Sincronizzazione dei Processi

Sistemi Operativi

Antonino Staiano
Email: antonino.staiano@uniparthenope.it

Introduzione (cont.)

- Semafori
- Monitor

Semafori

Semaforo. Una variabile intera condivisa con valori non negativi che può essere soggetta alle sole operazioni che seguono:

1. Inizializzazione (specificata come parte della sua dichiarazione)
2. Operazioni indivisibili *wait* e *signal* (o post)

```
procedure wait(S)
begin
     while S<=0
          do {nothing};
     S:=S-1;
end;
```

```
procedure signal(S)
begin
     S:=S+1;
end;
```

Semantica delle operazioni wait e signal su un semaforo (spinlock)

- Anche chiamati *semafori contatori* per le operazioni su S

Implementazione dei Semafori (NO valori negativi)

```
wait(semaphore *S) {

   if (S->value > 0)
         S->value--;
   else {
         aggiungi P a S->list;
         block();
        }
}
```

```
signal(semaphore *S) {

   if (qualche P bloccato su S){
      togli P da S->list;
      wakeup(P);
   }
   else
        S->value++;
}
```

```
typedef struct{
   int value;
   struct process *list;
   } semaphore;
```

Uso dei Semafori nei Sistemi Concorrenti

Use	Description
Mutual exclusion	Mutual exclusion can be implemented by using a semaphore that is initialized to 1. A process performs a *wait* operation on the semaphore before entering a CS and a *signal* operation on exiting from it. A special kind of semaphore called a *binary semaphore* further simplifies CS implementation.
Bounded concurrency	Bounded concurrency implies that a function may be executed, or a resource may be accessed, by n processes concurrently, $1 \leq n \leq c$, where c is a constant. A semaphore initialized to c can be used to implement bounded concurrency.
Signaling	Signaling is used when a process P_i wishes to perform an operation a_i only after process P_j has performed an operation a_j. It is implemented by using a semaphore initialized to 0. P_i performs a *wait* on the semaphore before performing operation a_i. P_j performs a *signal* on the semaphore after it performs operation a_j.

Uso: Mutua Esclusione

```
var sem_CS : semaphore := 1;
Parbegin
    repeat                              repeat
        wait (sem_CS);                      wait (sem_CS);
        { Critical Section }                { Critical Section }
        signal (sem_CS);                    signal (sem_CS);
        { Remainder of the cycle }          { Remainder of the cycle }
    forever;                            forever;
Parend;
end.
        Process P_i                         Process P_j
```

Figure 6.23 CS implementation with semaphores.

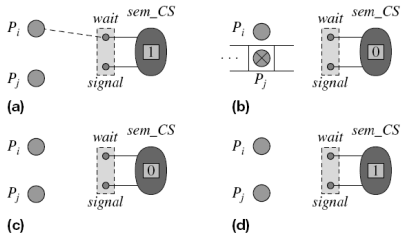


Figure 6.24 Snapshots of the concurrent system of Figure 6.23.

Uso: concorrenza limitata

- Fino a ***c*** processi possono eseguire concorrentemente op_i
- Implementata inizializzando un semaforo ***sem_c*** a ***c***
- Ogni processo che intende eseguire op_i
  - fa una *wait (sem_c)* prima di eseguire op_i e
  - una ***signal (sem_c)*** dopo averla eseguita

Uso: Segnalazione tra Processi

```
var sync : semaphore := 0;
Parbegin
    . . .                               . . .
    wait (sync);                        { Performaction a_j }
    { Performaction a_i }               signal (sync);
Parend;
end.
        Process P_i                         Process P_j
```

- Non possono verificarsi race condition poiché le operazioni *wait* e *signal* sono indivisibili
- *Semaforo binario*: può solo avere i valori 0 e 1

Semafori: invariante

- Siano
  - **nP** il numero di operazioni **wait** completare
  - **nV** il numero di operazioni **signal** completate
  - **init** il valore iniziale del semaforo

- Vale il seguente invariante:

$$nP \leq nV + init$$

- Distinguiamo die casi:
  - Eventi (**init =0**)
    - Il numero di attese dell'evento deve essere non superiore al numero di volte che l'evento si è verificato
  - Risorse (**init > 0**)
    - Il numero di richieste soddisfatte non deve essere superiore al numero iniziale di risorse + il numero di risorse da restituire

Semafori binari

Definizione

variante dei semafori in cui il valore può assumere solo i valori 0 e 1

Uso

servono a garantire mutua esclusione, semplificando il lavoro del programmatore hanno lo stesso potere espressivo dei semafori interi

Invariante

$0 \leq nV + init - nP \leq 1$

$0 \leq s.value \leq 1$

```
waitB(semaphore *S){
 if (S->value = 1)
      S->value = 0
 else{
      aggiungi P a S->list;
          block();
   }
```

```
signalB(semaphore *S){
 if S->list vuota
      S->value = 1;
 else {
        togli P da S->list;
        wakeup(P);
      }
```

Osservazione:
- la differenza è solo concettuale! Implementazione dei semafori generale (contatori)

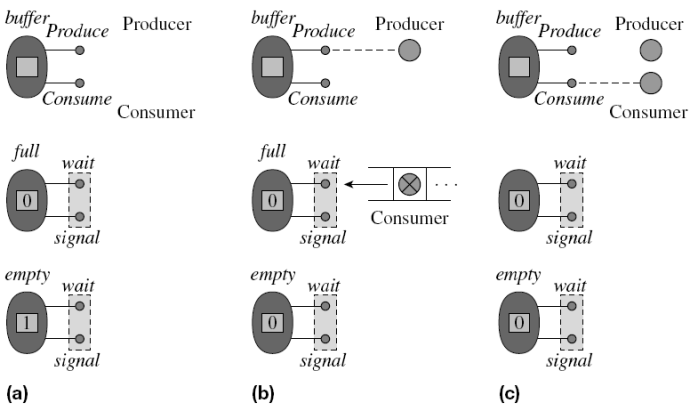
Produttori-Consumatori con Semafori

Produttori-consumatori con buffer singolo

```
type    item = . . .;
var
        full  : Semaphore := 0; { Initializations }
        empty : Semaphore := 1;
        buffer : array [0] of item;
begin
Parbegin
   repeat                                repeat
        wait (empty);                         wait (full);
        buffer [0] := . . .;                  x := buffer [0];
            { i.e., produce }                     { i.e., consume }
        signal (full);                        signal (empty);
        { Remainder of the cycle }            { Remainder of the cycle }
   forever;                              forever;
Parend;
end.
            Producer                              Consumer
```

- Evita l'attesa attiva poiché i semafori sono usati per controllare se i buffer sono pieni o vuoti
- La concorrenza totale nel sistema è 1

Esempio: Produttori-Consumatori a Buffer Singolo con Semafori



Esempio: Produttore-Consumatore ad n Buffer con Semafori

- Soluzione per il problema produttore-consumatore ad n-buffer

```
const   n = . . .;
type    item = . . .;
var
        buffer : array [0..n – 1] of item;
        full : Semaphore := 0; { Initializations }
        empty : Semaphore := n;
        prod_ptr, cons_ptr : integer;
begin
        prod_ptr := 0;
        cons_ptr := 0;
Parbegin
  repeat                                  repeat
    wait (empty);                           wait (full);
    buffer [prod_ptr] := . . .;             x := buffer [cons_ptr];
        { i.e. produce }                        { i.e. consume }
    prod_ptr := prod_ptr + 1 mod n;         cons_ptr := cons_ptr + 1 mod n;
    signal (full);                          signal (empty);
    { Remainder of the cycle }              { Remainder of the cycle }
  forever;                                forever;
Parend;
end.
        Producer                                Consumer
```


Lettori-Scrittori con Semafori

Soluzione rifinita per i lettori-scrittori

```
Parbegin
  repeat                                  repeat
    if runwrite ≠ 0                         if runread ≠ 0 or
    then                                        runwrite ≠ 0
        { wait };                           then { wait };
    { read }                                { write }
    if runread = 0 and                      if totread ≠ 0 or totwrite ≠ 0
        totwrite ≠ 0                        then
    then                                        activate either one waiting writer
        activate one waiting writer             or all waiting readers
  forever;                                forever;
Parend;
        Reader(s)                               Writer(s)
```

- Significato dei contatori
  - runread: numero di lettori in lettura
  - totread: numero di lettori che intendono leggere o in lettura
  - Similmente runwrite e totwrite

Lettori-Scrittori (preferenza ai Lettori) con Semafori

per la mutua esclusione → `sem_CS : semaphore := 1;`

per la segnalazione ← `reading, writing : semaphore := 0;`

```
var
        totread, runread, totwrite, runwrite : integer;
        reading, writing : semaphore := 0;
        sem_CS : semaphore := 1;
begin
        totread := 0;
        runread := 0;
        totwrite := 0;
        runwrite := 0;
Parbegin
  repeat                                  repeat
    wait (sem_CS);                          wait (sem_CS);
    totread := totread + 1;                 totwrite := totwrite + 1;
    if runwrite = 0 then                    if runread = 0 and runwrite = 0 then
        runread := runread + 1;                 runwrite := 1;
        signal (reading);                       signal (writing);
    signal (sem_CS);                        signal (sem_CS);
    wait (reading);                         wait (writing);
    { Read }                                { Write }
    wait (sem_CS);                          wait (sem_CS);
    runread := runread–1;                   runwrite := runwrite–1;
    totread := totread–1;                   totwrite := totwrite–1;
    if runread = 0 and                      while (runread < totread) do
        totwrite > runwrite                 begin
        then                                    runread := runread + 1;
            runwrite := 1;                      signal (reading);
            signal (writing);               end;
    signal (sem_CS);                        if runread = 0 and
  forever;                                  totwrite > runwrite then
                                                runwrite := 1;
                                                signal (writing);
                                            signal (sem_CS);
                                          forever;
Parend;
end.
        Reader(s)                               Writer(s)
```


Implementazione dei Semafori

Type declaration for Semaphore

```
type
    semaphore =   record
                    value : integer;    { value of the semaphore }
                    list : . . .        { list of blocked processes }
                    lock : boolean;     { lock variable for operations on this semaphore }
                  end;
```

Procedures for implementing wait and signal operations

```
procedure wait (sem)
begin
    Close_lock (sem.lock);
    if sem.value > 0
        then
            sem.value := sem.value–1;
            Open_lock (sem.lock);
        else
            Add id of the process to list of processes blocked on sem;
            block_me (sem.lock);
end;

procedure signal (sem)
begin
    Close_lock (sem.lock);
    if some processes are blocked on sem
        then
            proc_id := id of a process blocked on sem;
            activate (proc_id);
        else
            sem.value := sem.value + 1;
    Open_lock (sem.lock);
end;
```

Implementazioni di *wait* e *signal*:
- Kernel-Level: mediante system call
- User-Level: System calls solo per bloccare/attivare
- Ibrido: combinazione dei due

Implementazione dei Semafori (**SI valori negativi**)

```
wait(semaphore *S) {
   S->value--;
   if (S->value < 0) {
      aggiungi P a S->list;
      block();
   }
}
```

```
signal(semaphore *S) {
   S->value++;
   if (S->value <= 0) {
      togli P da S->list;
      wakeup(P);
   }
}
```

```
typedef struct{
   int value;
   struct process *list;
   } semaphore;
```


I Monitor

- Rappresentano una primitiva di alto livello di supporto alla stesura di programmi concorrenti corretti
  - Brinch Hansen (1973), Hoare (1974)
- Un Monitor è una raccolta di dati, strutture dati e procedure raggruppate in un modulo o pacchetto
  - I processi possono invocare le procedure di un Monitor in ogni momento
  - Non possono accedere ai dati di un Monitor se non attraverso le procedure del Monitor
  - In un Monitor può essere attivo un solo processo alla volta
    - Un Monitor implementa la mutua esclusione

```
monitor monitor-name
{
  // shared variable
  declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…)
  { … }
  }
}
```

// Simile ad una classe C++ o Java

Monitor: definizione formale

- Un Monitor definisce un lock (implicito) e zero o più variabili di condizione per gestire l'accesso concorrente ai dati
  - Usa il lock per assicurare che solo un processo è attivo nel monitor in ogni momento
  - Il lock fornisce anche la mutua esclusione per i dati condivisi
  - Le variabili di condizione permettono ai processi di bloccarsi nella sezione critica, rilasciando il loro lock e, allo stesso tempo, bloccando il processo
- Operazioni del Monitor
  - Incaspula i dati condivisi da proteggere
  - Acquisisce il lock all'inizio
  - Opera sui dati condivisi
  - Rilascia il lock temporaneamente se non può completare
  - Riacquisisce il lock appena può continuare
  - Rilascia il lock alla fine

I Monitor: variabili di condizione

- Implementano la mutua esclusione (ci pensa il compilatore), ma
  - come è possibile bloccare i processi quando non possono entrare?
- **Variabili di condizione**
  - Variabile a cui è associata una condizione del monitor
  - Sono definite due funzioni sulla variabile di condizione
    - **cond_wait** e **cond_signal**

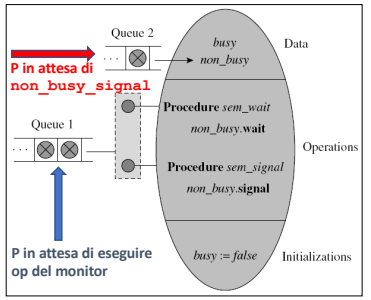
- **`condition x;`**
  - **`x.cond_wait()`** un processo che la invoca si blocca fino ad una **`x.cond_signal()`**
  - Il lock mantenuto dal processo è rilasciato atomicamente quando processo si blocca
  - **`x.cond_signal()`** risveglia un processo (se esiste) che ha invocato **`x.cond_wait()`**
    - Se non ci sono state **`x.cond_wait()`** sulla variabile, non ha effetto

Monitor: regole per le variabili di condizione

- Dopo una **`cond_signal`** è necessario avere solo un processo attivo nel monitor
  - Abbiamo bisogno di una regola per decidere chi risvegliare
- **Hoare**
  - Eseguiamo il processo appena risvegliato, sospendendo l'altro (che ha invocato **`cond_signal`**)
- **Brinch Hansen**
  - Il processo che invoca **`cond_signal`** deve uscire immediatamente
    - **`cond_signal`** può apparire solo come istruzione finale di una procedura del monitor
  - Concettualmente più semplice e più facile da implementare
  - Dopo la **`cond_signal`**, lo scheduler seleziona uno solo dei processi in attesa
- Alternativa (**Mesa**)
  - Il processo, P, che invoca **`cond_signal`** continua
  - Il processo in attesa comincia dopo che P è uscito dal monitor

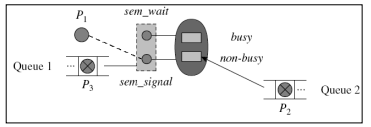
Semaforo Binario con Monitor

```
type Sem_Mon_type = monitor
  var
     busy:boolean;
     non_busy: condition;
  procedure sem_wait;
  begin
        if busy = true then non_busy.cond_wait;
        busy := true;
  end
  procedure sem_signal;
  begin
        busy := false;
        non_busy.cond_signal;
  end
  begin { initialization }
        busy := false;
  end
```



Monitor per il semaforo binario

```
var binary_sem : Sem_Mon_type;
begin
Parbegin
repeat                    repeat                    repeat
  binary_sem.sem_wait;      binary_sem.sem_wait;      binary_sem.sem_wait;
  { Critical Section }      { Critical Section }      { Critica lSection }
  binary_sem.sem_signal;    binary_sem.sem_signal;    binary_sem.sem_signal;
  { Remainder of            { Remainder of            { Remainder of
    the cycle }               the cycle }               the cycle }
forever;                  forever;                  forever;
Parend;
end.
Process P1                Process P2                Process P3
```



P_1 accede in SC, P_2 cerca di eseguire sem_wait, P_3 prova ad eseguire sem_wait prima che P_1 finisca di eseguire sem_signal

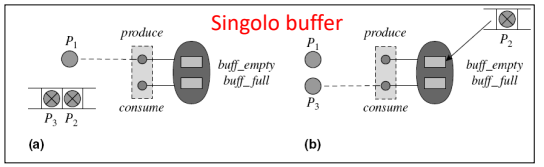
Esempio: Produttori-Consumatori con Monitor

```
type Bounded_buffer_type = monitor
    const
        n = . . .;                          { Number of buffers }
    type
        item = . . .;
    var
        buffer : array [0..n–1] of item;
        full, prod_ptr, cons_ptr : integer;
        buff_full : condition;
        buff_empty : condition;
    procedure produce (produced_info : item);
    begin
        if full = n then buff_empty.wait;
        buffer [prod_ptr] := produced_info;          { i.e., Produce }
        prod_ptr := prod_ptr + 1 mod n;
        full := full + 1;
        buff_full.signal;
    end;
    procedure consume (for_consumption : item);
    begin
        if full = 0 then buff_full.wait;
        for_consumption := buffer[cons_ptr];         { i.e., Consume }
        cons_ptr := cons_ptr + 1 mod n;
        full := full–1;
        buff_empty.signal;
    end;
    begin { initialization }
        full := 0;
        prod_ptr := 0;
        cons_ptr := 0;
end;
```

Attendere quando no buffer vuoti

Per attendere quando no buffer pieni

Esempio: Produttori-Consumatori con Monitor




```
begin
var B_buf : Bounded_buffer_type;

Parbegin
  var info : item;          var info : item;          var area : item;
  repeat                    repeat                    repeat
     info := . . .             info := . . .             B_buf.consume (area);
     B_buf.produce (info);     B_buf.produce (info);     { Consume area }
     { Remainder of            { Remainder of            { Remainder of
       the cycle }               the cycle }               the cycle }
  forever;                  forever;                  forever;
Parend;
end.
     Producer P1               Producer P2               Consumer P3
```

Problema del barbiere addormentato

- In un negozio di barbiere c'è
  - Un barbiere
  - N sedie per i clienti in attesa
  - 1 poltrona da barbiere
- Specifiche
  - Se non ci sono clienti, il barbiere si addormenta sulla poltrona
  - Quando arriva, un cliente sveglia il barbiere e si fa tagliare i capelli sulla poltrona
  - Se arriva un cliente mentre il barbiere sta tagliando i capelli ad un altro cliente, il cliente attende su una delle sedie libere
  - Se tutte le sedie sono occupate, il cliente, contrariato, se ne va!

Problema del barbiere addormentato (cont.)

```
   sem_CS: semaphore := 1;
   clienteDisponibile, poltrona : semaphore := 0;
   inAttesa : integer;
   SEDIE : const;
begin
   SEDIE := N;
   inAttesa := 0;
Parbegin
   repeat

      wait(ClienteDisponibile);
      wait(sem_CS);
      inAttesa := inAttesa - 1;
      signal(poltrona);
      signal(sem_CS);
      {taglia capelli};

   forever
```

Barbiere

```
   wait(sem_CS);
   if inAttesa < SEDIE then
      inAttesa := inAttesa + 1;
      signal(ClienteDisponibile);
      signal(sem_CS);
      wait(poltrona);
      {taglio dei capelli}
   else signal(sem_CS);
   {lascia il negozio};

Parend
```

Clienti

Monitor in Java

- Una classe Java diventa un tipo monitor quando l'attributo *synchronized* è associato con uno o più metodi nella classe
- Un oggetto di una tale classe è un monitor
- Ogni monitor contiene una singola variabile di condizione senza nome
  - Può portare ad attese attive in applicazioni con molte condizioni

Casi di Studio di Sincronizzazione di Processi

- Sincronizzazione dei Thread POSIX
- Sincronizzazione dei processi Unix
- Sincronizzazione dei processi Linux
- Sincronizzazione dei processi Solaris
- Sincronizzazione dei processi Windows

Sincronizzazione dei Thread POSIX

- I thread POSIX forniscono
  - Mutex per la mutua esclusione
    - Un mutex è come un semaforo binario
  - Variabili di condizione per la sincronizzazione di controllo tra processi
- Un SO può implementare i thread POSIX come thread di livello kernel o thread di livello utente

Sincronizzazione di Processi in Unix

- Unix System V fornisce una implementazione di livello kernel dei semafori
  - Il nome di un semaforo è chiamato key
    - Key è associato con un array di semafori
    - I processi condividono un semaforo usando la stessa key
- Unix SVR4 tiene traccia di tutte le operazioni eseguite da un processo su ogni semaforo che usa
  - Esegue un undo su di essi quando il processo termina
  - Aiuta a rendere i programmi più affidabili
- Unix 4.4BSD pone un semaforo in un'area di memoria condivisa ed usa un'implementazione ibrida

Sincronizzazione di processi Linux

- Linux ha semafori Unix-like per i processi utente
- Fornisce anche semafori usati dal kernel
  - Semaforo convenzionale
    - Usa uno schema efficiente per un'implementazione di livello kernel
  - Semaforo Lettore-scrittore (read-write lock)
- I kernel precedenti alla versione 2.6 implementavano la mutua esclusione nello spazio kernel mediante system call
- Il kernel 2.6 ha un mutex dello spazio utente veloce chiamato futex
  - Usa un'implementazione ibrida
  - Fornisce un'operazione wait limitata temporalmente

Sincronizzazione dei Processi Solaris

- Caratteristiche interessanti
  - Semafori lettore-scrittore
    - Analogo a quello in Linux
  - Mutex adattivi
    - Utili nei SO multi-processore
  - Usa il protocollo di ereditarietà della priorità per ridurre i ritardi di sincronizzazione
  - Struttura dati chiamata turnstile
    - Mantiene informazioni che riguardano i thread bloccati su un mutex o semaforo lettore-scrittore
      - Usato per la sincronizzazione e l'ereditarietà della priorità

Sincronizzazione Processi Windows

- Un oggetto dispatcher è integrato in ogni oggetto su cui è necessaria la sincronizzazione
  - Può trovarsi nello stato segnalato/non-segnalato
  - Oggetti dispatcher usati in processo, file, ecc.
- Fornisce spinlock, spinlock accodati, mutex veloci e lock push
- Vista fornisce un lock lettore-scrittore