesso à secção crítica.



SEMÁFOROS – LEITORES-ESCRITORES (2)

- Solução 1: acesso preferencial aos leitores.




```
int NLeitores=0;
/*semáforos*/
semaforo mutex; /*exclusão mútua no acesso a NLeitores*/
semaforo escrita; /*exclusão mútua na escrita*/
escrita.valor=1;
mutex.valor=1;
```

```
/*escritores*/
...
espera(&escrita);
/* escreve dados */
...
assinala(&escrita);
...
```

```
/*leitores*/
...
espera(&mutex);
NLeitores++;
if (NLeitores==1) espera(&escrita);
assinala(&mutex);
/* leitura de dados */
...
espera(&mutex);
NLeitores--;
if (NLeitores==0) assinala(&escrita);
assinala(&mutex);
...
```

- Análise da solução:
  - A variável NLeitores guarda a informação sobre o número de processos leitores que estão a aceder à secção crítica. O semáforo mutex (inicializado a 1) controla o acesso em modo exclusivo à variável NLeitores. O semáforo escrita (inicializado a 1) controla o acesso em modo exclusivo para os processos escritores.
  - Se um processo escritor está na secção crítica e N processos leitores pretendem aceder, o primeiro fica bloqueado no semáforo escrita e os restantes N-1 ficam bloqueados no semáforo mutex.
  - Enquanto existirem processos leitores, todos os processos escritores ficam bloqueados no semáforo escrita. Apenas o último processo leitor irá assinalar o semáforo escrita (possibilidade de privação para os processos escritores).

SEMÁFOROS: JANTAR DOS FILÓSOFOS

- Cinco filósofos estão sentados à volta de uma mesa redonda. Cada filósofo tem à sua frente uma taça de arroz. Entre cada par de filósofos existe um pauzinho chinês. A vida de um filósofo alterna entre comer e pensar, mas para comer necessita de dois pauzinhos. Quando um filósofo tem fome, tenta adquirir um pauzinho da direita e outro da esquerda, independentemente da ordem. Se conseguir obter ambos, o filósofo come durante um tempo, depois pousa os pauzinhos e recomeça a pensar. Caso contrário o filósofo tem que aguardar, com fome, até ter ambos os pauzinhos disponíveis. (Nota: alguns preferem "spaghetti" e dois garfos para o comer, mas o problema é o mesmo!!)

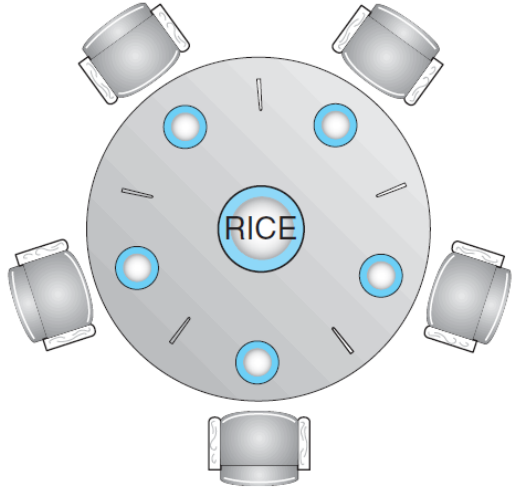


Figure 5.13 The situation of the dining philosophers.

Fonte da imagem: [Silberschatz2014]

- Trata-se de um exemplo clássico de sincronização entre processos, por representar uma classe de vários problemas de controlo de concorrência entre processos.

SEMÁFOROS: JANTAR DOS FILÓSOFOS (1)

- **A única variável partilhada é um vetor de semáforos**
  - pauzinho[N], sendo N=5
  - **todos inicializados a 1**

```
/* Semáforo */
#define N 5
/* semáforo por pauzinho */
semaforo pauzinho[N];
for(i=0;i<N;i++)
    pauzinho[i].valor=1;
    /* exclusão mútua */
```

```
while (true)            {
    wait(pauzinho[i]);//aguarda lock 1º pauzinho
    wait(pauzinho[(i+1)%N]); //aguarda lock 2º pauzinho
    /*          comer       */
    signal(pauzinho[i]); //liberta lock 1º pauzinho
    signal(pauzinho[(i+1)%N]);//liberta lock 2º pauzinho
     /*pensar      */
}
```

- Esta solução garante que dois filósofos vizinhos não estão simultaneamente a comer, mas pode criar uma situação de ***interblocagem (deadlock)***. Os cinco filósofos detêm o pauzinho da esquerda. Não garante, ainda, que um ou mais filósofos morram à fome (privação).
- **Algumas soluções para a *interblocagem***:
  - Permitir apenas que quatro filósofos se sentem à mesa;
  - Permitir apenas que um filósofo adquira os dois pauzinhos em simultâneo;
  - Um filósofo com número par adquire primeiro o pauzinho da direita e depois o da esquerda, um filósofo com número ímpar adquire primeiro o pauzinho da esquerda e depois o da direita;

SEMÁFOROS: JANTAR DOS FILÓSOFOS (2)

- **Solução: Permitir apenas que um filósofo adquira os dois pauzinhos em simultâneo.**
  - Um filósofo só consegue apanhar os pauzinhos se ambos os vizinhos (esquerda e direita) não estiverem a comer. Caso contrário fica bloqueado.
  - Quando termina de comer, verifica se os seus vizinhos o podem fazer.
  - A privação pode ocorrer.

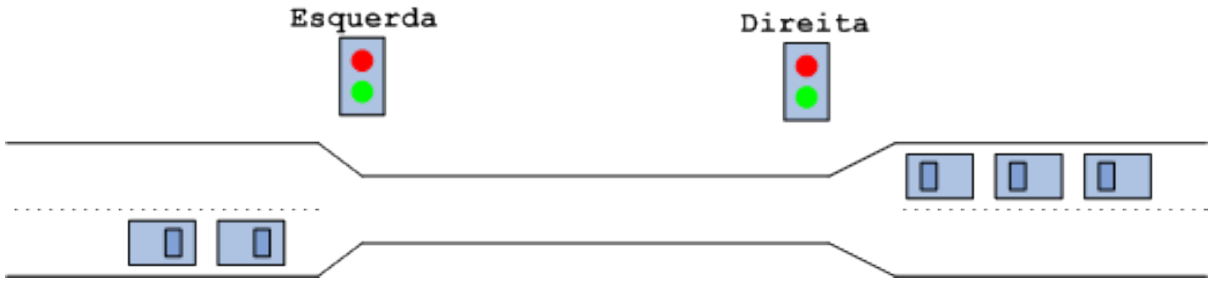
```
#define N 5
#define ESQUERDA (i+N-1)%N
#define DIREITA (i+1)%N
#define PENSAR 0
#define FOME 1
#define COMER 2
semaforo semFilosofo [N];        /*
inicializado a 0 */
semaforo mutex;                  /*
inicializado a 1 */
int estado [N];          /*inicializado a
PENSAR*/
/* Filosofo i*/
while (1) {
        /* pensar */ ...
        ApanharPauzinhos (i); /
        * comer */ ...
        Pousar Pauzinhos (i);
}
```

```
void teste (int i) {
if (estado [i] == FOME &&
        estado [ESQUERDA] != COMER &&
        estado [DIREITA] != COMER) {
        estado [i]= COMER;
                assinala (&semFilosofo [i]);
}
```

```
void Apanhar Pauzinhos (int i) {
        espera (&mutex);
        estado [i]= FOME;
        teste (i);
        assinala (&mutex);
        espera (&semFilosofo [i]);
}
void Pousar Pauzinhos (int i) {
        espera (&mutex);
        estado [i] PENSAR;
        teste (ESQUERDA);
        teste (DIREITA);
        assinala (&mutex);
}
```

SEMÁFOROS: PONTE ESTREITA (1)

Considere que existe uma ponte com uma única via por onde passam carros em ambos os sentidos. Para evitar colisões existe um semáforo luminoso em cada extremidade da ponte. Por cada carro que se aproxima da ponte do lado esquerdo é criado um processo que executa a rotina CarroEsquerda() e coloca o semáforo da esquerda a vermelho. Quando um carro da esquerda sai da ponte invoca uma rotina CarroSaiEsquerda() de modo a sinalizar a sua saída da ponte, colocando o semáforo a verde. Para os carros provenientes da direita, o procedimento é semelhante, chamando neste caso as rotinas CarroDireita() e CarroSaiDireita(). São permitidos vários carros na mesma direcção, mas não em direcções opostas.



SEMÁFOROS: PONTE ESTREITA (2)

- O primeiro carro da esquerda fica bloqueado no semáforo ponte, se existirem carros da direita. Os restantes ficam bloqueados no semáforo mutexEsquerda. O último carro da esquerda a passar na ponte liberta o semáforo ponte. O mesmo é realizado pelos carros da direita.

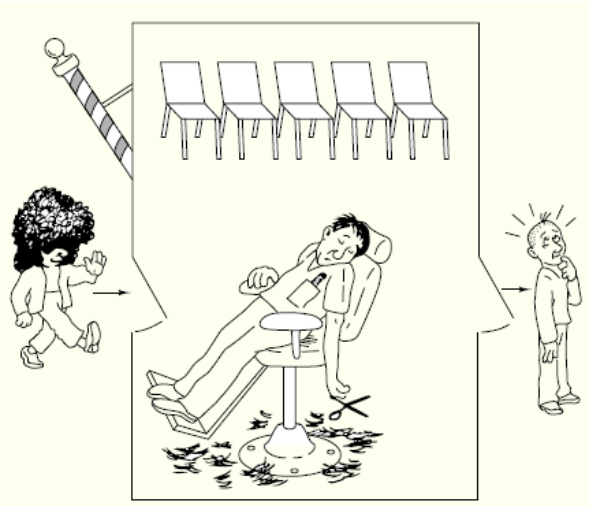
```
semaforo mutexEsquerda;
mutexEsquerda.valor=1;
semaforo mutexDireita;
mutexDireita.valor=1;
semaforo ponte;
ponte.valor=1;
int
CarrosEsquerda=0,CarrosDireita=0;
```

```
void carroEsquerda(){
  espera(&mutexEsquerda);
  CarrosEsquerda++;
  if (CarrosEsquerda == 1)
      espera(&ponte);
  assinala(&mutexEsquerda);
}

void carroSaiEsquerda(){
  espera(&mutexEsquerda);
  CarrosEsquerda--;
  if (CarrosEsquerda == 0)
      assinala(&ponte);
  assinala(&mutexEsquerda);
}
```

```
void carroDireita(){
  espera(&mutexDireita);
  CarrosDireita++;
  if (CarrosDireita == 1)
      espera(&ponte);
  assinala(&mutexDireita);
}

void carroSaiDireita(){
  espera(&mutexDireita);
  CarrosDireita--;
  if (CarrosDireita == 0)
      assinala(&ponte);
  assinala(&mutexDireita);
}
```

SEMÁFOROS: BARBEARIA

Numa barbearia existe uma cadeira onde o barbeiro corta cabelo e N cadeiras para os clientes que estão à espera aguardarem sentados. Se não existem clientes, o barbeiro senta-se na cadeira e adormece. Quando um cliente chega, ele tem que acordar o barbeiro dorminhoco para lhe cortar o cabelo. Se entretanto chegarem mais clientes, enquanto o barbeiro estiver a cortar o cabelo ao primeiro, ou esperam numa cadeira livre ou vão-se embora se já não houver mais cadeiras livres



```
#define N 5
semaforo mutex;mutex.valor=1;
semaforo cadeira; cadeira.valor=0;
semaforo barbeiro; barbeiro.valor=0;
int cliente=0;
```

```
void TarefaCliente(){
  espera(&mutex);
  if (clientes < N){
    clientes++;
    assinala((&barbeiro);
    assinala(&mutex);
    espera(&cadeira);
    <*corta cabelo*>
  }
  else assinala(&mutex);
}
```

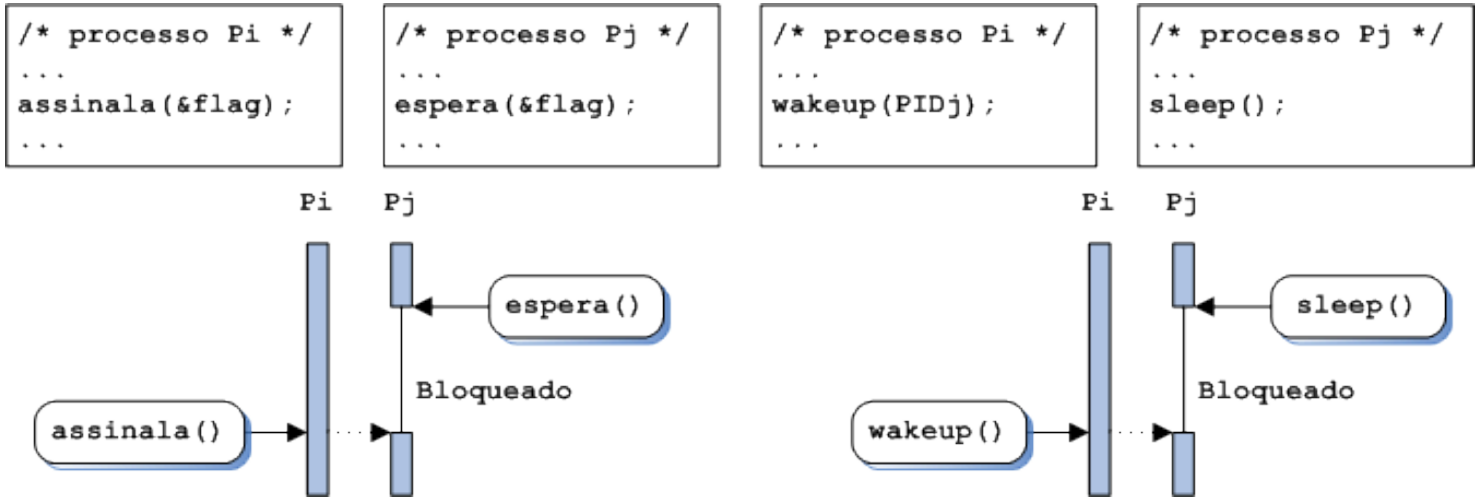
```
void TarefaBarbeiro(){
  while (1) {
    espera(&barbeiro);
    espera((&mutex);
    clientes--;
    assinala(&mutex);
    assinala(&cadeira);
    <*corta cabelo*>
  }
}
```

COOPERAÇÃO ENTRE PROCESSOS:SEMÁFOROS VS SINAIS

- Semáforos: sincronização indireta. Um dos processos deve assinalar o semáforo para que outro, qualquer que seja, não fique bloqueado aquando da invocação da função espera(). O assinalar de um semáforo não é endereçável a um processo em particular.

- Sinais: sincronização direta. O envio de sinais só é possível para processos devidamente identificados (PID). Duas operações básicas estão disponíveis:
  - sleep(): causa o bloqueio do processo ("adormece") que a invoca até ser desbloqueado ("acordado") aquando da recepção de um sinal;
  - wakeup(PID): desbloqueia o processo identificado por PID.

- Ambas as soluções evitam a espera activa.



COOPERAÇÃO ENTRE PROCESSOS:SEMÁFOROS VS SINAIS

- O semáforo tem capacidade de memória. Pelo contrário, um sinal de wakeup() pode ser perdido quando um processo manda "acordar" o outro sem este ter "adormecido".
- Por esse motivo os sinais não garantem exclusão mútua como os semáforos.
  - Use semáforos quando precisar de sincronização forte e controle sobre o acesso a recursos compartilhados.
  - Use sinais quando precisar de notificação rápida de eventos, como interrupções ou mensagens entre processos.