Esercizio 1

Siano *A*, *B*, *C*, *D* ed *E* le procedure che un insieme di processi *P1*, *P2*, ..., *PN* possono invocare e che devono essere eseguite rispettando i seguenti vincoli di sincronizzazione:

Sono possibili solo due sequenze di esecuzioni delle procedure, sequenze tra loro mutuamente esclusive:

- la prima sequenza prevede che venga eseguita per prima la procedura *A*. a cui puo' seguire esclusivamente líesecuzione di una o piu' attivazioni concorrenti della procedura *B;*

- la seconda sequenza e' costituita dallíesecuzione della procedura *C* a cui puo' seguire esclusivamente l'esecuzione della procedura *D*, o in alternativa a *D* della procedura *E*.

Una volta terminata una delle due sequenze una nuova sequenza puo' essere di nuovo iniziata.

utilizzando il meccanismo dei semafori, realizzare le funzioni *StartA*, *EndA*, *StartB*, *EndB*, ... , *StartE*, *EndE* che, invocate dai processi clienti *P1*, *P2*, ..., *PN* rispettivamente prima e dopo le corrispondenti procedure, garantiscano il rispetto dei precedenti vincoli. Nel risolvere il problema non e' richiesta la soluzione ad eventuali problemi di starvation.

Esercizio 2

In un sistema organizzato secondo il modello a memoria comune viene definita una risorsa astratta R sulla quale si puo` operare mediante tre procedure identificate, rispettivamente, da ProcA, ProcB e Reset. Le due procedure ProcA e ProcB, operano su variabili diverse della risorsa R e pertanto possono essere eseguite concorrentemente tra loro senza generare interferenze. La procedura Reset opera su tutte le variabili di R e quindi deve essere eseguita in modo mutuamente esclusivo sia con ProcA che con ProcB.

1) Se i tre processi PA, PB e PR invocano, rispettivamernte, le operazioni ProcA, ProcB e Reset, descrivere una tecnica che consente ai processi PA e PB di eseguire le procedure da essi invocate senza vincoli reciproci di mutua esclusione, garantendo invece la mutua esclusione con l'esecuzione della procedura Reset invocata da PR. Nel risolvere il problema garantire la priorita` alle esecuzioni di Reset rispetto a quelle di ProcA e ProcB.

2) Qualora i processi che invocano le procedure ProcA e ProcB siano piu` di due (PA1,.....PAn e PB1,....,PBm) riscrivere la soluzione garantendo anche la mutua esclusione tra due o piu` attivazioni di ProcA e tra due o piu` attivazioni di ProcB.

Esercizio 3

In un sistema organizzato secondo il modello a memoria comune si vuole realizzare un meccanismo di comunicazione tra processi che simula una *mailbox* a cui *M* diversi processi mittenti inviano messaggi di un tipo *T* predefinito e da cui prelevano messaggi *R* diversi processi riceventi.
Per simulare tale meccanismo si definisce il tipo *busta* di cui si suppone di usare *N* istanze (costituenti un pool di risorse equivalenti). Un gestore *G* alloca le buste appartenenti al pool ai processi mittenti i quali, per inviare un messaggio eseguono il seguente algoritmo:

send (*messaggio*) => 1 - richiesta al gestore *G* di una *busta* vuota;
2 - inserimento nella busta del *messaggio*;
3 - accodamento della *busta* nella *mailbox*;

Analogamente ogni processo ricevente, per ricevere un messaggio, esegue il seguente algoritmo:

messaggio = *receive*() => 1 - estrazione della *busta* dalla *mailbox*;
2 - estrazione del *messaggio* dalla *busta*;
3 - rilascio della *busta* vuota al *gestore*

Realizzare il precedente meccanismo utilizzando i semafori e garantendo che la *receive* sia bloccante quando nella *mailbox* non ci sono *buste*, e che la *send* sia bloccante quando non ci sono piu` *buste* vuote disponibili. Indicare, in particolare, come viene definita la *busta*, il codice del *gestore* e della *mailbox*, il codice delle due funzioni *send* e *receive*.

Per garantire la ricezione FIFO dei messaggi, organizzare le buste nella *mailbox* mediante una coda concatenata. Il gestore alloca le buste vuote ai processi mittenti adottando una politica prioritaria in base ad un parametro *priorita`* che ciascun processo indica nel momento in cui chiede una *busta* vuota al gestore. La priorita` che ogni processo indica puo` essere 0 (priorita` massima, 1 (priorita` intermedia) oppure 2 (priorita` minima) Per richieste specificate con uguale priorita` viene seguita la politica FIFO. Si puo` supporre che le code semaforiche siano gestite FIFO.

Esercizio 4

Scrivere un programma multi-thread che simuli il gioco della morra cinese. In tale programma ci devono essere 3 thread:

- 2 thread simulano i giocatori;
- 1 thread simula l'arbitro.

Il thread arbitro ha il compito di:

1. "dare il via" ai due thread giocatori;
2. aspettare che ciascuno di essi faccia la propria mossa;
3. controllare chi dei due ha vinto, e stampare a video il risultato;
4. aspettare la pressione di un tasto da parte dell'utente;
5. ricominciare dal punto 1.

Ognuno dei due thread giocatori deve:

1. aspettare il "via" da parte del thread arbitro;
2. estrarre a caso la propria mossa;
3. stampare a video la propria mossa;
4. segnalare al thread arbitro di aver effettuato la mossa;
5. tornare al punto 1.

Per semplicita', assumere che la mossa sia codificata come un numero intero con le seguenti define:

```
#define CARTA     0
#define SASSO     1
#define FORBICE   2
```

e che esista un array di stringhe cosi' definito:

```
char *nomi_mosse[3] = {"carta", "sasso", "forbice"};
```

Esercizio 5

In un programma multithread, ogni thread esegue il seguente codice:

```
void *thread(void *arg)
{
  int voto = rand()%2;

  vota(voto);

  if (voto == risultato()) printf("Ho vinto!\n");
  else printf("Ho perso!\n");

  pthread_exit(0);
}
```

cioe' ogni thread:

- esprime un voto, che puo' essere 0 o 1, invocando la funzione vota(), la quale registra il voto in una struttura dati condivisa che per comodita' chiameremo "urna";

- aspetta l'esito della votazione invocando la funzione risultato(), la quale controlla l'urna e ritorna 0 o 1 a seconda che ci sia una maggioranza di voti 0 oppure di voti 1.

- se l'esito della votazione e' uguale al proprio voto, stampa a video la stringa "Ho vinto", altrimenti stampa la stringa "Ho perso";

Supponiamo che ci siano un numero dispari di threads nel sistema. Il candidato deve implementare la struttura dati

```
struct {
  ...
} urna;
```

e le funzioni:

```
void vota(int v);

int risultato(void);
```

in modo che i thread si comportino come segue:

- Se l'esito della votazione non puo' ancora essere stabilito, la funzione risultato() deve bloccare il thread chiamante. Non appena l'esito e' "sicuro" (ovvero almeno la meta' piu' uno dei threads ha votato 0, oppure almeno la meta' piu' uno dei threads ha votato 1) il thread viene sbloccato e la funzione risultato() ritorna l'esito della votazione. I thread vengono sbloccati il piu' presto possibile, quindi anche prima che abbiano votato tutti.

Utilizzare i costrutti pthread_mutex_xxx e pthread_cond_xxx visti a lezione.

Esercizio 6

In questo compito verrà affrontato il celebre problema dei filosofi a tavola, che può essere così schematizzato :

Un certo numero N di filosofi siede intorno ad un tavolo circolare al cui centro c'è un piatto di spaghetti e su cui sono disposte N forchette (in modo che ogni filosofo ne abbia una alla sua destra e una alla sua sinistra).

Ogni filosofo si comporta nel seguente modo :

- Trascorre il suo tempo pensando e mangiando.
- Dopo una fase di riflessione passa a una di nutrizione.
- Per mangiare acquisisce prima la forchetta alla sua destra, quindi quella alla sua sinistra e mangia usando entrambe.
- Una volta che ha finito di mangiare rimette a posto le due forchette che ha usato.

Il candidato :

- modelli le forchette come risorse condivise, associando quindi un semaforo ad ogni forchetta, ed ogni filosofo come un thread e ne scriva quindi il relativo codice.
- modelli le fasi di pensiero e nutrizione come dei cicli for a vuoto di lunghezza definita dalla macro DELAY.
- definisca il numero di filosofi (e quindi anche di forchette) usando la macro NUM_FILOSOFI.
- si sincronizzi con la fine di tutti i thread.

Si tenga presente che è stato dimostrato che, per evitare situazioni di deadlock, uno dei filosofi deve invertire l'ordine di acquisizione delle forchette (quindi acquisirà prima quella alla sua sinistra e poi quella alla sua destra).

Esercizio 7

Un negozio di barbieri ha tre barbieri, e tre poltrone su cui siedono i clienti quando vengono per tagliarsi la barba.

C'è una sala d'aspetto con un divano (max 4 persone; gli altri aspettano fuori dalla porta).

C'e' un cassiere, che può servire solamente un cliente (con barba tagliata) alla volta.

Scrivere il processo cliente che cerca di entrare nel negozio per farsi tagliare la barba,

Suggerimenti:
- considerare i barbieri, il cassiere ed il divano come risorse condivise.
- modellare il processo di taglio barba come un ciclo di SHAVING_ITERATIONS iterazioni
- modellare il processo di pagamento come un ciclo di PAYING_ITERATIONS iterazioni
- dopo che un cliente ha pagato esce (th thread muore) oppure, dopo un delay di alcuni secondi (usare la primitiva sleep()), si accoda nuovamente per farsi tagliare la barba.

Esercizio 8

In un sistema a memoria comune quattro processi applicativi *P1, P2, P3, P4* competono per l'uso di due risorse equivalenti. Si chiede di scrivere il codice del gestore per allocare dinamicamente le risorse ai richiedenti tenendo conto che, all'atto della richiesta, il richiedente specifica anche un parametro *T*:*integer* che denota un timeout (in termini di tick di orologio) scaduto il quale, se la richiesta non è stata esaudita il processo richiedente viene comunque svegliato pur non avendo disponibile la risorsa richiesta.

identificate le procedure del gestore con i necessari parametri, scrivere il codice del gestore supponendo che ad ogni tick di orologio vada in esecuzione il relativo processo orologio. Se necessaria, è disponibile la primitiva di sistema *PE* che restituisce l'indice del processo in esecuzione. Non è specificata nessuna politica per quanto riguarda la priorità dei processi richiedenti.

Esercizio 9

Utilizzando i semafori, realizzare un meccanismo di comunicazione fra 5 processi mittenti ciclici (*M0*, *M1*, …, *M4*) e un processo ricevente ciclico *R*. Il tipo di dati (messaggi) che i processi si scambiano è costituito da un array di 5 elementi di tipo *T*. Il *buffer* di comunicazione può contenere un unico messaggio (ovviamente costituito da un array di 5 elementi di tipo *T*). All'interno di ogni suo ciclo il processo *R* può ricevere il messaggio quando questo è pronto nel *buffer*. All'interno di ogni suo ciclo il processo mittente *Mi* (*i*=0, 1, .., 4) invia un valore di tipo *T* che va riempire l'elemento di indice *i* del *buffer*. Quando tutti gli elementi del *buffer* contengono un valore inviato dal corrispondente mittente (tutti valori inviati dai mittenti all'interno dello stesso ciclo della loro esecuzione) il *buffer* è pronto per essere letto dal processo *R*.

1) scrivere il codice delle funzioni *send* eseguite da ciascun mittente, della funzione *receive* eseguita dal ricevente e dettagliare la struttura dati del meccanismo di comunicazione. Per quanto riguarda i semafori indicare, per ciascuno di essi, lo scopo per cui vengono usati e, nell'ipotesi che questi siano semafori di mutua esclusione, giustificarne la necessità.
2) indicare come dovrebbe essere modificata la precedente soluzione se i processi mittenti fossero costituiti da due gruppi di 5 processi ciascuno MA0, MA1,..,MA4 e MB0, MB1,…,MB4 con il vincolo che, se uno dei processi mittenti di un gruppo, durante uno dei propri cicli, riesce per primo a inserire il proprio dato nel buffer, allora il buffer deve essere riempito con i soli dati provenienti dai processi di quel gruppo.

Esercizio 10

In un sistema organizzato secondo il modello a memoria comune due risorse *RA* e *RB* sono allocate dinamicamente ai processi *P1*, *P2*, …., *Pn* tramite un comune gestore *G*. Ogni processo può richiedere al gestore l'uso esclusivo della risorsa *RA* (tramite la funzione *RicA*) o della risorsa *RB* (tramite la funzione *RicB*) oppure l'uso esclusivo di una qualunque delle due (tramite la funzione *RicQ*). Un processo può però richiedere anche l'uso esclusivo di entrambe le risorse (tramite la funzione *Ric2*). Chiaramente, dopo che la risorsa (o le risorse) è stata (sono state) utilizzata (utilizzate), il processo la (le) rilascia al gestore tramite le opportune funzioni di rilascio.
Utilizzando il meccanismo semaforico, realizzare il gestore *G* con tutte le funzioni di richiesta e di rilascio necessarie per implementare quanto sopra specificato. Nel realizzare il gestore si tenga conto delle seguenti regole di priorità: vengono privilegiate le richieste generiche rispetto alle richieste della risorsa *RA* e queste nei confronti delle richieste della risorsa *RB* e infine queste nei confronti delle richieste di entrambe le risorse. Durante l'allocazione di una risorsa, in seguito ad una richiesta generica, se sono disponibili entrambe le risorse, allocare per prima la risorsa *RA*.

Esercizio 11

All'interno di un'applicazione, vari processi (ciclici) cooperano, secondo i criteri specificati nel seguito, accedendo ad una risorsa condivisa *R*.
Alcuni fra questi processi, arrivati ad un certo punto della loro esecuzione, per procedere devono controllare che si sia verificato un certo evento (che indicheremo come *eventoA*). In caso positivo procedono senza bloccarsi altrimenti attendono che l'evento si verifichi. Questo controllo viene effettuato accedendo alla risorsa *R* tramite la funzione *testaA*. Altri processi sono preposti a segnalare l'occorrenza dell'evento invocando la funzione *segnalaA*. Gli eventi segnalati sono però, per così dire, "consumabili". In particolare, se un processo, invocando *testaA* verifica che *eventoA* si è già verificato, procede senza bloccarsi ma resetta l'evento, cioè, da quel momento in poi se un altro processo invoca *testaA*, questo si deve bloccare in attesa che, tramite *segnalaA* l'evento sia di nuovo segnalato. Se all'atto dell'invocazione di *testaA* l'evento non si è ancora verificato, o se verificato è già stato consumato, allora il processo richiedente si blocca. Viceversa, quando viene invocata *segnalaA*, se non ci sono processi in attesa dell'evento, allora questa segnalazione resta disponibile per il primo processo che invochi *testaA*, se viceversa uno o più processi sono in attesa dell'evento, tutti vengono riattivati e l'evento viene consumato. Infine, se all'atto dell'invocazione di *segnalaA* e` ancora presente un precedente evento non consumato, il processo segnalante si deve bloccare in attesa che il precedente evento venga consumato.

1- utilizzando il meccanismo semaforico implementare *R* con le due funzioni *testaA* e *segnalaA*.
2- Supponendo che i processi che segnalano l'evento siano *P1*, …., *PN*, e che ciascun processo nell'invocare la funzione *segnalaA* passi il proprio nome (intero tra 1 e N) come parametro, modificare la precedente soluzione imponendo una priorità tra questi processi quando devono essere risvegliati dopo un blocco in modo tale che *P1* abbia priorità su *P2* ecc.

Esercizio 12
Un gestore di risorse equivalenti alloca dinamicamente ad *N* processi clienti 9 istanze di uno stesso tipo di risorse (*R1*, *R2*, …., *R9*).
Gli *N* processi clienti (P_0.,……., P_{N-1}) vengono serviti dal gestore privilegiando i processi di indice più basso.
Un processo che chiede una risorsa si blocca se all'atto della richiesta non ci sono risorse disponibili in attesa che una risorsa venga rilasciata. Però il processo rimane bloccato, al massimo, per *t* quanti di tempo successivi al blocco (dove *t* denota un parametro intero passato in fase di chiamata della funzione *richiesta*). Quando un processo viene svegliato per lo scadere del tempo massimo da lui indicato, la funzione *richiesta* termina senza allocare niente al richiedente.

Utilizzando il meccanismo semaforico realizzare il codice del *gestore* che offre le tre seguenti funzioni membro:

- int *richiesta* (int *t,* int *p*) dove *t* denota il time-out espresso come numero di quanti di tempo, mentre *p* denota l'indice del processo richiedente. La funzione restituisce l'indice della risorsa allocata (1,…, 9) oppure 0 se il processo termina la funzione dopo un time-out senza aver allocato niente.

- void *rilascio* (int *r*) dove r denota l'indice della risorsa rilasciata.

- void *tick* () funzione chiamata dal driver dell'orologio in tempo reale ad ogni interruzioni di clock.

Esercizio 13

Alcuni processi *mittenti* inviano periodicamente messaggi di un tipo *mes* (che si suppone già definito), a due processi riceventi (*R1* ed *R2*). I messaggi vengono inviati tramite una *mailbox* da realizzare per mezzo di un buffer circolare di *N* posizioni. Tutti i messaggi inviati devono essere ricevuti da entrambe i processi riceventi e ciascun processo deve ricevere tutti i messaggi in ordine FIFO (quindi una posizione della mailbox una volta riempita con un messaggio, potrà essere resa libera solo dopo che tale messaggio sia stato ricevuto dai due processi *R1* ed *R2*). Chiaramente, un generico messaggio potrà essere ricevuto prima da *R1* e poi da *R2* mentre un altro messaggio potrà essere ricevuto in ordine inverso a seconda dei rapporti di velocità tra i processi.

a) utilizzando il meccanismo dei semafori, realizzare la precedente *mailbox* con le tre funzioni *send*, *receive1* e *receive2* invocate, rispettivamente, da ciascun mittente e da *R1* e *R2*.
b) ripetere la soluzione con il vincolo che ogni messaggio debba sempre essere ricevuto prima da *R1* e poi da *R2*.

Esercizio 14

Un insieme di *N* processi periodici P_0, P_1, …., P_{N-1} eseguono tutti il seguente programma (dove le porzioni <codice privato> identificano sequenze di istruzioni non specificate e in genere diverse da processo a processo):

```
Process Pi                          // (i=1,2,….,N)
    {   While (true)
{
     <codice privato>;
     A( );
              <codice privato>;
               B( );
              <codice privato>;
              C( ); oppure D ( );
              <codice privato>;
     fine_ciclo( );
          }
     }
```

con i seguenti vincoli di sincronizzazione:

- le funzioni A( ) devono essere eseguite in mutua esclusione privilegiando, in caso di competizione P_0, su P_1, ecc.;
- le funzioni B ( ) devono essere eseguite non solo in mutua esclusone ma anche nell'ordine P_0, prima di P_1, ecc.;
- alcuni processi eseguono la funzione C ( ) ed altri la funzione D ( ) col vincolo che le C possono essere eseguite tra loro in concorrenza e analogamente le D ma non deve esserci concorrenza tra le esecuzioni di C e quelle di D;
- l'operazione fine_ciclo( ) sincronizza i processi nel senso che i processi che la eseguono per primi attendono che anche gli altri siano arrivati. Quando tutti gli N processi hanno invocato la funzione, tutti ripartono e possono rieseguire il ciclo successivo.

Utilizzando il meccanismo semaforico indicare i prologhi e gli epiloghi che dovranno contraddistinguere le funzioni A, B, C e D affinche' abbiano il comportamento desiderato, i parametri delle funzioni (se servono), le strutture dati necessarie e il codice della funzione fine_ciclo( ).

Esercizio 15

Simulare il comportamento di conto corrente bancario sul quale piu` utenti (processi) possono effettuare prelievi e/o depositi, con il vincolo che il conto non possa mai andare in rosso (il totale depositato deve essere sempre maggiore o uguale a zero. Utilizzare il costrutto monitor per simulare il conto corrente, con due procedure: *deposito* e *prelievo.*.

Evitare di svegliare un processo quando non si Ë sicuri che possa ripartire.

Fornire due soluzioni: una che prevede che un processo non sia bloccato quando le condizioni logiche per la sua esecuzione sono verificate ed una che prevede che i prelievi siano serviti rigorosamente FIFO, cioË che un processo che puÚ essere servirto si blocchi se qualcuno ha richiesto un prelievo prima di lui che non puÚ essere servito.

Esercizio 16

Supponiamo che nella facolta` vi sia un solo locale destinato ai servizi igienici, costituito da 5 toilette. Il locale Ë condiviso da tutti gli studenti, uomini e donne, con la regola che in ogni istante l'intero locale puÚ essere utilizzato escusivamente o da donne o da uomini.

a) Utilizzando il meccanismo semaforico, simulare quanto necessario per sincronizzare gli utenti del locale servizi. (nel descrivere la soluzione non preoccuparsi di individuare particolari strategie di allocazione o di evitare la starvation).

b) Riscrivere la soluzione implementando una strategia di accesso che elimini problemi di starvation.

Esercizio 17

Ciascuno degli *N* processi *P1*, *P2*, ..., *PN*, contribuisce a fornire il risultato di una computazione. Nel corpo del generico processo *Pi* viene eseguita una funzione *Fi* destinata a contribuire, per la sua parte, al risultato finale. Supponendo che la generica esecuzione della *Fi* possa fallire, dopo la sua esecuzione il processo esprime un voto per indicare se la sua parte di risultato e` corretta oppure no. Successivamente il processo si pone in attesa di sapere qual'e` l'esito delle votazioni degli altri processi. Se tale esito e` positivo il processo continua per la sua strada. Se l'esito e` negativo allora esegue una operazione *Gi* atta ad annullare gli effetti relativi all'esecuzione della *Fi* e quindi continua per la sua strada.
Per questo motivo si suppone di avere a disposizione la risorsa astratta *Controllo* sulla quale i processi operano mediante le due seguenti operazioni:

void entry *parere* (int *i*, int *voto*)

chiamata da ogni processo per espremere il proprio parere sulla correttezza del risultato. Il parametro i va da 1 ad *N* e serve al processo chiamante per indicare il proprio indice, il parametro *voto* serve per indicare se il suo risultato e` corretto (valore 1) oppure no (valore 0)

int *esito(void)*;

funzione invocata da ogni processo, dopo aver espresso il proprio voto, per sapere l'esito della consultazione. Se la funzione restituisce il valore 1 l'esito e` positivo e il processo continua indipendentemente dal suo specifico risultato. Se il valore restituito e` 0 il processo prima di continuare esegue la funzione di recupero *Gi*.
L'esito della conultazione dovra` essere positivo se almeno *M* (con $M<N$) processi hanno espresso un voto positivo, altrimenti l'esito dovra` essere negativo.

```
void *Pi (void *arg)   // per ogni (1≤i≤N)
{
        Fi;
        voto(i,risultato);
        if (!esito) Gi;
       .....
       .....
}
```

Se quando un processo invoca la funzione *esito* non e` ancora possibile conoscere l'esito finale della consultazione il processo si blocca e verra` svegliato non appena sara` possibile conoscere l'esito finale.

a) realizzare la risorsa astratta *Controllo*, con le due operazioni *parere* ed *esito*, utilizzando il meccanismo semaforico.

Esercizio 18

F1, F2, F3, F4, F5 ed *F6* sono sei procedure condivise tra i processi *P1, P2, ..., PN*. Ciascun processo puo` chiamare, quando necessario, una qualunque delle sei procedure. Le stesse pero` sono soggette ai seguenti vincoli di sincronizzazione:

1) le esecuzioni di *F1* ed *F2* sono tra loro mutuamente esclusive;

2) l'esecuzione di *F3* puo` iniziare solo dopo che e` terminata l'esecuzione di *F1* o di *F2* e, viceversa, dopo che e` terminata l'esecuzione di *F1* o di *F2* l'unica procedura che puo` essere eseguita e` *F3*;

3) durante l'esecuzione di *F3* altre attivazioni di *F3* possono procedere concorrentemente;

4) l'esecuzione di *F4* puo` iniziare soltanto quando e` terminata l'esecuzione dell'ultima attivazione concorrente di *F3*;

5) Per evitare la starvation dovuta alle esecuzioni concorrenti di *F3*, una nuova attivazione di *F3* non viene consentita se ci sono chiamate pendenti di *F4*. Cio` significa che al termine di *F1* o di *F2* vengono attivate tutte le invocazioni pendenti di *F3*; successivamente, durante le loro esecuzioni, nuove chiamate di *F3* vengono accettate solo se non ci sono richieste pendenti di *F4*; se alla fine di *F1* o di *F2* non ci sono chiamate pendenti di *F3* mentre ci sono chiamate pendenti di *F4* e` comunque necessario, per rispettare il vincolo 2, attendere una chiamata di *F3* prima di poter accettare una chiamata di *F4*;'

6) *F6* puo` essere eseguita solo dopo che *F5* e` stata terminata e viceversa, dopo la terminazione di *F5* la sola procedura che puo` essere esegiuita e` *F6*

7) le sequenze di esecuzioni di:
   - *F1* o *F2* seguita da una o piu` esecuzioni di *F3* e quindi da *F4*
     e
   - *F5* seguita da *F6*

devono avvenire in modo mutuamente esclusivo.

Quindi, inizialmente puo` iniziare solo l'esecuzione di *F1* o di *F2* o di *F5*. Se inizia *F1* oppure *F2*, la sola procedura eseguibile successivamente e` la *F3*, eventualmente in concorrenza con altre *F3*, e quindi *F4*. Se, viceversa, inizia *F5* la sola procedura eseguibile successivamente e` la *F6*. Alla fine dell'esecuzione di *F4* in un caso, o di *F6* nell'altro, si torna nella condizione iniziale.

a) utilizzando il meccanismo dei semafori scrivere *StartF1, EndF1, StartF2, EndF2, ... , StartF6, EndF6* concepite come procedure di una risorsa condivisa *R*.

Esercizio 19

I processi appartenenti ad certo un insieme (mittenti) inviano messaggi (tutti dello stesso tipo *T*) ai processi appartenenti ad un secondo insieme (riceventi)
Il meccanismo di scambio di messaggi viene realizzato tramite un'unica *mailbox* di dimensione fissa contente al piu` *N* messaggi (gia` inviati e non ancora ricevuti).
Ciascun mittente puo` inviare messaggi di tipo normale (tramite la procedura *sendN*) oppure messaggi prioritari (tramite la procedura *sendP*). I messaggi vengono ricevuti tramite la procedura *receive* che:

- se ci sono messaggi prioritari estae in ordine FIFO uno di essi,
- se non ci sono messaggi prioritari e ci sono messaggi normali, estrae in ordine FIFO uno di questi;
- se non ci sono messaggi la receive e` bloccante.

Le operazioni send sono bloccanti quando la mailbox e` piena. Quando piu` processi mittenti sono bloccati e diventa possibile svegliarne uno, si privilegia, se ce ne sono, uno di quelli che vogliono inviare messaggi prioritari rispetto a quelli che vogliono inviare messaggi normali.

a) risolvere il problema implementando la *mailbox* con le operazioni *sendN*, *sendP* e *receive* tramite il meccanismo semaforico. Per semplicita` si suppone di conoscere che la strategia di gestione delle code semaforiche e` FIFO

b) ripetere la soluzione nell'ipotesi che i processi riceventi possano disporre, oltre che della *receive*, che funziona come sopra specificato, anche della *receiveP*, per ricevere escusivamente un messaggio prioritario (ed in questo caso essa e` bloccante se messaggi prioritari non sono presenti nella *mailbox*) e della *receiveN*, per ricevere esclusivamente messaggi normali (ed in questo caso essa e` bloccante se messaggi normali non sono presenti nella *mailbox*).
   In questa seconda soluzione, se all'atto di una *sendP* si deve svegliare qualcuno si devono privilegiare, se ce ne sono, riceventi bloccati sulla *receiveP* rispetto a quelli bloccati sulla *receive.* Viceversa se alla fine di una *sendN* si puo` svegliare qualcuno devono essere privilegiati riceventi bloccati su *receive*, se ce ne sono, rispetto ai bloccati su *receiveN.*

Esercizio 20

Un meccanismo di comunicazione tra processi e` costituto da una mailbox di dimensione massima pari ad N messaggi (per semplicita` supponiamo i messaggi di tipo intero). In tale mailbox inviano messaggi molti processi mittenti ciascuno dei quali ha a disposizione due diverse funzioni (*send1* e *send2*). La *send1* ha un parametro *m* che denota il messaggio da inviare e funziona come una generica send (bloccante se la mailbox e` piena, altrimenti inserisce il messaggio *m* nella mailbox in ordine FIFO). La *send2* ha due parametri *m1* ed *m2* che denotano due diversi messaggi che vengono inseriti nella mailbox, anch'essi in ordine FIFO, prima *m1* e poi *m2*, e se non c'e` spazio per i due la *send2* e` bloccante. Vi sono poi *R* processi riceventi, identificati tramite gli indici da 0 ad *R*-1, che usano la funzione *receive* per estrarre messaggi dalla mailbox. La *receive*, al solito, e` bloccante se non ci sono messaggi nella mailbox altrimenti restituisce il primo tra i messaggi presenti. Supponiamo che la *receive* abbia un parametro *j* che denota l'indice del ricevente che l'ha invocata.

a) utilizzando il meccanismo semaforico, realizzare la mailbox con le precedenti funzioni adottando le seguenti politiche:
   - fra i processi riceventi deve essere privilegiato quello di indice minore;
   - nessun processo ricevente deve essere ovviamente bloccato se vi sono messaggi nella mailbox;
   - nessuna send deve essere bloccante se nella mailbox c'e` spazio per il (oppure i) messaggi da inviare;
   - nell'accesso alla mailbox devono essere privilegiati i processi mittenti che invocano *send2* rispetto a quelli che invocano *send1* se le condizioni lo consentono.

Esercizio 21

Un gestore *G* alloca dinamicamente le risorse *R1*,...., *RM*, appartenenti ad un pool di risorse equivalenti, ai processi *P1*,..., *Pn*. Ogni processo puo` richiedere al gestore l'uso di una risorsa, mediante la procedura *Richiesta1*, oppure di due risorse contemporaneamente, mediante la procedura *Richiesta2*. Dopo l'uso il processo restituisce la (o le) risorse al gestore mediante le procedure *Rilascio1* e *Rilascio2*.

a) utilizzando il meccanismo semaforico, realizzare il gestore *G*, con le 4 precedenti procedure, e implementando le seguenti strategie: ogni richiesta deve essere bloccante solo se la (o le) risorse richieste non sono disponibili e adottando inoltre la seguente priorita`: fra le richieste pendenti devono essere privilegiate le richieste di due risorse rispetto alle richieste di una sola risorsa e, al loro interno, le richieste pendenti provenienti dal processo di indice piu` basso (*P1* privilegiato rispetto a *P2* ecc).

b) riscrivere la precedente soluzione privilegiando ulteriormente le richieste di due risorse rispetto alle richieste di una sola risorsa nel seguente modo: una richiesta singola deve essere bloccante anche se una risorsa e` disponibile qualora ci siano richieste pendenti per due risorse.

Esercizio 22

Un processo *mittente* invia periodicamente messaggi di un tipo *T* precedentemente definito, a tre processi riceventi (*ricevente1, ricevente2, ricevente3*). I messaggi inviati devono essere ricevuti (ovviuamente in ordine FIFO) da tutti e tre i processi riceventi.
Il meccanismo di comunicazione e` costituito da una *mailbox* realizzata tramite un buffer circolare di *N* posizioni.

a) utilizzando il meccanismo semaforico, realizzare la precedente *mailbox* rispettando i seguenti criteri:

   a1) ogni messaggio deve essere ricevuto dai tre riceventi prima di essere eliminato dalla mailbox senza che debba essere previsto nessun ordine particolare fra i processi riceventi (ad esempio un messaggio puo` essere ricevuto prima da *ricevente1,* poi da *ricevente3* e quindi da *ricevente2* mentre per altri messaggi puo` essere diverso l'ordine di ricezione da parte dei vari riceventi).

   a2) ripetere la soluzione ma con il vincolo che ogni messaggio deve essere ricevuto esattamenrte nel seguente ordine: prima da *ricevente1,* poi da *ricevente2* e infine da *ricevente3.*

   Nel fornire le due soluzioni, si richuiede di utilizzare eventuali semafori di mutua esclusione al fine di garantire l'assenza di interferenze soltanto se necessario.

Esercizio 23

Utilizzando i semafori realizzare un gestore di due risorse equivalenti *R1* ed *R2* Il gestore alloca le risorse ai processi *P1* ,...., *Pn* che possono richiedere una qualunque delle due ricorse oppure le due risorse contemporaneamente tramite le procedure *Richiesta1, Richiesta2.*
La, o le, risorse sono rilasciate tramite le procedure di rilascio: *Rilascio1, Rilascio2.* Al rilascio, se ci sono processi bloccati che possono essere svegliati, devono essere privilegiati i processi richiedenti entrambe le risorse rispetto a quelli che richiedono risorse singole. Indicare il tipo di semantica della signal che viene utilizzata.