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*.