UNIVERSITÀ DEGLI STUDI DI NAPOLI
PARTHENOPE

Sistemi Operativi

Sincronizzazione dei Processi

LEZIONE 8

prof. Antonino Staiano

Corso di Laurea in Informatica – Università di Napoli Parthenope

antonino.staiano@uniparthenope.it

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

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

Implementazione dei Semafori (NO valori negativi)

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

```
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++;
}
```

Implementazione dei Semafori (SI valori negativi)

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

```
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);
   }
}
```

Uso dei Semafori nei Sistemi Concorrenti

Table 6.3 Uses of Semaphores in Implementing Concurrent Systems

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.

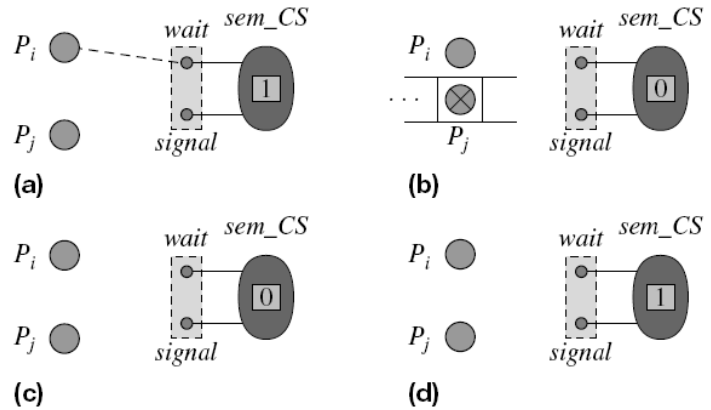

Figure 6.24 Snapshots of the concurrent system of Figure 6.23.

```
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++;
}
```

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 effettua
  - 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 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 contatore

```
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)

Produttore-Consumatore con Semafori

Produttore-consumatore con buffer singolo

- 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

```
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
```

Produttore-Consumatore a Buffer Singolo con Semafori

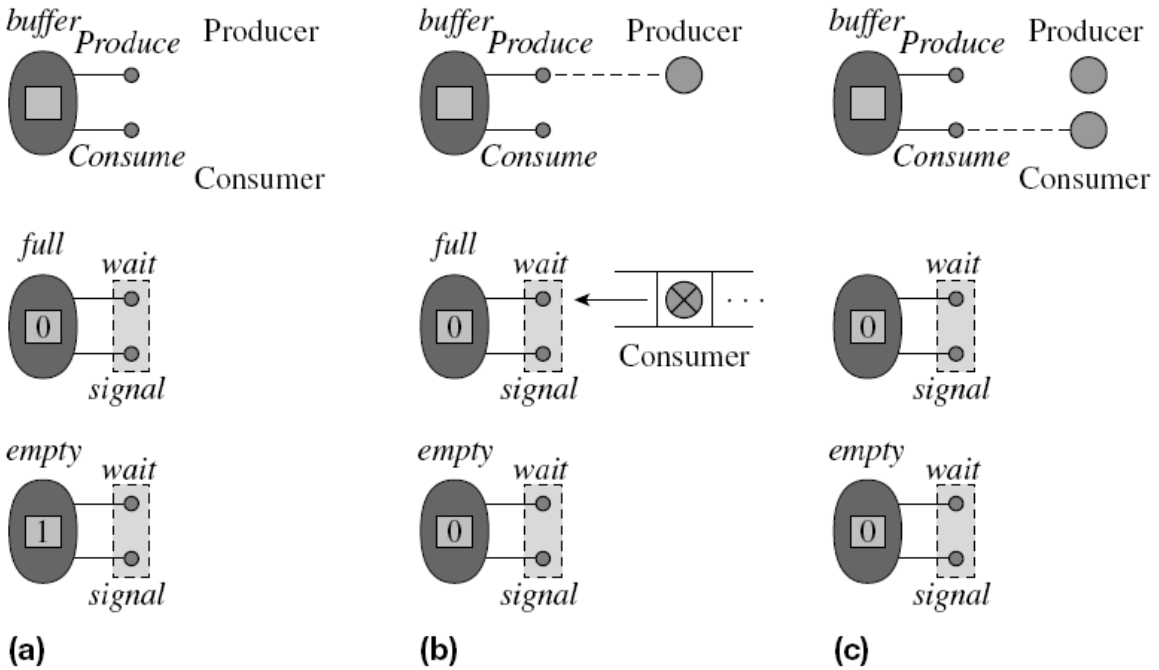

Figure 6.27 Snapshots of single buffer producers–consumers using semaphores.

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*

- Cosa succede se abbiamo
  - M produttori
  - N consumatori

Lettori-Scrittori con Semafori

Soluzione rifinita per i lettori-scrittori

```
Parbegin
  repeat
    if runwrite ≠ 0
    then
        { wait };
    { read }
    if runread = 0 and
        totwrite ≠ 0
    then
        activate one waiting writer
  forever;
Parend;
```

Reader(s)

```
repeat
  if runread ≠ 0 or
      runwrite ≠ 0
  then { wait };
  { write }
  if totread ≠ 0 or totwrite ≠ 0
  then
      activate either one waiting writer
      or all waiting readers
forever;
```

Writer(s)

```
Parbegin
  repeat
      If a writer is writing
      then
          { wait };
      { read }
      If no other readers reading
      then
          if writer(s) waiting
          then
              activate one waiting writer;
  forever;
Parend;
end.
```

Reader(s)

```
repeat
    If reader(s) are reading, or a
        writer is writing
    then
        { wait };
    { write }
    If reader(s) or writer(s) waiting
    then
        activate either one waiting
        writer or all waiting readers;
forever;
```

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 con Semafori

Preferenza ai lettori

```
var
      totread, runread, totwrite, runwrite : integer;
      reading, writing : semaphore := 0;        <-- per la segnalazione
      sem_CS : semaphore := 1;                  <-- per la mutua esclusione
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 call per bloccare solo quando tutti i thread sono bloccati
- Ibrido: combinazione dei due

Figure 6.31 A scheme for implementing *wait* and *signal* operations on a semaphore.