**ESERCIZIO PRODUTTORE CONSUMATORE (semafori)**

In un sistema operativo, i thread A (produttore) e B (consumatore) cooperano scambiandosi messaggi attraverso un buffer di 10 celle, ciascuna capace di contenere un messaggio e utilizzando i semafori *mutex* (valore iniziale 1), *CelleLibere* (valore iniziale 10) e *MessaggiGiacenti* (valore iniziale 0).
A meno delle operazioni di sincronizzazione e di quelle per la mutua esclusione (necessarie perché si suppone che le operazioni *insert(item)* e *remove_item* non siano indivisibili), i frammenti rilevanti dei thread A e B sono i seguenti:

```
A (produttore):
........
While (true) {
      item = produce_item();
      insert(item);
      }
........

B (consumatore):
........
While (true) {
      item= remove_item();
      consume(item);
      }
........
```

Completare questi frammenti con le operazioni per la sincronizzazione e la mutua esclusione.

SOLUZIONE

```
A (produttore):
........
While (true) {
      item = produce_item();
      P(&CelleLibere);
      P(mutex);
      insert(item);
      V(mutex);
      V(&MessaggiGiacenti);
      }
........

B (consumatore):
........
While (true) {
      P(&MessaggiGiacenti)
      P(mutex);
      item= remove_item();
      V(mutex);
      V(&CelleLibere);
      consume(item);
      }
........
```

## ESERCIZIO parcheggio

## Variabili condizione

Un parcheggio per auto della capacità di 10 posti è dotato di un unico ingresso I e un'unica uscita U. L'ingresso e l'uscita sono controllate da sbarre. Le auto sono thread di uno stesso processo.
Per l'utilizzo del parcheggio, i thread condividono la variabile *PostiDisponibili,* con valore iniziale 10, la variabile *lockParcheggio* di tipo *lock*, e la variabile *PostoLibero,* di tipo *condition.*
Si chiede di risolvere il problema completando lo pseudo codice sotto riportato con le opportune operazioni di sincronizzazione.

## Semafori

Risolvere il problema precedente utilizzando i semafori opportuni.

## SOLUZIONE (variabili condizione)

<l'auto arriva all'ingresso I>
**lock.acquire**(lockParcheggio)
**while** PostiDisponibili == 0 **PostoLIbero.wait**(LockParcheggio);
*//l'auto attende la disponibilità di un posto//*
<la sbarra si alza e l'auto entra nel parcheggio>
<l'auto sceglie un posto libero e lo occupa >
PostiDisponibili - -
**lock.release**(LockParcheggio);
<auto parcheggiata>
**lock.acquire**(LockParcheggio);
<l'auto libera il posto che aveva occupato e si dirige all'uscita U>
PostiDisponibili ++
**PostoLibero.signal**(LockParcheggio);
**lock.release**(LockParcheggio);
<la sbarra si alza e l'auto esce dal parcheggio>

## SOLUZIONE (semafori)

<l'auto arriva all'ingresso I>

**P(postiDisponibili);**

*//l'auto attende la disponibilità di un posto//*

<la sbarra si alza e l'auto entra nel parcheggio>

<l'auto sceglie un posto libero e lo occupa >

<auto parcheggiata>

<l'auto libera il posto che aveva occupato e si dirige all'uscita U>

**V(postiDisponibili);**

<la sbarra si alza e l'auto esce dal parcheggio>

## ESERCIZIO lettori/scrittori con variabili condizione

Nel problema dei lettori-scrittori, *n* processi lettori e *m* processi scrittori competono per l'accesso ad una struttura dati condivisa. I processi lettori utilizzano la struttura dati esclusivamente per leggere, senza modificare i dati, mentre i processi scrittori la utilizzano senza restrizioni, con la possibilità di modificare i dati.
Per evitare interferenze, i processi scrittori accedono alla base dati in mutua esclusione (rispetto agli altri scrittori e ai lettori), mentre i processi lettori accedono alla base dati in mutua esclusione rispetto agli scrittori, ma senza vincolo di mutua esclusione rispetto agli altri lettori (in altre parole, più lettori possono accedere concorrentemente alla struttura dati).
Per la soluzione del problema, si utilizzano i seguenti dati condivisi da tutti i processi:

- *LettoriAmmessi*: intero non negativo; valore iniziale 0
- *LettoriInAttesa*: intero non negativo; valore iniziale 0
- *ScrittoreAmmesso*: intero nel intervallo [0, 1]
- *ScrittoriInAttesa*: intero non negativo; valore iniziale 0

Nell'ipotesi che i lettori e gli scrittori siano thread di uno stesso processo, si chiede di risolvere il problema utilizzando la variabili *P_mutex,* di tipo *lock,* per la mutua esclusione e le variabili *AttesaLettori* e *AttesaScrittori*, di tipo *condition*, rispettivamente per la sospensione dei lettori e degli scrittori.

**SOLUZIONE (variabili condizione)**
*{nota: questa soluzione dà precedenza ai lettori e non evita l'attesa indefinita per gli scrittori}*

```
Lettore_i
{……..
    // prologo dell'accesso in lettura//
    lock.acquire(P_mutex);
    LettoriInAttesa++;
    while  ScrittoreAmmesso<> 0  AttesaLettori.wait(P_mutex);
    //il lettore attende fino al rilascio della base dati da parte dello scrittore//
    LettoriInAttesa - -; LettoriAmmessi ++;
    lock.release(P_mutex);

   < esegue accesso in lettura >

    // epilogo dell'accesso in lettura//
    lock.acquire(P_mutex);
    LettoriAmmessi - -;
    if (LettoriAmmessi== 0)  AttesaScrittori.signal(P_mutex);
    //si riattiva, se esiste, uno scrittore in attesa di accedere//
    lock.release(P_mutex);
    ……..
}

Scrittore_j
     {
    ……..
    // prologo dell'accesso in scrittura//
    lock.acquire(P_mutex);
    ScrittoriInAttesa ++;
     while (LettoriAmmessi > 0 or ScrittoreAmmesso<> 0)      AttesaScrittori.wait(P_mutex);
    //lo scrittore attende fino al rilascio della base dati da parte dello scrittore o dei lettori che la utiilizzano//
    ScrittoriInAttesa - -;  Scrittore Ammesso= 1,
    lock.release(P_mutex);

  < esegue accesso in scrittura >

  // epilogo dell'accesso in scrittura//
    lock.acquire(P_mutex);
    ScrittoreAmmesso= 0;
    if (LettoriInAttesa>0)
    AttesaLettori.broadcast(P_mutex);
    //si riattivano tutti i lettori  in attesa di accedere//
    else  AttesaScrittori.signal(P_mutex);
```

```
    //si riattiva, se esiste, uno scrittore in attesa di accedere//
    lock.release(P_mutex);
    ……..
}
```

**ESERCIZIO Problema dei lettori e degli scrittori con semafori**

Una struttura dati (*base dati*) è condivisa da un insieme di thread, che appartengono a una delle due seguenti categorie:

1) *Thread lettori*: accedono alla *base dati* esclusivamente per leggere, senza modificare i dati;

2) *Thread scrittori*: accedono alla *base dati* senza restrizioni, con la possibilità di modificare i dati.

Per evitare interferenze, sono imposti seguenti vincoli:

- i thread scrittori accedono alla base dati in mutua esclusione, rispetto agli altri scrittori e ai lettori;
- i thread lettori accedono alla base dati in mutua esclusione rispetto agli scrittori, ma senza vincolo di mutua esclusione rispetto agli altri lettori (in altre parole, più lettori possono utilizzare la base dati concorrentemente).

Il problema viene risolto con una politica che consente l'accesso alternativamente a uno scrittore e a un insieme di lettori, ed evita l'attesa indefinita per gli scrittori. Precisamente, per le richieste valgono le seguenti clausole:

a) Quando la base dati non è utilizzata da nessun processo:
   a1) accede il primo processo (lettore o scrittore) che ne fa richiesta;

b) quando la base dati è utilizzata da uno scrittore, oppure da uno o più lettori:
   b1) se un lettore richiede l'accesso e la base dati è utilizzata da uno scrittore, il lettore richiedente viene sospeso;
   b2) se un lettore richiede l'accesso e la base dati è utilizzata da uno o più lettori, il lettore ottiene l'accesso *a condizione che non vi siano scrittori in attesa*;
   b3) se un lettore richiede l'accesso, la base dati è utilizzata da uno o più lettori e vi è almeno uno scrittore in attesa di accedere, il lettore richiedente viene sospeso *(questa clausola evita l'attesa indefinita per gli scrittori)*;
   b4) se uno scrittore richiede l'accesso e la base dati è utilizzata da uno scrittore oppure da uno o più lettori, lo scrittore si sospende.

Per i rilasci valgono le seguenti clausole:

c) quando uno scrittore rilascia la base dati:
   c1) se esistono lettori in attesa, tutti questi lettori sono riattivati e ottengono l'accesso;
   c2) se non esistono lettori in attesa ma esiste almeno uno scrittore in attesa, il primo di questi scrittori ottiene l'accesso;
   c3) se non esistono lettori o scrittori in attesa, si torna al caso a).

d) quando un lettore rilascia la base dati:
   d1) se altri lettori stanno ancora utilizzando la base dati, continua l'accesso in lettura da parte di questi thread;
   d2) se non vi sono altri lettori che utilizzano la base dati ed esiste almeno uno scrittore in attesa, il primo di questi scrittori ottiene l'accesso;
   d3) altrimenti si torna al caso a).

Per la soluzione del problema, si utilizzano i seguenti dati condivisi da tutti i thread:

- *LettoriAmmessi*: intero non negativo; valore iniziale 0
- *LettoriInAttesa*: intero non negativo; valore iniziale 0
- *ScrittoreAmmesso*: intero non negativo; valore iniziale 0
- *ScrittoriInAttesa*: intero non negativo; valore iniziale 0

e i seguenti semafori:

- *mutex* (valore iniziale 1): semaforo utilizzato per la mutua esclusione sulle sezioni critiche con le quali i thread verificano la condizione di accesso;
- *AttesaLettori (*valore iniziale 0)*:* semaforo utilizzato per la sospensione dei lettori;
- *AttesaScrittori (*valore iniziale 0)*:* semaforo utilizzato per la sospensione degli scrittori; valore iniziale 0.

Nota importante: per evitare lo stallo, la sospensione sui semafori *AttesaLettori* e *AttesaScrittori* deve avvenire dopo aver rilasciato la mutua esclusione instaurata per la verifica delle condizioni di accesso alla base dati. A questo scopo i thread registrano l'esito della verifica della condizione di accesso nella variabile locale *OK* e a seconda del valore assegnato a questa variabile, eseguiranno eventualmente la primitiva *P* dopo aver rilasciato la mutua esclusione.

Si chiede di completare lo pseudo-codice sotto riportato, inserendo opportunamente le operazioni sui semafori *mutex, AttesaLettori* e *AttesaScrittori.*

**SOLUZIONE semafori**

```
ProcessoLettore_i
{
   ……..
   // prologo dell'accesso in lettura//
   OK = 0 ;
   P(mutex);
   if ScrittoriInAttesa == 0 and ScrittoreAmmesso == 0      {
   // clausole a1 e  b2  //
          LettoriAmmessi ++; OK= 1;
                                                              }
   else LettoriInAttesa ++;  / /clausola b1 //
   V(mutex);
   if OK == 0  P(AttesaLettori);
   // la sospensione avviene dopo il rilascio della mutua esclusione, per evitare lo stallo //
  < esegue accesso in lettura >
   // epilogo dell'accesso in lettura//
   P(mutex);
   LettoriAmmessi -- ;
   if LettoriAmmessi== 0  and  ScrittoriInAttesa > 0          {
          ScrittoreAmmesso = 1; ScrittoriInAttesa --;
          // clausola d2 //
           V(AttesaScrittori);
                                                              }
   V(mutex)
   ……..
}

ProcessoScrittore_j
    {
   ……
   // prologo dell'accesso in scrittura//
   OK = 0;
   P(mutex);
   if LettoriAmmessi== 0 and ScrittoreAmmesso == 0 {
           ScrittoreAmmesso =1; OK= 1;  // clausola a1 //
                                                              }
   else ScrittoriInAttesa ++;   // clausola b4 //
   V(mutex);
   if OK == 0
       P(AttesaScrittori);
  < esegue accesso in scrittura >
  // epilogo dell'accesso in scrittura//
   P(mutex);
   ScrittoreAmmesso = 0;
   if LettoriInAttesa > 0       {
            while LettoriInAttesa> 0  {
            LettoriInAttesa --; LettoriAmmessi++;   // clausola c1 //
            V(AttesaLettori);
                                          }
   else if ScrittoriInAttesa > 0 {
                ScrittoreAmmesso = 1; ScrittoriInAttesa --;   // clausola c2 //
                V(AttesaScrittori);
                                  }
                                 }
   V(mutex);
```

## ESERCIZIO Problema del barbiere dormiglione
## variabili condizione

Nel problema del "barbiere dormiglione" si considera un negozio gestito da un unico barbiere, con una poltrona per il servizio dei clienti e un numero illimitato di sedie per l'attesa. Il barbiere e i clienti sono thread di uno stesso processo e la poltrona è una risorsa, che può essere assegnata a un cliente per il taglio dei capelli, oppure utilizzata dal barbiere per sonnecchiare. All'apertura del negozio e quando non ci sono clienti in attesa di servizio, il barbiere occupa la poltrona, fino all'arrivo del primo cliente.

Quando entra nel negozio, il generico cliente ha il seguente comportamento:

- se il barbiere è addormentato, lo risveglia provocando il rilascio della poltrona, che occupa immediatamente;
- altrimenti (ciò avviene quando il barbiere è attivo per eseguire il taglio dei capelli ad un altro cliente) si blocca, accomodandosi su una delle sedie in attesa del suo turno;
- quando arriva il suo turno, è riattivato dal barbiere ed occupa la poltrona;
- dopo il taglio dei capelli, paga ed esce dal negozio.

Dopo che un cliente ha occupato la poltrona, il barbiere esegue il taglio dei capelli e al termine:

- se ci sono clienti in attesa del proprio turno, riattiva il primo;
- altrimenti occupa la poltrona e si addormenta.

Il problema viene risolto utilizzando i seguenti dati condivisi:

- *BarbiereAttivo, SediaOccupata: booleano;*
- *ClientiInAttesa: intero; valore iniziale 0;*

e inoltre le seguenti variabili:

- *Mux:, di tipo lock;*
- *AttesaTurno, di tipo condition;*
- *AttesaBarbiere, di tipo condition;*
- *TaglioCapelli, di tipo condition. (il cliente si sospende su questa variabile condizione all'inizio del taglio dei capelli e viene riattivato dal barbiere alla fine. Questa attesa implica l'assegnazione della poltrona al cliente).*

Si chiede di completare il programma del barbiere e il frammento di codice eseguito da ogni cliente che entra nel negozio.

## Semafori

Oltre ai dati condivisi di cui sopra, si usino i seguenti semafori:

- *mutex: valore iniziale 1 (per la mutua esclusione);*
- *AttesaBarbiere: valore iniziale 0 (utilizzato dal barbiere per addormentarsi sulla poltrona. L'attesa su questo semaforo implica l'assegnazione della poltrona al barbiere);*
- *AttesaTurno: valore iniziale 0 (per i clienti che attendono il taglio. Le modalità di utilizzo garantiscono che il valore di questo semaforo sia sempre 0)*
- *TaglioCapelli: valore iniziale 0 (il cliente si sospende su questo semaforo all'inizio del taglio dei capelli e viene riattivato dal barbiere alla fine. L'attesa su questo semaforo implica l'assegnazione della poltrona al cliente).*

Per ogni cliente è inoltre definita la variabile privata *attende* utilizzata per evitare la sospensione all'interno della sezione critica nella quale viene decisa la sospensione. Allo stesso fine il barbiere utilizza la variabile privata *dorme.*
Notare che la risorsa poltrona non viene gestita esplicitamente, perché le assegnazioni e i rilasci sono implicite nelle operazioni sui semafori *AttesaBarbiere* e *TaglioCapelli*.
Si chiede di completare il programma del barbiere e il frammento di codice che controlla i clienti che entrano nel negozio, inserendo le opportune operazioni sui semafori.

**SOLUZIONE (variabili condizione)**
**Barbiere**

```
{
    lock.acquire(Mux); BarbiereAttivo= false; lock.release(Mux);
    /inizializzazione; eseguita prima della generazione dei processi "cliente" //
    while(true) {
        AttesaTurno.signal();
        // se vi sono clienti inattesa riattiva il primo; altrimenti non ha effetto //
        lock.acquire(Mux);
        SediaOccupata= true; <si siede>;
        while(BarbiereAttivo== false) AttesaBarbiere.wait(Mux);
            // si sospende rilasciando la mutua esclusione; sarà risvegliato da un cliente e riacquisirà la mutua
            esclusione //
        SediaOccupata= false ; <si alza>;
        // un cliente e' pronto per il taglio dei capelli: quello riattivato dal barbiere o quello che ha risvegliato il
        barbiere //
```

```
        while(SediaOccupata== false) InizioTaglio.wait(Mux);
        <esegue il taglio dei capelli>
        BarbiereAttivo= false;
        TaglioCapelli.signal();
        lock.release(Mux);
        <presenta il conto al cliente, che libera la poltrona>
        // ripete il ciclo, attivando un cliente in attesa o addormentandosi //
    }
}
```

**Cliente**

```
{
……
//Frammento di codice//
< entra nel negozio >
lock.acquire(Mux);
while (BarbiereAttivo) AttesaTurno.wait(Mux);
BarbiereAttivo= true;
AttesaBarbiere.signal(Mux);
// il barbiere è in attesa e viene risvegliato; il cliente che risveglia è appena entrato o è stato riattivato dal barbiere //
// un cliente e' pronto ad occupare la poltrona: si tratta di quello appena entrato in negozio, o di quello riattivato dal barbiere //
SediaOccupata= true ; // occupa la poltrona //
InizioTaglio.signal(Mux);
while (BarbiereAttivo) TaglioCapelli.wait(Mux);
// attende il taglio dei capelli rilasciando la mutua esclusione, che riacquista dopo la riattivazione //
<paga ed esce dal negozio>
lock.release(Mux);
// Fine del frammento di codice //
……
}
```

**SOLUZIONE (semafori)**

**Barbiere**

```
{
BarbiereAddormentato= true; P(AttesaBarbiere);
// la riga precedente inizializza il negozio ed è eseguita prima della generazione dei  thread "cliente" //
while(true)         {
        <esegue il taglio dei capelli>
        V(taglioCapelli);
        P(mutex);
        if ClientiInAttesa> 0         {
                // riattiva un cliente in attesa del turno e libera la poltrona //
                ClientiInAttesa --;  dorme= false;
                V(AttesaTurno);
                }
        else  { BarbiereAddormentato= true; dorme= true};
        // occupa la poltrona per dormire, sospendendosi //
        V(mutex);
        if dorme  P(AttesaBarbiere);  // la sospensione avviene dopo il rilascio della mutua esclusione //
        }
}
```

**Cliente** *//Frammento di codice//*

```
{
……
< entra nel negozio >
P(mutex);
if BarbiereAddormentato {
        BarbiereAddormentato= false; attende= false;
        V(AttesaBarbiere);
        // sveglia il barbiere e occupa la poltrona rilasciata dal barbiere //
```

```
        }
else    {
        // il barbiere è sveglio e sta servendo un cliente //
        ClientiInAttesa ++; attende= true;
        }
V(mutex);
if attende
        P(AttesaTurno); // la sospensione avviene dopo il rilascio della mutua esclusione //
        // Il cliente attende il suo turno. Sarà riattivato dal barbiere e troverà la poltrona libera //
P(TaglioCapelli);
// siede sulla poltrona e attende il taglio dei capelli. Al termine…//
<paga ed esce dal negozio>
// Fine del frammento di codice //
}
```

## ESERCIZIO fast food (variabili condizione)

Un ristorante Fast-Food che somministra un unico tipo di hamburger è gestito da un commesso e da un cuoco, che interagiscono attraverso uno scaffale, dove possono essere accumulati fino a 10 hamburger pronti per essere serviti. II cuoco prepara gli hamburger in sequenza e li depone sullo scaffale, eventualmente attendendo che ci sia un posto disponibile. Il generico cliente fa il suo ordine al commesso e quindi attende la consegna dell'hamburger. Il commesso riceve in sequenza gli ordini, preleva gli hamburger dallo scaffale (eventualmente attendendo la disponibilità) e li consegna ai clienti, riattivandoli.

Il ristorante è un processo, i cui thread sono i clienti, il commesso e il cuoco. Lo scaffale è un buffer di 10 celle, ciascuna capace di contenere un hamburger.

Per l'interazione tra i thread si utilizzano le variabili *MutexOrdini* e *MutexScaffale* di tipo *lock,* e le variabili *AttesaOrdine, consegna, HamburgerNelloScaffale* e *PostoLiberoNelloScaffale* di tipo *condition.*

Si utilizano inoltre le variabili intere (condivise) *OrdiniPendenti* e *HamburgerPronti,* con valore iniziale 0.

Il generico cliente esegue la seguente funzione:

```
thread GenericoCliente
.............
<entra nel negozio>;
lock.acquire(MutexOrdini);
OrdiniPendenti ++;
<ordina un hamburger e paga>;
AttesaOrdine.signal (MutexOrdini); //ha effetto solo se l'commesso è sospeso//
lock.release(MutexOrdini);
lock.acquire(MutexFittizio);
//l'associazione della condition a un mutex è obbligatoria, anche quando, come in questo caso, si voglia una sospensione incondizionata/
consegna.wait (MutexFittizio)
//attende incondizionatamente la consegna//
lock.release(MutexFittizio);
.........
```

dove la variabile *MutexFittizio* è introdotta unicamente per motivi sintattici e non viene utilizzata altrove.

Si chiede di completare le funzioni eseguite dal thread *commesso* e dal thread *Cuoco,* inserendo negli schemi sotto riportati le operazioni sulle variabili di tipo *lock* e sulle variabili di tipo *condition*.

## Semafori

Per l'interazione tra i thread si utilizzano i semafori *commesso, consegna, HamburgerNelloScaffale* e *PostoLiberoInScaffale*.

Il generico cliente esegue la seguente funzione:

```
thread GenericoCliente
……
<entra nel negozio>;
V(clienti);
P(commesso);
<ordina un hamburger e paga>;
P(consegna);
<prende l'hamburger e va al tavolo a mangiare>
<esce dal fast food>
……
```

Si chiede di completare le funzioni eseguite dal thread Commesso e dal thread Cuoco, inserendo negli schemi sotto riportati le operazioni opportune sui semafori

### SOLUZIONE (variabili condizione)

```
thread Commesso
while true{
        lock.acquire(MutexOrdini);
        while (OrdiniPendenti== 0) AttesaOrdine.wait (MutexOrdini);
         //attende un ordine//
        OrdiniPendenti --;
        //gestisce gli ordini con politica FIFO (vero???)//;
        lock.release(MutexOrdini);
        lock.acquire(MutexScaffale);
        while (HamburgerPronti== 0) HamburgerNelloScaffale.wait (MutexScaffale)
        //attende la disponibilità di almeno un hamburger nello scaffale//
        <preleva un hamburger dallo Scaffale>
        HamburgerPronti --;
```

```
            PostoLiberoNelloScaffale.signal (MutexScaffale);
            lock.release(MutexScaffale);
            consegna.signal (MutexFittizio);
             //consegna un hamburger, riattivando i clienti in ordine FIFO (vero???)//
}

thread Cuoco
while true{
            <prepara un hamburger>
            lock.acquire(MutexScaffale);
            while (HamburgerPronti== 10)   PostoLiberoNelloScaffale.wait (MutexScaffale);
            //attende la disponibilità di almeno un posto libero nello scaffale //
            <deposita un hamburger sullo Scaffale>
            HamburgerPronti ++;
            HamburgerNelloScaffale.signal (MutexScaffale);
            //ha effetto solo se il commesso è sospeso//
            lock.release(MutexScaffale);
                        }
```

## Soluzione (semafori)

**Commesso**:

while true{

**V(commesso);**

***P(clienti);***

//attende un ordine//

**P(HamburgerNelloScaffale)**

//attende la disponibilità di almeno un hamburger nello scaffale//

<preleva un hamburger dallo Scaffale e lo consegna la cliente>

**V(consegna);**

//consegna un hamburger, riattivando i clienti //

**V(PostoLiberoInScaffale);**

// segnala al cuoco disponibilità nello scaffale//

}

**Cuoco:**

while true{

<prepara un hamburger>

**P(PostoLiberoInScaffale);**

//attende la disponibilità di almeno un posto libero nello scaffale //

<deposita un hamburger sullo Scaffale>

**V(HamburgerNelloScaffale);**

//segnala al commesso disponibilità di hamburger//

}