Sincronizzazione nel modello ad ambiente globale

Corso di Laurea in Ingegneria Informatica

Università degli Studi di Napoli Federico II

Anno Accademico 2024/2025, Canale San Giovanni

Sincronizzazione nel modello ad ambiente globale

- Sommario
  - Problema della mutua esclusione (competizione)
  - Soluzioni al problema della mutua esclusione
  - Semafori
- Riferimenti
  - P. Ancilotti, M.Boari, A. Ciampolini, G. Lipari, "Sistemi Operativi", Mc-Graw-Hill (Cap.3; par. 3.2, 3.3, 3.4)
  - Dispensa su mutua esclusione (tratta da: A. Tanenbaum, "Operating Systems Desing and Implementation", terza ed (Cap 2, par. 2.2.3))

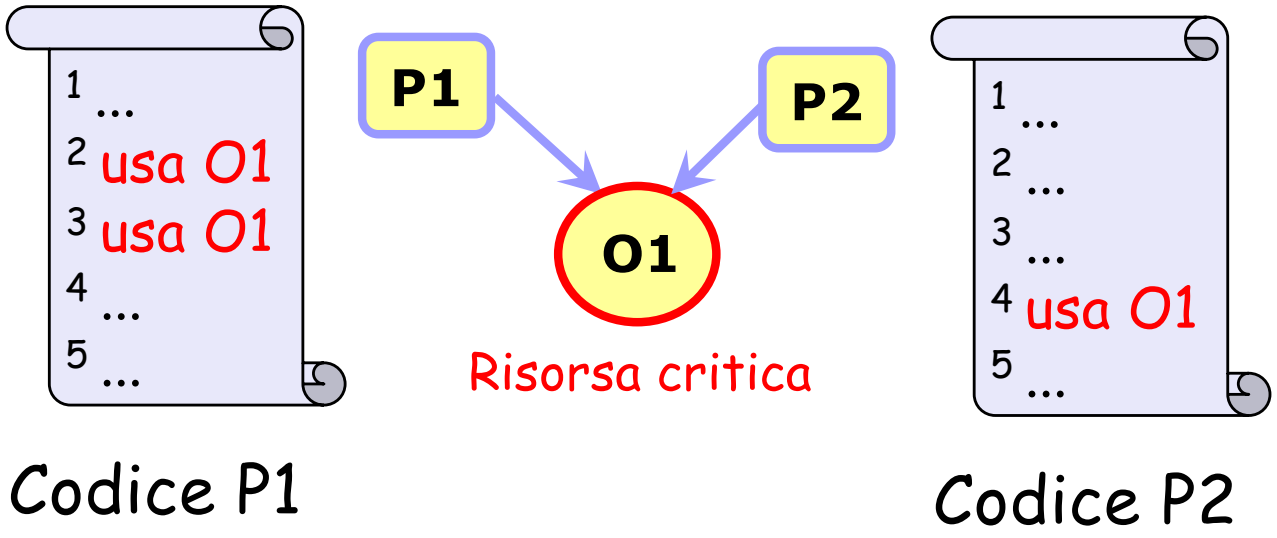
Problemi di competizione

- Più processi "competono" nell'uso di una stessa risorsa
- Per garantire l'uso corretto della risorsa, essa deve essere **acceduta da al più un processo alla volta**

es. nel caso di una risorsa "stampante", è necessario che due richieste di stampa non si "mischino" tra loro

Mutua Esclusione

- Due o più processi vogliono utilizzare una risorsa ad uso esclusivo, detta **Risorsa Critica**
- La porzione di codice che usa la risorsa è detta **Sezione Critica**



es. in questo caso la sezione critica è costituita da:
- le righe 2-3 di P1
- la riga 4 di P2

La sezione critica può essere composta da più parti di codice, eseguite da processi differenti

Mutua Esclusione

```
int counter = 0;

processo1() {
     for(1 ... 1,000,000)
          counter++;
}

processo2() {
     for(1 ... 1,000,000)
          counter++;
}
```

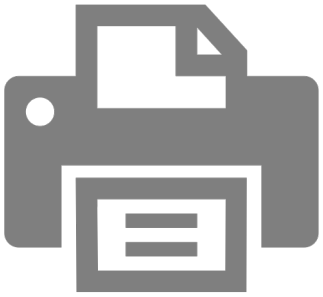
risorsa critica

sezione critica

Mutua esclusione VS comunicazione

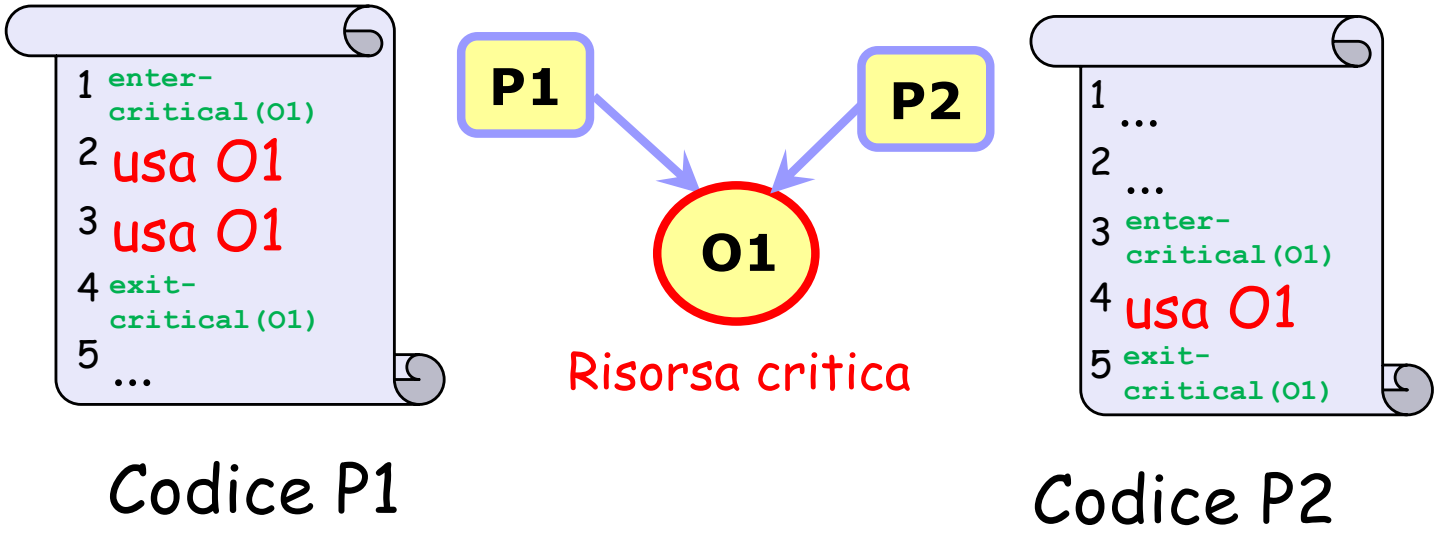
- **Mutua esclusione**: l'ordine con cui devono avvenire due eventi non è fissato
  - È sufficiente che i processi non utilizzino contemporaneamente la risorsa
- **Comunicazione**: si pone un ordinamento tra gli eventi

es. nell'esempio della risorsa "stampante", non è importante l'ordine delle stampe (purché vi sia la mutua esclusione)

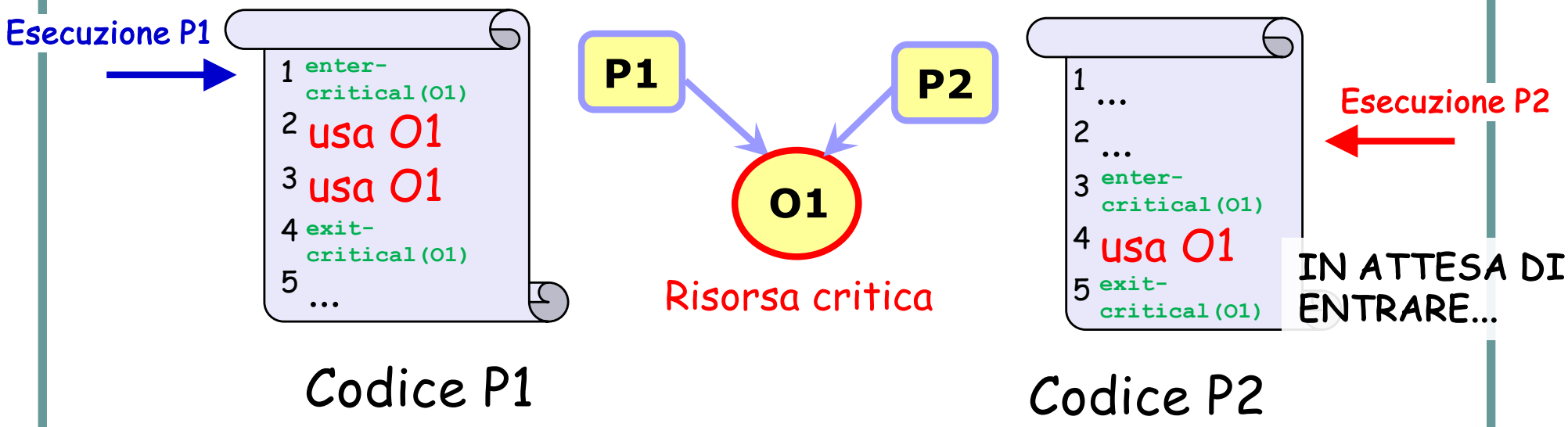


Gestione della competizione

- Il SO fornisce ai programmi meccanismi per **richiedere accesso in mutua esclusione** alle risorse
- Vi sono varie soluzioni (hardware/software)



Gestione della competizione



quando **P1 entra nella sezione critica**, "acquisisce" il possesso della risorsa

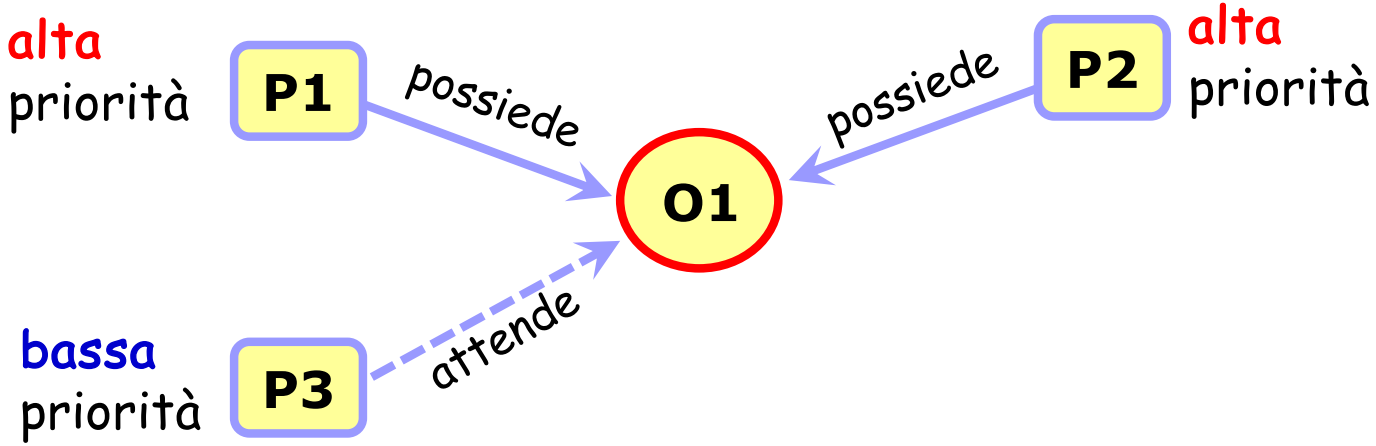
se **P2 tenta di entrare** nella sezione critica **prima che P1 sia uscito**, P2 viene posto in **attesa** dal SO

Requisiti per la mutua esclusione

1. Un solo processo alla volta può accedere alla sezione critica
2. Il corretto funzionamento non dipende dal numero o dalla velocità di esecuzione dei processi (no "Race Condition")
3. Evitare deadlock e starvation dei processi

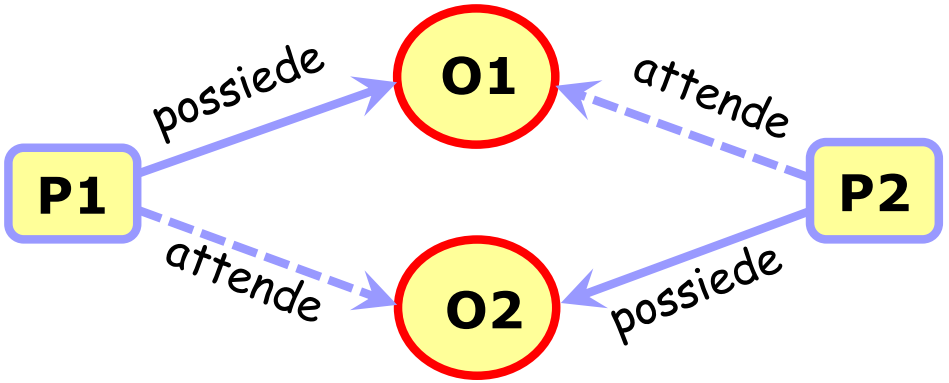
Starvation

- Indica una condizione di **attesa indefinita** da parte di un processo a bassa priorità
- I processi ad alta priorità possono alternarsi a lungo nel possesso



Deadlock

- Indica la presenza di una condizione di blocco permanente (stallo, deadlock) di un gruppo di processi in competizione



Ogni processo "possiede" una risorsa in mutua esclusione, bloccando il progresso dell'altro processo

Requisiti per la mutua esclusione

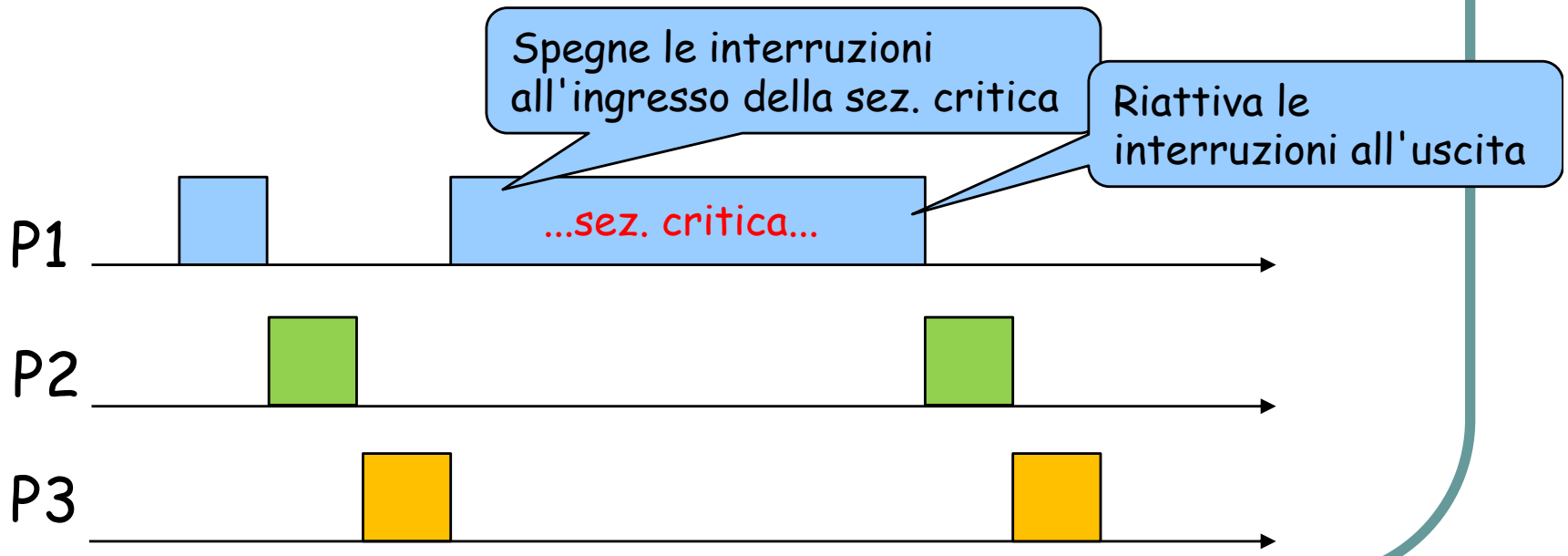
4. Se non vi sono altri processi nella sezione critica, un processo deve poter accedere immediatamente alla risorsa
5. Quando un processo non è nella sezione critica, esso non interferisce con l'uso della risorsa da altri processi
6. Un processo può rimanere in una sezione critica solo per un tempo finito

Supporto Hardware alla mutua esclusione

- Disabilitazione degli Interrupt
- Istruzione Test-and-Set-Lock

Disabilitazione degli Interrupt

- In un sistema **monoprocessore**, per garantire la mutua esclusione è sufficiente **disabilitare le interruzioni** (no timer, no I/O)
- Si evita che un processo in una sezione critica venga **prelazionato**



Disabilitazione degli Interrupt

- Problemi:
  - In un sistema **multiprocessore**, la disabilitazione delle interruzioni in uno dei processori **non garantisce** la mutua esclusione
  - L'approccio viola i principi di **protezione**: non è sicuro consentire ai processi utente di disabilitare le interruzioni!
- Tuttavia, è conveniente per **uso interno nel kernel**, quando deve aggiornare delle variabili condivise interne

Variabili Lock

- Soluzione basata su una **variabile condivisa (lock)**, inizialmente 0
- Se lock=1 (indica che la risorsa è occupata), il processo attende (linee 1-2)
- Quando lock=0, termina il ciclo ed esegue la sezione critica (linea 4)
- Si pone lock=1 (linea 3) per impedire ad altri processi di entrare

```
1  while (lock == 1)
2        /* nulla */

3  lock = 1

4  ...sez. critica...

5  lock = 0
```

ciclo di "attesa attiva" se lock è 1

esce dal ciclo quando lock è 0

Variabili Lock

- **Questa soluzione non è sufficiente**!
- In caso di context switch prima di porre "lock = 1", un altro processo può entrare nel frattempo nella sezione critica

la mutua esclusione è violata se si verifica un **context switch** qui

```
1  while (lock == 1)
2        /* nulla */
     prelazione!
3  lock = 1

4  ...sez. critica...

5  lock = 0
```

Variabili Lock

- Causa del problema: la lettura e la scrittura del lock sono eseguite in momenti diversi
- **Sono operazioni divisibili**

la mutua esclusione è violata se si verifica un **context switch** qui

```
1  while (lock == 1)        LETTURA
2      /* nulla */
       prelazione!
3  lock = 1                 SCRITTURA
4  ...sez. critica...
5  lock = 0
```

Esempio: variabili lock

Process P1	Process P2	lock
.	.	0
x = lock	.	0
if x == 0, vai avanti	.	0
.	x = lock	0
.	if x == 0, vai avanti	0
.	lock = 1	1
.	...sezione critica...	1
lock = 1	.	1
...sezione critica...	.	1
lock = 0	.	0
.	lock = 0	0

RACE CONDITION!

Le letture e scritture su lock sono divisibili

Istruzione Test and Set Lock (TSL)

- Per risolvere via **hardware** il problema delle variabili lock, molti processori forniscono la istruzione macchina:

```
TSL RX,LOCK
```

- Equivale a

```
MOV RX,LOCK     // lettura lock
MOV LOCK, 1     // scrittura lock
```

- Le operazioni di lettura e scrittura di LOCK sono **indivisibili** (eseguono in un solo ciclo di CPU)
- Utilizzabile anche nei sistemi multiprocessore (inibisce l'accesso alla memoria alle altre CPU)

Mutua esclusione con istruzione TSL

```
loop:   TSL RX, LOCK          |copia LOCK in RX, setta LOCK a 1
        CMP RX, 0             |LOCK era 0?
        JNE loop              |NO, ripete il ciclo
        ...                   |SI, entra nella regione critica



        MOVE LOCK, 0          |posiziona LOCK a 0
        ...                   |esce dalla sezione critica
```

lettura+scrittura indivisibili

ciclo di attesa attiva

La soluzione è caratterizzata da **attesa attiva** (*busy wait*)

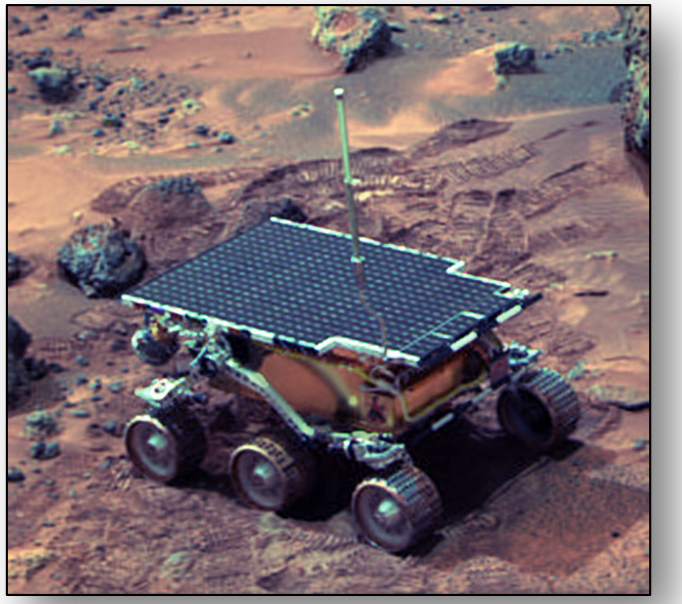
Esempio: Test and Set Lock (TSL)

Process P1	Process P2	lock
.	.	0
tsl x, lock	.	1
if x == 0, vai avanti	.	1
.	tsl x, lock	1
. ATTESA ATTIVA	if x == 0, vai avanti	1
.	tsl x, lock	1
.	if x == 0, vai avanti	1
.	*...si ripete...*	1
...sezione critica...	.	1
lock = 0	.	0
.	tsl x, lock	1
.	if x == 0, vai avanti	1
.	...sezione critica...	1

Priority inversion

- Oltre al problema dell'attesa attiva, la soluzione con TSL presenta il problema della "**priority inversion**".

- **Curiosità**: il più famoso problema di priority inversion si è verificato su Marte!



Priority inversion

- Due processi accedono alla **stessa sezione critica**:
  - Processo H con priorità alta
  - Processo L con priorità bassa
- Si supponga che vi sia inizialmente in esecuzione solo L, e che entri nella **sezione critica** (imposta LOCK=1)
- Il processo H diviene "**pronto**". H ha priorità maggiore, quindi il processo L viene prelazionato, e la CPU è assegnata ad H
- H inizia il **busy wait** tramite TSL (LOCK è stata lasciata ad 1)
- H rimane bloccato nel ciclo di **attesa attiva per sempre**, poiché L non avrà mai la possibilità di eseguire quando H è in esecuzione

La soluzione…

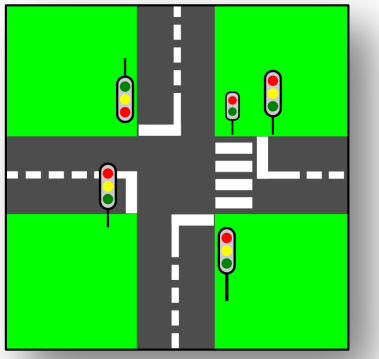
- I problemi di **attesa attiva** e di **priority inversion** vengono risolti "forzando" il processo che trova la variabile LOCK al valore 1 a **sospendersi** (transita nello stato bloccato) in attesa che la risorsa diventi disponibile

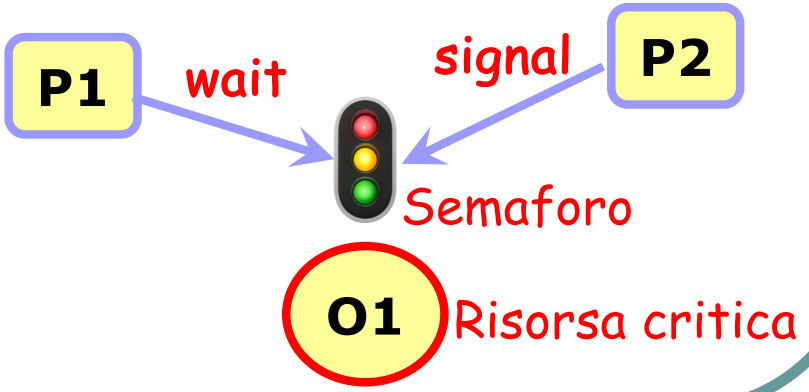
Utilizzo di due system call:

- **suspend(P)**: il processo P che la chiama si auto-sospende, in attesa di un segnale di risveglio
- **wake-up(P)**: un processo invia un segnale di risveglio al processo P (che è stato sospeso mediante la suspend)

Semafori

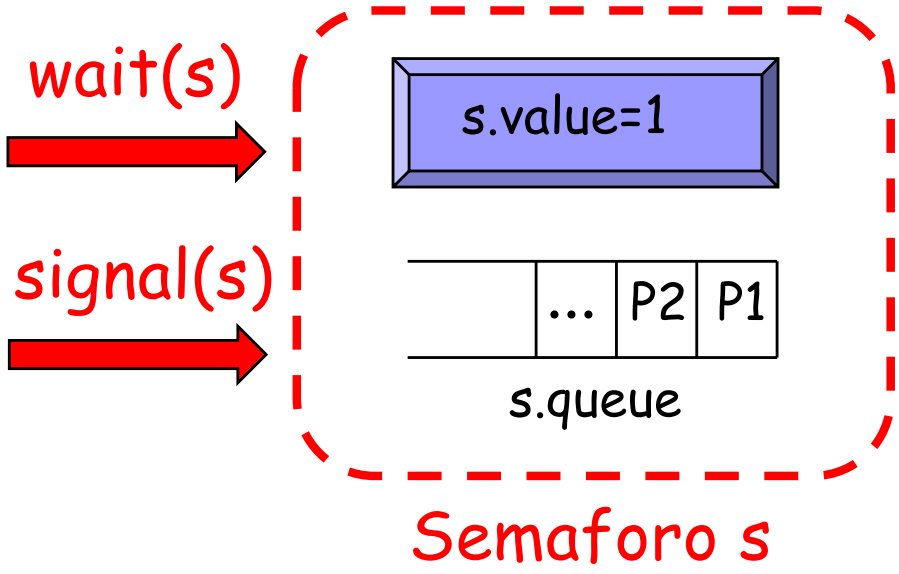
- I **semafori** sono variabili speciali utilizzate per la **competizione** e la **cooperazione** tra processi
  - I processi condividono una **stessa istanza** di semaforo **s** per coordinarsi
  - Un processo può eseguire la primitiva **wait(s)** sul semaforo **s**, per sospendersi in attesa di ricevere un segnale, o proseguire se il segnale è stato già ricevuto
  - Un altro processo può inviare un segnale sul semaforo **s** tramite una procedura **signal(s)**





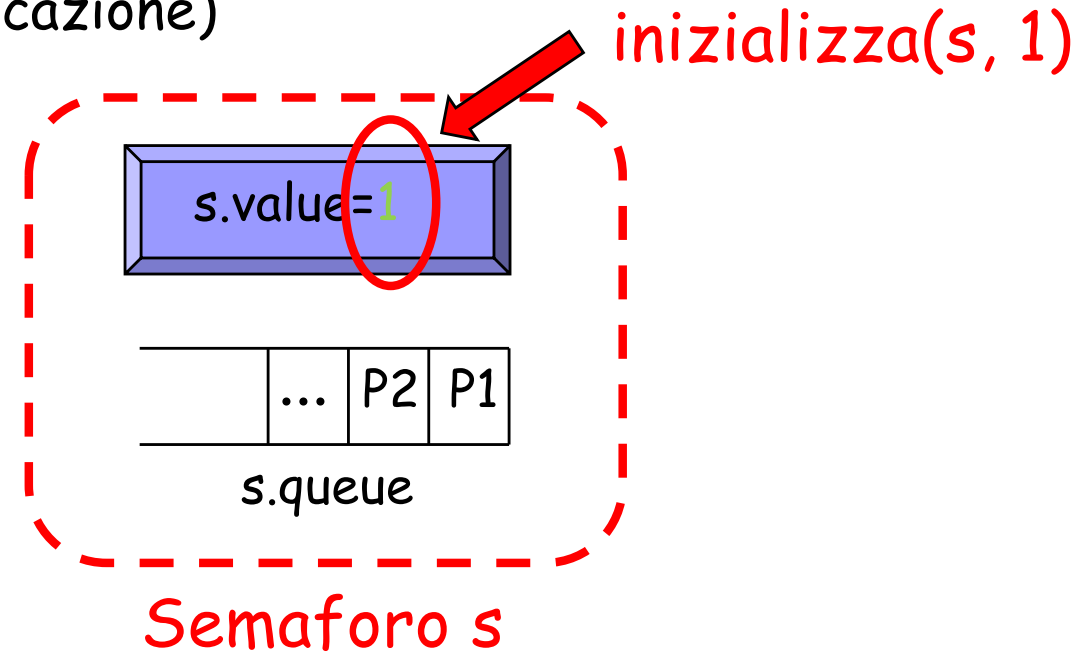
Semafori

- Un tipo di dato astratto **s** che incapsula:
  - una variabile di tipo intero (**s.value**)
  - una coda (**s.queue**), per tenere traccia dei processi che si sono sospesi con wait(s), nell'attesa di una signal(s)



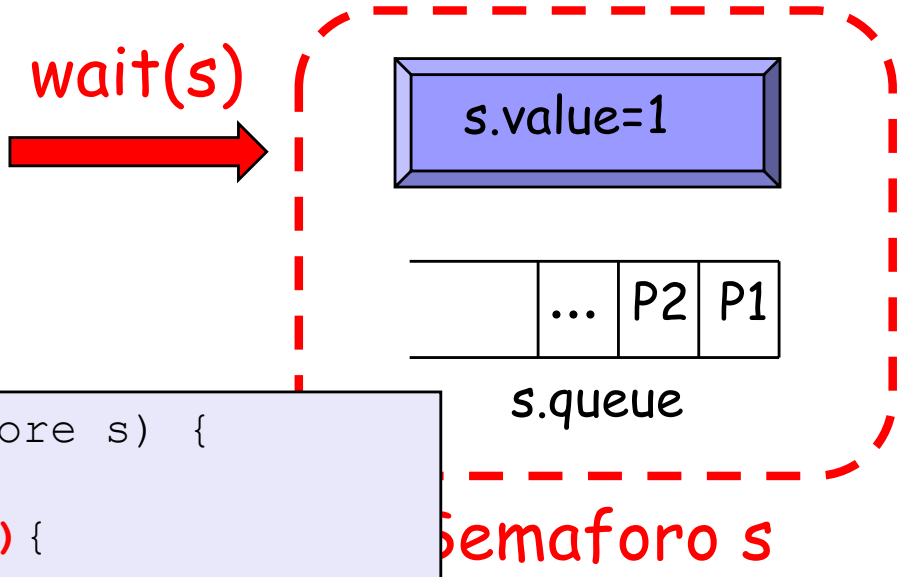
Semafori

- Sono definite le seguenti operazioni:
  - **Inizializzazione** della variabile ad un valore non negativo
  - Il valore iniziale è scelto dal programmatore, in base al tipo di interazione che si vuole realizzare (mutua esclusione, comunicazione)



Semafori

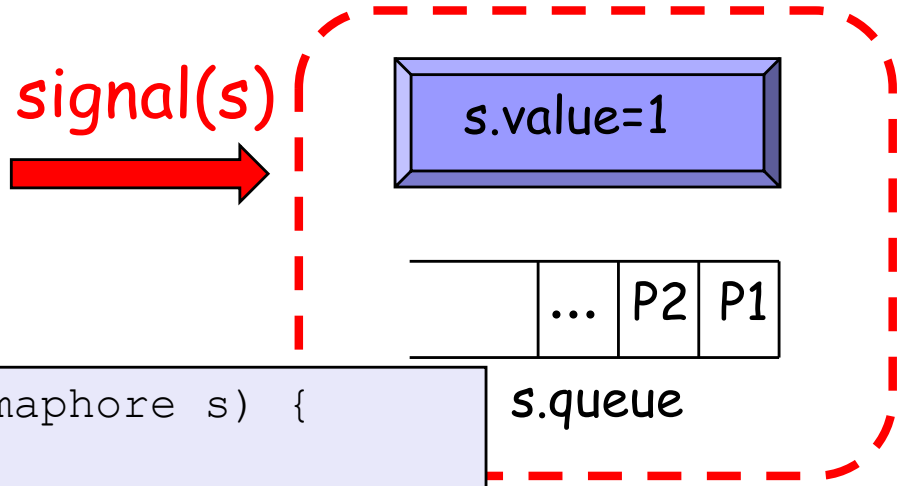
- Sono definite le seguenti operazioni:
  - L'operazione di **wait** ha l'effetto di **decrementare** il valore del semaforo
  - Se il valore del semaforo diventa **negativo**, il processo che ha chiamato wait viene **bloccato**




```
void wait(semaphore s) {
    s.value--;
    if (s.value<0){
         s.queue.insert(Process);
         suspend(Process);
    }
}
```

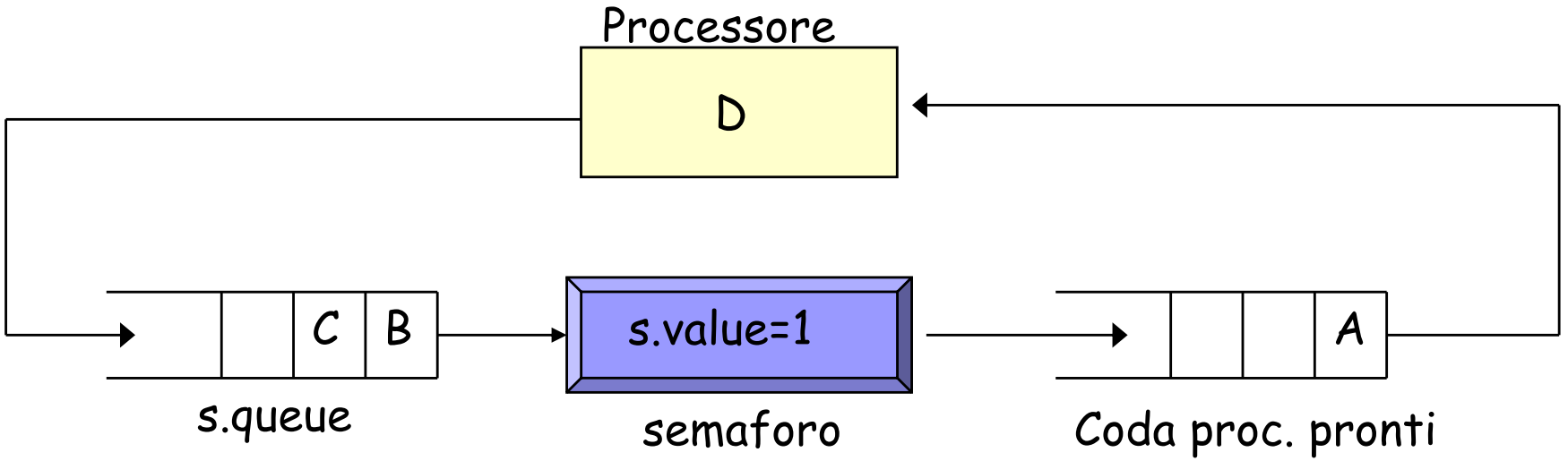
Semafori

- Sono definite le seguenti operazioni:
  - L'operazione di **signal** ha l'effetto di **incrementare** il valore del semaforo
  - Se il valore del semaforo è **minore o uguale a zero**, viene "sbloccato" uno dei processi sospesi




```
void signal(semaphore s) {
    s.value++;
    if (s.value<=0){
         s.queue.remove(Process);
         wake-up(Process);
    }
}
```

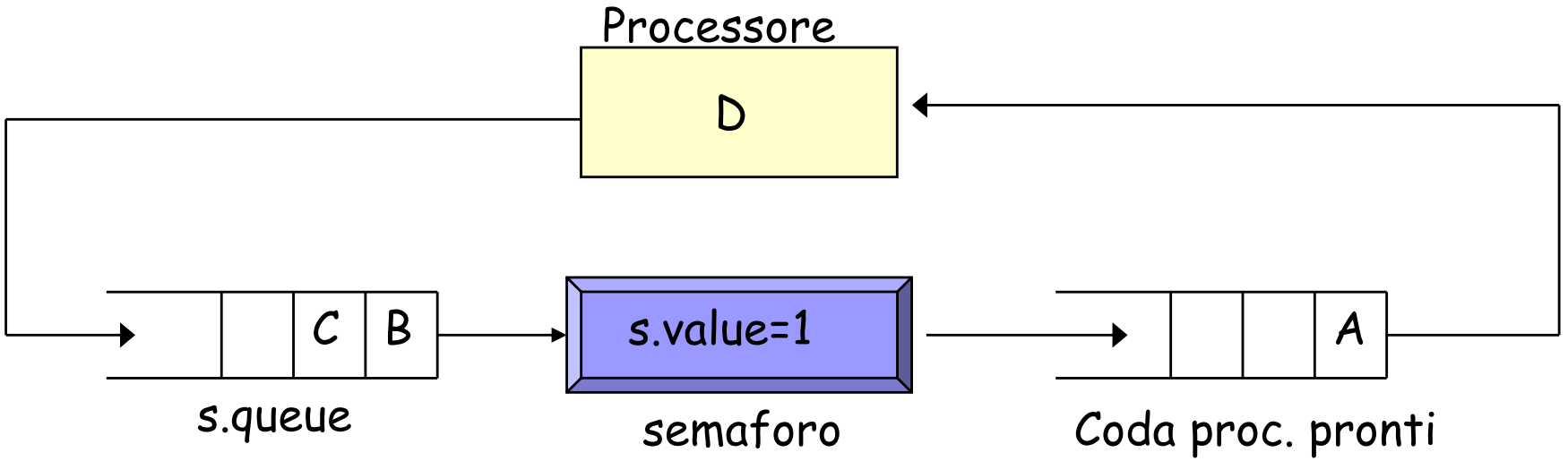
Semafori: modello concettuale




```
void wait(semaphore s) {
    s.value--;
    if (s.value<0){
        s.queue.insert(Process);
        suspend(Process);
    }
}
```

```
void signal(semaphore s) {
    s.value++;
    if (s.value<=0){
        s.queue.remove(Process);
        wake-up(Process);
    }
}
```

Semafori: modello concettuale




```
void wait(semaphore s) {
  s.value--;
  if (s.value<0){
       s.queue.insert(Process);
       suspend(Process);
  }
}
```

```
void signal(semaphore s) {
  s.value++;
  if (s.value<=0){
       s.queue.remove(Process);
       wake-up(Process);
```

Nota: è necessario che l'accesso alla **variabile s** sia a sua volta in modo mutuamente esclusivo (es. tramite utilizzo di TSL). La sospensione allevia comunque i problemi di busy-waiting e priority inversion.

Semafori Binari (Mutex)

- Caso speciale di semaforo, il cui valore intero può essere soltanto 0 oppure 1
- È più semplice da utilizzare (es. per soli problemi di mutua esclusione)

```
void waitB(Binary_sem s) {
   if (s.value==1) {
        s.value=0;
   } else {
        s.queue.insert(Process);
        suspend(Process);
   }
}
```

```
void signalB(Binary_sem s) {
   if (s.queue.is_empty()) {
        s.value=1;
   } else {
        s.queue.remove(Process);
        wake-up(Process);
   }
}
```

Mutua esclusione con l'utilizzo dei semafori

```
semaphore s;     /* condiviso tra i processi */

int main() {
        s.value = 1;    /* inizializzazione */

        /* avvia l'esecuzione concorrente di
           P(1) … P(n) */
}

void P(int i) {

        …
        wait(s);
        /* sezione critica */
        signal(s);
        …
}
```

Questa soluzione richiede di inizializzare "s.value=1" (il valore iniziale del semaforo è importante!)

Mutua esclusione con l'utilizzo dei semafori

```
semaphore s;     /* condiviso tra i processi */

int main() {
        s.value = 1;    /* inizializzazione */

        /* avvia l'esecuzione concorrente di
           P(1) … P(n) */
}

void P(int i) {
        …
        wait(s);
        /* sezione critica */
        signal(s);
        …
}
```

P1 P2 OK STOP

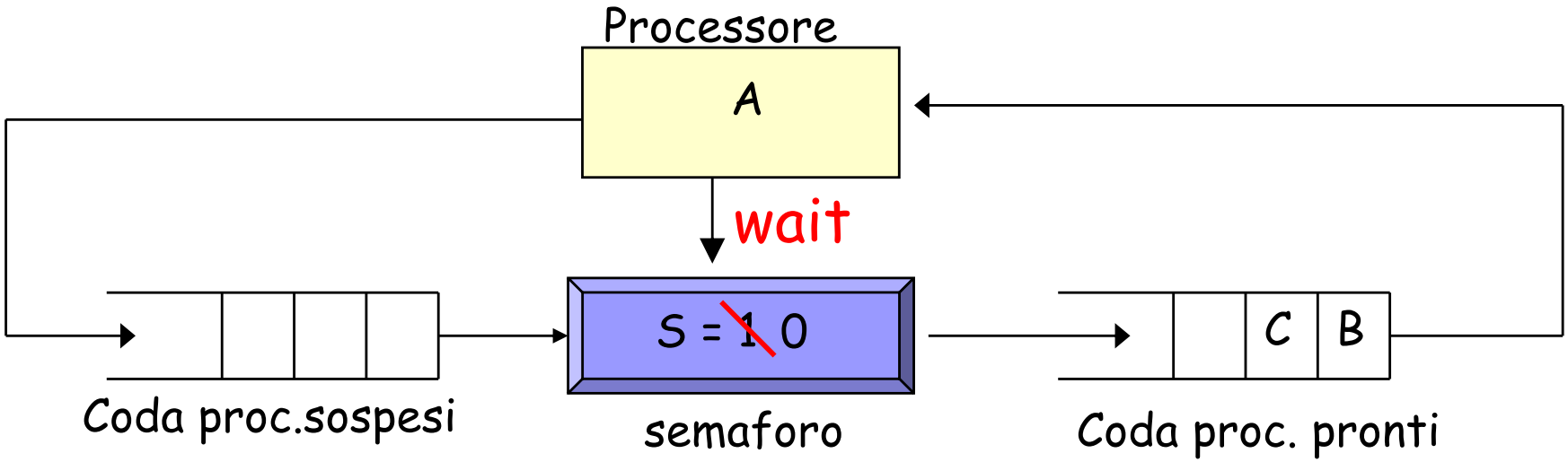
P1 trova "s.value" pari a 1.
Pone "s.value--" (diventa 0),
entra nella sezione critica.

P2 trova "s.value" pari a 0.
Viene sospeso (in attesa di signal)

Esempio

Si hanno 3 processi concorrenti, denominati *A*, *B*, *C*

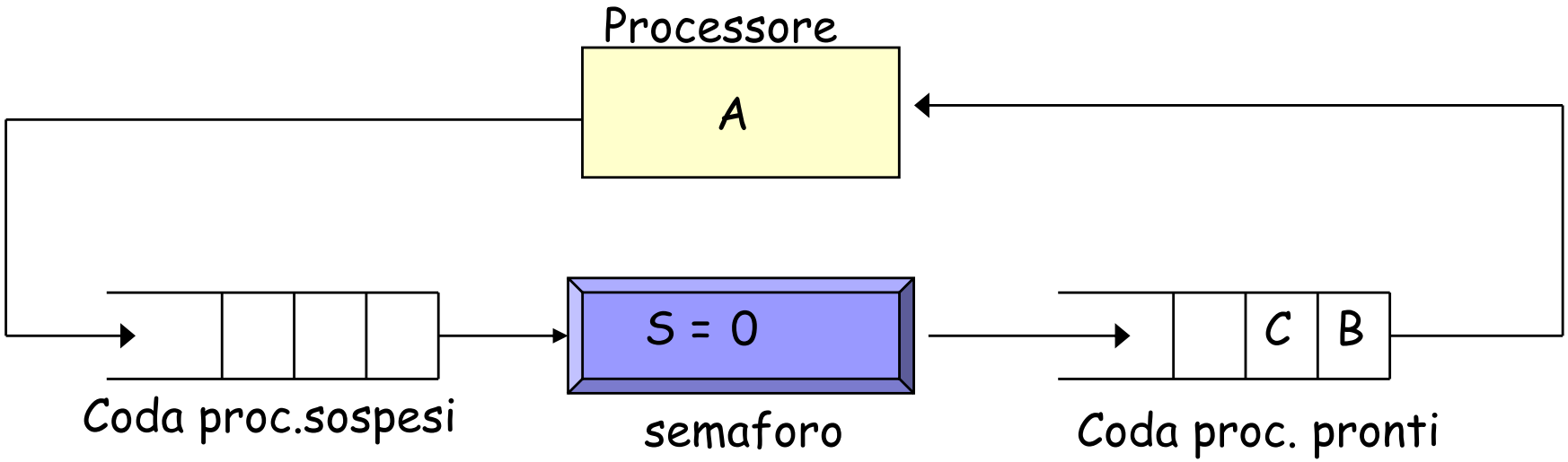
Si supponga che tutti i processi eseguano una sezione critica in mutua esclusione, usando un semaforo "mutex"



1) Inizialmente A è in esecuzione, B e C sono ready.
Il valore iniziale del semaforo è 1

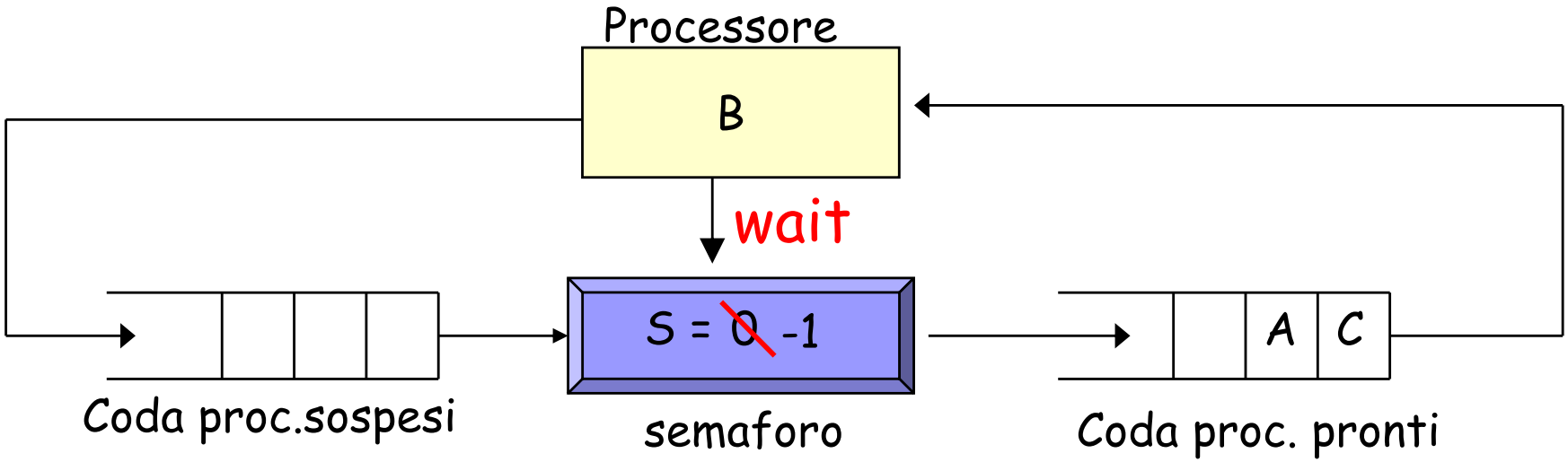
A esegue una wait. Il valore di S viene decrementato.

Poiché il semaforo non è negativo, A non viene sospeso.



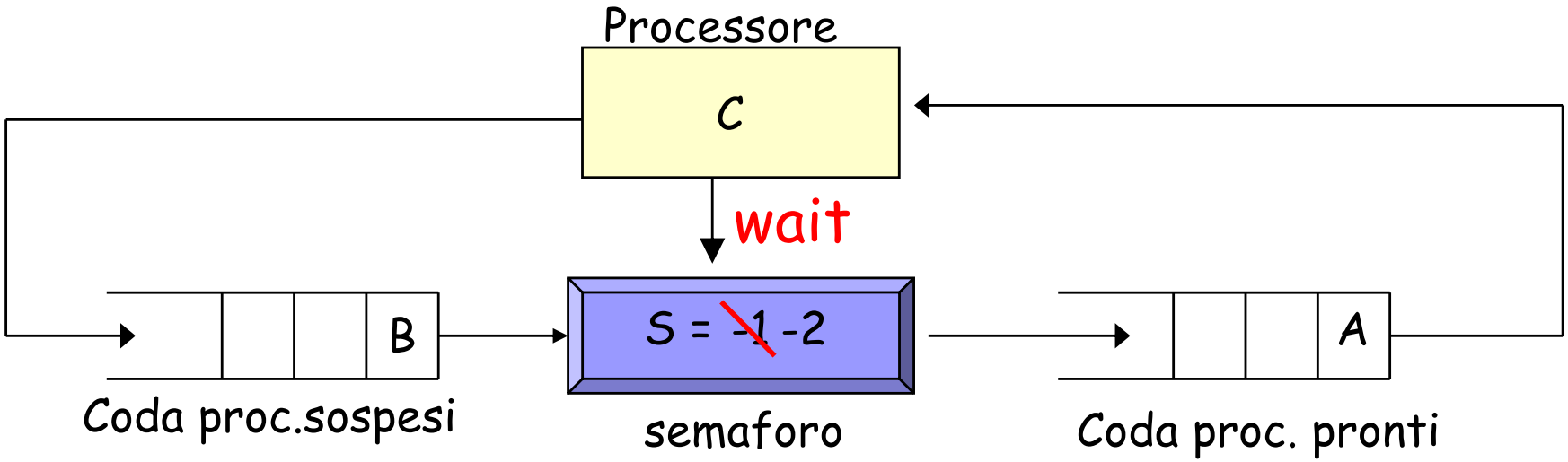
2) Si supponga che il processo A sia prelazionato in favore di B.

Nota: la sezione critica è ancora occupata da A.

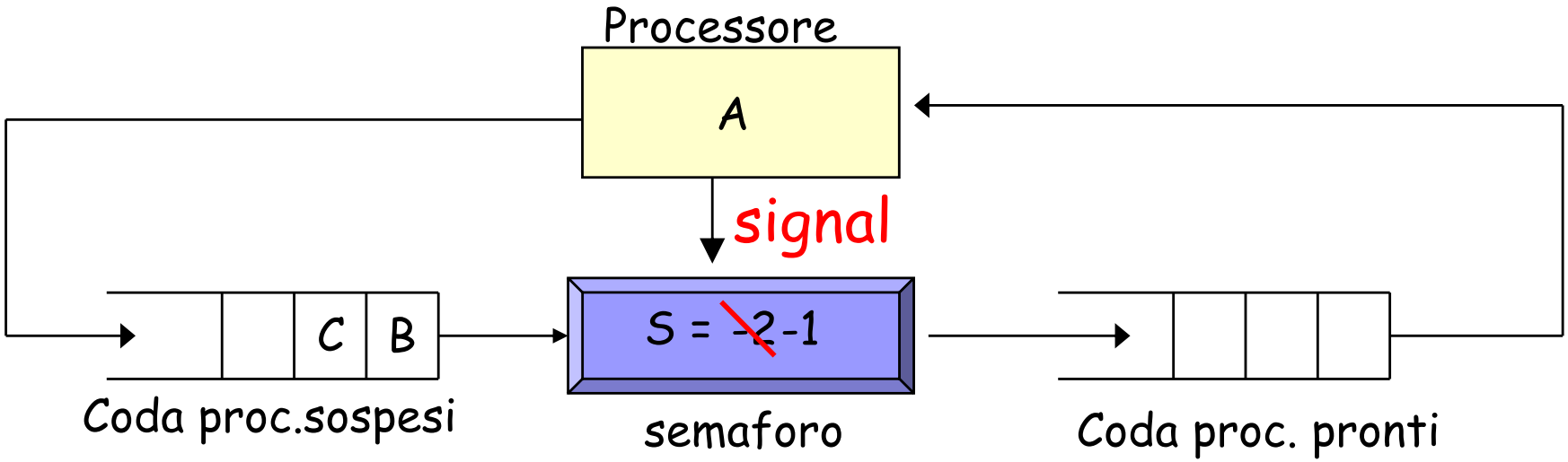


3) B esegue una wait. Il valore del semaforo è decrementato e B viene sospeso.

In questo modo, viene garantita la mutua esclusione tra A e B.



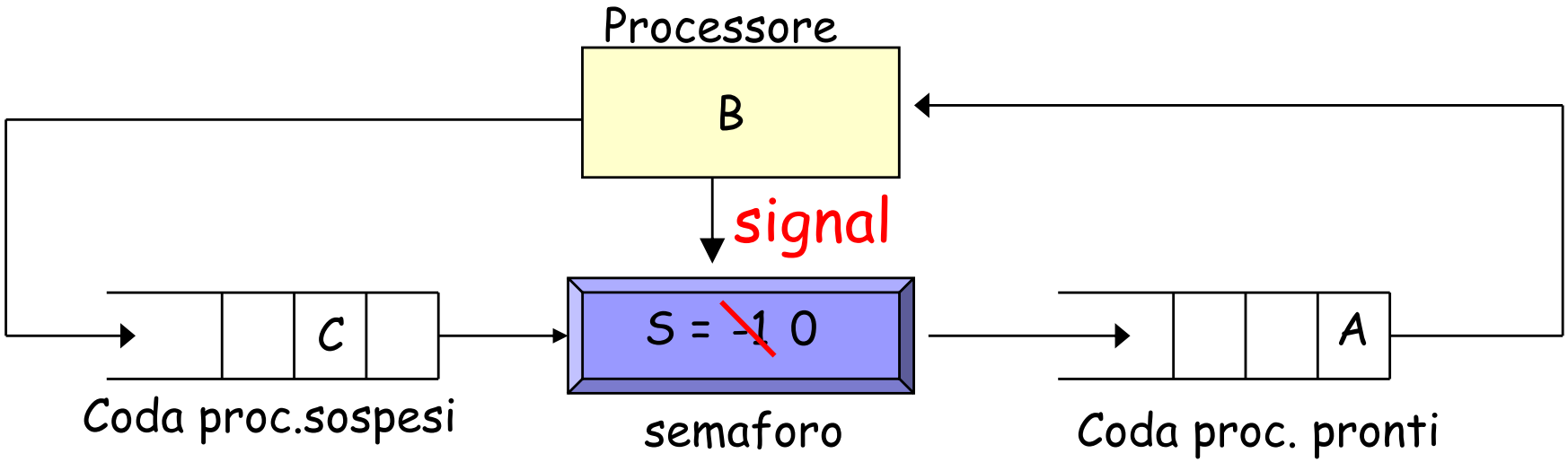
3) Anche C esegue una wait. Il valore del semaforo è decrementato, e anche C viene sospeso



4) Quando A riprende ad eseguire ed esce dalla sezione critica, effettua una signal.

Il processo B viene inserito tra i processi pronti.

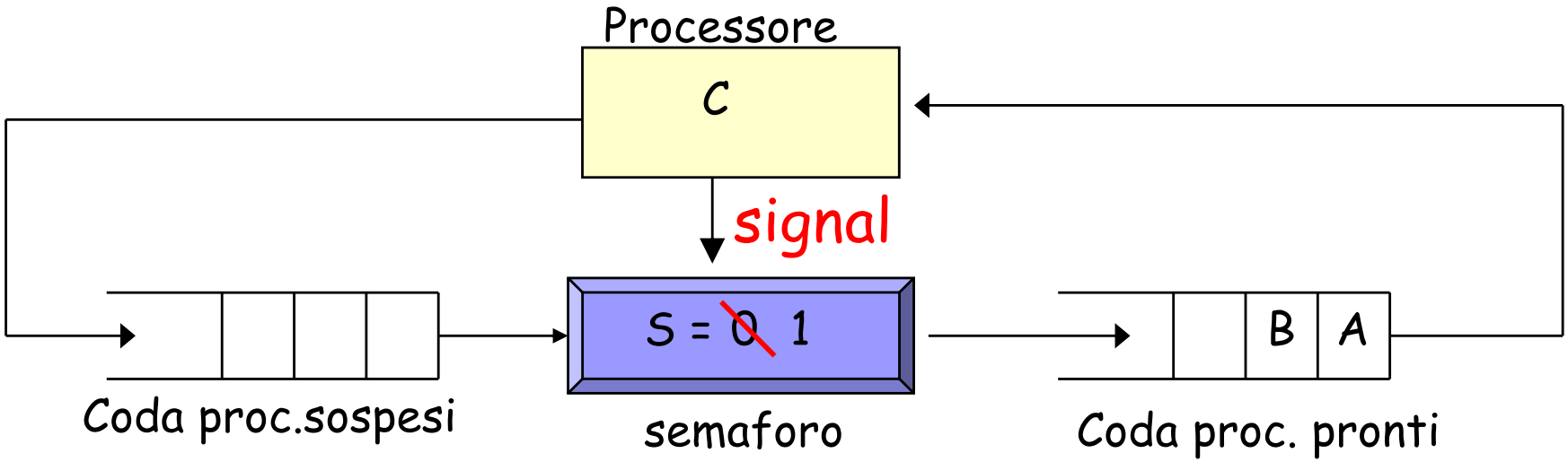
Si noti che A non viene sospeso.



5) B raggiunge il processore, e conclude la wait (durante cui era stato sospeso).

Dopo aver eseguito anch'esso la sezione critica, anche B effettua una signal.

C verrà spostato di nuovo tra i processi pronti



6) C esegue la sezione critica, ed effettua una signal.

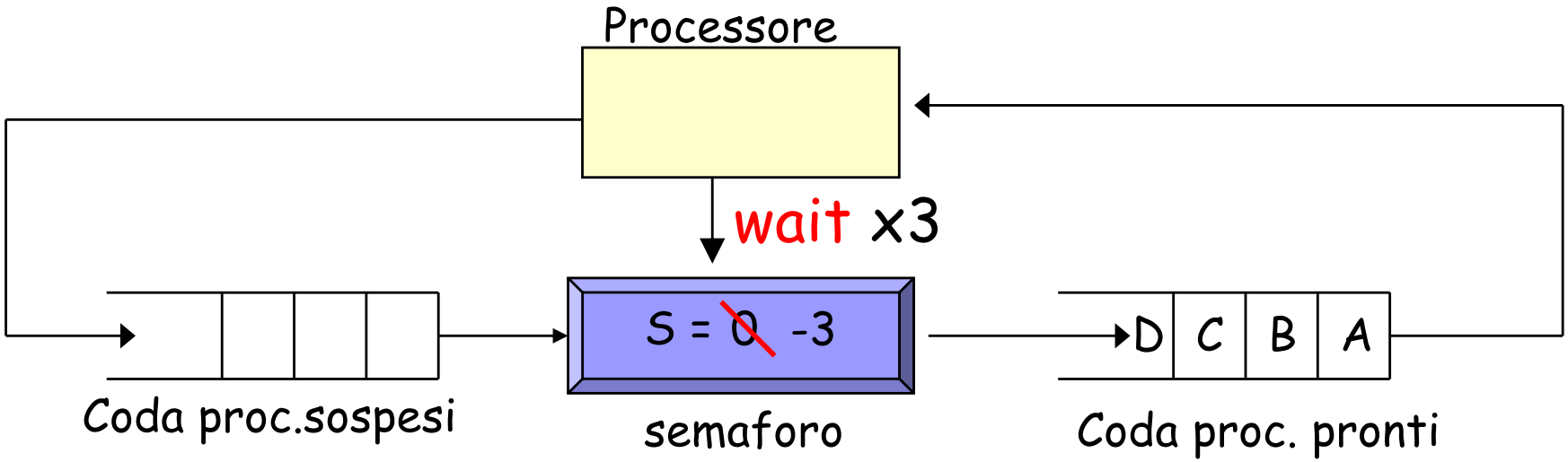
Viene ripristinato il valore iniziale del semaforo (1).

Esempio

Si hanno 4 processi concorrenti, denominati *A*, *B*, *C* e *D*.

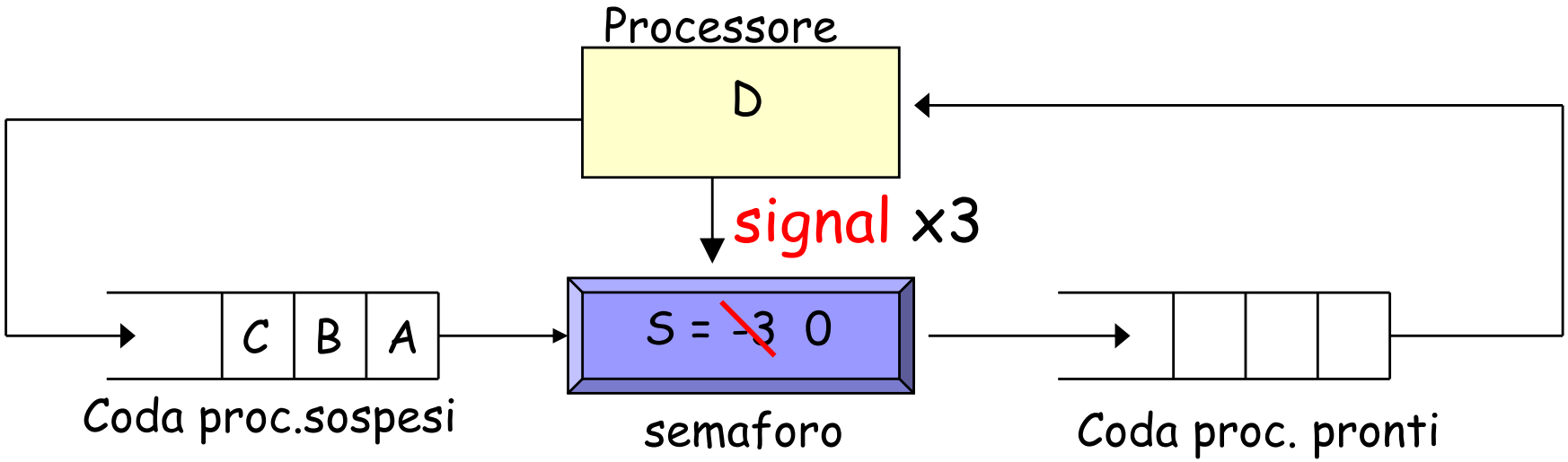
Si vuole che l'esecuzione di *A*, *B* e *C* dipenda dall'esecuzione del processo *D*.

A, *B*, *C* eseguiranno solo quando *D* avrà concluso (problema di cooperazione).



1) Il semaforo è inizialmente posto a 0.

I processi A, B e C si pongono in attesa effettuando wait (in tutti e 3 i casi la wait sospende i processi).



1) Il processo D può cadenzare i processi A, B e C eseguendo 3 volte signal

Quiz

1. L'insieme delle porzioni di codice che utilizza una stessa risorsa è detto:

   - ○ Risorsa Critica
   - ○ Sezione Critica

2. Quali di queste soluzioni al problema della mutua esclusione comporta una attesa attiva? (selezionare più di una)

   - ☐ Disabilitazione degli interrupt
   - ☐ Variabile lock
   - ☐ Istruzione Test-and-Set-Lock
   - ☐ Semafori

https://forms.office.com/r/YffRf87zp4