Sistemas Operativos: Concorrência

Pedro F. Souto (pfs@fe.up.pt)

March 16, 2012

Sumário

Sumário

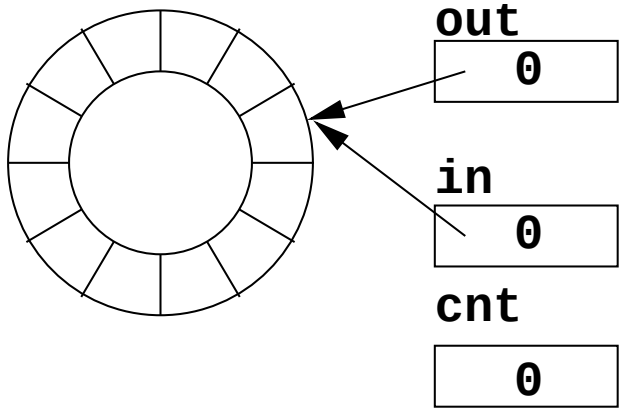
Interferência

- Em sistemas concorrentes, um processo (ou *thread*) pode estar sujeito à interferência de outros processos (ou *threads*) com os quais partilha estruturas de dados **e**

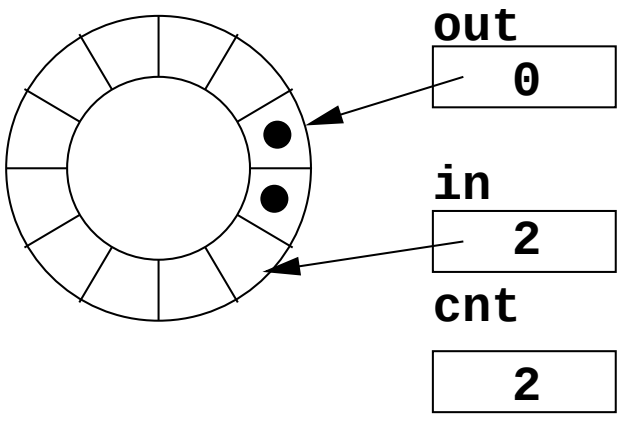
  pelo menos um dos acessos a essas estruturas é de escrita.
- Um exemplo clássico onde este problema pode ocorrer é o problema do produtor/consumidor:
  - o *thread* produtor "produz" estruturas de dados;
  - o *thread* consumidor "consome" essas estruturas de dados.
- O servidor de *Web* com múltiplos *threads* é um dos muitos exemplos práticos deste tipo de problema:
  - o *dispatcher* recebe os pedidos dos clientes e passa-os aos *workers* para os processarem.

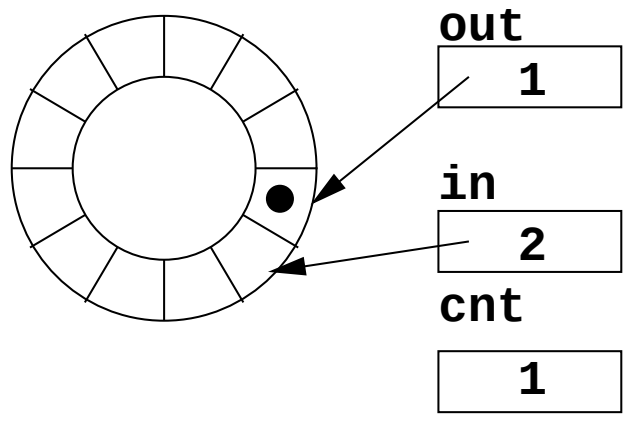
Problema do Produtor/Consumidor

- Implementação usando um vector circular:



- 2 inserções ... seguidas por uma remoção





Solução de *Bounded Buffer* em C (1/2)

Quando a comunicação entre o produtor e o consumidor é feita através dum *buffer* de capacidade finita, o problema do produtor/consumidor é designado por bounded buffer problem.

```
#define BUF_SIZE     100
typedef struct {
    int cnt, in, out;
    void *buffer[BUF_SIZE];
} bbuf_t;
void enter(bbuf_t *bbuf_p, void *obj_p) {
    assert( cnt < BUF_SIZE );
    bbuf_p->buffer[bbuf_p->in] = obj_p;
    bbuf_p->in = (bbuf_p->in + 1) % BUF_SIZE;
    bbuf_p->cnt++;
}
```

Solução do *Bounded Buffer* em C (2/2)

```
void *remove(bbuf_t *bbuf_p) {
    void *obj_p;
    assert( cnt > 0 );
    obj_p = bbuf_p->buffer[bbuf_p->out];
    bbuf_p->out = (bbuf_p->out + 1) % BUF_SIZE;
    bbuf_p->cnt--;
    return obj_p;
}
```

- O produtor deverá usar `enter()`, enquanto que o consumidor deverá usar `remove()`.

Actualização do Número de Elementos

- Actualização do número de objectos no *buffer* (`cnt`):

  Pelo produtor:

```
LD   (D),A
INC  A
ST   A,(D)
```

  Pelo consumidor:

```
LD  (D), A
DEC A
ST A,(D)
```

- Onde:
  - `A` e `D` são registos do CPU;
  - `(D)` designa o conteúdo da posição de memória apontada por `D`.

Race Condition

- O produtor e o consumidor executam “em simultâneo”, i.e. concorrentemente, num PC com um único CPU.
- Por causa da comutação de *threads*, a ordem de execução das instruções pode ser:

```
LD   (D),A
INC  A
                                  LD  (D), A
                                  DEC A
                                  ST  A,(D)
ST   A,(D)
```

- O valor de `cnt` passa a estar incorrecto.
- Este tipo de situação, na qual a correcção da execução dum segmento de código depende da ordem da execução dos *threads*, designa-se por *race condition*.

Desenvolvimento de Aplicações Concorrentes

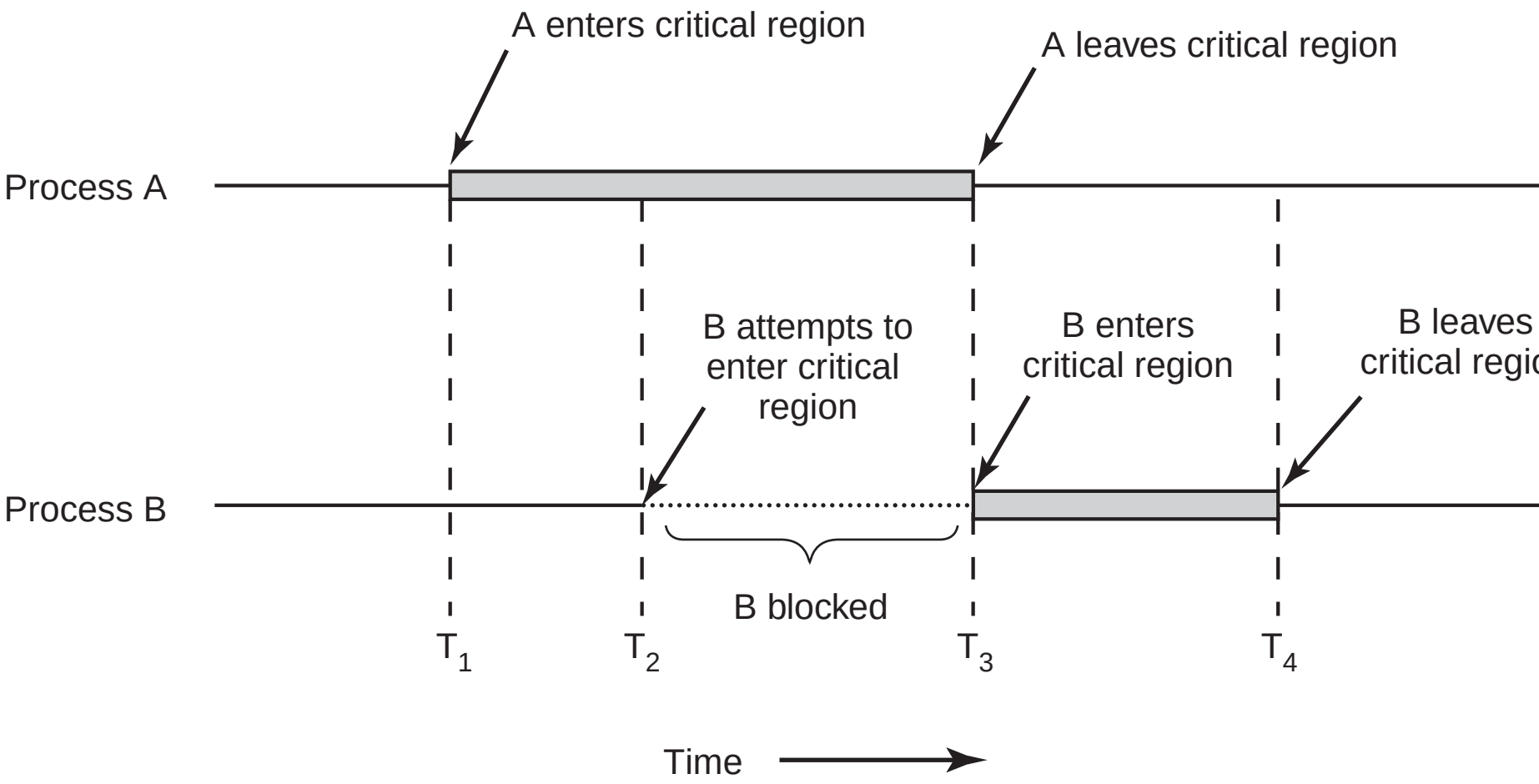
- Uma tarefa fundamental no desenvolvimento de aplicações concorrentes consiste:
  - na identificação de possíveis condições de competição;
  - na eliminação dessas condições de competição.
- Para eliminar condições de competição é necessário garantir exclusão mútua no acesso a variáveis partilhadas modificáveis.
  - Isto aplica-se a quaisquer objectos (p.ex. ficheiros) partilhados.
- Encontrar condições de competição que escaparam na concepção e na codificação é extremamente difícil:

  quase todos os testes terminam com sucesso, mas uma vez num milhão ocorrem coisas inexplicáveis.

Secções Críticas

- Um segmento de código em que um *thread* acede a um objecto partilhado modificável designa-se por secção crítica.
- Secções críticas "aparecem" em conjuntos, tipicamente associados a objectos partilhados:
  - secções críticas dentro do mesmo conjunto não podem ser executadas "simultaneamente";
  - secções críticas pertencentes a conjuntos diferentes, podem ser executadas "simultaneamente".

Secções Críticas



- As secções críticas devem ser tão grandes quanto o necessário para garantir a correcção do programa, mas não maiores.

Secções Críticas: Critérios para uma Solução

- Tanenbaum sugere que uma solução para o problema das condições de competição deverá satisfazer as seguintes condições:
  1. Não mais do que um *thread* poderá estar numa secção crítica (dum dado conjunto).
  2. A solução não deverá depender da velocidade relativa de execução dos *threads*.
  3. *Threads* que pretendem entrar numa secção crítica não devem ser bloqueados por *threads* que não executem dentro duma secção crítica (desse conjunto).
  4. Nenhum *thread* deverá esperar indefinidamente para entrar na sua secção crítica.
- Mas, nem sempre estes são os critérios mais apropriados.

Sumário

Secções Críticas: *SW-only non-solution*

- Uma não-solução ingénua baseada em programação é:

```
while(busy); // wait if busy
busy = 1;    // keep out other processes
...          // critical section
busy = 0;    // leaving critical section
...          // non-critical section
```

- Obviamente, há uma condição de competição no acesso a **`busy`**.

Secções Críticas: *SW-only solution*

- Podem encontrar uma solução (de Peterson) para o caso de 2 processos em livros sobre SO.
- Há também várias soluções para *n* processos: entre as mais eficientes, a mais simples parece ser de Lamport (*bakery algorithm*):
  - O algoritmo baseia-se no uso de números de forma semelhante ao usado para atendimento de clientes (daí o seu nome).
  - Requer que os diferentes processos tenham identificadores distintos, para resolver "empates".
- Soluções baseadas em *HW* são mais simples e eficientes.

Sumário

Inibição de Interrupções

- Cada *thread* deverá:
  - inibir interrupções imediatamente antes de entrar na secção crítica; e
  - permiti-las imediatamente depois de sair.
- Por exemplo, na actualização de **`cnt`** do problema produtor/consumidor:

```
DI                      DI
LD   (D),A              LD (D), A
INC A                   DEC A
ST   A,(D)              ST A,(D)
EI                      EI
```

- Com as interrupções inibidas não há comutação entre *threads*, e consequentemente não há possibilidade de condições de competição.

Inibição de Interrupções: Problemas

- Enquanto um *thread* estiver numa secção crítica, o CPU não pode responder a pedidos de interrupção:

  E se o thread se "esquecer" de permitir interrupções de novo?

- Inibição/permissão de interrupções é realizada através de *instruções protegidas*.
- Um único conjunto de secções críticas (SC).
- Não funciona em sistemas multiprocessador (ver à frente).
- Inibir interrupções:
  - não é uma solução apropriada ao nível da aplicação;
  - mas é uma técnica frequentemente usada no núcleo (*kernel*) dum sistema operativo.

Sumário

Locks

- Um *lock* pode ser "definido" em C pelo seguinte *tipo abstracto*:

```
typedef struct {
  unsigned short locked;
  struct thrd *waitq; /* queue of waiting threa
} lock_t;
void unlock(lock_t *lock_p);
void lock(lock_t *lock_p);
```

- `lock()` bloqueia o *thread* e insere-o na fila de *threads* `waitq`, se `locked == 1`, senão altera o valor de `locked` para `1`.
- `unlock()` desbloqueia um dos *threads* (em geral, o primeiro) da fila `waitq`, se algum, senão altera o valor de `locked` para `0`.

Locks: algumas questões pertinentes

Problema Mas, e `lock()` e `unlock()` não têm secções críticas?

Resposta Sim, mas podemos inibir interrupções.

Pergunta Mas, e agora já não há esquecimentos?

Resposta Se `lock()` e `unlock()` forem implementadas como chamadas ao sistema (*system calls*), não há problemas.

- E o código de `unlock()` e `lock()` é curto, pelo que não deve afectar significativamente o tempo de resposta a interrupções.

Locks: Exemplo

- Consideremos o problema do *bounded buffer* de novo:

```
[...]
lock(lp);
cnt++;
lock(lp);
[...]
```

```
[...]
lock(lp);
cnt--;
unlock(lp);
[...]
```

- E se nos esquecermos de invocar `unlock()`?
  - :) Boas notícias primeiro: as interrupções não ficam inibidas.
  - :(Más notícias depois: um ou mais *threads* poderão bloquear, possivelmente indefinidamente, apesar de não haver qualquer *thread* na secção crítica.
- E se nos esquecermos de invocar `lock()`?
  - Há a possibilidade de ocorrer uma *race condition*.

Locks (Mutexes) em *libpthread*

- Um mutex é uma variável do tipo **`pthread_mutex_t`**

```
#include <pthread.h>
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

- Funções que operam sobre mutexes:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- `pthread_mutex_trylock()` tenta fazer o *lock*, mas não bloqueia o *thread* que o executa se o *mutex* já estiver no estado *locked*, ao contrário de `pthread_mutex_lock()`.
- Um *mutex* tem que ser inicializado antes de ser usado.

Sumário

Leitura Adicional

Sistemas Operativos

- Secções: 5.1 a 5.5 (excepto subsecções 5.4.2, 5.4.3 e 5.5.3)

Modern Operating Systems, 2nd. Ed.

- Subsecções 2.3.1, 2.3.2 e 2.3.3 (excepto último parágrafo: *The TSL Instruction*)

Operating Systems Concepts

- Secções 6.1 e 6.2