Sincronizzazione fra processi

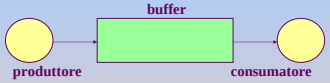
- Background
- Il problema della sezione critica
- Semafori
- Problemi classici di sincronizzazione
- Monitor

Background

- ***Race condition***: Situazioni nelle quali più processi accedono in concorrenza, e modificano, dati condivisi.
  - L'accesso concorrente a dati condivisi può causare incoerenza nei dati.
  - L'esito dell'esecuzione (il valore finale dei dati condivisi) dipende dall'ordine nel quale sono avvenuti gli accessi.
- Per garantire la coerenza dei dati occorrono meccanismi che assicurano l'esecuzione ordinata dei processi cooperanti.
  - Per evitare race conditions occorre che processi concorrenti siano **sincronizzati**.

Esempio di cooperazione: produttore-consumatore

- È un paradigma classico per processi cooperanti: processi **produttori** generano dati che sono utilizzati dai processi **consumatori**



- Esempi:
  - Un programma di stampa produce caratteri consumati dal driver della stampante
  - Il compilatore produce un codice che viene consumato da un assemblatore
- Nel caso di esecuzione concorrente dei processi è necessario un meccanismo di sincronizzazione
- Possibili approcci per lo scambio di dati:
  - Uso di un canale di comunicazione e di scambio di messaggi
  - Uso di memoria condivisa (buffer)
    - Buffer limitato: si assume che la dimensione del buffer sia fissata
    - Buffer illimitato: non si pongono limiti pratici alla dimensione

Soluzione con memoria condivisa e buffer limitato

- Consideriamo il caso in cui sia presente un solo produttore e un solo consumatore
  - buffer = un vettore di BUFFER_SIZE elementi, con gestione FIFO (circolare)
  - *in* = prima posizione libera per l'inserimento di un nuovo dato
  - *out* = posizione del prossimo dato da estrarre
- Dati condivisi

```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Note
  - La soluzione consiste nel garantire che i due processi non accedano contemporaneamente alle stesse variabili (o alle stesse locazioni del buffer)
  - La soluzione consente l'utilizzo di soli BUFFER_SIZE–1 elementi ….

Soluzione con buffer limitato

```
item nextProduced;

while (1) {
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Processo Produttore

```
item nextConsumed;

while (1) {
        while (in == out)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
}
```

Processo Consumatore

Busy waiting

Soluzione 2 con buffer limitato

- Una soluzione in cui vengano impiegati tutti gli *n* elementi del buffer non è banale.
  - Si modifica il codice del produttore–consumatore aggiungendo una variabile *counter*, inizializzata a 0.
  - *counter* viene incrementata ogni volta che un nuovo elemento viene inserito nel buffer, decrementata dopo un'estrazione.
- Dati condivisi

```
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Soluzione 2 con buffer limitato

Processo produttore

in: indice della successiva posizione libera nel buffer.

```
item nextProduced;
while (1) {
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter ++;
}
```

```
item nextConsumed;
while (1) {
        while (counter == 0)
                ; /* do nothing */
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter – –;
}
```

Processo consumatore

out: indice della prima posizione piena nel buffer.

Soluzione 2 con buffer limitato

- Le istruzioni:

```
counter++;
counter– –;
```

  devono essere eseguite *atomicamente*.
- Un'operazione è atomica quando viene completata senza subire interruzioni.
- Le istruzioni di aggiornamento del contatore vengono realizzate in linguaggio macchina come…

```
register1 = counter
register1 = register1 + 1
counter = register1
```

```
register2 = counter
register2 = register2 – 1
counter = register2
```

I due registri potrebbero coincidere… ma ciò è ininfluente

Soluzione 2 con buffer limitato

- Problema:
  - Se sia il produttore che il consumatore tentano di accedere al buffer contemporaneamente, le istruzioni in linguaggio assembler possono risultare *interfogliate*.
  - La sequenza effettiva di esecuzione dipende da come i processi produttore e consumatore vengono schedulati.

- **Esempio:** inizialmente counter = 5

  T_0 *produttore*: $register_1 = counter$ ($register_1 = 5$)
  T_1 *produttore*: $register_1 = register_1 + 1$ ($register_1 = 6$)
  T_2 *consumatore*: $register_2 = counter$ ($register_2 = 5$)
  T_3 *consumatore*: $register_2 = register_2 - 1$ ($register_2 = 4$)
  T_4 *produttore*: $counter = register_1$ ($counter = 6$)
  T_5 *consumatore*: $counter = register_2$ ($counter = 4$)

  - Il valore di counter sarà pari a 4 o 6, mentre dovrebbe rimanere correttamente 5 (un elemento prodotto ed uno consumato).

Modellazione con sezioni critiche

- n processi che competono per utilizzare dati condivisi.
- ciascun processo è costituito da un segmento di codice, chiamato **sezione critica**, in cui accede a dati condivisi.
- Esempio produttore-consumatore
  - Nella formulazione considerata la sezione critica riguarda l'accesso alla variabile counter
  - Nel caso ci siano più produttori, anche l'accesso alle celle del buffer deve essere controllato

Problema della sezione critica

- Problema: quando un processo esegue la sua sezione critica, a nessun altro processo deve essere concesso eseguire la propria.
  - L'esecuzione di sezioni critiche da parte di processi cooperanti è mutuamente esclusiva nel tempo.
  - Nessun vincolo sull'ordine in cui le sezioni critiche sono eseguite
  - Esempio: A, B sezioni critiche

- Soluzione: progettare un protocollo di cooperazione:
  - Ogni processo deve chiedere il permesso di accesso alla sezione critica, tramite una entry section;
  - La sezione critica è seguita da una exit section; il rimanente codice è non critico.

Problema della sezione critica

- Struttura generale del processo P_i (l'altro processo è P_j).

```
do {
    entry section
        sezione critica
    exit section
        sezione non critica
} while (1);
```

- I processi possono condividere alcune variabili per sincronizzare le loro azioni.

Problema della sezione critica: requisiti della soluzione

1. **Mutua esclusione:** un solo processo alla volta può accedere alla propria sezione critica
   - Se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può eseguire la propria sezione critica.
2. **Progresso**: se esistono richieste pendenti di accesso alla sezione critica, bisogna soddisfarne ameno una
   - Se nessun processo è in esecuzione nella propria sezione critica ed esiste qualche processo che desidera accedervi, allora la selezione del prossimo processo che entrerà nella propria sezione critica non può essere rimandata indefinitamente.
3. **Attesa limitata:** l'accesso alla sezione critica deve essere garantito a tutti I processi (eventualmente con diverse priorità)
   - se *P* ha chiesto di accedere alla sezione critica, si deve limitare il numero di volte che altri processi entrano nelle loro sezioni critiche prima che la richiesta di *P* sia stata soddisfatta

Primi tentativi di soluzione: Algoritmo 0

occupato = 1 ➔ risorsa occupata
occupato = 0 ➔ risorsa libera

P_1

```
prologo: while (occupato==1);
         occupato =1;
         <sezione. critica A>;
epilogo: occupato =0;
```

P_1

```
prologo: while (occupato==1);
         occupato =1;
         <sezione critica B>;
epilogo: occupato =0;
```

Algoritmo 0

Problema: Race condition sulla variabile *occupato*

Esempio:

T0: P1 esegue while e trova occupato=0
T1: P2 esegue while e trova occupato=0
T2: P1 pone occupato=1 ed entra in A
T3: P2 pone occupato=1 ed entra in B

P_1

```
prologo: while (occupato==1);
         occupato =1;
         <sezione. critica A>;
epilogo: occupato =0;
```

Entrambi i processi sono contemporaneamente nella loro sezione critica: *violazione della mutua esclusione*

Istruzione TSL (Test and Set Lock)

- Si introduce una istruzione del linguaggio macchina per leggere e modificare una cella di memoria in un solo ciclo
  - Istruzione atomica: non interrompibile
  - P1 può leggere e modificare la variabile *occupato* senza interferenze da parte di P2
- Istruzione "TSL R, x":
  - Il valore contenuto in x viene copiato nel registro R del processore e viene scritto in x un valore diverso da zero.
  - Le operazioni di lettura e scrittura sono eseguite in modo indivisibile (hardware)

Algoritmo 1

- Variabili condivise:
  - **int turn**;
    (inizialmente **turn = 0**).
  - **turn = i** ⇒ P_i può entrare nella propria sezione critica.
- Processo P_i

```
do {
   while (turn != i);
      sezione critica
   turn = j;
      sezione non critica
} while (1);
```

- Soddisfa la **mutua esclusione**, ma non il **progresso**.
  - Richiede una stretta alternanza fra i due processi
  - Es.: Se *turn=0*, P_1 non può entrare nella propria sezione critica, anche se P_0 si trova nella propria sezione non critica.

Algoritmo 2

- Variabili condivise:
  - **boolean flag[2]**;
    (inizialmente **flag [0] = flag [1] = false**).
  - **flag [i] = true** ⇒ P_i è pronto ad entrare nella propria sezione critica.
- Processo P_i

```
do {
   flag[ i ] = true;
   while (flag[ j ]);
      sezione critica
   flag [ i ] = false;
      sezione non critica
} while (1);
```

- Soddisfa la **mutua esculsione**, ma non il **progresso**.
  - I due processi possono porre entrambi *flag[i] = true*, bloccandosi indefinitamente.

Algoritmo 3

- Combina le variabili condivise degli algoritmi 1 e 2.
- Processo P_i

```
do {
   flag[ i ] = true;
   turn = j;
   while (flag[ j ] and turn = j);
      sezione critica
   flag[ i ] = false;
      sezione non critica
} while (1);
```

- Risolve il problema della sezione critica per due processi. E soddisfa tutte le condizioni.
  - P_i entra nella sezione critica (**progresso**) al massimo dopo un'entrata da parte di P_j (**attesa limitata**).

Algoritmo del fornaio

Soluzione del problema della sezione critica per *n* processi.

- Prima di entrare nella loro sezione critica, i processi ricevono un numero. Il possessore del numero più basso entra in sezione critica.
- Se i processi P_i e P_j ricevono lo stesso numero, se $i < j$, allora P_i viene servito per primo, altrimenti tocca a P_j.
- Lo schema di numerazione genera sempre numeri non decrescenti; ad esempio, 1,2,3,3,3,3,4,5...
- Notazione $\leq$ ordine lessicografico (# biglietto, # processo)
  - $(a,b) < (c,d)$, se $a < c$ o se $a = c$ e $b < d$;
  - max $(a_0,\ldots, a_{n-1})$ è un numero k, tale che $k \geq a_i$ per $i = 0, \ldots, n-1$.
- Variabili condivise:

  boolean choosing[n];

  int number[n];

  I vettori sono inizializzati rispettivamente a **false** e **0**.

Algoritmo del fornaio

```
do {
   choosing[i] = true;
   number[i] = max(number[0], number[1], …, number [n – 1])+1;
   choosing[i] = false;
   for (j = 0; j < n; j++) {
            while (choosing[j]);
            while ((number[j] != 0) && (number[ j],j) < (number[i],i));
   }
         sezione critica
   number[i] = 0;
         sezione non critica
} while (1);
```

number[i] = 0 indica che P_i non vuole entrare in sezione critica.

Soluzione con lock e unlock

- Due operazioni per acquisire/rilasciare il diritto di accesso alla sezione critica: lock/unlock
  - Sono associate ad una variabile x nella memoria centrale

P_1		P_2	
prologo:	`lock(x);`	*prologo*:	`lock(x);`
	<*sez. critica A*>;		<*sez. critica B*>;
epilogo:	`unlock(x);`	*epilogo*:	`unlock(x);`

- Caratteristiche
  - Attesa attiva (busy form of waiting)
  - Applicabilità al caso multiprocessore

lock e unlock

```
lock(x):
•  TSL registro, x      (copia x nel registro e pone x=1)
•  CMP registro,0       (il valore di x era 0 ? )
•  JNE lock             (altrimenti, ricomincia il ciclo)
•  RET                  (ritorna al chiamante)

unlock(x):
–  MOVE x, 0           (inserisce 0 in x)
–  RET                  (ritorna al chiamante)
```

Semafori

- I semafori sono strumenti di sincronizzazione
  - associati ad unavariabile intera.
  - si può accedere al semaforo solo attraverso due operazioni indivisibili (operazioni atomiche, primitive).
- Implementazione con *busy waiting:*

```
wait (S):
        while S≤ 0 do no–op;
        S– –;
```

```
signal (S):
        S++;
```

Sezione critica con *n* processi

- Variabili condivise:
  - **semaphore mutex; // inizialmente mutex = 1**
- Processo P_i:

```
do {
   wait(mutex);
      sezione critica
   signal(mutex);
      sezione non critica
} while (1);
```

Gli *n* processi condividono un semaforo comune: ***mutex***

…da ***mut***ual ***ex***clusion.

Implementazione dei semafori

- Per evitare busy waiting si definisce un semaforo come una coppia <variabile intera, coda di processi> :

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

- Si assume che siano disponibili due operazioni (syscall):
  - **block** sospende il processo che la invoca;
  - **wakeup(*P*)** riprende l'esecuzione di un processo bloccato ***P***:
    - Il processo diventa READY

Implementazione dei semafori

- Le operazioni sui semafori possono essere definite come…

```
wait(S):
        S.value– –;
        if (S.value < 0) {
                aggiungi il processo P a S.L;
                block;
        }
```

|S.value| è il numero di processi in coda.

```
signal(S):
        S.value++;
        if (S.value <= 0) {
                rimuovi il processo P da S.L;
                wakeup(P);
        }
```

Atomicità di wait e signal

- wait e signal devono essere azioni indivisibili (azioni atomiche):
  - Modifica al variare del semaforo ed eventuale sospensione o riattivazione di un processo devono avvenire in modo indivisibile
- Indivisibilità di wait e signal:
  - processi eseguiti sullo stesso processore: disabilitazione delle interruzioni durante la loro esecuzione (system call).
  - processi eseguiti su processori diversi: utilizzo di lock(x), unlock(x).

Semaforo: uno strumento di sincronizzazione

- Si esegue *B* in P_j solo dopo che *A* è stato eseguito in P_i
- Si impiega un semaforo *flag* inizializzato 0
- Codice:

P_i	P_j
⋮	⋮
A	***wait(flag)***
signal(flag)	***B***

Due tipi di semafori

- Semaforo *contatore*
  - intero che può assumere valori in un dominio non limitato.
- Semaforo *binario*
  - intero che può essere posto solo a 0 o 1
- Un semaforo contatore può essere realizzato per mezzo di semafori binari.

Deadlock e starvation

- **Deadlock** — due o più processi sono in attesa indefinita di un evento che può essere generato solo da uno dei due processi in attesa
  - Es.: Siano *S* e *Q* due semafori inizializzati a 1:

P_0	P_1
wait(S);	***wait(Q);***
wait(Q);	***wait(S);***
⋮	⋮
signal(S);	***signal(Q);***
signal(Q)	***signal(S);***

  Se dopo *wait(S)* di P_0 si esegue *wait(Q)* di P_1 si ha un deadlock.

- **Starvation** — blocco indefinito;
  - un processo attende indefinitamente ad un semaforo, e non può essere rimosso dalla coda del semaforo su cui è sospeso.

Problemi classici di sincronizzazione

- **Problema del buffer limitato**
- **Problema dei lettori e degli scrittori**
- **Problema dei cinque filosofi**

Problema del buffer limitato

- Variabili condivise:
  semaphore full, empty, mutex;
  // inizialmente **full = 0, empty = n, mutex = 1**
- Il buffer ha *n* posizioni, ciascuna in grado di contenere un elemento.
- Il semaforo *mutex* garantisce la mutua esclusione sugli accessi al buffer.
- I semafori *empty* e *full* contano, rispettivamente, il numero di posizioni vuote ed il numero di posizioni piene nel buffer.

Problema del buffer limitato

Processo produttore

```
do {
        …
    produce un elemento in nextp
        …
    wait(empty);
    wait(mutex);
        …
    inserisce nextp nel buffer
        …
    signal(mutex);
    signal(full);
} while (1);
```

Processo consumatore

```
do {
    wait(full)
    wait(mutex);
        …
    sposta un elemento dal buffer in nextc
        …
    signal(mutex);
    signal(empty);
        …
    consuma un elemento in nextc
        …
} while (1);
```

Problema dei lettori e degli scrittori

- Variabili condivise:
  semaphore mutex, wrt;
  int readcount;
  // inizialmente **mutex = 1, wrt = 1, readcount = 0**
- Un insieme di dati (ad es. un file) deve essere condiviso da più processi concorrenti che possono richiedere la sola lettura del contenuto, o anche un aggiornamento.
- Se due *lettori* accedono contemporaneamente ai dati condivisi non ha luogo alcun effetto negativo, se uno *scrittore* accede simultaneamente ad altro processo si può avere incoerenza dell'informazione.
- 1° problema dei lettori–scrittori: nessun processo lettore deve attendere, salvo che uno scrittore abbia già ottenuto l'accesso ai dati condivisi (possibilità di starvation per gli scrittori).

Problema dei lettori e degli scrittori

Processo scrittore

```
wait(wrt);
    …
    si effettua la scrittura
    …
signal(wrt);
```

readcount **conta i lettori attualmente attivi.**

Processo lettore

```
wait(mutex);
readcount++;
if (readcount == 1)
    wait(wrt);
signal(mutex);
    …
    si effettua la lettura
    …
wait(mutex);
readcount – –;
if (readcount == 0)
    signal(wrt);
signal(mutex);
```

Problema dei cinque filosofi

Schematizza la classe dei problemi di allocazione di risorse multiple.

- Cinque filosofi passano la vita pensando e mangiando, attorno ad una tavola rotonda. Al centro della tavola c'è una zuppiera di riso e la tavola è apparecchiata con cinque bacchette.
- Quando un filosofo pensa non interagisce con i colleghi. Quando ha fame tenta di impossessarsi delle bacchette che stanno alla sua destra ed alla sua sinistra. Il filosofo può prendere una sola bacchetta per volta.
- Quando un filosofo affamato possiede due bacchette contemporaneamente, mangia. Terminato il pasto, appoggia le bacchette e ricomincia a pensare.

Soluzione con semafori

- **Si può rappresentare ciascuna bacchetta con un semaforo.**
- **Ogni filosofo tenta di afferrare una bacchetta con un'operazione di wait e la rilascia eseguendo signal sul semaforo appropriato.**

Problema dei cinque filosofi

- Variabili condivise: un vettore di semafori
  semaphore chopstick[5]; // tutti I semafori inizializzati a 1
- Filosofo *i*:

```
do {
   wait(chopstick[ i ])
   wait(chopstick[(i+1) % 5])
      …
    mangia
      …
   signal(chopstick[ i ]);
   signal(chopstick[(i+1) % 5]);
      …
    pensa
      …
} while (1);
```

Problema dei cinque filosofi

- Non esclude il deadlock, ad esempio se tutti i filosofi hanno fame contemporaneamente e prendono prima la bacchetta alla loro destra.
- Alcune soluzioni:
  - Solo quattro filosofi possono essere seduti contemporaneamente a tavola.
  - Un filosofo può prendere le sue bacchette solo se sono entrambe disponibili.
  - Soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra, un filosofo pari prende prima la bacchetta di destra.

Monitor

- È un costrutto di sincronizzazione di alto livello che permette la condivisione sicura di un tipo astratto di dati fra processi concorrenti.
- La mutua esclusione non deve essere codificata esplicitamente

I valori delle variabili definiscono lo stato di un istanza del tipo.

Gli operatori sono definiti dal programmatore.

```
monitor monitor–name
{
    dichiarazione delle variabili condivise
    procedure body P1 (…) {
                . . .
    }
    procedure body P2 (…) {
                . . .
    }
    procedure body Pn (…) {
                . . .
    }
    {
                codice di inizializzazione
    }
}
```

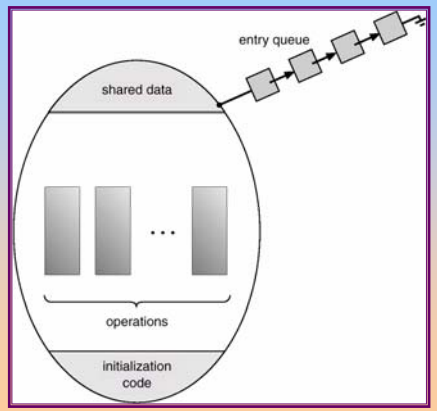
Monitor

- La definizione base di monitor non è sufficiente per modellare alcuni schemi di sincronizzazione…
- ⇒ per permettere ad un processo di attendere dentro al monitor, devono essere dichiarate apposite variabili ***condition***

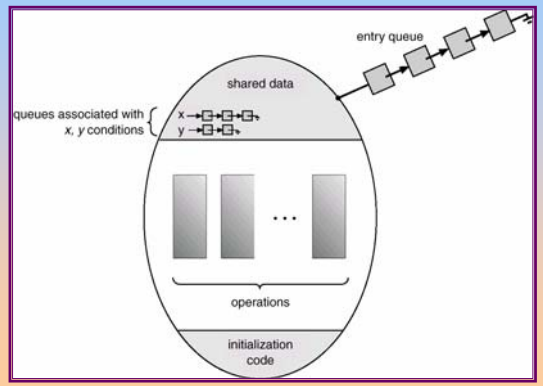
  condition x, y;

- Le variabili condition possono essere usate solo in relazione a specifiche operazioni ***wait*** e ***signal***.
  - **x.wait( )** fa sì che il processo chiamante rimanga sospeso
    - … fino a che un diverso processo non effettui la chiamata **x.signal( )**
  - **x.signal** attiva esattamente un processo sospeso.
    - … se non esistono processi sospesi *signal* non ha alcun effetto

Schema di un monitor



Monitor con variabili condition



Esempio dei cinque filosofi

- Soluzione senza deadlock: al filosofo è permesso di prelevare le bacchette solo se sono entrambe disponibili.
- La distribuzione delle bacchette è controllata dal monitor dp.

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i)
  void putdown(int i)
  void test(int i)
  void init( ) {
       for (int i = 0; i < 5; i++)
              state[ i ] = thinking;
  }
}
```

Esempio dei cinque filosofi

```
void pickup(int i) {
        state[ i ] = hungry;
        test[ i ];
        if (state[ i ] != eating)
                self[ i ].wait();
}
```

```
void putdown(int i) {
        state[ i ] = thinking;
        // controlla i vicini
          destro e sinistro
        test((i+4) % 5);
        test((i+1) % 5);
}
```

```
void test(int i) {
        if ( (state[(i + 4) % 5] != eating) &&
          (state[ i ] == hungry) &&
          (state[(i + 1) % 5] != eating)) {
                state[ i ] = eating;
                self[ i ].signal( );
        }
}
```

Implementazione del monitor con semafori

- Variabili

  semaphore mutex; // (inizialmente **= 1**)
  semaphore next; // (inizialmente **= 0**)
  int next–count = 0;

- Ciascuna procedura esterna ***F*** viene rimpiazzata con

```
wait(mutex);
   …
  corpo di F;
   …
if (next–count > 0)
  signal(next)
else
  signal(mutex);
```

- La mutua esclusione è assicurata all'interno del monitor.

Implementazione del monitor con semafori

- Per ogni variabile condition ***x***, si ha:

  semaphore x–sem; // (inizialmente **= 0**)
  int x–count = 0;

- Le operazioni **x.wait** e **x.signal** possono essere implementate rispettivamente come:

```
x–count++;
if (next–count > 0)
      signal(next);
else
      signal(mutex);
wait(x–sem);
x–count– –;
```

```
if (x–count > 0) {
     next–count++;
     signal(x–sem);
     wait(next);
     next–count– –;
}
```

Implementazione del costrutto monitor

- Costrutto *Conditional–wait*: **x.wait(c);**
  - ***c*** — espressione intera valutata quando viene eseguita l'operazione **wait**.
  - Il valore di *c* (*numero di priorità*) viene memorizzato insieme al nome del processo che è stato sospeso.
  - Quando viene eseguita **x.signal**, il processo associato al numero di priorità più basso viene riavviato.
- Si controllano due condizioni per stabilire la correttezza del sistema:
  - I processi utente devono sempre effettuare chiamate al monitor con una sequenza corretta.
  - È necessario assicurare che un processo non cooperativo non ignori la porta di mutua esclusione fornita dal monitor, e provi ad accedere direttamente alle variabili condivise, senza impiegare i protocolli di accesso.