Sistemas Operacionais I

Sincronização

Prof. Leandro Marzulo

O problema da região crítica

- Processos concorrentes acessam dados compartilhados: mecanismos para garantir consistência são necessários

- Condição de corrida: vários processos acessando e manipulando dados compartilhados concorrentemente; resultado da execução depende da ordem em que os acessos ocorreram

O problema da região crítica

Produtor

```
while (true)
{
    /* produz um item e põe em
    nextProduced */
    while (count == BUFFER_SIZE)
    ; // nada a fazer
    buffer [in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    count++;
}
```

Consumidor

```
while (true)
{
    while (count == 0)
    ; // nada a fazer
    nextConsumed=  buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    count--;
    /*  consume o item em nextConsumed
}
```

O problema da região crítica

count++ poderia ser implementado assim:

registrador1 = count
registrador1 = registrador1 + 1
count = registrador1

count-- poderia ser implementado assim:

registrador2 = count
registrador2 = registrador2 - 1
count = registrador2

Considere a seguinte sequencia de acessos, inicialmente **count=5:**

Passo 0: producer execute registrador1 = count {registrador1 = 5}
Passo 1: producer execute registrador1 = registrador1 + 1 {registrador1 = 6}
Passo 2: consumer execute registrador2 = count {registrador2 = 5}
Passo 3: consumer execute registrador2 = registrador2 -1 {registrador2 = 4}
Passo 4: producer execute count = registrador1 {count = 6 }
Passo 5: consumer execute count = registrador2 {count = 4} !!

O problema da região crítica

- A região crítica é a porção de código em que o processo pode modificar variáveis compartilhadas, atualizar uma tabela, escrever um arquivo etc.

- O SO tem que assegurar que não mais do que um único processo (ou thread) possa executar na região crítica em qualquer dado momento.

O problema da região crítica

- Uma solução para o problema da região crítica deve garantir três (3) condições: exclusão mútua, progresso e espera limitada

O problema da região crítica

- Exclusão mútua: não mais do que um único processo pode executar na região crítica em qualquer dado momento
- Progresso: Se nenhum processo está a executar na sua secção crítica e existem processos que pretendem entrar na sua secção crítica, então apenas estes podem participar na decisão do processo que irá entrar na secção crítica e esta decisão não pode ser adiada indefinidamente.
- Espera limitada: Deve existir um limite de espera para o número de vezes em que é permitido a entrada a outros processos na sua secção crítica depois de um processo ter solicitado entrar na secção crítica e antes de o pedido ser garantido.

Operações atômicas

- Comandos count++ and count-- são comandos de alto-nível que são traduzidos em múltiplas instruções de máquina. Estas instruções podem ser interrompidas, podendo ter a execução entrelaçadas com instruções de outros processos.

- A solução deve assegurar exclusão mútua através de operações atômicas, i.e., initerruptíveis, como test-and-set e lock.

Solução de Peterson

Processo i

```
do {
    flag[ i ] = TRUE;
    turn = j;
    while (flag[ j ] && turn == j);
        seção crítica
    flag[ i ] = FALSE;
        seção remanescente
} while (TRUE)
```

Processo j

```
do {
    flag[ j ] = TRUE;
    turn = i;
    while (flag[ i ] && turn == i);
        seção crítica
    flag[ j ] = FALSE;
        seção remanescente
} while (TRUE)
```

Hardware de Sincronização - locks

Test and Set

```
boolean TestAndSet(boolean * target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}

do {
    while (TestAndSet(&lock)); //nada!!!
    //seção crítica
    lock=false;
    // seção remanescente
} while (TRUE)
```

Swap

```
void Swap(boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}

do {
    key=TRUE;
    while (key) Swap(&lock, &key);
    //seção crítica
    lock=false;
    // seção remanescente
} while (TRUE)
```

Hardware de Sincronização - locks

Test and Set com espera limitada

```
do {
      waiting[i] = true;
      key = true;
      while (waiting[i] && key)
            key=TestAndSet(&lock));
       waiting[i] = false;
      //seção crítica
       j = (i+1)%n;
      while((j !=i) && !waiting[j])
            j=(j+1)%n
      if (j ==i) lock=false;
      else waiting[j] = false;
      // seção remanescente
} while (TRUE)
```

Semáforos

- Mecanismo de sincronização provido pelo SO, de mais alto-nível (e fácil de usar) que test-and-set e lock.
- Um semáforo é uma variável inteira que, excetuando-se a inicialização, é acessada somente através de duas operações atômicas: wait() e signal(), também chamadas P() e V(), do holandês (Dijkstra) “proberen” (testar) e “verhogen” (incrementar).
- Principal mecanismo de sincronização do UNIX original

Semáforos

- Semáforo de contagem: valor inteiro que pode variar de modo irrestrito; bloqueia somente quando o valor é zero (0).

- Semáforo binário: valor inteiro pode ser somente 0 ou 1; para exclusão mútua, também chamdo mutex locks; também usado para sincronização entre linhas de execução (threads).

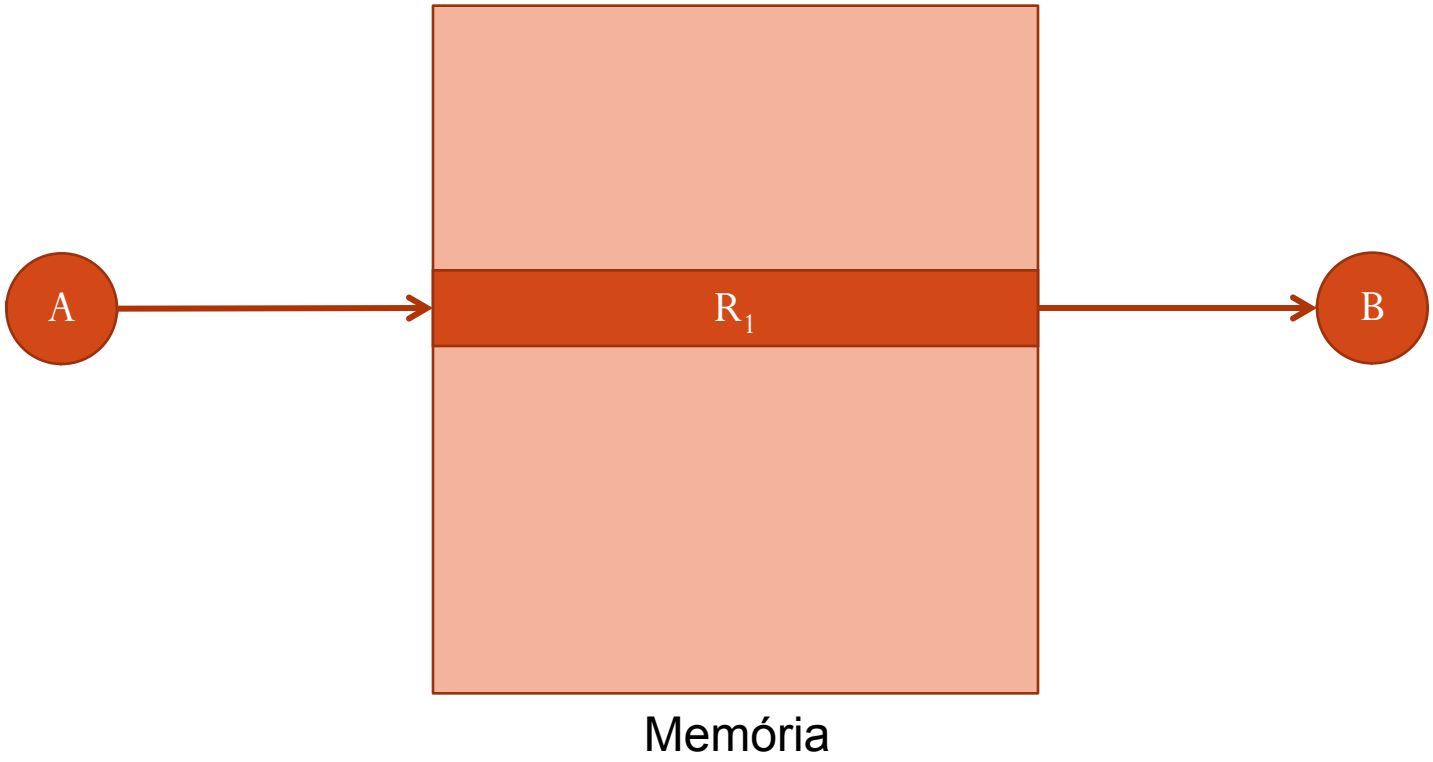
Semáforos

- Provê exclusão mútua

```
Semaphore S;    //  iniciado com 1
wait(S);
    Região Crítica
signal(S);
```

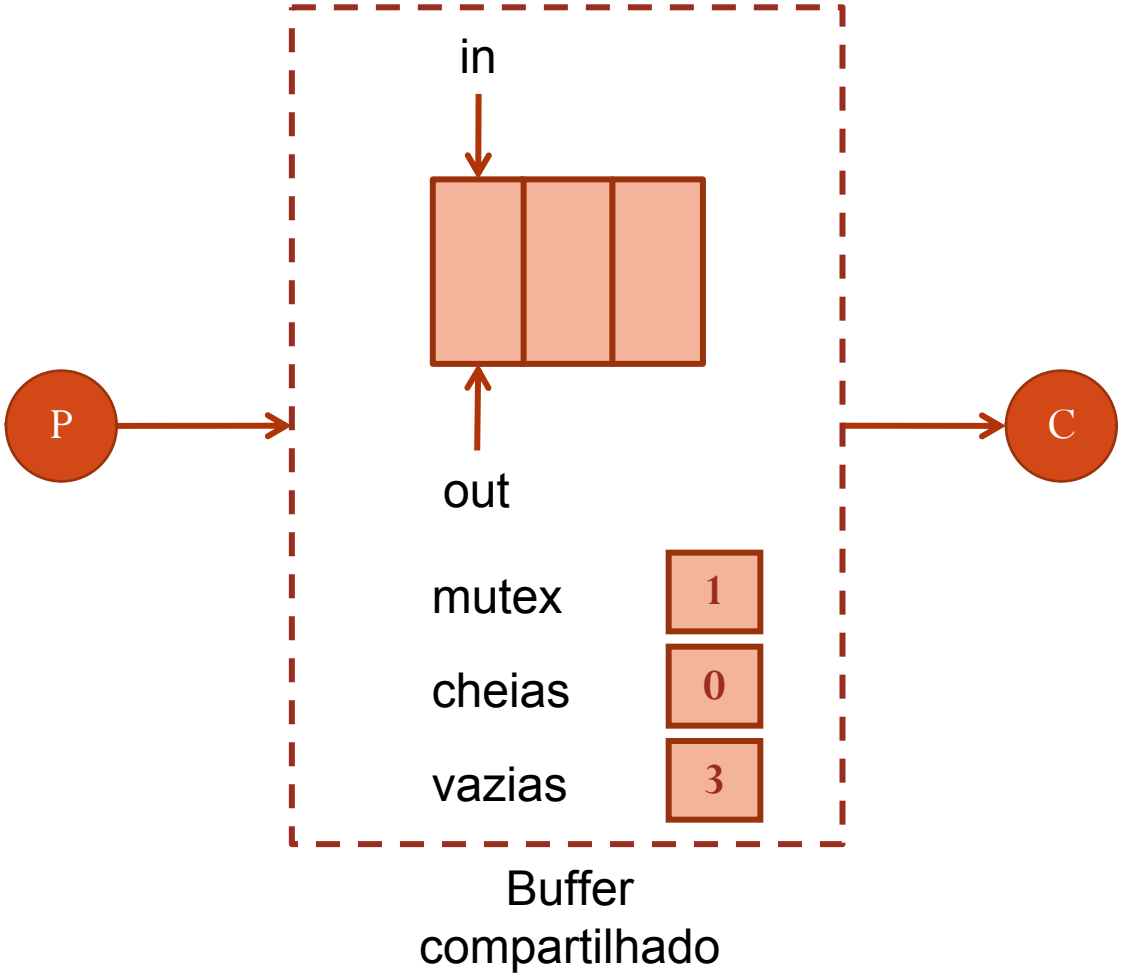
Semáforos

- Produtor / Consumidor



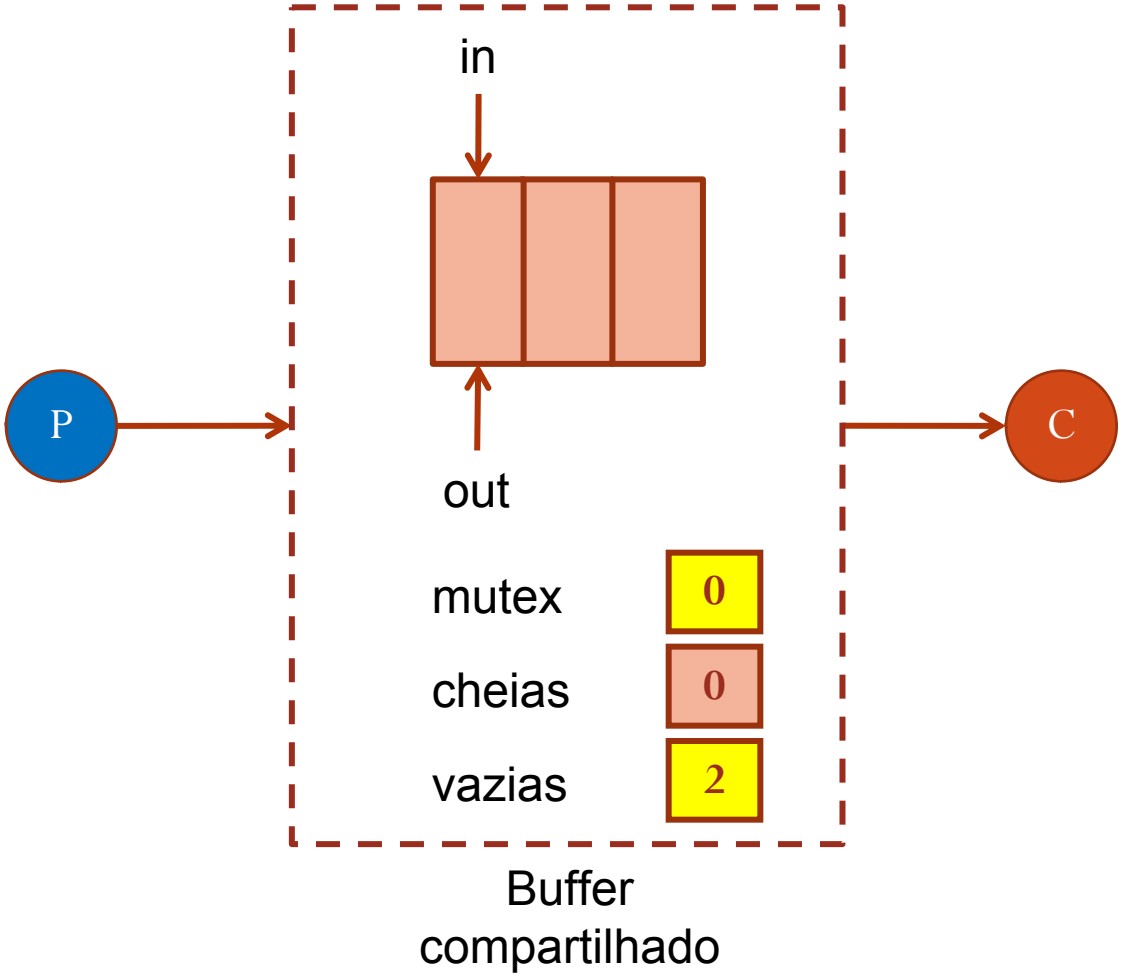
Semáforos

- Produtor / Consumidor



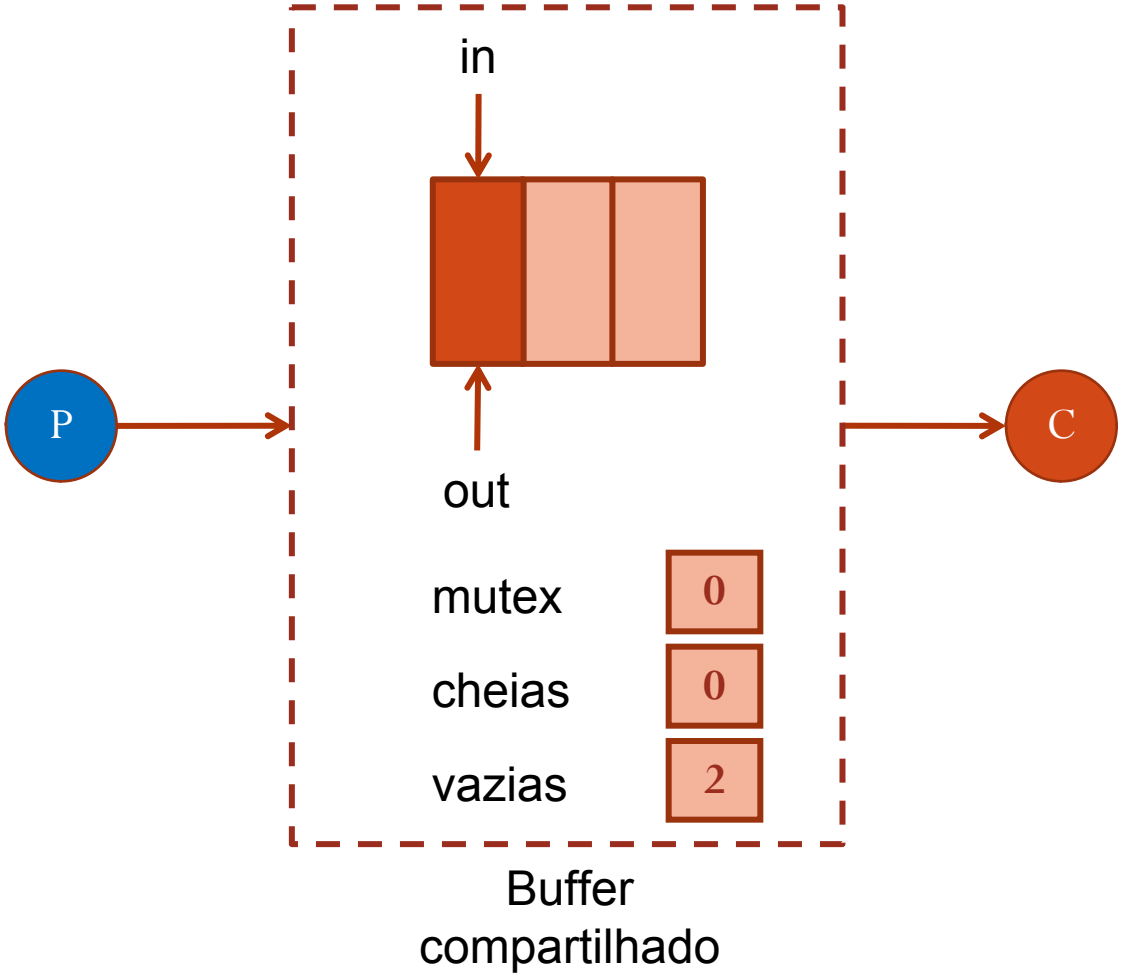
Semáforos

- Produtor / Consumidor



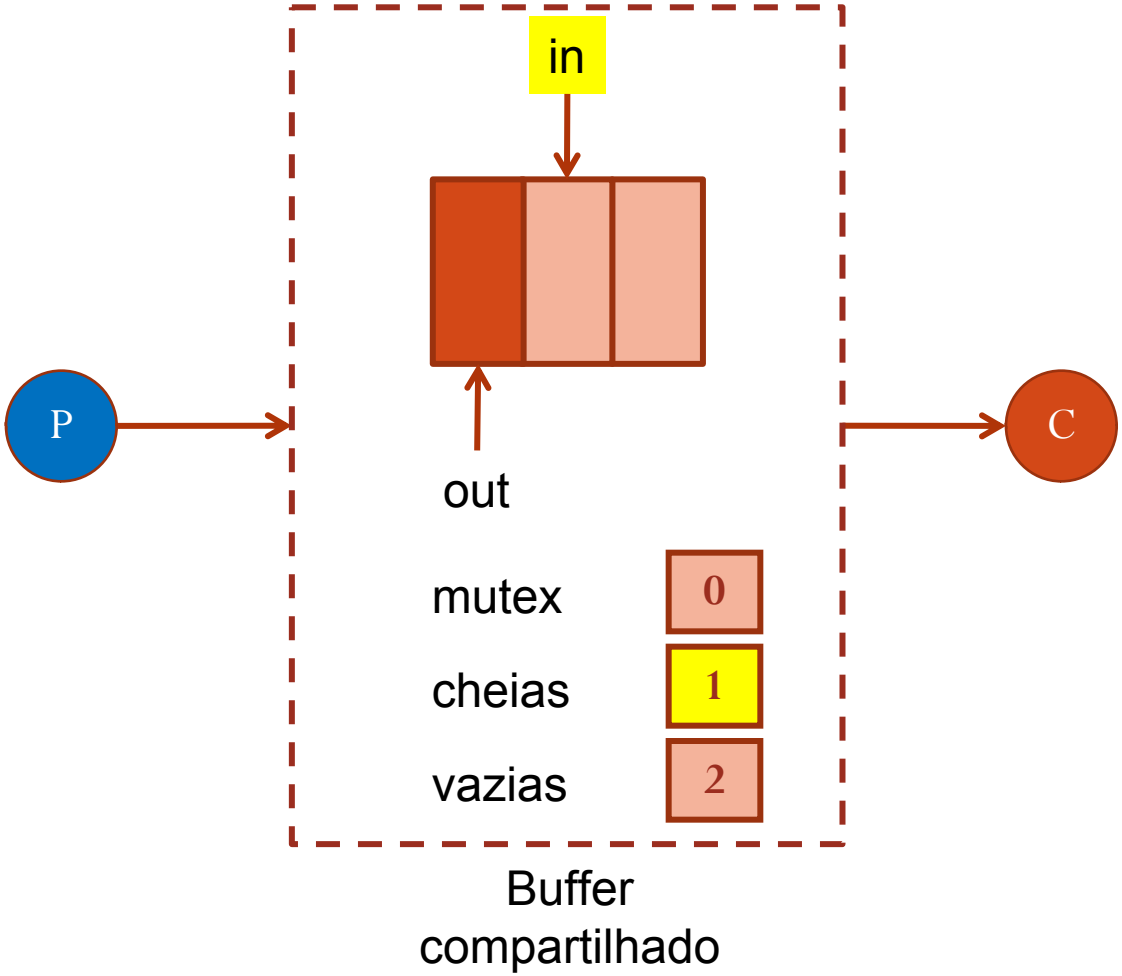
Semáforos

- Produtor / Consumidor



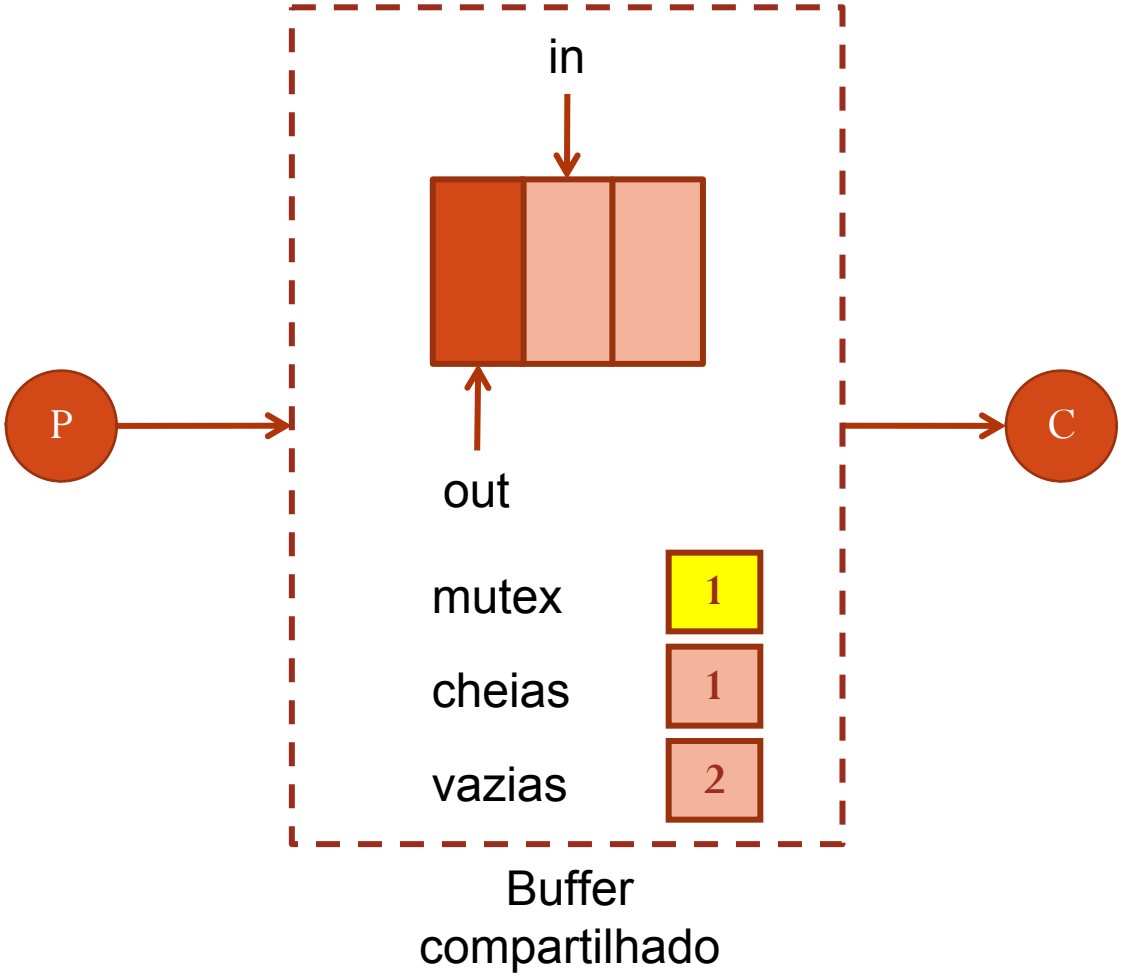
Semáforos

- Produtor / Consumidor



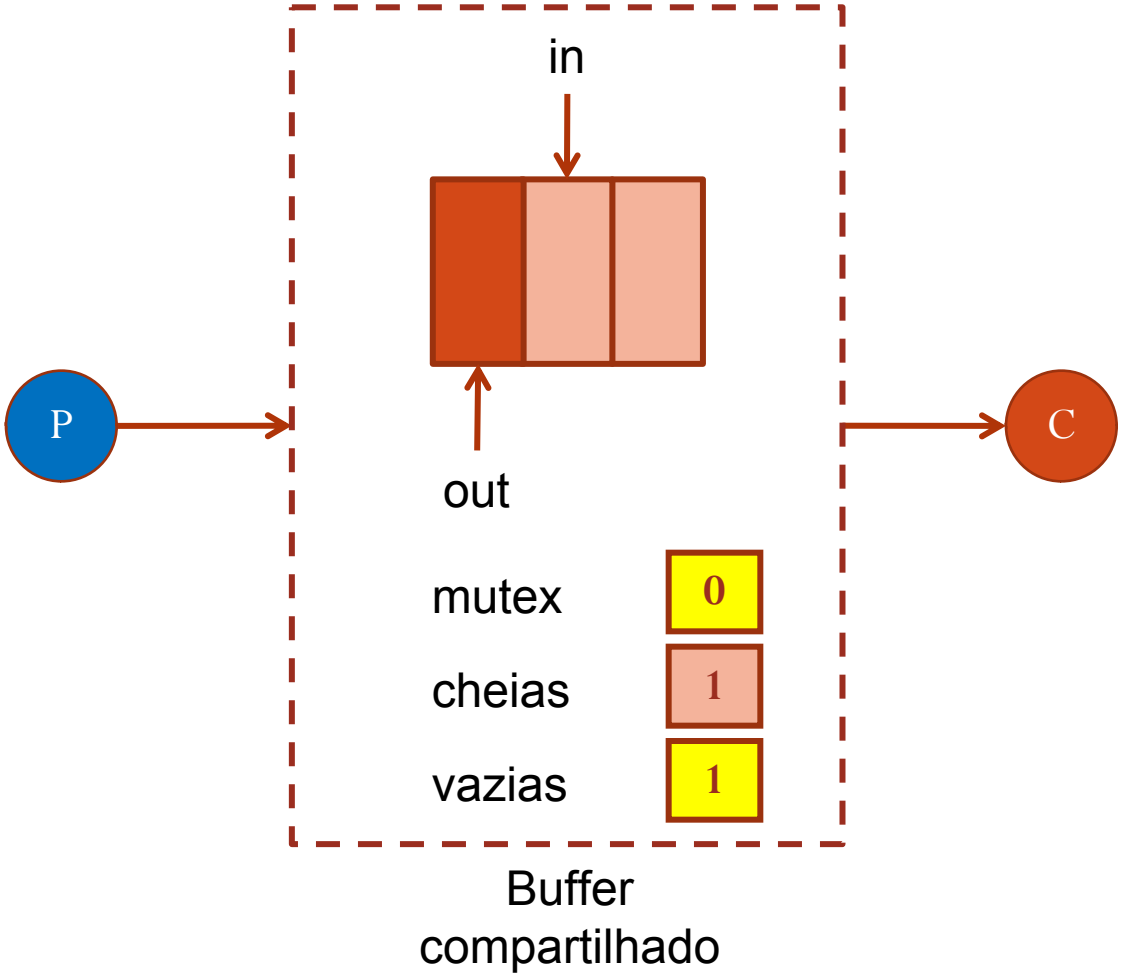
Semáforos

- Produtor / Consumidor



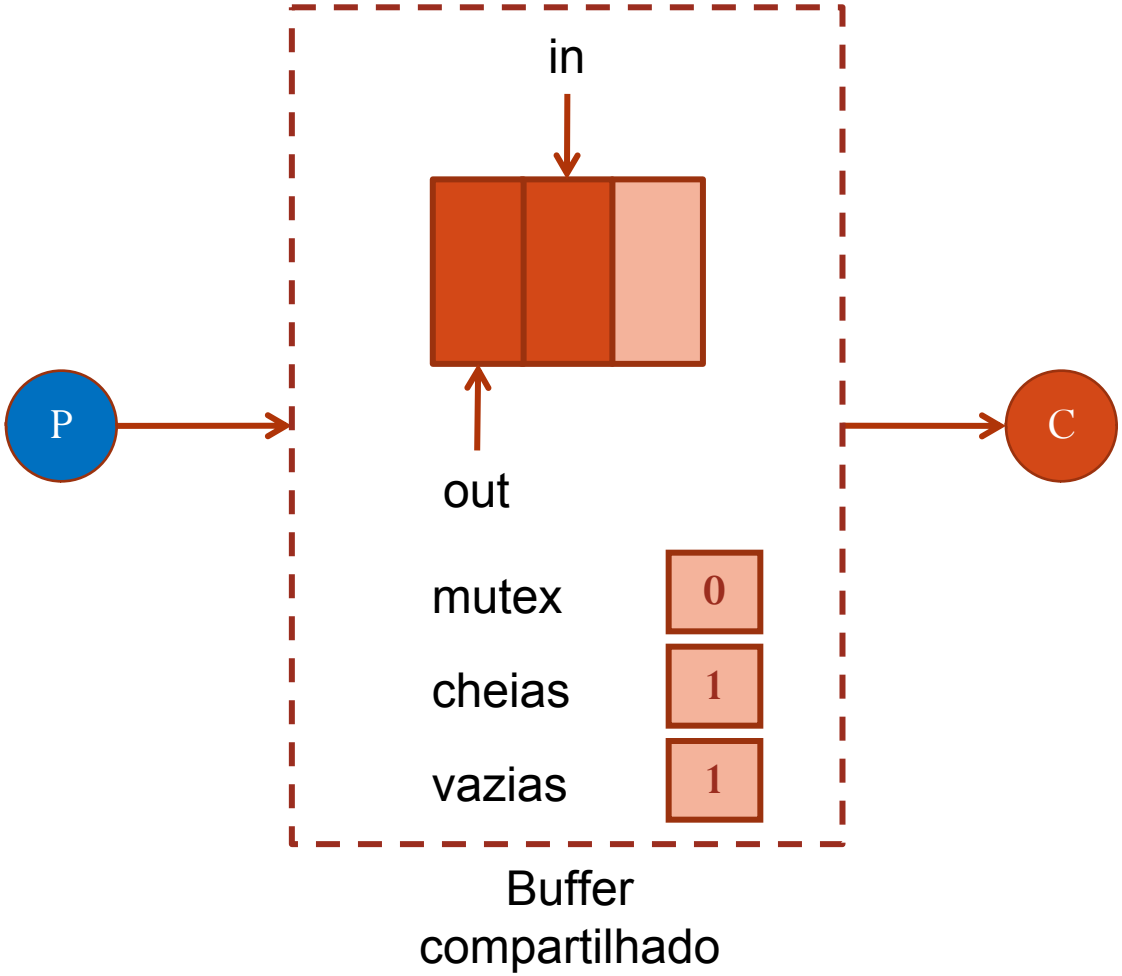
Semáforos

- Produtor / Consumidor



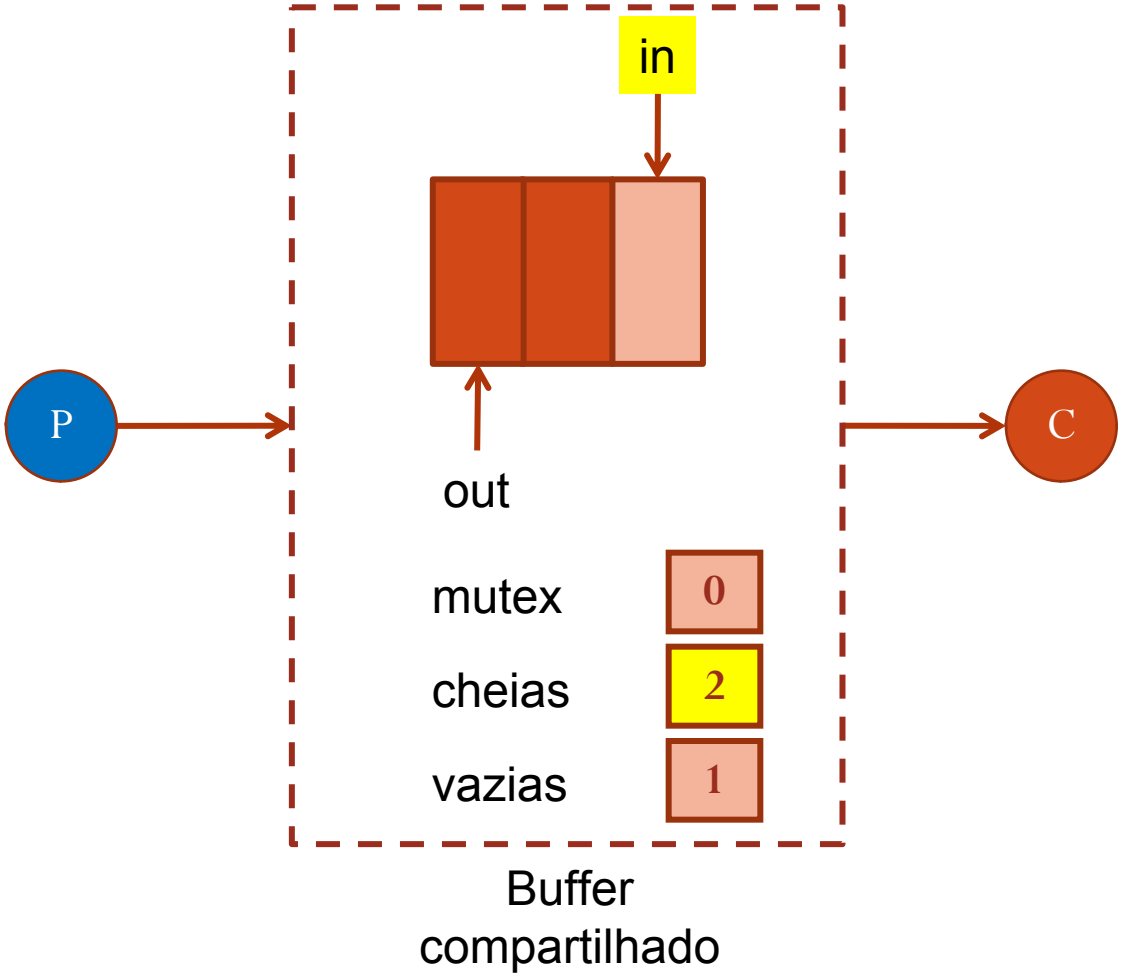
Semáforos

- Produtor / Consumidor



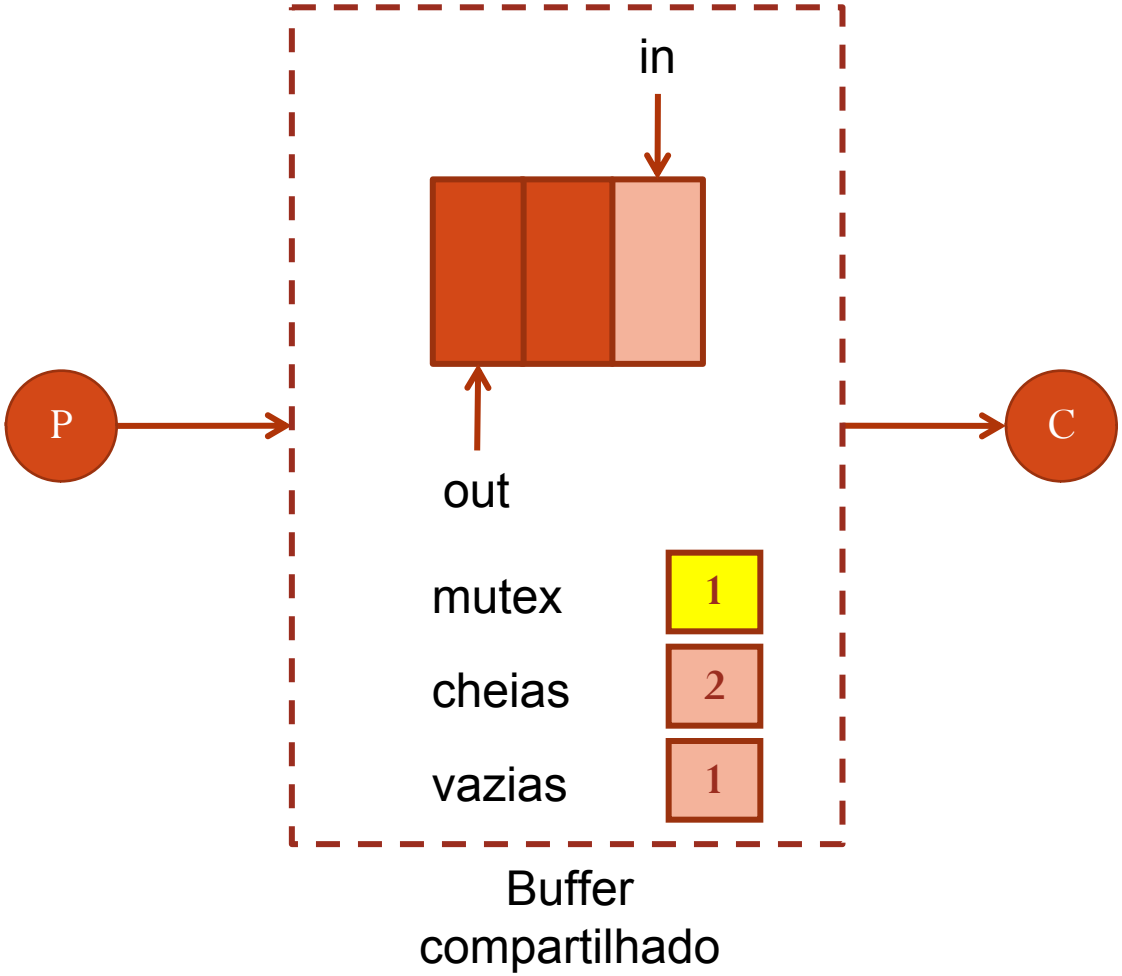
Semáforos

- Produtor / Consumidor



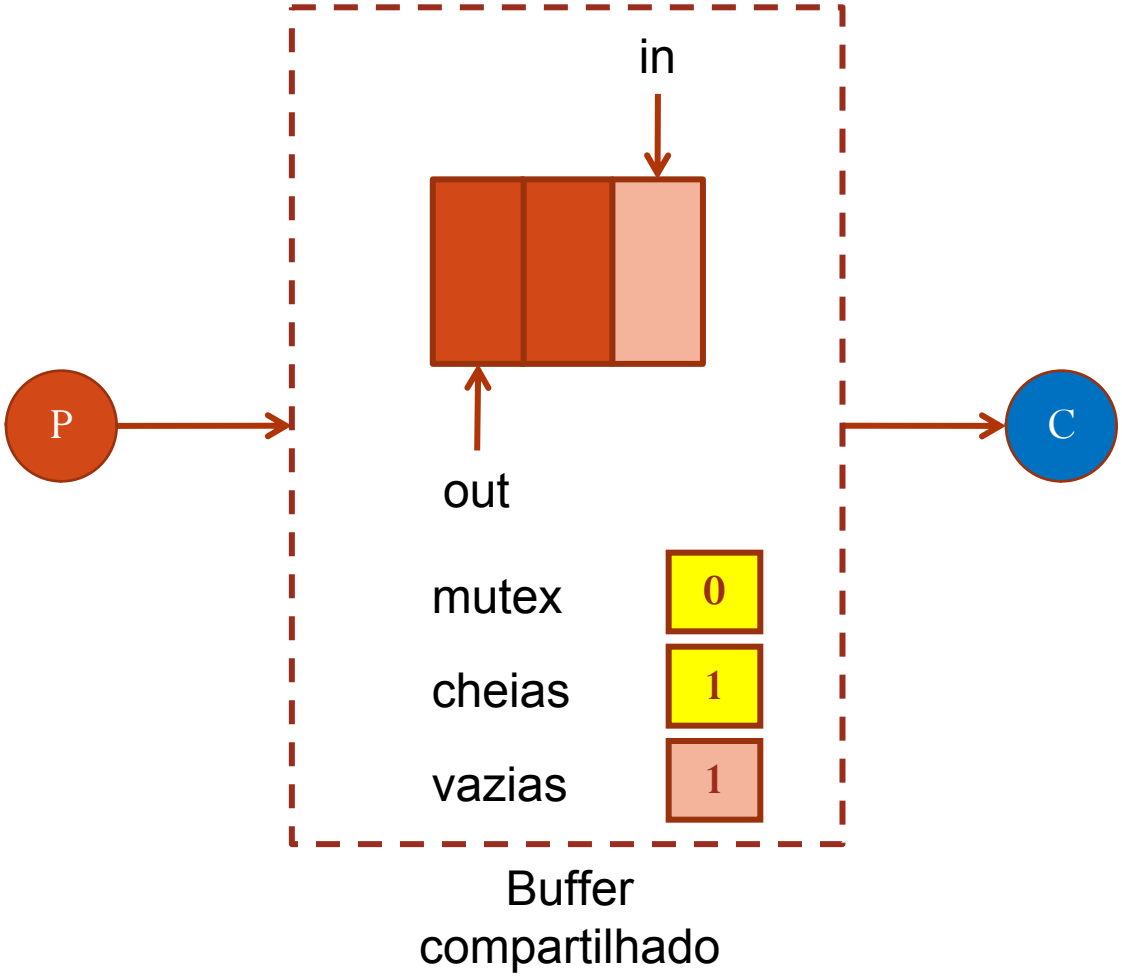
Semáforos

- Produtor / Consumidor



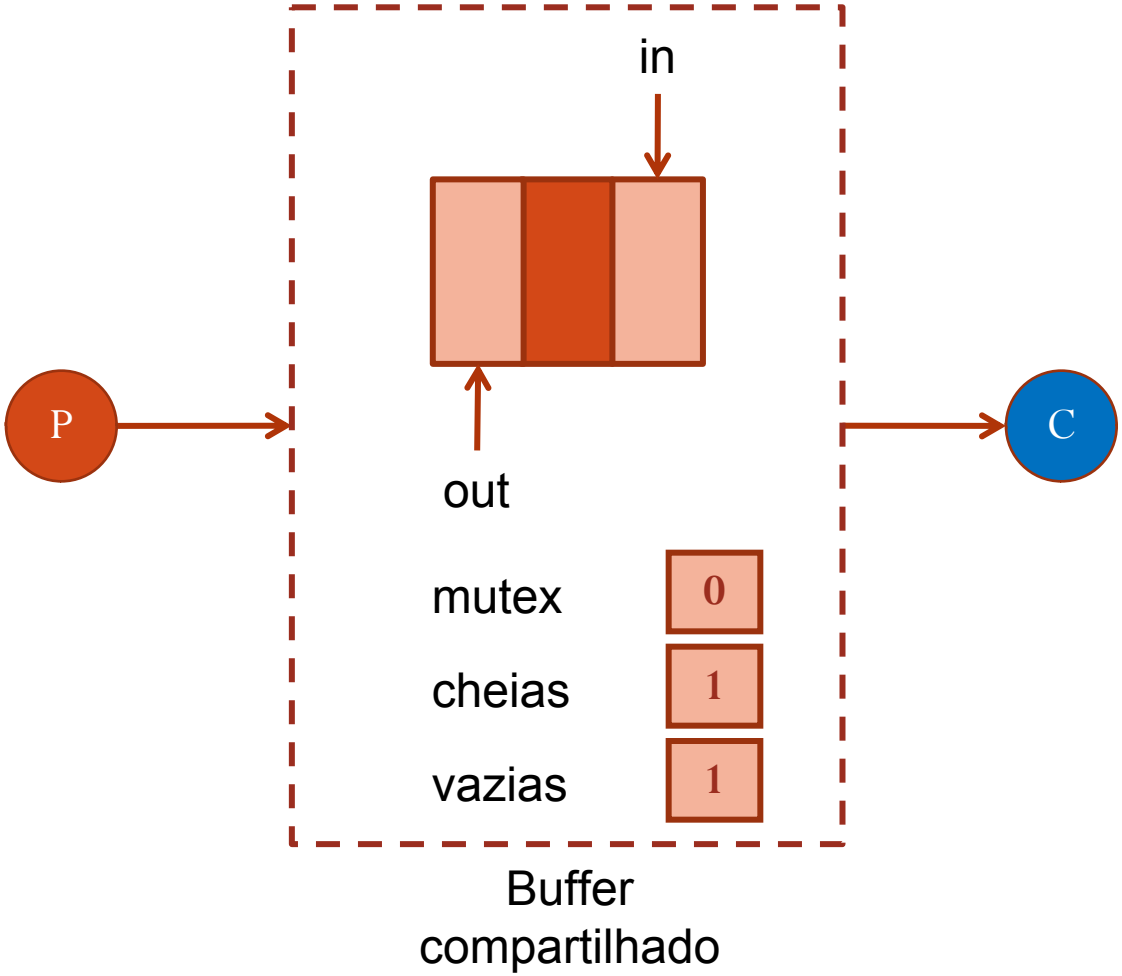
Semáforos

- Produtor / Consumidor



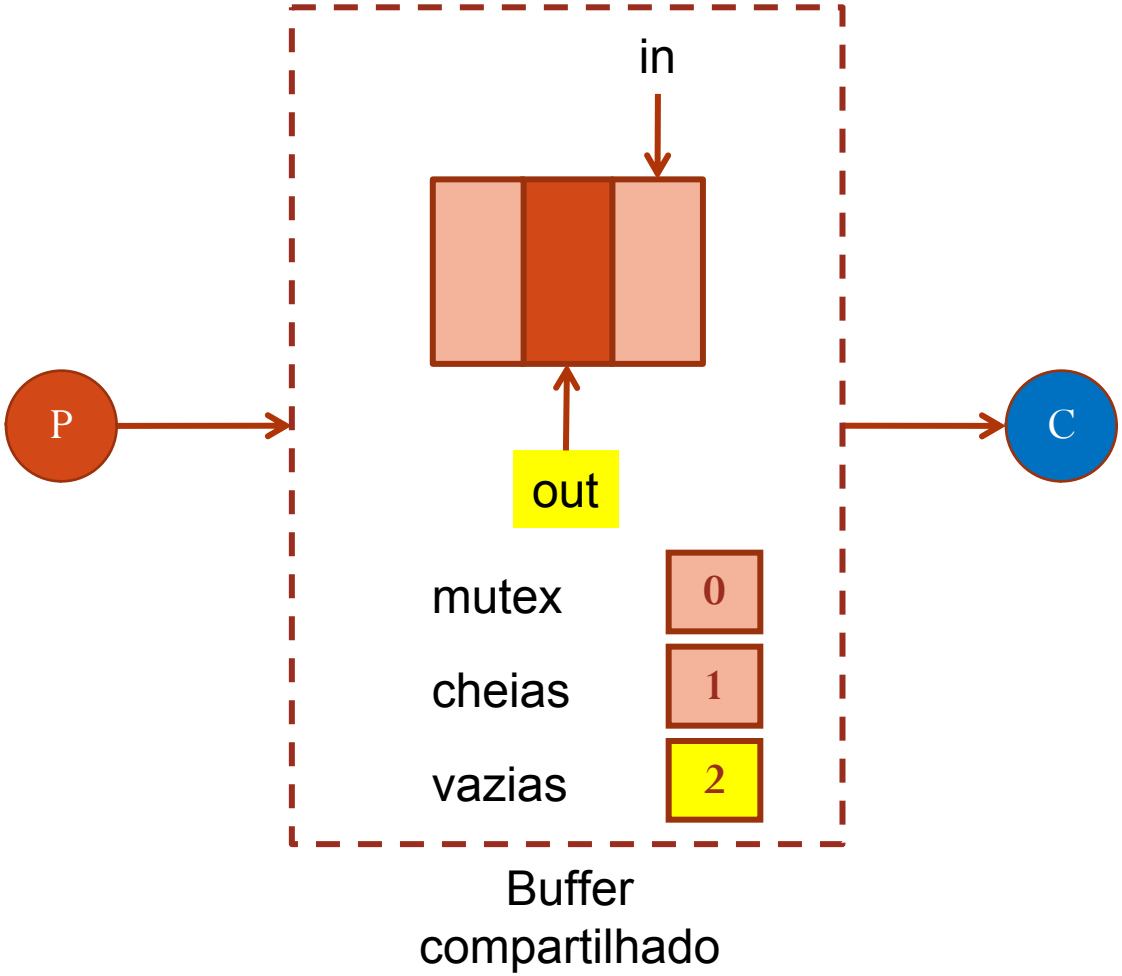
Semáforos

- Produtor / Consumidor



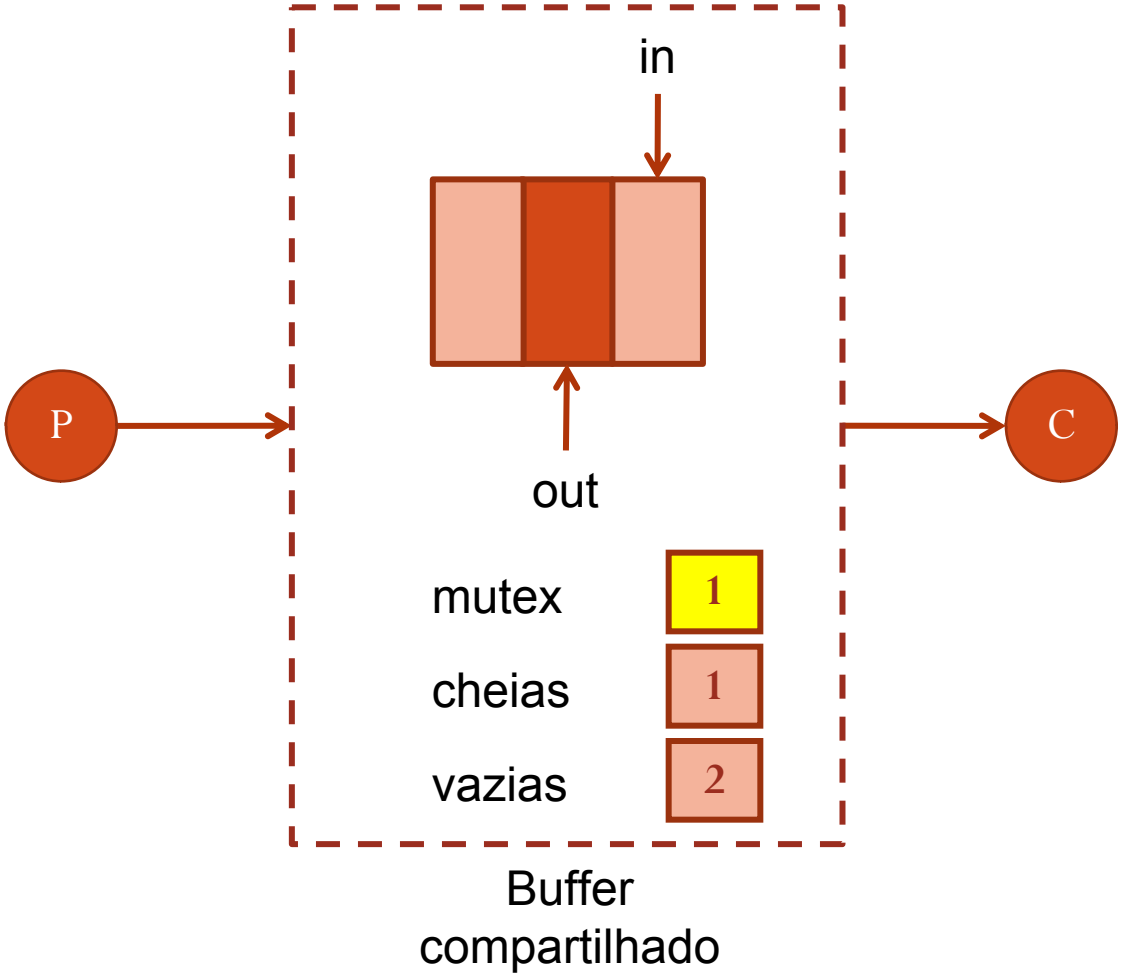
Semáforos

- Produtor / Consumidor



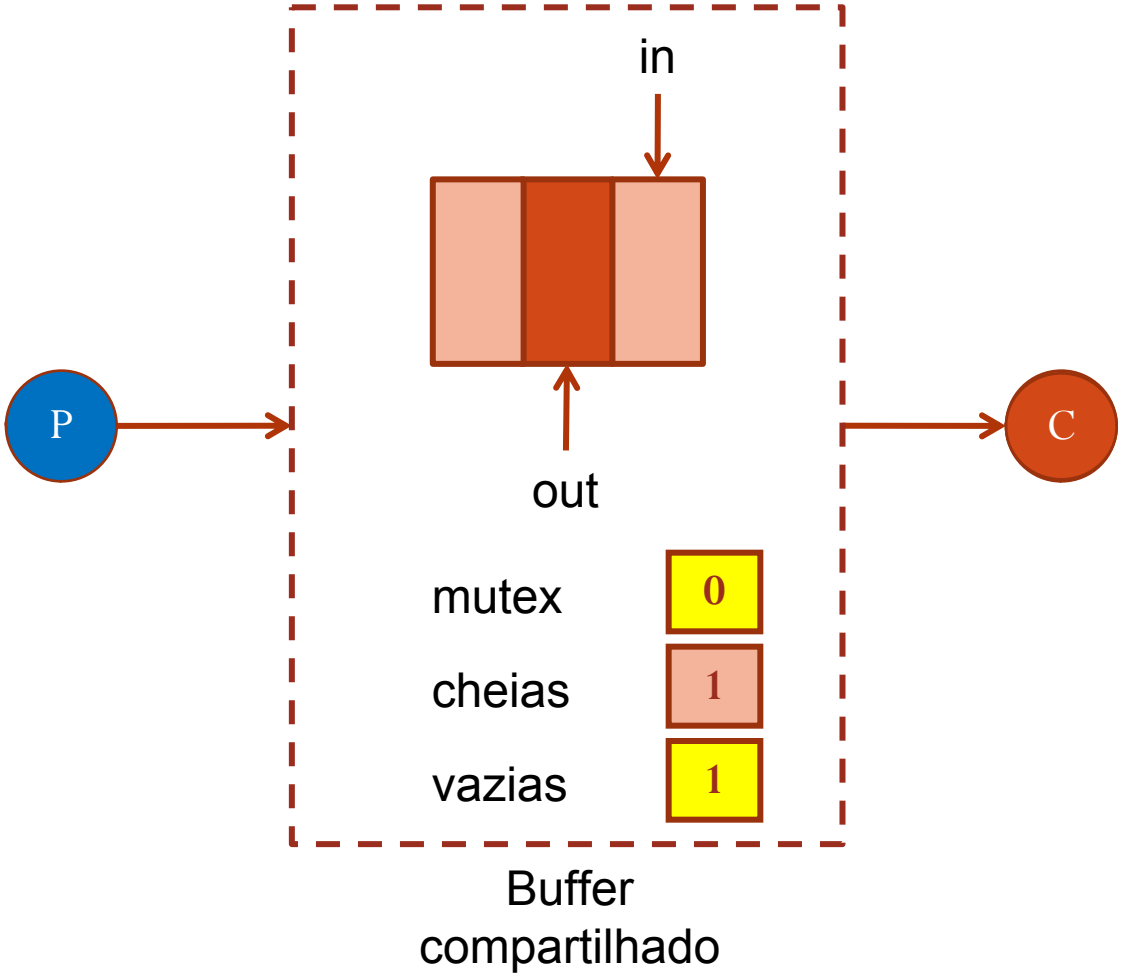
Semáforos

- Produtor / Consumidor



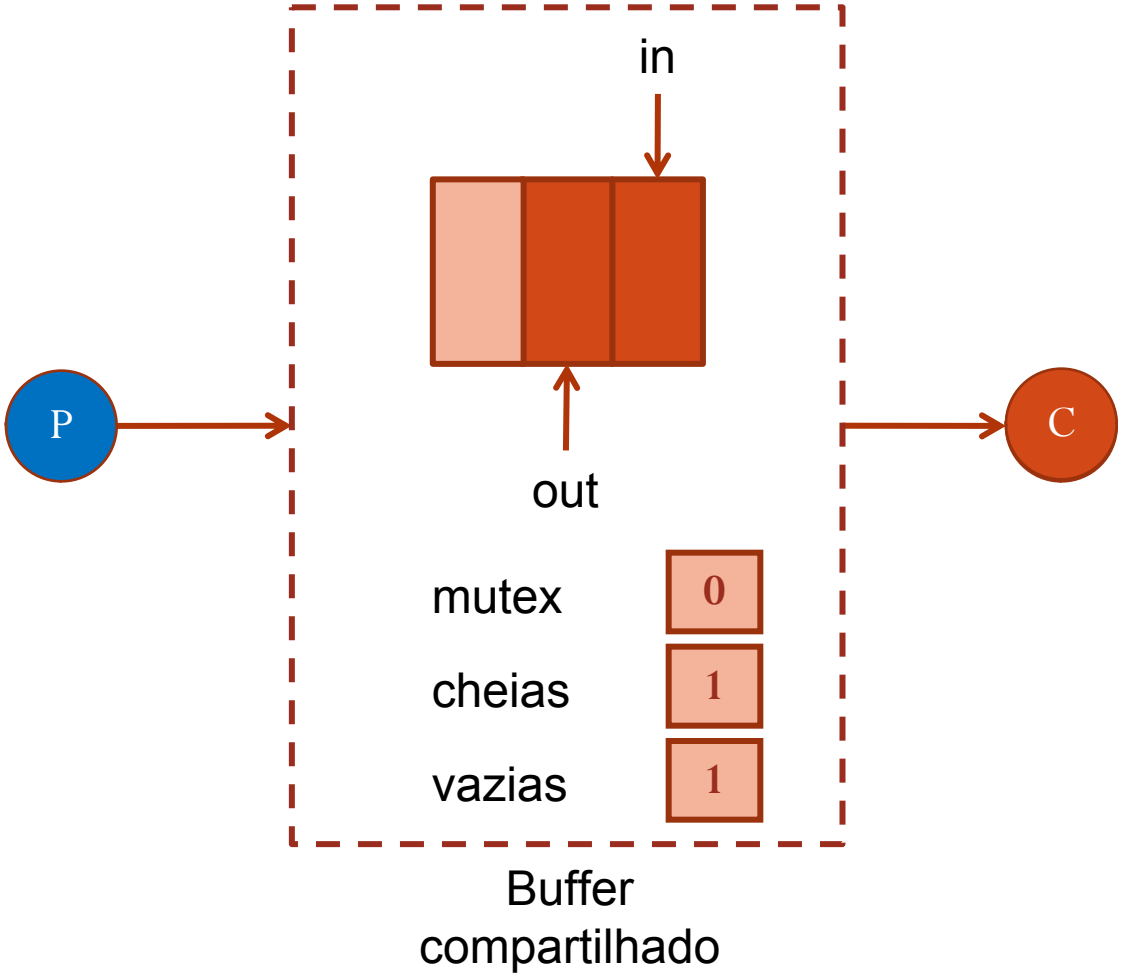
Semáforos

- Produtor / Consumidor



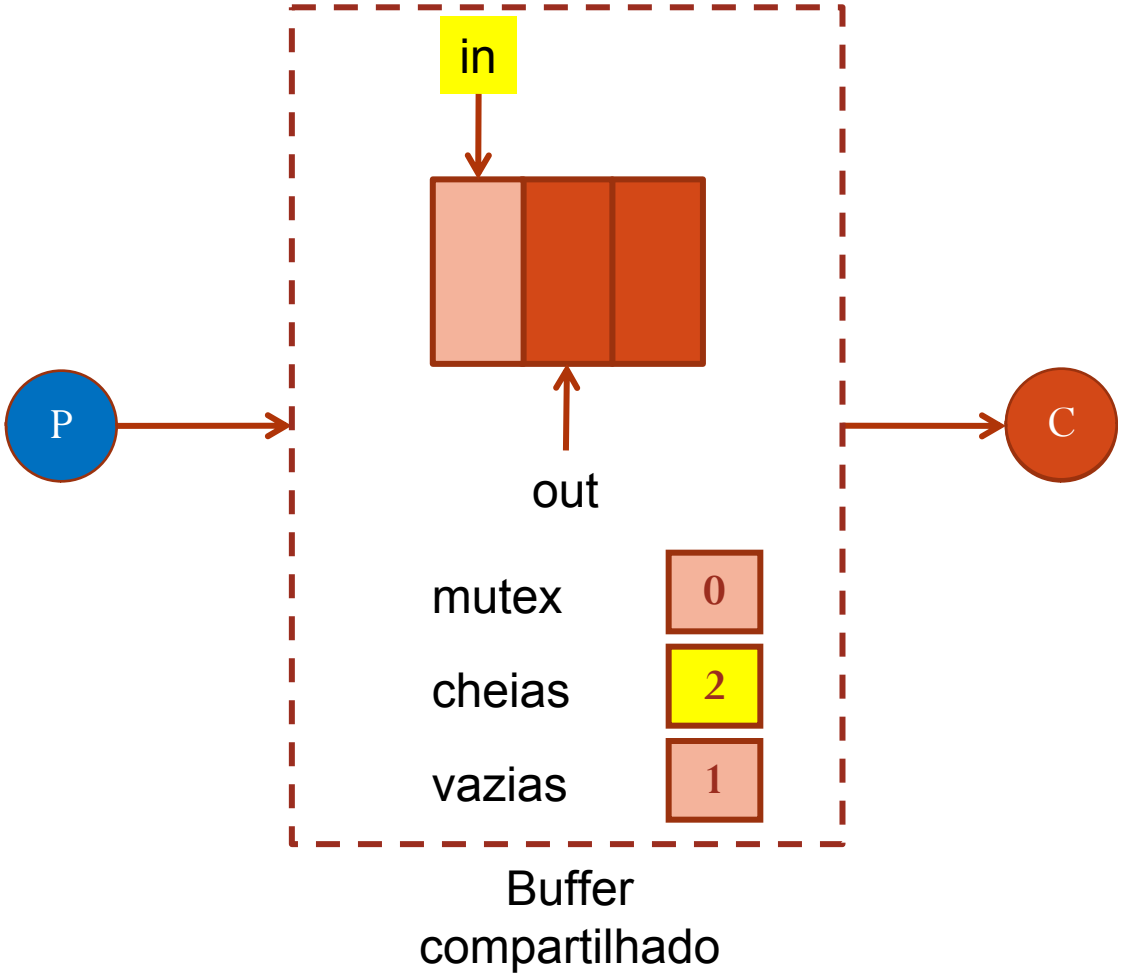
Semáforos

- Produtor / Consumidor



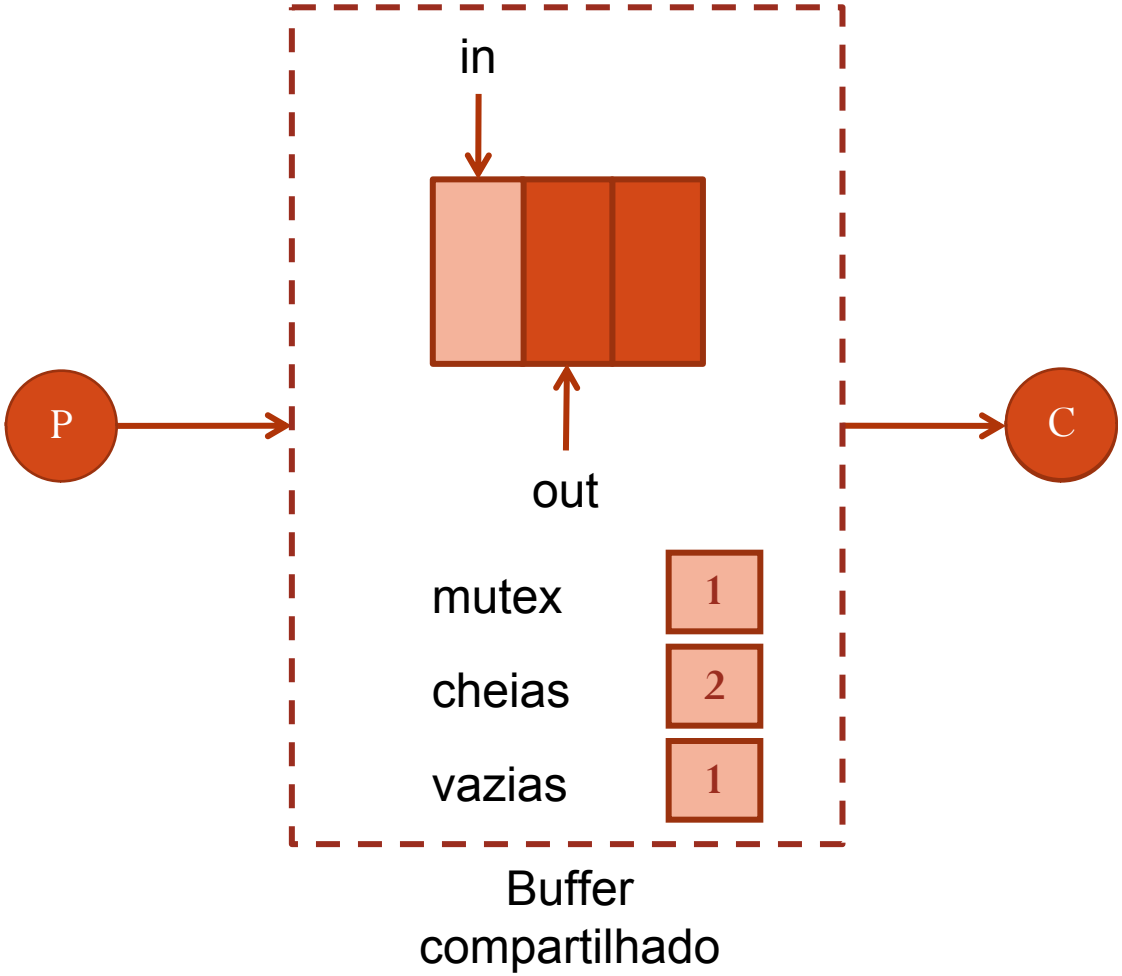
Semáforos

- Produtor / Consumidor



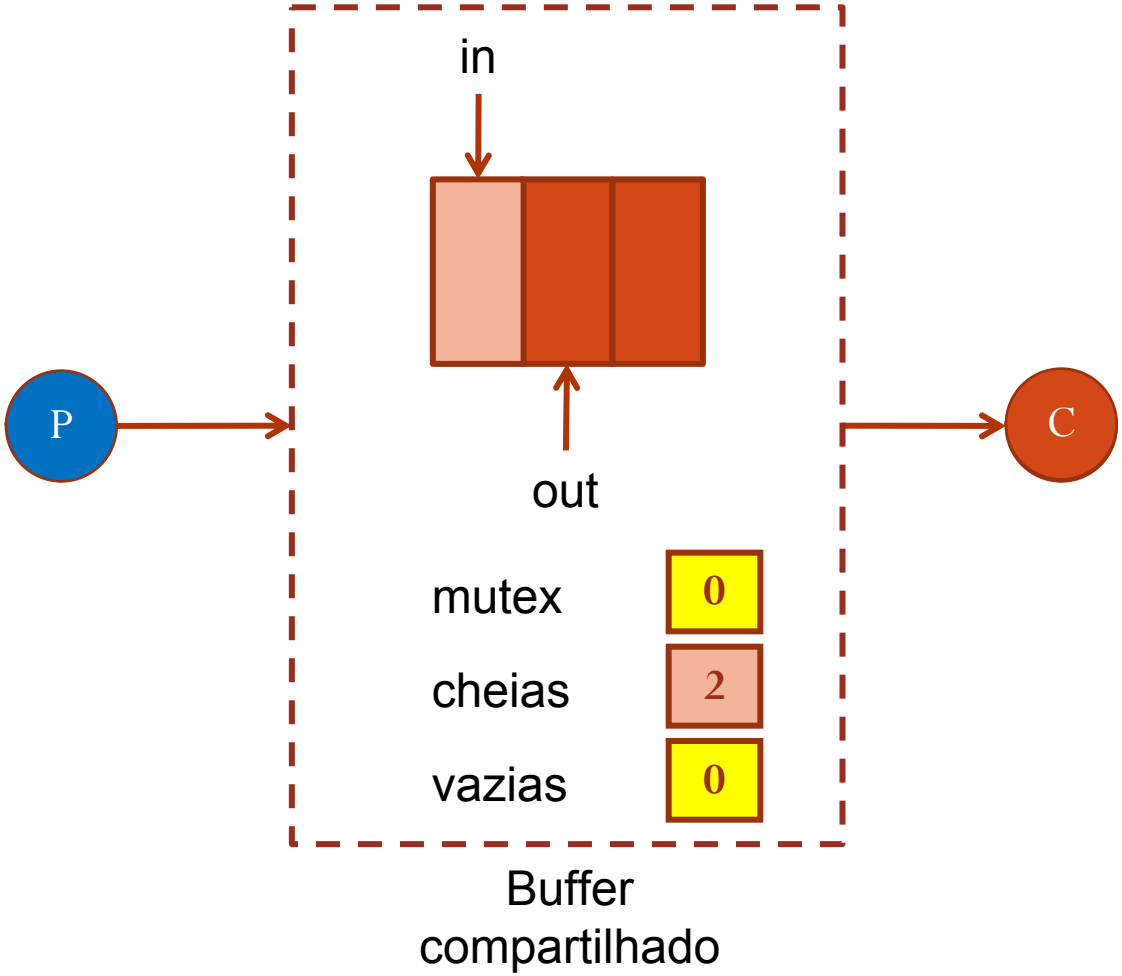
Semáforos

- Produtor / Consumidor



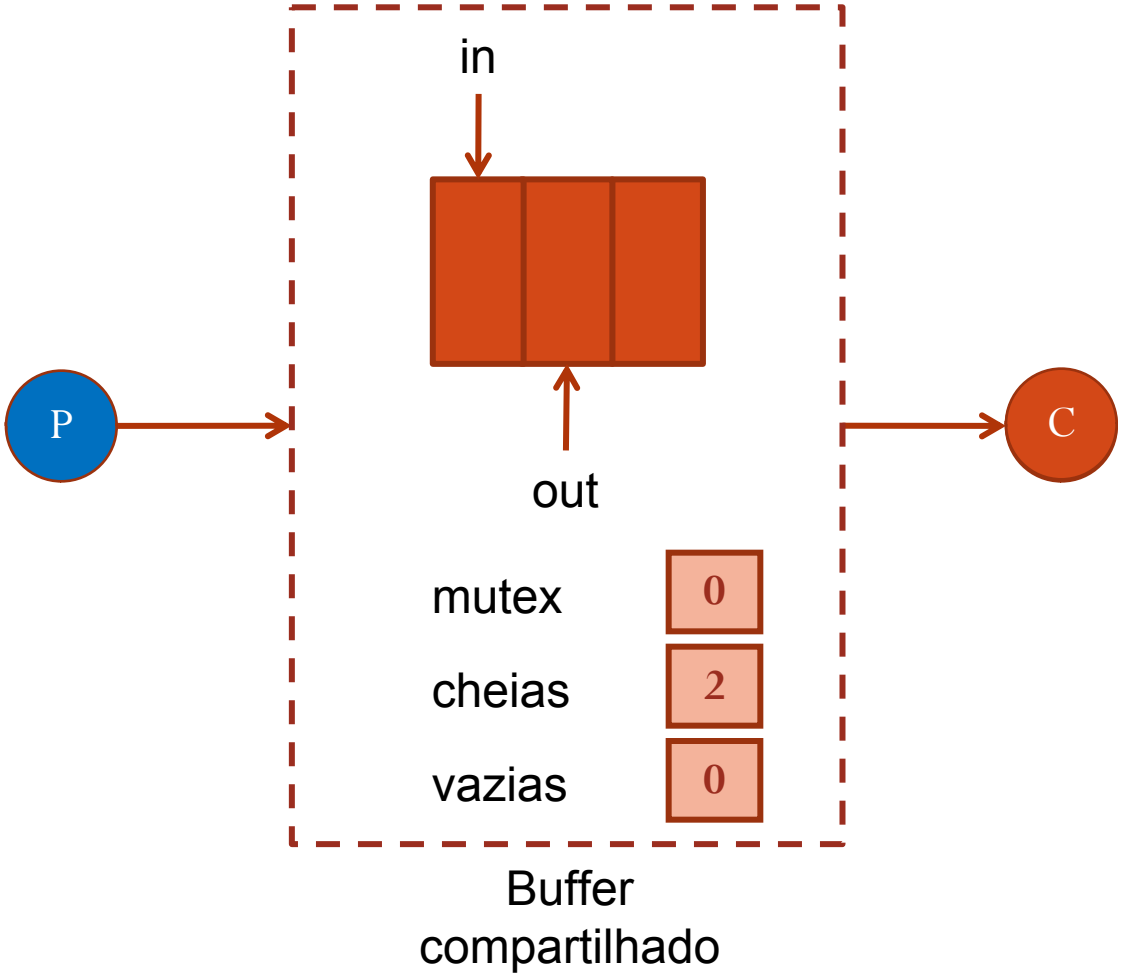
Semáforos

- Produtor / Consumidor



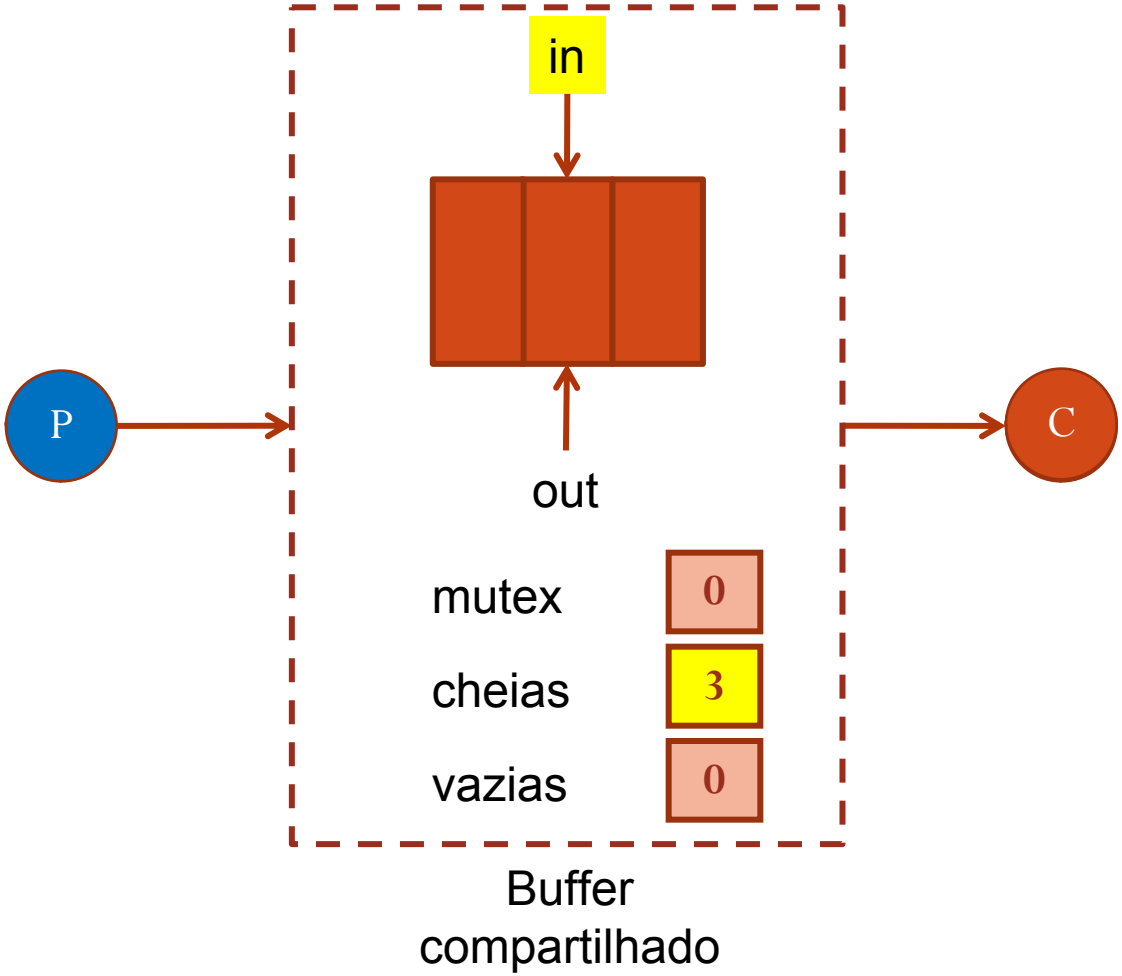
Semáforos

- Produtor / Consumidor



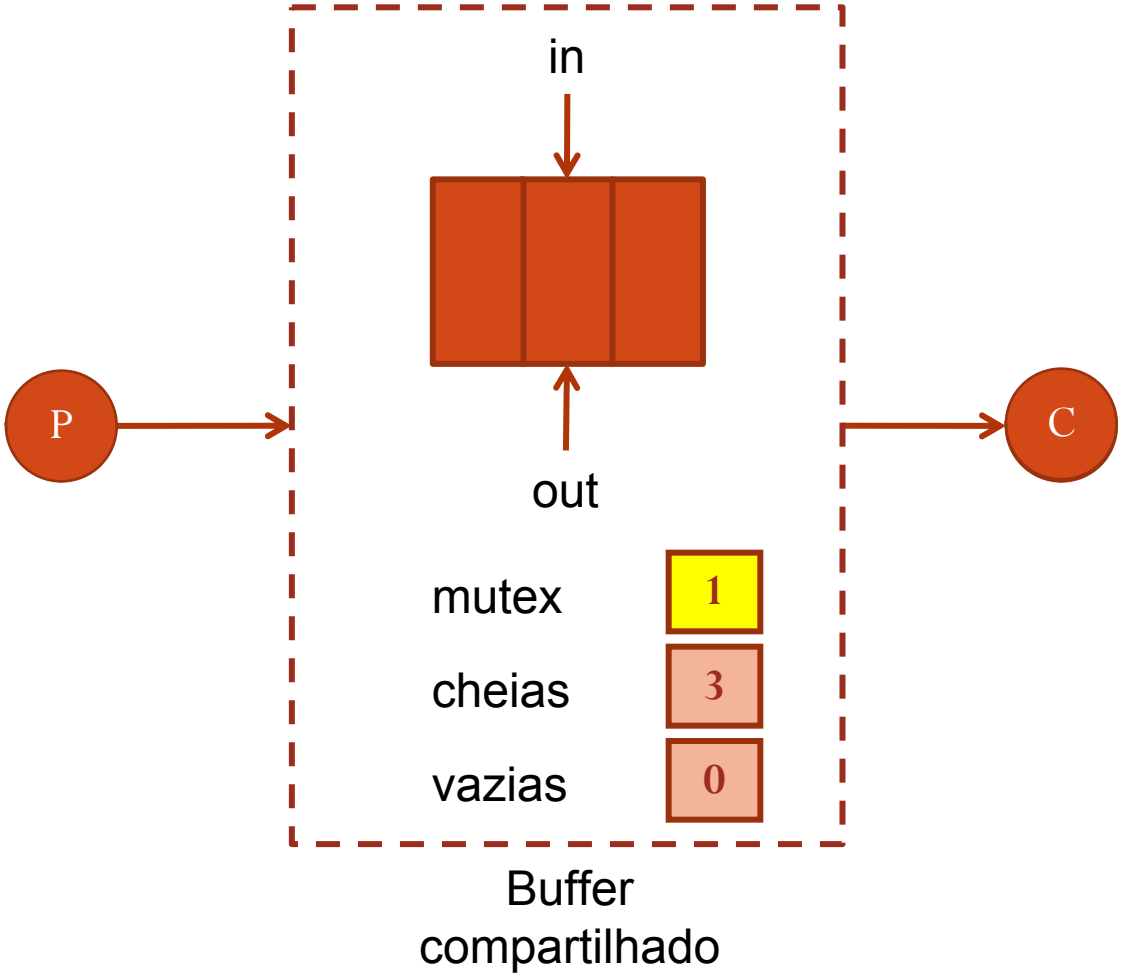
Semáforos

- Produtor / Consumidor



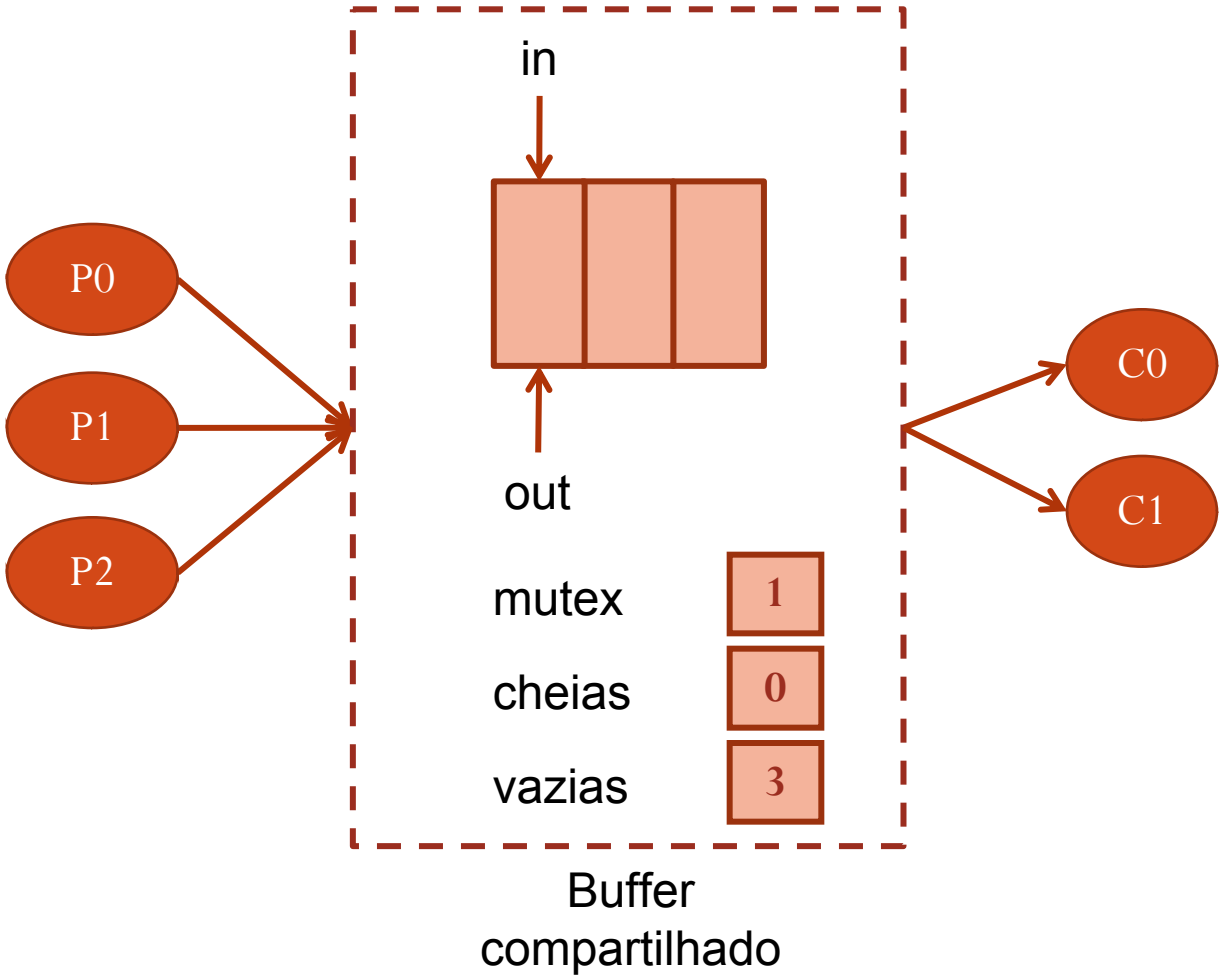
Semáforos

- Produtor / Consumidor



Semáforos

- Produtor / Consumidor (múltiplos processos/threads de cada tipo)



Semáforos

- Produtor / Consumidor – Código

Produtor

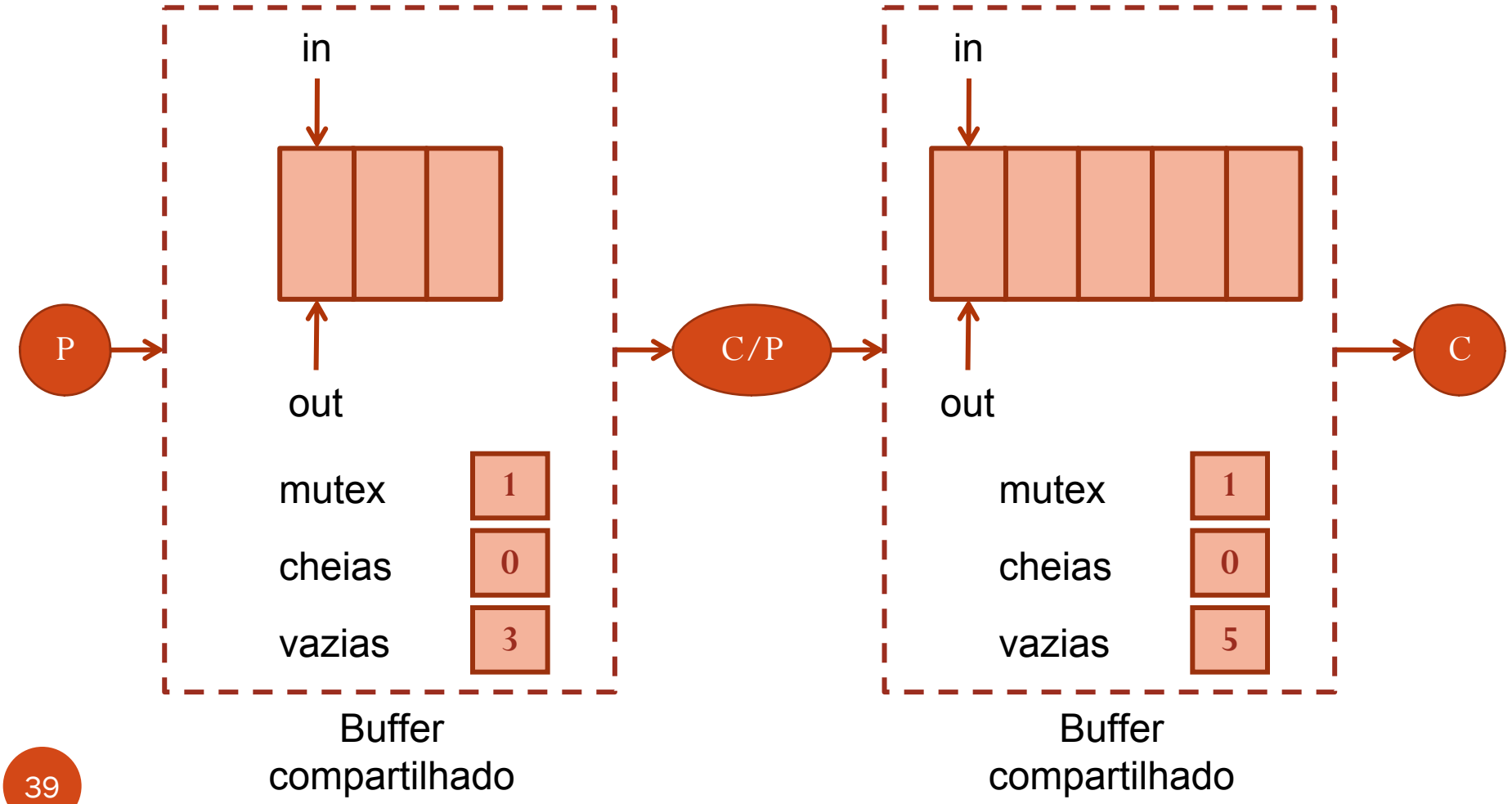
```
while (true)
{
     /* produz um item e põe em
    nextProduced */
     /* Espera por uma posição
     vazia*/
     sem_wait(&vazias);
     /* Espera acesso exclusivo ao
     buffer */
     sem_wait(&mutex);
     buf[in] = nextProduced;
     in = (in+1)%BUFF_SIZE;
     /* Libera o buffer */
     sem_post(&mutex);
     /* Incrementa posições cheias */
     sem_post(&cheias);
}
```

Consumidor

```
while (true)
{
     /* Espera por uma posição
     cheia*/
     sem_wait(&cheias);
     /* Espera acesso exclusivo ao
     buffer */
     sem_wait(&mutex);
     nextProduced=buf[out];
     out = (out+1)%BUFF_SIZE;
     /* Libera o buffer */
     sem_post(&mutex);
     /* Incrementa posições vazias*/
     sem_post(&vazias);
     /* consome o item de nextProduced */
}
```

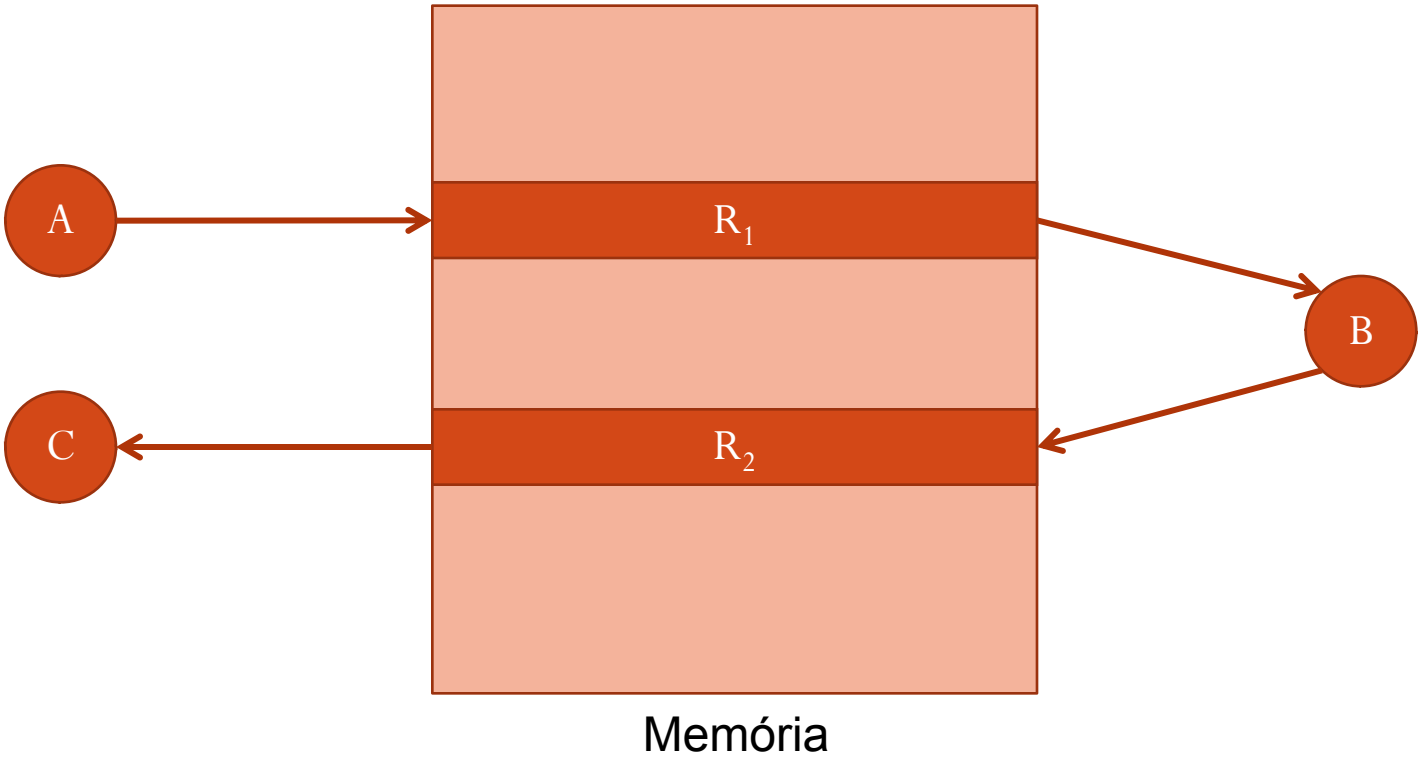
Semáforos

- Produtor – Consumidor&Produtor - Consumidor



Semáforos

- Produtor – Consumidor&Produtor - Consumidor



Semáforo – Implementação

- Com espera ocupada – Spinlock
- Ou com bloqueio