Sistemas Operativos: Concorrência (Parte 2)

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

March 23, 2012

Sumário

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Leitura Adicional

Sumário

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

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 AL, #1
    xchg AL, _lock  | copy _lock to AL register
                    |  and set it to 1
    cmp AL, #0      | was _lock zero?
    jnz spin_lock   | if it was non-zero,
                    |  loop (lock was set)
    ret             | else 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();             spin_lock();
cnt++;                   cnt--;
unlock();                unlock();
[...]                    [...]
```

- O uso de *spin-locks* justifica-se pelo custo associado à comutação de processos.
  - Se a secção crítica for “longa”, será vantajoso bloquear um processo que tente fechar um *lock* já fechado
- O uso de *spin-locks* em sistemas mono-processador raramente faz sentido. Porquê?
  - Contudo, o uso de instruções *read-modify-write* atómicas ainda é vantajoso comparado com o uso da inibição de interrupções. Porquê?

Sumário

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

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?

Sinais e o Problema do *Lost Wakeup*

- Em Linux pode usar-se

  `sigsuspend()`

  `kill()`

  em vez de `sleep()` e `wakeup()`
- Uma alternativa a `sigsuspend()` é usar `pause()`
  - Contudo, a solução resultante sofre do problema do *lost wakeup*
- Para evitar o problema do *lost wakeup* deverá bloquear-se os sinais. (Exercício para o leitor.)
  - E isso não é possível fazer com `pause()`

Sumário

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

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).

Outras Definições de Semáforos

Semáforos binários O valor da variável de contagem não pode ser superior a 1

Questão Qual o efeito dum `signal()` sobre um semáforo binário com valor 1?

Resposta Várias soluções possíveis:

1. O processo bloqueia
2. O valor do semáforo não é afectado (semelhante a um mutex)

Processo desbloqueado Soluções possíveis:

FCFS Os processos são inseridos numa fila, e desbloqueados por ordem

Não determinista Os processos são inseridos num conjunto, e o processo desbloqueado é escolhido aleatoriamente

API do Unix System V

- Concebida para a sincronização de processos (não *threads*) partilhando memória.
- Generalização das operações sobre semáforos

  `up/down` Permitem somar/subtrair um valor diferente de 1

  Conjuntos de semáforos É possível realizar operações sobre conjuntos de semáforos de forma atómica. P. ex.:
  - Decrementar 2 semáforos de valores diferentes, ou
  - Incrementar 1 semáforo e decrementar outro

  Estas operações tornam mais simples e eficientes alguns problemas de sincronização entre processos.

Sumário

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

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
   }
}
```

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
   }
}
```

`class BoundedBuffer` usando Monitores: Ahá!!!

```
class BoundedBuffer implements Monitor{
   int cnt, in, out;
   Object[] buffer;
   public boolean enter(Object item) { // returns false,
      if (cnt == buffer.length)        // if full
         return false;
      buffer[in] = item;
      in = (in + 1) % buffer.length;
      cnt++;
      return true;
   }
   public Object remove() {            // returns null,
      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

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

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).
- 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.

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;
    }
}
```

- Por que razão `wait()/signal()` não sofrem dum problema semelhante ao *lost wakeup()*?

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:**

  Signal and Exit (Brinch Hansen): ao executar `signal()` o processo sai imediatamente do monitor, i.e. `signal()` implicitamente retorna do monitor;

  Signal and (Urgent) Wait (Hoare):
  - O processo que executa `signal()` bloqueia, se houver algum processo bloqueado nessa condição.
  - Assim que este último sair do monitor, ou executar `wait()`, o processo que o desbloqueou continua.

  Signal and Continue (Lampson & Redell):
  - O processo que executa `signal()` continua.
  - O processo desbloqueado tem que "entrar" de novo no monitor e verificar se a condição continua válida

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

Instruções read-modify-write Atómicas.

Problema do Lost Wakeup

Semáforos

Monitores

Variáveis de Condição

Leitura Adicional

Leitura Adicional

Sistemas Operativos

- Subsecções 5.4.2, 5.4.3
- (Sub)Secções 6.1, 6.2.2, 6.5, 6.6.1, 6.7 (6.7.1 e 6.7.4 não será avaliada)

Modern Operating Systems, 2nd. Ed.

- Subsecção 2.3.3 (último parágrafo: *The TSL Instruction*)
- Subsecções 2.3.4, 2.3.5, 2.3.6, 2.3.7

Operating Systems Concepts, 7th. Ed.

- Secções 6.4, 6.5, 6.7