8. SINCRONIZZAZIONE TRA PROCESSI

Nei SO moderni, i processi vengono eseguiti in maniera concorrente, cioè si hanno più processi che vengono mandati in esecuzione e questi concorrono all'utilizzo della CPU e possono essere interrotti in qualunque momento (per esempio per priorità, etc.). I processi cooperanti possono condividere uno spazio logico di indirizzi, cioè codice e dati, oppure semplicemente dati, attraverso i file. Come si stabilisce chi deve manipolare per prima i dati? Ciò avviene mediante la sincronizzazione, che serve a mantenere la consistenza dei dati.

PRODUTTORE – CONSUMATORE:

Supponiamo di avere un processo produttore che produce informazioni che sono poi consumatore. I due processi hanno un buffer in comune in cui possono scrivere e dal quale possono leggere. Un esempio pratico di questa idea è il server-client: un server web fornisce (produce) pagine html e immagini, le quali vengono lette (consumate) dal browser web (client) che le richiede.

Definiamo il buffer nel seguente modo →

Supponiamo di avere a disposizione una certa struttura contenente al suo interno un certo numero di informazioni. Dichiariamo il *buffer* come un array di tanti dati di tipo elemento, dove il numero di elementi è 10.

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} elemento;
elemento buffer[BUFFER_SIZE];
```

Abbiamo altre variabili in comune tra il produttore e il consumatore che sono, oltre al buffer: *in*, *out*, *contatore* che vengono inizializzate a 0, capiremo a breve il loro funzionamento. Per adesso ci basta capire che questi sono i dati condivisi tra produttore e consumatore.

```
#define BUFFER_SIZE 10
typedef struct {
. . .
} elemento;
elemento buffer[BUFFER_SIZE];

int in = 0;
int out = 0;
int contatore = 0;
```

Vediamo ora come il processo **PRODUTTORE** è definito:

Supponiamo che il produttore abbia a disposizione un elemento appena prodotto che è di tipo elemento. Il produttore deve scrivere questo oggetto di tipo elemento all'interno del buffer. Per fare ciò, il produttore fa sempre un while(1) in quanto il suo compito è quello di inserire continuamente prodotti all'interno del buffer. All'interno di questo while notiamo che c'è subito l'uso della variabile *contatore* che abbiamo visto precedentemente come variabile comune che conta il numero di elementi che sono già presenti all'interno del buffer. Poi abbiamo una variabile *in* che rappresenta l'indice della successiva posizione libera nel buffer. Dunque, *contatore* ci dice quanti ce ne sono dentro, mentre *in* mi dice la prima posizione libera.

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

Nel codice del produttore c'è: *while(contatore == BUFFER_SIZE); /*do nothing*/,* significa che deve girare a vuoto nel momento in cui il buffer è pieno. Supponiamo che per un qualche motivo, il buffer comincia a svuotarsi, dunque il *contatore* è diverso da *BUFFER_SIZE*, di conseguenza il ciclo while fallisce e si procede nel codice. Quello che succede è che *buffer[in]* che ricordiamo indica la prima posizione libera all'interno di buffer, viene eguagliato ad *appena_prodotto*. Dopodiché si incrementa *in* in modulo *BUFFER_SIZE* (per poter ricominciare dall'inizio eventualmente scrivo nell'ultima posizione) *in* quanto la posizione *in* è stata appena occupata. Viene incrementato *contatore* in quanto ho incrementato anche il *buffer*.

Vediamo adesso come è definito il processo **CONSUMATORE**:

Naturalmente lo scopo del consumatore è quello di prelevare elementi dal buffer. Per fare ciò, fa sempre un *while(1)*. Appena entra in questo while nel momento in cui il *contatore* vale 0, e dunque non c'è nessun elemento all'interno del buffer, si rimane bloccati all'interno perché non deve fare niente.
Appena *contatore* diventa diverso da 0, significa che è stato inserito un elemento dal buffer, dunque esce dal while e procede con il codice. Quello che fa è andare a mettere all'interno della variabile *da_consumare*, il valore *buffer[out]*, dove *out* è l'indice della prima posizione piena del buffer. Naturalmente viene incrementato *out* per indicare che l'elemento in posizione *out* è stato prima preso correttamente. Viene decrementato *contatore* in quanto è stato eliminato un elemento da *buffer*.

```
elemento da_consumare;
while (1) {
  while (contatore == 0)
          ; /* do nothing */

  da_consumare = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  contatore --;
}
```

Prese separatamente, le procedure del produttore e del consumatore sono corrette, ma possono "non funzionare" se eseguite in concorrenza, sebbene siano corrette. In particolare, le istruzioni: *contatore--;* e *contatore++;* possono causare problemi se non atomiche (o si fa tutto o niente).

RACE CONDITION:

Quando si effettua l'incremento del contatore, quello che succede realmente è che: si prende il valore del contatore e lo si mette in un registro, si incrementa il valore del registro e si mette il valore del registro incrementato all'interno del contatore. Stesso ragionamento per il decremento.
L'incremento e il decremento del contatore non sono operazioni atomiche, dunque se il processo produttore/consumatore perde la CPU in una posizione intermedia dell'operazione di incremento/decremento del contatore, questo può causare problemi, del tipo:

```
registro1 = contatore
registro1 = registro1 + 1
contatore = registro1
```

```
LOAD R1, CONT
ADD 1, R1
STORE R1, CONT
```

```
registro2 = contatore
registro2 = registro2 - 1
contatore = registro2
```

```
LOAD R2, CONT
SUB 1, R2
STORE R2, CONT
```

Supponiamo che contatore valga inizialmente 5. La CPU viene assegnata al produttore al tempo T_0 e mettiamo che è arrivata al punto in cui deve fare *contatore++*. Per fare questa operazione deve fare le 3 operazioni descritte precedentemente, supponiamo che riesce a fare solo le prime 2 e poi perde la CPU, il valore di *registro* è 6. Subito dopo, la CPU viene presa dal processo consumatore la quale deve effettuare *contatore--*. Anch'esso fa solo le prime due operazioni di un decremento, quindi registro varrà 4. Una volta persa la CPU, viene assegnata nuovamente a produttore che deve fare il terzo passo dell'operazione di incremento, secondo cui deve assegnare a contatore, il valore di registro, dunque contatore vale 6.

Esempio: inizialmente contatore=5

T_0 produttore: registro1=contatore (registro1=5)
T_1 produttore: registro1=registro1+1 (registro1=6)
T_2 consumatore: registro2=contatore (registro2=5)
T_3 consumatore: registro2=registro2-1 (registro2=4)
T_4 produttore: contatore=registro1 (contatore=6)
T_5 consumatore: contatore=registro2 (contatore=4)

Il produttore riperde la CPU e viene assegnata al processo consumatore il quale deve completare l'operazione precedente e consumatore varrà 4. Quello che è successo, e che, tenendo conto che inizialmente contatore valesse 5, quello che ho fatto è stato aggiungere e togliere un elemento, di conseguenza ci si aspetta che contatore continui a valere 5. In realtà contatore vale 4, ma è un valore errato chiaramente. Tutto dipende dallo scheduling della CPU.

SEZIONE CRITICA:

In casi di questo genere, cioè quando si hanno parti di codice in comune (come nel nostro caso sull'alterazione di contatore) vengono chiamate **sezioni critiche o corse critiche**. Per evitare le sezioni critiche occorre **sincronizzare** i processi. Questo è un fenomeno di **Race Condition** secondo cui più processi accedono in concorrenza e modificano dati condivisi, l'esito dell'esecuzione dipende dall'ordine nel quale sono avvenuti gli accessi.

Supponiamo di avere *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. Il problema è assicurarsi che, quando un processo accede alla propria sezione critica, a nessun altro processo sia concessa l'esecuzione sulla propria sezione critica. L'esecuzione di sezioni critiche da parte di processi cooperanti è mutuamente esclusiva nel tempo.

La **soluzione** è quella di progettare un protocollo di cooperazione fra processi, a livello di codice:

- Ogni processo deve chiedere il permesso di accesso alla sezione critica, tramite una **entry section**, in modo che nessun altro processo può accedere alla propria sezione critica.
- La sezione critica è seguita da una **exit section**, in maniera tale da far capire agli altri processi che c'è la possibilità di accedere alla propria sezione critica.
- Il rimanente codice è non critico.

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

Le **entry section** ed **exit section** devono garantire:

- **Mutua esclusione**: se il processo P_i è in esecuzione nella sezione critica, nessun altro processo può eseguire la propria sezione critica.
- **Progresso**: se nessun processo è in esecuzione nella propria sezione critica ed esiste qualche processo che desidera accedervi, allora la sezione del processo che entrerà prossimamente nella propria sezione critica non può essere rimandata indefinitamente (evitare il deadlock).
- **Attesa limitata**: se un processo ha effettuato la richiesta di ingresso nella sezione critica, è necessario porre un limite al numero di volte che si consente ad altri processi di entrare nelle proprie sezioni critiche, prima che la richiesta del primo processo sia stata accordata (politica fai per evitare la starvation).

8.1 SOLUZIONE CON ALTERNANZA STRETTA

Questa è una soluzione semplice, in cui ci sono 2 processi P_0 e P_1 che utilizzano una variabile turn inizializzata a 0 inizialmente che serve a gestire la entry section e la exit section. Questi 2 codici garantiscono la stretta alternanza di ingresso alla sezione critica. Nel codice di P_0 inizialmente si va a verificare qual è il valore di turn, se è diverso da 0 allora non è il suo turno e cicla sul while. Appena turn vale 0 (viene settato dal processo P_1) allora è arrivato il suo turno, attende che la CPU gli venga data, e quando si entra nel primo while naturalmente fallirà e P_0 può entrare nella sua sezione critica.

```
do {
   while (turn != 0);
   sezione critica
   turn = 1;
   sezione non critica
} while (1);
```

algoritmo per il processo P_0

```
do {
   while (turn != 1);
   sezione critica
   turn = 0;
   sezione non critica
} while (1);
```

algoritmo per il processo P_1

Quando finisce, nella sua exit section, setterà turn uguale a 1 in modo da permettere al processo P_1 di entrare nella sua sezione critica.

Molti degli aspetti precedenti vengono risolti.

Ma, se tocca a P_0 (cioè P_1 ha finito con la sua sezione critica ed ha messo *turn=0*) ed intanto P_0 si attarda ancora nella sua sezione non critica (per un qualsiasi motivo) → P_1 rimane bloccato → si può violare il progresso; quindi, possibile deadlock. Di conseguenza, è stata violata una condizione per la buona scrittura di un codice, dunque questa non è corretta.

8.2 SOLUZIONE DI PETERSEN (1981)

Oltre ad utilizzare la variabile turn, si utilizza un vettore contenente due variabili booleane (*boolean flag[2]*) dove:

l'array *flag* si usa per indicare se un processo è pronto ad entrare nella propria sezione critica, se *flag[i]=TRUE* questo implica che il processo P_i è pronto per accedere alla propria sezione critica.

Rispetto alla soluzione con alternanza stretta, è stata modificata la entry section e la exit section di entrambi i processi. Il flag di P_0 risulta essere a true quando P_0 è pronto ad entrare nella propria sezione critica, setta *turn=1*. Queste due istruzioni ci fanno capire che P_0 vorrebbe entrare nella sua sezione critica (*flag[0]=true*), in questo momento è il turno del processo P_1 (*turn=1*), posso entrare nella sezione critica?

Questa domanda si traduce nel seguente modo: si va a controllare se *flag[1]* è uguale a true e *turn==1*. Se queste due condizioni sono vere allora P_1 si trova nella sua sezione critica e dunque cicla questo while (rimane bloccato a questa condizione).

Poiché P_1 si trova nella sua sezione critica, quando la finisce setta *flag[1]=false* e procede normalmente con la sua sezione non critica, successivamente perde la CPU che viene nuovamente data a P_0.

```
do {
   flag[0] = true;
   turn = 1;
   while (flag[1] && turn == 1);
   sezione critica
   flag[0] = false;
   sezione non critica
} while (1);
```

P_0

P_1

```
do {
   flag[1] = true;
   turn = 0;
   while (flag[0] && turn == 0);
   sezione critica
   flag[1] = false;
   sezione non critica
} while (1);
```

P_0, che ricordiamo era rimasto bloccato in *while(flag[1] && turn == 1).* Siccome ora *flag[1]=false* il while fallisce e il processo P_0 può entrare nella sua sezione critica.

Questa modalità risolve il problema della soluzione con alternanza stretta perché il processo P_1 termina la propria sezione non critica e ritorna alla prima istruzione della sua porzione di codice, dunque setta *flag[1]=true* e *turn=0*. Nello stesso tempo il processo P_0 è rimasto nella sua sezione non critica. Mentre nella soluzione precedente P_1 rimaneva bloccato, ora invece lui ha messo *flag[1]=true* per dire che è interessato all'utilizzo della sezione critica, ha messo *turn=0* per indicare che attualmente dovrebbe essere il turno di P_0, però quando fa a fare il test del while è ovviamente duplice e, turn vale ovviamente 0, invece, avendo supposto precedentemente che il processo P_0 si trovasse nella propria sezione non critica (quindi non ha ricominciato la propria porzione di codice), *flag[0]=false*, questo comporta che il while fallisce e il processo P_1 può entrare nella propria sezione critica.

Questa soluzione è perfettamente valida per 2 processi che tentano di utilizzare variabili comuni, ma se invece di essere 2 i processi fossero di più? In questo caso la soluzione di Petersen non sarebbe più corretta.

In generale, qualsiasi soluzione al problema della sezione critica richiede l'uso di un semplice strumento, detto **lock** (lucchetto). Per accedere alla propria sezione critica, il processo deve acquisire il processo di un lock, che restituirà al momento della sua uscita e uno dei tanti processi in attesa lo prenderà.

```
do {
   acquisisce il lock
   sezione critica
   restituisce il lock
   sezione non critica
} while (1);
```

Molti sistemi hanno dei sistemi hardware che mettono a disposizione per evitare i problemi descritti precedentemente, per esempio interdire le interruzioni mentre si modificano le variabili condivise. Inoltre, molte architetture forniscono speciali istruzioni atomiche implementate in hardware: queste permettono di controllare e modificare il contenuto di una parola di memoria (***TestAndSet***), o di scambiare il contenuto di due parole di memoria (***Swap***).

8.3 HARDWARE DI SINCRONIZZAZIONE (TestAndSet)

Questo è il codice della funzione *TestAndSet*. Viene eseguita atomicamente, ossia viene tutto eseguito insieme. La funzione prendere in input un puntatore ad un oggetto booleano e restituisce un valore booleano. La funzione salva in una variabile booleana *value* il valore della variabile booleana *object* passata in input. Successivamente setta a true il contenuto della variabile booleana *object* passata in input e come ultima istruzione ritorna la variabile *value* ossia il valore booleano passato in input alla funzione.

```
boolean TestAndSet (boolean *object)
{
boolean value = *object;
*object = TRUE;
return value;
}
```

Vediamo questa porzione di codice →

```
boolean lock = FALSE;
do {
   while TestAndSet(&lock);
   sezione critica
   lock = FALSE;
   sezione non critica
} while (1);
```

Inizialmente abbiamo una variabile booleana *lock* (rappresenta il **token**) inizializzata a **FALSE**.
Successivamente c'è do-while: nell'entry section c'è un while il cui contenuto è la chiamata a funzione definita precedentemente a cui gli passiamo in input il puntatore della variabile *lock*. Per quello che abbiamo detto prima sappiamo che la funzione restituirà **FALSE** e assegnerà a *lock* il valore TRUE. Poiché *TestAndSet* restituisce **FALSE** si esce dal while e si entra nella sezione critica.
lock=***TRUE*** quando un processo è nella sua sezione critica.
Nella exit section *lock* viene settato a **FALSE** in modo tale che un altro processo può entrare nella propria sezione critica (viene restituito il **token**).
Tutto funziona però bisogna considerare che il sistema operativo deve supportare tale operazione atomica (*TestAndSet*).

8.4 SEMAFORI

Un semaforo non è nient'altro che una variabile intera, chiamata *S*. Dopo l'inizializzazione si può accedere al semaforo solo attraverso due **operazioni atomiche predefinite**: **wait() – signal()** (da non confondere con le omonime funzioni Unix).

```
wait(S) {
  while S<=0 ; // do no+op
  S--;
}
```

```
signal(S){
  S++;
}
```

Il codice della **wait** è il seguente: prende come argomento il semaforo e quindi la variabile *S*. Non fa altro che andare a verificare il contenuto di *S*. Se è minore o uguale a 0 allora non fa niente. Se *S* è maggiore di 0, viene decrementata di uno e se ne esce. La **signal**, che prende come argomento il semaforo e quindi la variabile *S* non fa altro che incrementare *S* di uno.
L'obiettivo della variabile semaforo è quello di contare il numero di risorse che si hanno a disposizione.
Immaginiamo il caso in cui abbiamo 3 stampanti a disposizione e condivise.
Poiché le stampanti sono condivise, decido di mettere un semaforo a guardia di queste 3 stampanti. Di conseguenza la variabile *S* varrà 3. Se arriva un processo che decide di effettuare una stampa e dunque esegue la **wait** per verificare se c'è una risorsa (stampante) disponibile. Siccome *S* vale 3, il while fallisce e decremento *S* di uno, quindi le stampanti a disposizione sono diventate 2 in quanto una è stata occupata.
Supponiamo che altri 2 processi richiedono una stampa: di conseguenza viene chiamata due volte la **wait**, per due volte il while fallisce e alla fine ci ritroveremo che la variabile *S* vale 0. Quando *S* vale 0 significa che le 3 stampanti sono occupate. Se arriva un quarto processo che richiede una stampante, naturalmente invoca la **wait** però in questo caso rimarrà bloccato nel while. Questo processo quarto processo si sblocca quando uno dei 3 processi che stava utilizzando la stampante, finisce di utilizzarla. Appena finisce di utilizzarla viene invocata la **signal** che incrementa *S* di uno. In questo modo una stampante diventa nuovamente disponibile. Tutte le modifiche al valore del semaforo contenute nelle operazioni **wait()** e **signal()** devono essere eseguite in modo atomica. Mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore. Nel caso di **wait()** devono essere effettuate atomicamente sia la verifica del valore del semaforo che il suo decremento.

8.4.1 SEMAFORI BINARI

Un semaforo binario assume soltanto i valori 0 e 1.

```
Semaphore mutex=1;    // inizializzazione del semaforo mutex =1
```

Inizialmente la variabile viene inizializzata a 1 in quanto è in guardia di un'unica risorsa. Quando diventa uguale 0 allora l'unica risorsa è in mano ad un processo e gli altri processi che la richiedono devono aspettare. Generalmente i semafori binari vengono chiamati *mutex*.

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

Ricorda:

```
wait(S) {
   while S<=0;
   S--;
}
```

```
signal(S){
  S++;
}
```

Quando un processo vuole utilizzare questo semaforo, nell'entry section chiama la **wait** passando come parametro *mutex*, mentre nella exit section chiama la **signal** passandogli *mutex*.
Il semaforo binario, come lo abbiamo appena visto, risulta essere utile nel momento in cui si vuole andare a proteggere una sezione critica. Può essere utilizzato anche per la **sincronizzazione**: stabilire un ordine di esecuzione tra i processi.
Vogliamo in qualche modo condizionare l'esecuzione dei processi, utilizzando i semafori. Supponiamo che si voglia prima far eseguire il processo P1 e poi quello P2, indipendentemente da come lo scheduling della CPU selezioni i processi. Allora quello che si fa in questo caso è di inizializzare un semaforo *synch* a 0. Il processo P1 è composto da tutte le istruzioni di cui è composto e solo alla fine ci sta una signal su *synch* che non fa nient'altro che settare *synch* a 1.
Il processo P2 è fatto in modo che la sua prima istruzione sia la **wait** su *synch* in modo tale che se venisse schedulato prima del processo P1, si bloccherebbe al while della **wait** (perche *synch* vale 0).

Processo P_i

```
Semaphore synch=0;
```

```
P1:
      S1;
      signal(synch);

P2:
      wait(synch);
      S2;
```

Ricorda:

```
wait(S) {
   while S<=0;
   S--;
}
```

```
signal(S){
  S++;
}
```

Se il while fallisce allora significa che P1 è stato eseguito per prima, e dunque abbiamo applicato la **sincronizzazione**.

BUSY WAITING:

In generale, quando il valore di un semaforo è maggiore di 1, allora parliamo di **semaforo contatore** (esempio precedente riguardo le stampanti). I processi che desiderano utilizzare un'istanza della risorsa, invocano una **wait()** sul semaforo, decrementandone così il valore. I processi che ne rilasciano un'istanza, invocano **signal()**, incrementando il valore del semaforo. Quando il semaforo vale 0, tutte le istanze della risorsa sono allocate e i processi che le richiedono devono sospendersi sul semaforo fino a che esso ridiventa positivo.
Questo fenomeno viene chiamato **busy waiting**. Come fare per risolvere questo problema? l'idea potrebbe essere la seguente: quando un processo chiama la **wait** passando come parametro un semaforo che vale 0, anziché essere bloccato in quel while, si potrebbe pensare di far perdere la CPU e di farlo accodare (tramite il Process Control Block) ad una lista linkata di tutti i processi che stanno aspettando per una certa risorsa.

Tutto ciò si traduce nella seguente struttura: in questo caso il semaforo non è più costituito da un singolo valore numerico, ma piuttosto da un record costituito dal valore numerico e da una lista di record, dove ogni elemento della lista è un processo (PCB del processo stessao) che vuole utilizzare le risorse messe a disposizione dal semaforo, ma che in un certo momento non sono disponibili.

```
typedef struct {
  int valore;
  struct processo *lista;
} semaphore;
```

Un semaforo del genere appena descritto utilizza due operazioni:

- **block**: posiziona il processo che richiede di essere bloccato nell'opportuna cosa di attesa, ovvero sospende il processo che la invoca.
- **wakeup**: rimuove un processo dalla coda di attesa e lo sposta nella ready queue.

Il codice della **wait**, come conseguenza di quanto abbiamo appena detto, subisce delle modifiche: innanzitutto in questo caso si prende in input alla funzione un puntatore al semaforo che ricordiamo essere un record. Come prima cosa che si fa è decrementare *S->value*, ossia il numero di risorse a disposizione. Se *S->value* è diventato un valore minore di 0 allora significa che non ci sono più risorse a disposizione e allora viene aggiunto il processo alla lista e invoca la **block** in modo da fargli perdere la CPU.

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

Per quanto riguardo la **signal**, la prima cosa che si fa è incrementare *S->value*. Se adesso *S->value* è un valore minore o uguale a 0 (perché significa che c'è ancora qualche processo nella coda dei processi in attesa della risorsa) allora si rimuove il processo P da *S->list* e si invoca la **wakeup**.

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

Un'osservazione: se *S->value* è negativo, allora il valore assoluto di *S-value*: |*S-value*| rappresenta il numero dei processi che attendono il semaforo. Ciò avviene a causa dell'inversione dell'ordine delle operazioni di decremento e verifica del valore nella **wait()**.

La realizzazione di un semaforo con coda di attesa può condurre a situazioni in cui ciascun processo attende l'esecuzione di un'operazione **signal()**, che solo uno degli altri processi in coda può causare. Più in generale, si verifica una situazione di **deadlock**, quando due o più processi attendono indefinitamente un evento che può essere causato soltanto da uno dei due processi in attesa.

Supponiamo ci siano due semafori **S** e **Q** inizializzati entrambi ad 1 e che ci siano due processi P0 e P1 che hanno le seguenti porzioni di codice:

```
P0
wait(S);
wait(Q);
.....
signal(S);
signal(Q);
```

```
P1
wait(Q);
wait(S);
.....
signal(Q);
signal(S);
```

Supponiamo venga eseguito prima il processo P0, che fa la **wait** su **S**: tenendo conto che **S** vale 1, riesce a prendere la risorsa. Immaginiamo che dopo aver fatto la **wait** su **S**, perde la CPU in favore di P1: P1 fa la **wait** su Q che inizialmente vale 1 dunque continua. Successivamente perde la CPU che ritorna a P0. P0 fa la **wait** su Q che in questo momento vale 0, dunque decrementa Q e viene piazzato in coda.
Supponiamo che ora perde la CPU che ripassa a P1 che farà la **wait** su S che valeva 0, verrà decrementato il valore e andrà nella coda dei processi S. Per come abbiamo costruito i codici, nessuno li sbloccherà (P0 può essere sbloccato da P1 e viceversa) e dunque ci siamo messi in una situazione di deadlock. È importante stare attenti quando si scrive il codice.

Bisogna stare anche attenti al fenomeno della **Starvation**, questa si può verificare qualora i processi vengano rimossi dalla lista di attesa associata al semaforo in modalità LIFO. In teoria si potrebbe gestire mediante FIFO.

Facciamo ora un passo indietro e vediamo come risolvere il problema **produttore-consumatore** utilizzando i semafori:

```
semaphore full, empty, mutex;

// inizialmente full = 0, empty = n, mutex = 1
```

Abbiamo a disposizione 3 semafori *full, empty e mutex*. 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.

Questo è il codice di produttore, che continuamente produce un elemento. Dopo aver prodotto l'elemento fa una **wait** su *empty*. Successivamente esegue una **wait** su *mutex*. Si può osservare come *mutex* sia a guardia del buffer. Successivamente c'è una **signal** su *buffer* con cui si indica che le operazioni sul buffer sono terminate e poi una **signal** su *full*, in maniera tale da incrementare il numero delle posizioni piene.

```
do {
... /* produce un elemento in
next_produced */
...
wait(empty);
wait(mutex);
... /* inserisce next_produced nel
buffer */
...
signal(mutex);
signal(full);

} while (true);
```

Consumatore esegue le operazioni simmetriche secondo cui: innanzitutto le operazioni sul buffer sono controllare dal semaforo *mutex.* Inizialmente il consumatore può andare ad eseguire le operazioni quando all'interno del buffer c'è qualcosa al suo interno, dunque si effettua una **wait** su *full.* Una volta effettuate le operazioni sul buffer c'è una **signal** su *empty* in modo da far capire che è stato preso un elemento, c'è una posizione libera in più nel buffer.

```
do {

wait(full);
wait(mutex);
... /* sposta un elemento dal buffer
in next_consumed */
...
signal(mutex);
signal(empty);
... /* consuma un elemento in
next_consumed */

... } while (true);
```

In questo modo sono risolti tutti i problemi e abbiamo di conseguenza risolvo il problema **produttore-consumatore**.