Sistemas Operativos 2

2021/22

Fundamentos de Sincronização (revisão sobre semáforos)

Produtor/Consumidor – Resolução com recurso a semáforos

1

Tópicos

Fundamentos de sincronização

Problemas e soluções

Semáforos

Aplicações e exemplos

Bibliografia específica:

- *Fundamentos de Sistemas Operativos*; 3ª Ed.; Marques & Guedes Capítulos 2
- *Operating Systems Concepts* ; Silberschatz & Galvin Capítulo 6

2

Sincronização

Este conjunto de slides foca-se no conceito de semáforos e num caso de exemplo de uso

Parte 1 - É feita uma **revisão de conceitos de base** sobre sincronização (mas baseia-se no conhecimento prévio de *mutexes*)

Parte 2 - O **exemplo dos produtores consumidores** poderá, mais tarde, ser implementado com *threads* (**local a um mesmo processo**) ou com recurso a **memória partilhada** (envolvendo **processos diferentes**), e essa questão é **independente** do que aqui é abordado

Não é abordado API específico – apenas funções esperar/assinalar genéricas (aplica-se a qualquer sistema operativo)

Sincronização – Semáforos como gestores de recursos

Parte 1

Sincronização e semáforos

Conceitos sobre **exclusão mútua**, **competição** e **cooperação**. Conceitos sobre **semáforos**

Sincronização

Motivos para sincronização

1. Sistemas compostos por mais do que uma entidade activa (processos, *threads*) que usam recursos e dados partilhados entre si exigem a coordenação do acesso a recursos e dados

2. Sistemas compostos por mais do que uma entidade activa em que uma das entidades é dependente de acontecimentos originados por outra(s) vai exigir mecanismos para coordenar a sua execução em função da execução das outras entidades

Exemplos típicos da bibliografia
- Gestores de recursos (memória partilhada), gestão de produtores/consumidores

Exemplos concretos da vida real
- Sistemas cliente servidor envolvendo vários utilizadores ou postos; programas *multi-threaded*

Sincronização

Situações típicas (exemplos) que envolvem sincronização

Cooperação
- Diversas actividades concorrem para a conclusão de uma aplicação comum
- Situação que pode ser resolvida algoritmicamente para as aplicações envolvidas com recurso a mecanismos do sistema de suspensão e sinalização

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
- É complicado e/ou ineficiente resolver esta situação sem ajuda do sistema operativo

Sincronização

Situações típicas (exemplos) que envolvem sincronização

Exclusão mútua

- A utilização concorrente de uma zona de dados (ou recurso) partilhada pode levar a que os dados fiquem inconsistentes
- A utilização desses dados (ou recurso) 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
- Este caso é muito frequente e muito importante. Está também na base da tomada de decisões críticas que podem ter um resultado incorrecto se não for devidamente acautelado

Sincronização

Abordagem às questões sobre sincronização

1 Exclusão mútua

- Pode ser vista como um caso particular de competição:
  - Competição pelo acesso exclusivo secção de código (secção crítica)
  - O problema que se pretende resolver é normalmente o acesso concorrente a dados

  O caso particular é o facto de apenas se permitir um acesso de cada vez. Também pode ser visto ao contrário: a competição é uma generalização de exclusão mútua em que se permite N acessos em simultâneo
- Muitas vezes o problema reside numa decisão que se pretende tomar enquanto uma determinada variável não muda de valor

Sincronização

Abordagem às questões sobre sincronização

2 Competição

- Pode ser visto como uma generalização da exclusão mútua:
  A competição é feita sobre recursos com mais do que uma unidade.
  Pretende simplificar a espera/sinalização de recursos. Não envolve necessariamente secções críticas
- As soluções encontradas para a exclusão mútua servem de base para as soluções para a competição

Sincronização

Abordagem às questões sobre sincronização

3 Cooperação

- Os processos querem sincronizar as suas acções de forma explícita uns com os outros. Em vez de competição e autorização para avançar, podem voluntariamente auto-suspenderem-se até receberem uma notificação
- Os mecanismos utilizados para os casos anteriores podem ser utilizados com grande simplicidade para conseguir os objectivos da cooperação

Sincronização – Exclusão mútua – Semáforos

Semáforo - Conceito

- Recurso controlado pelo SO. Semelhante a um semáforo real, mas com a capacidade de ter um "número de autorizações" (número de processos/threads que o podem ter em simultâneo)
- Constituído por uma estrutura de dados que inclui
  - Variável de controlo (inteira) = número de autorizações restantes
  - Fila de espera (de processos / threads bloqueados)

Sincronização – Exclusão mútua – Semáforos

Semáforo

- Há dois tipos conceptuais de semáforos

Semáforo binário ou MUTEX

- Tem apenas uma "autorização" – serve essencialmente para resolver situações de exclusão mútua
- Implementado em alguns sistemas como mecanismo independente do semáforo genérico

Semáforo

- Genérico. Permite *n* "autorizações"
- Usado para qualquer tipo de situação (inclusive exclusão mútua).
- Uso comum em situações de competição e controlo de recursos (por exemplo, cenário produtor/consumidor)

Sincronização – Exclusão mútua – Semáforos

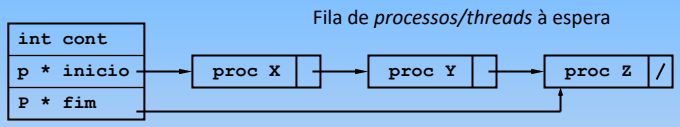
Semáforo

- As operações fundamentais sobre semáforos são
  - **Esperar** – Requisita o recurso / Requisita uma autorização / (pede acesso a uma secção crítica)
  - **Assinalar** – Liberta o recurso / Devolve uma autorização / (liberta a secção crítica)
- Se o semáforo já não tiver mais nenhuma autorização (contador a zero) o processo que efectua o esperar fica bloqueado. Posteriormente é automaticamente desbloqueado quando algum outro processo efectua assinalar / devolve uma autorização. Os "esperar" ficam memorizados e pode haver mais que um processo em espera
- O bloqueio / desbloqueio é transparente para o processo

Sincronização – Exclusão mútua – Semáforos

Semáforos

Aspecto conceptual de um semáforo



A implementação dos semáforos no núcleo garante que o teste-decisão-actualização é de carácter atómico (indivisível), através de instruções *test and set* ou similar

Sincronização – Exclusão mútua – Semáforos

Semáforos (em resumo)

- Objectivo: informar o sistema de que uma *thread/processo* está à espera de acesso a um recurso, podendo o SO bloquear a execução do processo até que o recurso esteja livre
- Método: Cada semáforo tem um contador interno. As operações sobre o semáforo são as de **esperar** (diminuir o contador) e **assinalar** (aumentar o contador). Esse contador actua como número de processos que ainda podem passar sem ficarem bloqueados.
- O sistema garante que o teste-decisão-acção interno à implementação do núcleo não é susceptível à ocorrência de comutação de *processo/thread*, seja por instruções do tipo *test and set*, seja por outros mecanismos disponíveis no núcleo

Sincronização – Exclusão mútua – Semáforos – Esperar

Lógica de funcionamento interno de um semáforo

`Esperar(Semáforo)`

- Decrementa a variável de controlo
  - Se o resultado for inferior a zero, então já outro processo está a utilizar a secção crítica guardada pelo semáforo
    - Neste caso o processo que invocou a função esperar é bloqueado, ficando na lista de espera desse semáforo
  - Caso contrário
    - A secção crítica guardada pelo semáforo estava livre. O processo avança

Os semáforos para controlo de secções de exclusão mútua são inicializados com a variável de controlo a 1 (apenas um de cada vez pode usar)

Sincronização – Exclusão mútua – Semáforos – Assinalar

Utilização de semáforos por parte dos processos

`Assinalar(Semáforo)`

- Incrementa a variável de controlo
  - Se houver processos à espera, o primeiro pode avançar
    - O primeiro processo da fila é desbloqueado e o seu estado passa a executável (não é necessariamente posto logo em execução) **(*)**
    - A secção crítica (ou recurso) guardada pelo semáforo fica novamente ocupada
  - Caso contrário
    - A secção crítica (ou recurso) guardada pelo semáforo fica livre

(*) Depende do algoritmo de escalonamento ser ou não preemptivo, da existência de prioridades, e outras políticas de escalonamento do SO em questão

Sincronização – Exclusão mútua – Secção crítica

Pode-se usar um semáforo binário (1 autorização) nas situações onde se usaria um *mutex*. No entanto o código seria menos claro e possivelmente menos optimizado

```
Semáforo sem = CriarSemaforo(1);
/* ... */
int escreve(tipo_dados valor) {
    esperar(sem);
    var_partilhada=valor;
    assinalar(sem);
}
```

Já não há espera activa

O processo bloqueia temporariamente se não puder avançar

O SO garante que apenas um (o número inicial no semáforo) processo estará aqui num dado instante

As funções genéricas

- *CriarSemaforo* (cria um semáforo – o parâmetro é o valor inicial do contador)
- *Assinalar* (operação *assinalar*)
- *Esperar* (operação *esperar*)

são simplificações das funções realmente existentes nos SO

Sincronização – Exclusão mútua – Semáforos

A utilização de semáforos leva ao bloqueio e desbloqueio dos processos/*threads* que os utilizam (isto é geralmente transparente para o algoritmo dos programas)

Bloquear um processo/*thread* envolve

- Retirá-lo de execução
- Salvaguardar o seu contexto (de hardware e de software)
- Marcar o estado do processo/*thread* como "Bloqueado"
- Colocá-lo no fim da lista de espera

Desbloquear um processo/*thread* envolve

- Retirá-lo da fila de espera de processos bloqueados onde se encontrava
- Marcar o estado do processo como "Executável"
- Colocá-lo no fim da lista de processos executáveis

  O processo será posto em execução quando o despacho o seleccionar

Mecanismos de apoio à sincronização – Algumas notas

O uso de semáforos tem as mesmas vantagens genéricas dos *mutexes*:

Bloqueia o processo (*thread*) e não ocupa o processador

- O processo pode ficar bloqueado indefinidamente porque o processador pode continuar a executar outros processos
- É um mecanismo justo
  - Existindo uma fila de espera, obtém-se a garantia de atendimento de todos mais cedo ou mais tarde -> permite evitar a situação de *starvation*
- É compatível com implementação de políticas de prioridades nos processos bloqueados

Mecanismos de apoio à sincronização – Algumas notas

O uso de semáforos tem as mesmas vantagens genéricas dos *mutexes*:

Elimina a espera activa

- As funções de manipulação dos semáforos devem pertencer ao SO para que este perceba as intenções dos processos
  - As variáveis internas do semáforo não devem ser directamente visíveis
  - Os nomes das funções não são, obviamente, esperar e assinalar, tendo em alguns sistemas um aspecto bastante mais complicado

Sincronização – Competição

Exclusão mútua X Competição

- **Exclusão mútua**: trata-se na verdade de um caso particular de competição por **um** recurso
  - O recurso é o **acesso** a uma secção de código (na verdade, **dados**)
  - **O número de unidades disponíveis desse recurso é 1**

Pode-se generalizar o conceito de exclusão mútua

- Qualquer recurso pode ser alvo de competição e não apenas secções de código
- O número de unidades disponíveis pode ser qualquer número inteiro maior que zero
- Segundo esta generalização, **os semáforos continuam a poder ser utilizados como mecanismo preferencial de gestão do recurso alvo da competição**
  - **O valor inicial do semáforo é o número inicial de unidades do recurso disponíveis**

Sincronização – Semáforos como gestores de recursos

Parte 2

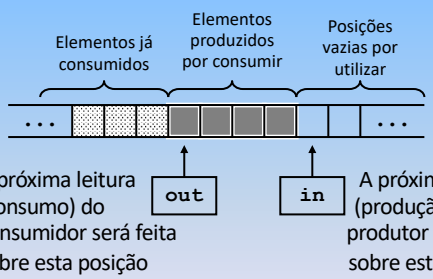
Exemplo: Produtor/Consumidor

Resolução de um problema clássico de **gestão de recursos partilhados** recorrendo exclusivamente a **mecanismos de sincronização**

Sincronização – Semáforos como gestores de recursos

Exemplo: Produtor/Consumidor

- *Buffer* infinito
- N produtores
- 1 consumidor



➢ Como fazer os produtores e consumidor sem que haja "*race condition*" (atropelamentos de consumo/produção, ou seja, **`out`** "atropelar" **`in`**) ?

Sincronização – Semáforos como gestores de recursos

Primeira "solução"

```
typedef ... item;       /* a estrutura exacta não é importante */
item buffer[∞];         /* apenas para simplificação da exposição */
∞ int in; ∞ int out;    /* "∞": infinito - não existe em C */

/* inicialização */
in = out = 0;
```

Produtor

```
while (COND) {
   item_p = produz_item();
   buffer[in] = item_p;
   in ++;
}
```

Consumidor

```
while (COND) {
   while (in <= out);
   item_c = buffer[out];
   out++;
   trata_item(item_c);
}
```


Sincronização – Semáforos como gestores de recursos

Problemas da primeira "solução"

- Existe ***espera activa*** (`while (in <= out);`) no consumidor.
- Sabendo que existem N produtores, tem-se que

  ```
  buffer[in] = item_p;
  in++;
  ```

  constitui uma ***secção crítica e não está protegida***

Como resolver estes problemas ?

- Soluções algorítmicas: complicadas e não são totalmente perfeitas
- Usar mecanismos de sincronização para gerir o acesso aos items
  - Trincos lógicos: mantém a espera activa
  - Semáforos: simples de usar e removem os problemas apontados

Sincronização – Semáforos como gestores de recursos

Produtor/Consumidor – Solução com semáforos

Semáforos necessários:

- Produtores
  - Um semáforo de exclusão mútua
    - Para o acesso simultâneo a **in**

    → `sem_mutex`
- Consumidor
  - Um semáforo para contabilizar o número de itens disponíveis
    - Impede o acesso a um item ainda não acabado de produzir (ou seja, o "atropelamento" de **in** por **out**)

    → `sem_itens`

Sincronização – Semáforos como gestores de recursos

Solução com semáforos

```
Semaforo sem_mutex, sem_itens;  // "Semaforo" já definido algures

/* inicialização */
sem_mutex = criar_semaforo(1);  // 1 -> semáforo de exclusão mútua
sem_itens = criar_semaforo(0);  // 0 itens produzidos inicialmente
```

Produtor

```
while (COND) {
    item_p = produz_item();
    esperar(&sem_mutex);
    buffer[in] = item_p;
    in ++;
    assinalar(&sem_mutex);
    assinalar(&sem_itens);
}
```

Consumidor

```
while (COND) {
    esperar(&sem_itens);
    esperar(&sem_mutex);
    item_c = buffer[out];
    out++;
    assinalar(&sem_mutex);
    trata_item(item_c);
}
```

Sincronização – Semáforos como gestores de recursos

Solução com semáforos – observações

`esperar/assinalar(&sem_mutex)` no consumidor é desnecessário

porque

- Só há um consumidor, o que significa que:

  Não há outro consumidor que, interrompendo o consumidor actualmente em execução dentro da secção crítica, possa consumir o mesmo item.

  Ou seja,

  ```
  item_c = buffer[out];
  out++;
  ```

  não é uma secção crítica considerando o consumidor isoladamente.

e

- Os produtores não manipulam a variável **`out`**, apenas a **`in`**.

e

- O semáforo **`sem_itens`** garante que nunca o consumidor e um produtor manipulam a mesma posição do buffer (isto é, **`in`** é sempre diferente de **`out`**).

Sincronização – Semáforos como gestores de recursos

Solução com semáforos - Versão final **N** produtores / **1** consumidor

```
Semaforo sem_mutex, sem_itens;  // "Semaforo" já definido algures

/* inicialização */
sem_mutex = criar_semaforo(1);  // 1 -> semáforo de exclusão mútua
sem_itens = criar_semaforo(0);  // 0 itens produzidos inicialmente
```

Produtor

```
while (COND) {
    item_p = produz_item();
    esperar(&sem_mutex);
    buffer[in] = item_p;
    in ++;
    assinalar(&sem_mutex);
    assinalar(&sem_itens);
}
```

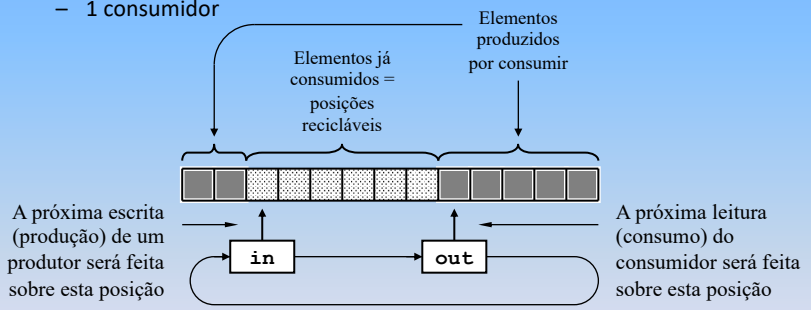
Consumidor

```
while (COND) {
    esperar(&sem_itens);
    item_c = buffer[out];
    out++;
    trata_item(item_c);
}
```

Sincronização – Semáforos como gestores de recursos

Produtor/Consumidor

- *Buffer* finito - mais realista
- N produtores
- 1 consumidor



➢ Novo problema: tem que se impedir que **`in`** "atropele" **`out`**

Ou seja, impedir que os produtores escrevem por cima de itens ainda não consumidos

Sincronização – Semáforos como gestores de recursos

➢ Como implementar um *buffer* aparentemente infinito sobre um que é finito ?

```
#define DIM ...   // tamanho do buffer
```

Em vez de

```
in++;
out++;
```

Faz-se

```
in = (in + 1) % DIM;
out = (out + 1) % DIM;
```

`in` e **`out`** variam sempre entre **`0`** e **`DIM-1`**

➢ Corresponde ao conceito de ***buffer* circular**

Sincronização – Semáforos como gestores de recursos

Produtor/Consumidor com *buffer* circular – Solução com semáforos

Continuam a ser necessários os semáforos:

sem_mutex
- De exclusão mútua (no produtor)
  - Previne o acesso simultâneo a **in**

sem_itens
- Para contabilizar o número de itens disponíveis
  - Impede o acesso a um item ainda não produzido (ou seja, o "atropelamento" de **in** por **out**)

Novo semáforo

sem_vazios
- Para contabilizar o número de elementos livres no *buffer*
  - Impede que os produtores tentem escrever um item sobre um que ainda não foi consumido (ou seja, o "atropelamento" de **out** por **in**)

Sincronização – Semáforos como gestores de recursos

Produtor/Consumidor – Versão com *buffer* circular (parte 1)

```
#define DIM ...      /* tamanho do buffer */

typedef ... item;    /* a estrutura exacta não é importante */
item buffer[DIM];    /* apenas para simplificação da exposição */
int in, out;         /* assumir que DIM cabe em "int" */

/* inicialização ("Semaforo" já definido algures) */
Semaforo sem_mutex, sem_itens, sem_vazios;

/* inicialização */
in = out = 0;
sem_mutex = criar_semaforo(1);     // 1 -> semáforo de exclusão mútua
sem_itens = criar_semaforo(0);     // 0 itens produzidos inicialmente
sem_vazios = criar_semaforo(DIM); // DIM elementos disponíveis */
```

Sincronização – Semáforos como gestores de recursos

Produtor/Consumidor – versão com *buffer* circular (parte 2)

Produtor

```
while (COND) {
    item_p = produz_item();
    esperar(&sem_vazios);
    esperar(&sem_mutex);
    buffer[in] = item_p;
    in = (in + 1) % DIM;
    assinalar(&sem_mutex);
    assinalar(&sem_itens);
}
```

Consumidor

```
while (COND) {
    esperar(&sem_itens);
    item_c = buffer[out];
    out = (out + 1) % DIM;
    assinalar(&sem_vazios);
    trata_item(item_c);
}
```

Observação: Esperar/assinalar no `sem_mutex` continua a ser desnecessário no consumidor, por isso foi omitido

Sincronização – Semáforos como gestores de recursos

Produtor/Consumidor – Nova versão

- *Buffer* finito circular
- N produtores
- M consumidores

Uma vez que existem vários consumidores, então

```
item_c = buffer[out];
out = (out + 1) % DIM;
```

passa a constituir uma secção crítica

- Pode acontecer que um consumidor interrompa outro durante o processo de extração do item
- Torna-se necessário guardar essa secção com o semáforo de exclusão mútua
- O código do produtor não sofre alterações

Sincronização – Semáforos como gestores de recursos

Produtor/Consumidor – versão **N** produtores / **M** consumidores

Consumidor

```
while (COND) {
    esperar(&sem_itens);
    esperar(&sem_mutex);
    item_c = buffer[out];
    out = (out + 1) % DIM;
    assinalar(&sem_mutex);
    assinalar(&sem_vazios);
    trata_item(item_c);
}
```


Sincronização – Semáforos como gestores de recursos

Produtor/consumidor, versão N/M – observações

➔ A ordem dos **`esperar()`** no produtor e no consumidor **é** significativa

Experiência: inverter a ordem dos **`esperar()`** no produtor. Fica:

```
esperar(&sem_mutex);
esperar(&sem_vazios);
```

Situação provável:

- Não existe, momentaneamente, nenhum elemento disponível no *buffer*, mas a secção crítica não está ocupada (nenhum processo esta a produzir ou consumir)
- Um produtor "decide" produzir um item e efectua **esperar(&sem_mutex)**
- Como não existe espaço disponível, o produtor fica bloqueado em **esperar(&sem_vazios)** até que algum consumidor consuma um item e efectue **assinalar(&sem_vazios)**
- Nenhum consumidor pode consumir itens porque fica bloqueados em **esperar(&sem_mutex)**

• **Entrou-se em interblocagem (*deadlock*)**

Sincronização – Semáforos como gestores de recursos

Produtor/consumidor, versão N/M – observações

Observa-se que:

- As secções críticas no produtor e no consumidor são distintas, **ao invés do que acontecia noutros exemplos**

Porque

- Os produtores e consumidores não manipulam as mesmas variáveis
  - Consumidor: **`out`**
  - Produtor: **`in`**

e

- O semáforo **`sem_itens`** garante que nunca um consumidor tentará utilizar a mesma posição no *buffer* que um produtor.

e

- O semáforo **`sem_vazios`** garante que nunca um produtor tentará utilizar a mesma posição no *buffer* que um consumidor.

Sincronização – Semáforos como gestores de recursos

Produtor/consumidor, versão N/M – observações (cont.)

- Os produtores podem utilizar um semáforo de exclusão mútua diferente do utilizado pelos consumidores (por razões análogas às que na primeira versão deste problema faziam com que o consumidor não precisasse de um semáforo de exclusão mútua)
  - Neste caso pode-se utilizar semáforos diferentes para cada uma das secções críticas
  - Vantagem: evita-se que um produtor esteja desnecessariamente à espera de um consumidor e vice-versa
  - Nota: este tipo de análise deve ser feito com bastante cuidado

O semáforo **`sem_mutex`** é substituído por

- **`sem_mutex_p`** – Semáforo de exclusão mútua dos produtores
- **`sem_mutex_c`** – Semáforo de exclusão mútua dos consumidores

Sincronização – Semáforos como gestores de recursos

N Produtores / M Consumidores / *buffer* circular (parte 1)

```
#define DIM ...      /* tamanho do buffer */

typedef ... item;    /* a estrutura exacta não é importante */
item buffer[DIM];    /* apenas para simplificação da exposição */
int in, out;         /* assumir que DIM cabe em "int" */

/* inicialização ("Semaforo" já definido algures) */
Semaforo sem_mutex_p, sem_mutex_c, sem_itens, sem_vazios;

/* inicialização */
in = out = 0;
sem_mutex_p = criar_semaforo(1);  // 1 -> semáforo de exclusão mútua
sem_mutex_c = criar_semaforo(1);  // 1 -> semáforo de exclusão mútua
sem_itens = criar_semaforo(0);    // 0 itens produzidos inicialmente
sem_vazios = criar_semaforo(DIM); // DIM elementos disponíveis
```


Sincronização – Semáforos como gestores de recursos

Produtor/Consumidor – versão com *buffer* circular (parte 2)

Produtor

```
while (COND) {
    item_p = produz_item();
    esperar(&sem_vazios);
    esperar(&sem_mutex_p);
    buffer[in] = item_p;
    in = (in + 1) % DIM;
    assinalar(&sem_mutex_p);
    assinalar(&sem_itens);
}
```

Consumidor

```
while (COND) {
    esperar(&sem_itens);
    esperar(&sem_mutex_c);
    item_c = buffer[out];
    out = (out + 1) % DIM;
    assinalar(&sem_mutex_c);
    assinalar(&sem_vazios);
    trata_item(item_c);
}
```

Obs.: Aqui a questão da ordem dos **esperar**() nem sequer se apresenta pois são semáforos independentes (cada caso é um caso; é necessário uma análise cuidadosa do problema). No entanto deve ser mantida a ordem apresentada por ser a mais lógica (a mais correcta).

Sincronização – Semáforos como gestores de recursos

Recomendações para a utilização correcta de semáforos

- Distinguir entre semáforos para exclusão mútua e semáforos para gestão de recursos
- Determinar correctamente o valor inicial dos semáforos
  - 1 para semáforos de exclusão mútua
  - N >= 0 para semáforos de gestão de recursos
- Identificar correctamente as secções críticas
  - Verificar se duas secções aparentemente distintas não são, na verdade, uma só que deve ser guardada pelo mesmo semáforo. O importante são os **dados acedidos serem ou não os mesmos**, e não a "igualdade" ou não das linhas de código
  - Verificar se secções críticas que aparentemente são a mesma não são, na verdade, independentes, podendo ser guardadas por semáforos distintos (ganho potencial em eficiência). Em caso de dúvida, deve-se optar pela situação mais segura (o mesmo semáforo)

Sincronização – Cooperação entre processos/threads

Competição vs cooperação

Na competição

- Os processos competem por recursos.
- A sincronização é concretizada pelo SO para manter a coerência
- A sincronização é feita de modo transparente para os algoritmos dos processos
  - Em princípio, os processos não se apercebem que ficaram bloqueados

Na cooperação

- Os vários processos colaboram para a conclusão de um objectivo
- A sincronização passa a ser explícita: gerida pelo algoritmo
- Usam-se mecanismos de sincronização pelas suas propriedades de sinalização (por exemplo, para situações onde também se poderiam usar sinais – tais como: eventos)

Tipicamente, consiste em bloquear um processo até que lhe seja assinalada a ocorrência de um "evento"