Sistemas Operativos: Concorrência

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

March 29, 2011

Sumário

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

Sumário

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

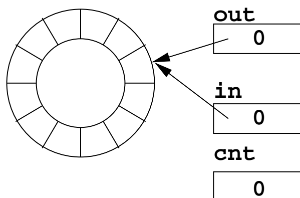
Interferência

- 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**

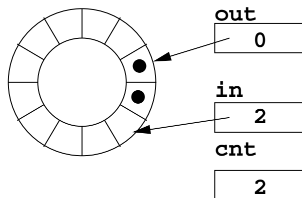
  pelo menos um dos acessos a essas estruturas é de escrita.
- Um exemplo clássico onde este problema pode ocorrer é o problema do produtor/consumidor:
  - o *thread* produtor "produz" estruturas de dados;
  - o *thread* consumidor "consome" essas estruturas de dados.
- O servidor de *Web* com múltiplos *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.

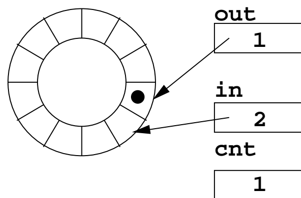
Problema do Produtor/Consumidor

- 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

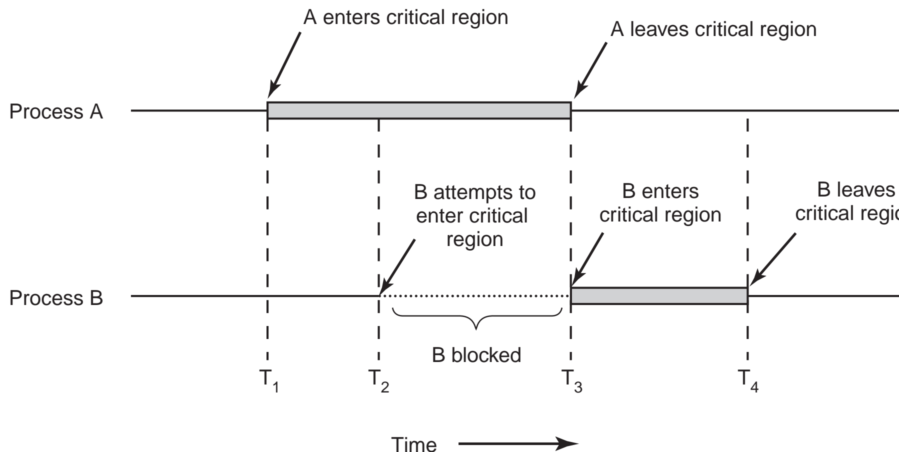
- Uma tarefa fundamental no desenvolvimento de aplicações concorrentes consiste:
  - na identificação de possíveis condições de competição;
  - na eliminação dessas condições de competição.
- Para eliminar condições de competição é necessário garantir exclusão mútua no acesso a variáveis partilhadas 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

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

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

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

Inibição de Interrupções

- Cada *thread* deverá:
  - inibir interrupções imediatamente depois de entrar na secção crítica; e
  - permiti-las imediatamente antes 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

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

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

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

Instruções *read-modify-write* Atómicas

- A inibição de interrupções é solução só para sistemas monoprocessador:

  a instrução de inibição de interrupções só afecta o processador que executa essa instrução.

- Sistemas multiprocessador requerem apoio adicional do *hardware*:

  através de instruções do tipo read-modify-write atómicas (bloqueiam o acesso ao barramento por outros processadores).

- Uma instrução típica é **`atomic-swap`**, (**`XCHG`** in IA32)

  permuta o valor dum registo com o duma posição de memória -a execução de `atomic-swap` é atómica.

spin-locks: Exemplo de Uso de `xchg`

```
spin_lock:
   mov register, #1
   xchg register, lock   | copy lock to register
                         |  and set it to 1
   cmp register, #0      | was lock zero?
   jnz spin_lock         | if it was non-zero,
                         |  loop (lock was set)
   ret                   | return to caller;
unlock:
   mov lock, #0          | store a 0 in lock;
   ret                   | return to caller
```

- Se outro processo já estiver na secção crítica, o processo que pretende entrar fica a testar `lock` continuamente, i.e., fica em espera activa (*busy waiting*).

Spin-locks: Exemplo

- Consideremos o problema do produtor/consumidor de novo:

```
[...]
spin_lock();
cnt++;
unlock();
[...]
```

```
[...]
spin_lock();
cnt--;
unlock();
[...]
```

- O uso de *spin-locks* justifica-se pelo custo associado à comutação de processos.
- O uso de *spin-locks* em sistemas mono-processador raramente faz sentido. Porquê?

Sumário

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

O Problema do *lost wakeup() (1/2)*

- O *thread dispatcher* do servidor de *Web* poderia incluir o seguinte código:

```
lock();
while(bbuf_p->cnt == BUF_SIZE) { /* Busy wait *
    unlock();
    lock();
}
enter(bbuf_p, (void *)req_p); /* Enter request
unlock();                     /* in buffer */
```

- Para evitar *espera activa*, o SO pode oferecer o par de chamadas ao sistema: `sleep()` e `wakeup()`.

O Problema do *lost wakeup()* (2/2)

- Para evitar desperdiçar o tempo do CPU, poderia usar-se:

```
lock();
while(bbuf_p->cnt == BUF_SIZE) {
    unlock();
    sleep(bbuf_p);          /* Block thread */
    lock();
}
enter(bbuf_p, (void *)req_p);
unlock();
```

- Para desbloquear o *dispatcher*, os *worker threads* executariam:

```
req_p = (req_t *)remove(bbuf_p);
if(bbuf_p->cnt == BUF_SIZE - 1) /* Buffer was ful
    wakeup(bbuf_p);      /* Wakeup dispatcher threa
```

- Este código tem uma *race condition* (lost wakeup) entre a aplicação e o SO, que pode bloquear o *dispatcher* para sempre. Qual é?

Sumário

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

Semáforos (*semaphores*)

- Semáforos são *objectos* de sincronização que podem ser usados para:
  - garantir exclusão mútua (como *locks*);
  - sincronização sem *busy waiting*.
- Um *semáforo* pode ser “definido” em C pelo seguinte tipo abstracto:

```
typedef struct {
  long int sem;        /* Counting variable */
  struct thrd *waitq; /* Queue of waiting thread
} sem_t;
void down(sem_t *sem_p);
void up(sem_t *sem_p);
```

Semáforos (*semaphores*): Semântica

- `down()` testa o valor de `sem`. Se for positivo, decrementa o seu valor e retorna. Senão, o *thread* é inserido na fila de *threads* `waitq` e bloqueia.
- `up()` desbloqueia um dos *threads* (tipicamente o primeiro) na fila `waitq`, se algum, senão incrementa o valor de `sem`.
- Os métodos `down()` e `up()` são atómicos e executados em secções críticas.
- O valor inicial do campo `sem` depende do problema em causa.

BoundedBuffer com *Semáforos* (1/3)

```
typedef struct {
   int in, out;
   sem_t mutex;          /* Semaphore initialized to 1
   sem_t slots;          /* Counter of empty slots */
   sem_t items;          /* Counter of items */
   void *buf[BUF_SIZE];
} bbuf_t;
void enter(bbuf_t *bbuf_p, void *obj_p) {
   down(&(bbuf_p->slots)); /* wait for some empty s
   down(&(bbuf_p->mutex)); /* keep other threads ou
   bbuf_p->buf[bbuf_p->in] = obj_p;
   bbuf_p->in = (bbuf_p->in + 1) % BUF_SIZE;
   up(&(bbuf_p->items));   /* update # of items */
   up(&(bbuf_p->mutex));   /* let other threads in
}
```

BoundedBuffer com *Semáforos*(2/3)

```
void *remove(bbuf_t *bbuf_p) {
    void *obj_p;
    down(&(bbuf_p->items)); /* wait for some items *
    down(&(bbuf_p->mutex)); /* keep other threads ou
    obj_p = bbuf_p->buf[bbuf_p->out];
    bbuf_p->out = (bbuf_p->out + 1) % BUF_SIZE;
    up(&(bbuf_p->slots));    /* update # of empty sl
    up(&(bbuf_p->mutex));    /* let other threads in
}
```

Bounded Buffer com Semáforos (3/3)

- O semáforo `mutex` é usado para assegurar exclusão mútua;
- Os semáforos `slots` e `items` são usados para sincronização:
  - `slots` conta o número de posições vazias: o produtor tem que esperar pelo consumidor, se esse número for zero;
  - `items` conta o número de items: o consumidor tem que esperar pelo produtor, se esse número for zero.
- Note-se que semáforos permitem sincronização:
  - sem espera activa (*busy waiting*);
  - nem *race conditions*.
- A ordem de execução das operações `down()` é fundamental para evitar bloqueio mútuo (*deadlock*).

Semáforos em *libpthreads*

- Um *semáforo* é uma variável do tipo `sem_t`:

```
#include <semaphore.h>
sem_t items;
```

- Funções que operam sobre *semáforos*:

```
int sem_init(sem_t *sem, int pshared, unsigned int va
int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);     /* down */
int sem_trywait(sem_t *sem); /* down, no wait */
int sem_post(sem_t *sem);     /* up */
int sem_getvalue(sem_t *sem, int *sval); /* use? */
```

- Um semáforo tem que ser inicializado antes de ser usado através de `sem_init()`:
  - se `pshared == 0`, o semáforo não pode ser partilhado por diferentes processos (em Linux, tem que ser 0).

Sumário

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

Mecanismos/Primitivas de Sincronização

- Instruções:
  - de inibição/permissão de interrupções;
  - *read-modify-write* atómicas;

  são mecanismos fornecidos ao nível do *HW* (CPU).
- *Locks* e semáforos são mecanismos:
  - suportados pelo SO através de chamadas ao sistema;
  - implementados usando aquelas instruções.
- O desenvolvimento de programas usando estas primitivas não é fácil:
  - O programador tem que as usar correctamente.

Monitores

- Um *monitor* é um mecanismo de linguagem de programação, enquanto que *locks* e *semáforos* são mecanismos fornecidos pelo sistema operativo.
- Um monitor é muito semelhante a um tipo abstracto (ou classe/objecto). Inclui:
  - variáveis ou estruturas de dados;
  - funções ou procedimentos.
- O único meio dum processo aceder às variáveis dum monitor é através da invocação das funções desse monitor.
- Apenas um processo pode estar *activo num monitor* em qualquer instante.
- A linguagem Java, que suporta múltiplos *threads*, usa um mecanismo semelhante ao conceito de monitor.

`class BoundedBuffer` usando Monitores: não-solução

```
class BoundedBuffer implements Monitor{
   private int cnt, in, out;
   Object[] buffer;
   public boolean empty() { // cnt cannot be
      return (cnt == 0);    // accessed outside
   }                        // the monitor
   public boolean full() {  // idem
      return (cnt == buffer.length);
   }
   public void enter(Object item) {
      buffer[in] = item;
      in = (in + 1) % buffer.length;
      cnt++;
   }
   public Object remove() {
      // left as an exercise to the reader
```

- Esta classe pode conduzir a *race-conditions*.

Monitores: mais uma tentativa ...

```
class BoundedBuffer implements Monitor{
   int cnt, in, out;
   Object[] buffer;
   public void enter(Object item) {
      while( cnt == buffer.length);  // wait if ful
      buffer[in] = item;
      in = (in + 1) % buffer.length;
      cnt++;
   }
   public Object remove() {
      Object o;
      while( cnt == 0);   // wait if empty buffer
      o = buffer[out];
      out = (out + 1) % buffer.length;
      cnt--;
      return o;            // missing 2 }
```

- Qual é o problema agora?

`class BoundedBuffer` usando Monitores: Ahá!!!

```
class BoundedBuffer implements Monitor{
   int cnt, in, out;
   Object[] buffer;
   public boolean enter(Object item) { // returns f
      if (cnt == buffer.length)         // if full
         return false;
      buffer[in] = item;
      in = (in + 1) % buffer.length;
      cnt++;
      return true;
   }
   public Object remove() {             // returns n
      Object o;                         // if empty
      // left as an exercise to the reader
```

- Os *threads* agora esperam fora do monitor: se a condição testada falhar, têm que sair e voltar a entrar.

Sumário

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

Variáveis de Condição (*Condition Variables*)

- A exclusão mútua na execução de métodos dum monitor, não é suficiente. Processos necessitam sincronizar:
  p.ex. na classe `BoundedBuffer`, `enter()` não pode inserir um item se o *buffer* estiver cheio.
- *Busy waiting* é sempre uma possibilidade, mas ...
- A solução é o uso de condições (*condition variables*), as quais são *objectos* com 2 operações:
  - wait() : bloqueia o processo que a executa;
  - signal() : desbloqueia um processo que executou a primitiva `wait()` sobre a mesma condição (se houver mais do que um processo, apenas um deles será desbloqueado).

BoundedBuffer: Monitores e Variáveis de Condição (1/2)

```
class BoundedBuffer implements Monitor{
   int cnt, in, out;
   Condition items, slots;          // condition variable
   Object[] buffer;
   public void enter(Object o) {   // eventually enters
                                   //  the object
      if (cnt == buffer.length)
         slots.wait();             // wait for free slots
      buffer[in] = o;
      in = (in + 1) % buffer.length;
      cnt++;
      if (cnt == 1)                // the buffer is not
         items.signal();           //  empty anymore
   }
```

BoundedBuffer: Monitores e Variáveis de Condição (2/2)

```
    public Object remove() {          // eventually
                                      //  returns an objec
      Object o;
      if (cnt == 0)                   // buffer empty,
        items.wait();                 //   wait for some i
      o = buffer[out];
      out = (out + 1) % buffer.length;
      cnt--;
      if (cnt == buffer.length - 1) // the buffer is not
        slots.signal();               //   full anymore
      return o;
    }
  }
```

Variáveis de Condição e Monitores

- **Problema:** se um processo executa a instrução `signal()` sobre uma condição com pelo menos um processo bloqueado, passará a haver mais de um processo activo no monitor.
- **Soluções alternativas:**
  - Hoare: o processo que executa `signal()` bloqueia, se houver algum processo bloqueado nessa condição;
  - Brinch Hansen: o processo que executa `signal()` tem que sair imediatamente do monitor (ver exemplo acima).
- Variáveis de condição não são contadores como semáforos:
  - Se uma condição for assinalada sem que qualquer processo esteja à espera, a sinalização não terá qualquer efeito.
- Por que razão `wait()/signal()` não sofrem dum problema semelhante ao *lost wakeup()*?

Variáveis de Condição em *libpthread* (1/4)

- Uma *variável de condição* é uma variável do tipo `pthread_cont_t`:

```
#include <pthread.h>
pthread_mutex_t bb_m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t bb_c = PTHREAD_COND_INITIALIZER;
```

- Em *libpthreads* uma variável de condição **tem** que ser usada em associação com um *mutex*:
  - o *mutex* assegura exclusão mútua na execução duma secção crítica;
  - a variável de condição evita/limita *busy waiting*.
- O uso do *mutex* evita o problema do *lost wakeup*.

Variáveis de Condição em *libpthread* (2/4)

- Funções que operam sobre *variáveis de condição*:

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
     pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
     pthread_mutex_t *mutex, ...);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- Uma *variável de condição* tem que ser inicializada antes de ser usada através de :
  - quer `PTHREAD_COND_INITIALIZER` (ver acima);
  - quer `pthread_cond_init()` (em Linux, o argumento `cond_attr` tem que ser NULL).

Variáveis de Condição em *libpthread* (3/4)

- `pthread_cond_wait()`:
  - faz o `unlock()` do *mutex* especificado no argumento `mutex`, o qual deverá ter sido *adquirido* previamente usando `pthread_mutex_lock()` e bloqueia o *thread* **atomicamente**;
  - quando o *thread* é desbloqueado, adquire de novo o `mutex`, conforme especificado em `pthread_mutex_lock()`.
- `pthread_cond_timedwait(cond, mutex, abstime)` é semelhante a `pthread_cond_wait()` excepto que o *thread* será desbloqueado no instante especificado em `abstime`, se a condição ainda não tiver sido assinalada.

Variáveis de Condição em *libpthread* (4/4)

- `pthread_cond_signal()` desbloqueia um *thread* bloqueado na variável de condição especificada, se algum.
- `pthread_cond_broadcast()` desbloqueia todos os *threads* bloqueados na variável de condição especificada, se algum.
- `pthread_cond_signal()` e `pthread_cond_broadcast()` podem ser invocados fora de qualquer secção crítica.
- A invocação de `pthtread_cond_wait()` deve ocorrer num ciclo `while`:

```
while( bbuf_p->cnt == 0 )
    pthread_cond_wait(&(bbuf->items),
                                &(bbuf->mutex));
```

Bounded Buffer com *condition variables*

```
#define BUF_SIZE     100
typedef struct
   int cnt, in, out;
   pthread_mutex_t mutex;
   pthread_cond_t  items, slots;
   void *buffer[BUF_SIZE];
 bbuf_t;
void enter(bbuf_t *bbuf_p, void *obj_p)
   pthread_mutex_lock(&(bbuf_p->mutex));
   while( bbuf_p->cnt == BUF_SIZE )
      pthread_cond_wait(&(bbuf_p->slots),
                             &(bbuf_p->mutex));
   bbuf_p->buffer[bbuf_p->in] = obj_p;
   bbuf_p->in = (bbuf_p->in + 1) % BUF_SIZE;
   bbuf_p->cnt++;
   if( cnt == 1 )
      pthread_cond_signal(&(bbuf_p->items));
   pthread_mutex_unlock(&(bbuf_p->mutex));
```

Primitivas de Sincronização em *libpthreads*

- *libpthread* suporta as seguintes primitivas de sincronização:

  locks: designados por *mutexes*, de *mutual exclusion*;
  semáforos;
  variáveis de condição, as quais têm que ser usadas em associação com *mutexes*.

- Como estas primitivas de sincronização deverão ser partilhadas por vários *threads*, devem ser
  - ou declaradas como variáveis globais;
  - ou membros das variáveis partilhadas, se se usar tipos abstractos (ou objectos) como no *bounded buffer problem*.

Sumário

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

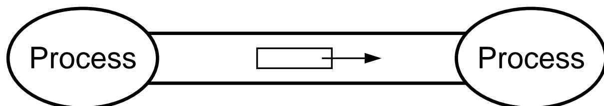
Leitura Adicional

Comunicação (IPC) com *Mensagens*

Problema Os mecanismos de sincronização considerados (*locks*, semáforos, monitores e variáveis de condição) não podem ser usados em sistemas onde *threads*/processos não partilham memória – p.ex. sistemas distribuídos.

Solução Usar *mensagens*:

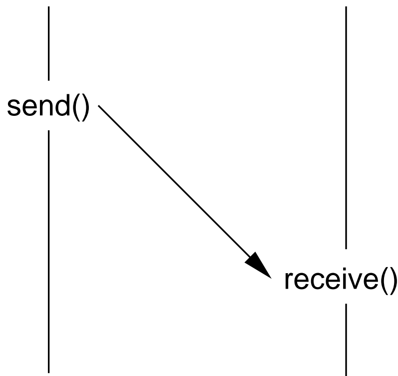
processos/*threads* enviam/recebem mensagens através dum canal de comunicação:



Primitivas para Comunicação com *Mensagens*

- Essencialmente, IPC com mensagens faz uso de 2 primitivas:

```
send(destination, &message)
receive(source, &message)
```



Semântica e Aplicação de IPC com Mensagens

- Embora genericamente a ideia seja simples, há muitas variantes possíveis:

  identificação: como identificar as extremidades do canal de comunicação?

  sincronização: os *threads* comunicantes sincronizam entre si?

  bufferização: o SO armazena as mensagens?

  propriedades do canal: perde mensagens? duplica-as?

- IPC com mensagens é estudada em redes ou em sistemas distribuídos.
- Pode-se usar mensagens em sistemas não distribuídos:
  - em sistemas paralelos (MPI *Message Passing Interface*);
  - entre processos/*threads* executando no mesmo CPU.

Sumário

Race Conditions e Secções Críticas

Exclusão Mútua via Programação

Exclusão Mútua via Inibição de Interrupções.

Locks

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Mensagens

Leitura Adicional

Leitura Adicional

- Secções 2.2 e 2.3 de *Modern Operating Systems*, 2nd Ed.
- Secções 5.1 a 5.5, 6.1, 6.2, 6.3, 6.5, 6.6.1 e 6.7 de José Alves Marques e outros, *Sistemas Operativos*, FCA - Editora Informática, 2009
- Outra documentação (transparências e enunciados dos TPs): `http://web.fe.up.pt/~pfs/aulas/so1011/`